INFORM 7

The Program

Complete Program

Build 5Z71 Graham Nelson

Inform is a natural-language design system for interactive fiction, first created in 1993. To most users it seems a single unified tool, but in fact is made up of core software, common to all platforms, combined with substantial user interfaces written independently for Mac OS X, Windows and Linux, and with documentation and examples. The core material is in turn divided into:

Chapters 1 to 14: the source code to the NI compiler, written in C

Appendix A: the Standard Rules, written in Inform 7

Appendix B: the template layer, written in Inform 6

Each of these chapter blocks is divided up into one or more named sections, which have both full names ("Grammar Lines") and abbreviations (`12/gl`*, the 12 signifying Chapter 12). Finally, each section is divided into numbered "paragraphs", some named and others not. Code can thus be approximately located by "postal codes" such as* `12/gl`*.§7.*

Of its nature, Inform must perform a computational task which is difficult to specify formally, particularly since part of its aim is to cope well with incorrect input from an inexperienced user. It has become a complex program some 120,000 lines in length, and like all such it must mitigate its complexity using internal stylistic conventions and principles of organisation. To this end it follows the "literate programming" dogma of Donald Knuth, an idea which had in any case influenced Inform's own design. In LP, a single source ("web") is both "tangled" into a functional form and also "woven" into a typeset form suitable for human readers. Inform uses its own LP tool, `inweb`*, an adaptation of Knuth's* `CWEB` *which scales better to large multi-target projects.*

The Inform project's main goal is to publish the entire core Inform code base, beginning in April 2008 with public drafts of Appendices A and B, approximately 600pp of material. These use only the simplest form of LP where tangling is minimal, and the reader needs no previous experience of the genre.

Preface P/pref

Purpose
An introduction to the reader.

P/pref.§2 Build number; §3 Basic method; §4 Organization into chapters; §5 Choice of language

§**1.** Inform 7 is a free open-source system for the design of interactive fiction. This is the source code to NI, which is the component of I7 used for turning a natural language description of a model world into a valid implementation in Inform 6 (I6). This is one of the two uninteractive steps on the assembly line:

$$\textit{Natural language} \longrightarrow \textbf{NI} \longrightarrow \textbf{I6} \longrightarrow \begin{cases} \textit{Z-machine story file} \\ \textit{Glulx story file} \end{cases}$$

The Inform user interface automates this process and also supplies both ends: editing the natural language source and running the appropriate interpreter to play the resulting story file. The three user interfaces, by Andrew Hunter for Mac OS X, by David Kinder for Windows, and by Philip Chimento and Adam Thornton for Linux, are not documented here.

There are a further three substantial software layers needed to use I7:

(i) the Standard Rules file, written in I7 source text but with certain permitted additional syntaxes: this is formally an Extension, though its presence is essential;

(ii) the `.i6` files, written in a metalanguage resembling that of I6, which serves both as a repository of useful I6 routines for NI-compiled programs, and also as a description at the top level of what NI should do and in what order;

(iii) the "natural" variant of the I6 library, which is fundamentally the standard I6 library but with large parts of the action verb subroutines removed, and with numerous minor modifications and hooks added to the parser and the actions engine.

Any reader who has used I6 can take (iii) for granted, but a reading of (i) and (ii) might well be more useful in giving an appreciation of I7's internals than reading the NI source would be.

This is not intended as a manual for Inform 7: *Writing with Inform* and *The Inform Recipe Book*, the interleaved volumes of documentation included in the application and also published online, are the user guide. Nor is there much comment here on how the design was made: for some of the early history, see the "white paper" for the Inform 7 project, *Natural Language, Semantic Analysis and Interactive Fiction*, which covers the basic decisions made. The commentary in the source code assumes that the overall strategy is wise and gets on with its practical implementation.

§**2. Build number.** Each time the master copy of NI is modified and recompiled, the build number (digit-letter-digit-digit) is advanced. Build numbers do not reflect a hierarchical branching of the source, but are simply a way to encode a large number in four printable digits. Letters I and O are skipped, and the tailing two digits run from 01 to 99.

Build 1A01 was the first rough draft of a completed compiler: but it did not synchronise fully with the OS X Inform application until 1G22 and private beta-testing did not begin until 1J34. Other milestones include time (1B92), tables (1C86), component parts (1E60), indexing (1F46), systematic memory allocation (1J53), pattern matching (1M11), the map index (1P97), extension documentation support (1S39) and activities (1T89). The first round of testing, a heroic effort by Emily Short and Sonja Kesserich, came informally to an end at around the 1V50 build, after which a general rewriting exercise began. Minor changes needed for David Kinder's Windows port began to be made with 1W80, but the main aims were to increase speed and to improve clarity of source code. Hashing algorithms adapted to word-based syntax were introduced in 1Z50; the prototype parser was then comprehensively rewritten using a unified system to handle ambiguities and avoid blind alleys. A time trial of 2D52 against 1V59 on the same, very large, source text showed a speed increase of a factor of four. A second stage of rewriting, to generalise binary predicates and improve grammatical accuracy, began with 2D70. By the time of the first public beta release, 3K27, the testing tool `inform-test` had been written (it subsequently evolved into today's `intest`), and Emily Short's extensive suite of Examples had been worked into the verification process for builds. The history since 3K27 is recorded in the published change log.

```
define NI_VERSION "Inform 7"
define NI_BUILD "[[Build Number]]"
```

§**3. Basic method.** NI is a large but simple program. First, it reads the source text and breaks it up lexically into words, quoted matter and punctuation. Then it divides this stream of words into sentences, marking some out as headings and others as requests to include extensions, requests which are immediately carried out. When there is nothing further to input, sentences are sorted into assertions (statements about the initial state of the world) and rules (instructions concerning play). Assertions are generally copular sentences relating things together, such as "two coins are in a closed box", and these are parsed first into "meaning list" tree structures and thence into propositions in predicate calculus which are declared to be true at the start of play. Each such proposition is then interpreted to extract single facts called "inferences", and reasonable "common sense" guesses are made to add further facts, which have a lower level of certainty. When all assertions have been processed, the many inferences are reconciled to construct a spatial model of the world which, following Occam's razor, is as simple as it can be, consistent with the inferences. Only if this can be done without inconsistency are the remaining data structures built: tables defined in the source, and the parsing grammar. At last the rules are identified, placed into relevant rulebooks, sorted by scope of applicability, and compiled into I6 routines. The final task is to produce a report and an index.

Though Inform uses propositions in predicate calculus as an intermediate state to hold the meaning of sentences, it does not store knowledge that way; though it makes elementary deductions and substitutions, it does not contain a theorem-prover by unification. In the 1960s divide between the "scruffy" school of storing domain-specific knowledge (McCarthy, Minksy, Papert, et al.) versus the "neat" school of purely propositional logic (Robinson, Kowalski, Colmerauer, et al.), Inform is scruffy. McCarthy thought that a program with "common sense" is one which "automatically deduces for itself a sufficiently wide class of immediate consequences of anything it is told and what it already knows". He wrote that back in 1959 and the "neat" methods have become astonishingly powerful since then – verifying the proof of the Prime Number Theorem via contour integration, for instance – but where the programmer knows in advance what sort of knowledge the program will be dealing with, I think the scruffy methods still have it. At any rate, Inform is a working program for its end users, not an experiment in AI.

§**4. Organization into chapters.** "I can't help feeling," wrote Michael Frayn recently, "that if someone had asked me before the universe began how it would turn out, I should have guessed something a bit less like an old curiosity shop and a bit more like a formal French garden – an orderly arrangement of straight avenues, circular walks, and geometrically shaped trees and hedges." I have an uneasy feeling that the same might be said about Inform, but at its most basic level, it is organised in six layers:

(L1) services for memory management, output streams, file input/output, and the handling of text as a sequence of words (chapters 1 to 3);

(L2) the A-parser, which gathers the source text from its various locations and divides it into sentences whose basic form can be recognised (chapter 4);

(L3) the S-parser, which resolves smaller excerpts of text into meanings, and the necessary apparatus to describe, manipulate and compile these meanings (chapters 5 to 7);

(L4) the world-builder, which uses the A-parser and S-parser in combination to break down the assertions in the source text to see what facts can be inferred about the initial state of the world, and then assembles those into a model, compiling it to a suitable Inform 6 description (chapters 8 to 10);

(L5) the change agent, compiling the phrases which are the means by which the state of the model world changes at run-time; and organising the rules which use those phrases, and the rulebooks holding those rules; and the actions and activities which structure those rulebooks, and the run-time grammar of commands which invoke those actions (chapters 11 to 13);

(L6) a thin control layer on top, following instructions from the command line parameters and from the "template" which lays out Inform's sequence of operation (chapter 14).

§**5. Choice of language.** NI is written in a literate programming extension of C called `inweb`. Besides interspersing documentation, this also provides concise syntax for parsing natural language; further apparent extensions, for logging to the debugging log and for automated memory allocation, are in fact provided by macros which are defined in the NI source.

To compile NI requires use of the Inform Tools, and in particular of `inweb`. A manual for each of the tools is included in the NI source (as documentation without code). Having compiled NI, it is important to check the result with `intest`: testing of bug fixes requires each change to be checked against a corpus of about 1200 test source texts. As of November 2008, this takes more than 3 minutes even using all four processors of a quad-3 GHz Mac Pro: it takes up to an hour on a Power PC-equipped iMac G4.

Thematic Index P/thix

Purpose

A gazetteer of interesting places to visit for the traveller who embarks into the Inform source code.

---

§**1.** Literate programs are traditionally typeset with an index, covering (mainly) each use of each identifier, and also some general concepts. In an age of highly searchable PDF files, that no longer seems helpful. Instead, this section is designed to help people who only want information on particular areas.

§**2. How NI interfaces with the Inform application.** These are independent programs and communicate at arm's length: the host application (usually providing a user interface) issues instructions at the command line when it launches NI, and then reads and acts upon its output. Each program has expectations about how the other will behave: these evolved gradually over the period 2003-07, and were for a long time never written down. The following indexes passages in the NI source which document these miscellaneous interface issues:

---

§**3. Miscellany.**

Licence and Copyright Declaration P/legal

Purpose

A verbatim copy of the Artistic License 2.0, which governs the use and distribution of Inform 7 materials.

---

§**1. Declaration of copyright.** The Inform 6 compiler and its associated library are copyright (c) Graham Nelson 1993-2007. The following materials associated with the Inform 7 project are copyright (c) Graham Nelson 2003-2007: the source code to the NI compiler; the I6T template files; the "natural" version of the I6 library; the icons, images and other non-executable resources supplied for use within Inform user interfaces; the Inform 7 documentation and website, including the Examples; the Inform Tools Suite; and the test cases used for regression-testing with `intest`, their associated material and ideal results. The *Inform [6] Designer's Manual, Fourth Edition* (DM4) is copyright (c) Graham Nelson 2001. We will call this whole body of intellectual property, with the exception of the typeset and printed form of the DM4, "the Package".

GN is the primary author of the Package, but not the only one: indeed the examples were very largely written by Emily Short, and others have also made modest contributions. However, GN holds the copyright to all of it so that there is a single point of reference for licencing of Inform.

Note that this does *not* include the officially recognised Inform user interface applications: the authors of these programs retain their own copyrights and are responsible for their own licencing. (It is a condition to be officially recognised by the Inform project that some form of free licence be used: in practice, most have chosen the GPL version 2.)

§**2. The former Inform licence, 1993-2007.** The Package was previously governed by a short, largely free but in some ways restrictive licence. (The text of this was published in the DM4, which is freely browsable online.)

The new licence is both more explicit and more liberal. The change was made for cultural reasons: over those years the Internet became a highly litigious domain, where even those who espouse freedoms are obsessed with the terms of licenses. Some users are now frightened off by any retention of rights by the author of software, worrying that the rug might be pulled at any moment. Some institutions enforce their ideological position by refusing to distribute software under a licence which they deem impure: for instance, under the old licence Inform could not be aggregated with Debian Linux, even though it had been completely free and open for 15 years. Also, there are some possible complications about the way NI and other components combine with GPL-licensed software for the user interfaces, and I feel reluctantly compelled to finally include a disclaimer of warranty, some degree of anti-patent-troll protection, and so forth – something I dislike doing since it implicitly acknowledges some validity in such distasteful legal claims.

There are two practical differences between the old and new licences. First, the new licence removes the requirement that any work of IF compiled by Inform should name "Inform" in its banner text. Authors are now legally free to suppress the banner text altogether, though I hope they won't: it's only a small gesture of respect to all those who have worked on Inform, and costs nothing. Second, the old licence forbade (at any rate did not permit) the making of a modified version of Inform, whereas the new licence largely permits this (see below).

§**3. The current licence, 2008-.** As from the first publication on the Inform website of the source code to Inform 7, in 2008:

(a) The Package is hereby placed under the Artistic License 2.0.
(b) For the avoidance of doubt, the Author additionally grants the right for all users of the Package to make unlimited use of story files produced by the Package.

To clarify (b): the structure of Inform means that it copies large pieces of the I6 library and the I6T template files almost verbatim into the I6 source code used to make a story file. Someone might then worry that any resulting story file is therefore a derivative work of the Package itself, and so inherits the Artistic License 2.0. The Author wishes to clarify that this is not the case, and that *people using Inform to make a story file can sell, distribute, modify or otherwise use it exactly as they please, and under any licence(s) they choose.* (The same issue arises with the Free Software Foundation's `bison`, and they solve it in the same way, by making an additional grant of rights on top of the licence for `bison` itself.)

§**4.** The Artistic License 2.0, used by Perl among other notable projects, is widely considered to be one offering generous permissions for the user, and was chosen because:

(a) it is recognised by the Open Source Initiative as an open source licence;
(b) it is recognised by the Free Software Foundation as a guarantee that the software is free not only in the economic way, but also in a more libertarian sense;
(c) by freely allowing modifications and derivative works, but requiring any such to be issued under different names, it allows the authors to retain the moral right of authorship: the ability to decide what is, and is not, the Inform design.

The licence is the sole legal document governing these matters, of course, but our interpretation of clause (9) is that, in particular, it is legal to distribute an aggregation of the Package along with any of the recognised user interface application(s) and also with Extensions by third parties which are subject to the Creative Commons Attribution licence (such as those published on the Inform website), and to call the result "Inform".

The verbatim text of the licence now follows. It is copyrighted by the Perl Foundation, 2006, whose work in creating it we acknowledge with thanks. (The boldface section numbers in the typeset form of this preface are just typography, and not a part of the licence: but the clause numbers, "(10)" and so forth, are. The American spelling "license" is used, since that's how the Foundation wrote it.)

§**5. Preamble.** This license establishes the terms under which a given free software Package may be copied, modified, distributed, and/or redistributed. The intent is that the Copyright Holder maintains some artistic control over the development of that Package while still keeping the Package available as open source and free software.

You are always permitted to make arrangements wholly outside of this license directly with the Copyright Holder of a given Package. If the terms of this license do not permit the full use that you propose to make of the Package, you should contact the Copyright Holder and seek a different licensing arrangement.

§**6. Definitions.** *Copyright Holder* means the individual(s) or organization(s) named in the copyright notice for the entire Package. *Contributor* means any party that has contributed code or other material to the Package, in accordance with the Copyright Holder's procedures. *You* and *your* means any person who would like to copy, distribute, or modify the Package. *Package* means the collection of files distributed by the Copyright Holder, and derivatives of that collection and/or of those files. A given Package may consist of either the Standard Version, or a Modified Version. *Distribute* means providing a copy of the Package or making it accessible to anyone else, or in the case of a company or organization, to others outside of your company or organization. *Distributor Fee* means any fee that you charge for Distributing this Package or providing support for this Package to another party. It does not mean licensing fees. *Standard Version* refers to the Package if it has not been modified, or has been modified only in ways explicitly requested by the Copyright Holder. *Modified Version* means the Package, if it has been changed, and such changes were not explicitly requested by the Copyright Holder. *Original License* means this Artistic License as Distributed with the Standard Version of the Package, in its current version or as it may be modified by The Perl Foundation in the future. *Source* form means the source code, documentation source, and configuration files for the Package. *Compiled* form means the compiled bytecode, object code, binary, or any other form resulting from mechanical transformation or translation of the Source form.

§**7. Permission for Use and Modification Without Distribution.**

(1) You are permitted to use the Standard Version and create and use Modified Versions for any purpose without restriction, provided that you do not Distribute the Modified Version.

§**8. Permissions for Redistribution of the Standard Version.**

(2) You may Distribute verbatim copies of the Source form of the Standard Version of this Package in any medium without restriction, either gratis or for a Distributor Fee, provided that you duplicate all of the original copyright notices and associated disclaimers. At your discretion, such verbatim copies may or may not include a Compiled form of the Package.

(3) You may apply any bug fixes, portability changes, and other modifications made available from the Copyright Holder. The resulting Package will still be considered the Standard Version, and as such will be subject to the Original License.

§**9. Distribution of Modified Versions of the Package as Source.**

(4) You may Distribute your Modified Version as Source (either gratis or for a Distributor Fee, and with or without a Compiled form of the Modified Version) provided that you clearly document how it differs from the Standard Version, including, but not limited to, documenting any non-standard features, executables, or modules, and provided that you do at least ONE of the following:

(a) make the Modified Version available to the Copyright Holder of the Standard Version, under the Original License, so that the Copyright Holder may include your modifications in the Standard Version.

(b) ensure that installation of your Modified Version does not prevent the user installing or running the Standard Version. In addition, the Modified Version must bear a name that is different from the name of the Standard Version.

(c) allow anyone who receives a copy of the Modified Version to make the Source form of the Modified Version available to others under

(i) the Original License or

(ii) a license that permits the licensee to freely copy, modify and redistribute the Modified Version using the same licensing terms that apply to the copy that the licensee received, and requires that the Source form of the Modified Version, and of any works derived from it, be made freely available in that license fees are prohibited but Distributor Fees are allowed.

§10. Distribution of Compiled Forms of the Standard Version or Modified Versions without the Source.

(5) You may Distribute Compiled forms of the Standard Version without the Source, provided that you include complete instructions on how to get the Source of the Standard Version. Such instructions must be valid at the time of your distribution. If these instructions, at any time while you are carrying out such distribution, become invalid, you must provide new instructions on demand or cease further distribution. If you provide valid instructions or cease distribution within thirty days after you become aware that the instructions are invalid, then you do not forfeit any of your rights under this license.

(6) You may Distribute a Modified Version in Compiled form without the Source, provided that you comply with Section 4 with respect to the Source of the Modified Version.

§11. Aggregating or Linking the Package.

(7) You may aggregate the Package (either the Standard Version or Modified Version) with other packages and Distribute the resulting aggregation provided that you do not charge a licensing fee for the Package. Distributor Fees are permitted, and licensing fees for other components in the aggregation are permitted. The terms of this license apply to the use and Distribution of the Standard or Modified Versions as included in the aggregation.

(8) You are permitted to link Modified and Standard Versions with other works, to embed the Package in a larger work of your own, or to build stand-alone binary or bytecode versions of applications that include the Package, and Distribute the result without restriction, provided the result does not expose a direct interface to the Package.

§12. Items That are Not Considered Part of a Modified Version.

(9) Works (including, but not limited to, modules and scripts) that merely extend or make use of the Package, do not, by themselves, cause the Package to be a Modified Version. In addition, such works are not considered parts of the Package itself, and are not subject to the terms of this license.

§13. General Provisions.

(10) Any use, modification, and distribution of the Standard or Modified Versions is governed by this Artistic License. By using, modifying or distributing the Package, you accept this license. Do not use, modify, or distribute the Package, if you do not accept this license.

(11) If your Modified Version has been derived from a Modified Version made by someone other than you, you are nevertheless required to ensure that your Modified Version complies with the requirements of this license.

(12) This license does not grant you the right to use any trademark, service mark, tradename, or logo of the Copyright Holder.

(13) This license includes the non-exclusive, worldwide, free-of-charge patent license to make, have made, use, offer to sell, sell, import and otherwise transfer the Package with respect to any patent claims licensable by the Copyright Holder that are necessarily infringed by the Package. If you institute patent litigation (including a cross-claim or counterclaim) against any party alleging that the Package constitutes direct or contributory patent infringement, then this Artistic License to you shall terminate on the date that such litigation is filed.

(14) Disclaimer of Warranty: THE PACKAGE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES. THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT ARE DISCLAIMED TO THE EXTENT PERMITTED BY YOUR LOCAL LAW. UNLESS REQUIRED BY LAW, NO COPYRIGHT HOLDER OR CONTRIBUTOR WILL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING IN ANY WAY OUT OF THE USE OF THE PACKAGE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

§**14.** That final burst of shouting concludes the verbatim text of the Artistic License 2.0, and we can now return to your scheduled program.

BNF Grammar P/bnfg

Purpose

A collated BNF grammar for the syntax of the language in which Inform source text is written.

---

§**1.** The following is derived by `inweb` from documentation, not code, and is intended primarily to provide a contents page showing how to find which sections handle which grammatical constructs. While this is close to being a formal syntactic description of I7, semantic considerations are in practice more important. For instance, it is syntactically valid for an action pattern to be used as a value, but one would not normally expect, say, "taking something portable" to be something which can be evaluated: but in sentences like "Taking something portable is permitted behaviour", the same words are construed as a referential noun phrase. Similarly, the description below does not explain how to resolve ambiguities (though in most cases productions are checked in the order shown). Finally, note that a number of the syntaxes below are undocumented in the public manual for Inform. This is intentional: they are provided only for the bootstrapping process which the Standard Rules goes through, and are not for public use. They may change without notice.

The notation used is mostly standard: see the Wikipedia page on EBNF for details. In particular "::=" can be read as "matches", and each angle-bracketed construction is followed by one or more possibilities. (A vertical stroke means "or".) Square brackets mean that material is optional. Braces mean "0 or more": thus "**on {and on}**" matches the texts "on", "on and on", "on and on and on", ...

Bold-face words in lower case are literal words, and bold-face parentheses are literal parentheses. Bold-face small caps are used for lexically defined patterns: the meaning of these is mostly obvious, or can be found elsewhere. (For instance, an I6 identifier is a string of 1 to 31 characters consisting of digits, upper or lower case unaccented letters or underscore signs, such that the first character must not be a digit.) Lower-case roman type denotes a name: thus "world-object" means the name of a world-object, which might be a thing, room, direction or region. Lower-case italic type denotes an inflected usage: thus *verb-usage* might match "has not been" or "carried" or "was supported by". Finally, the subscripts after angle-bracketed constructions refer to the NI source code: for instance, "5/lit" means the section "Literal Productions" in Chapter 5.

§**2. Literal values.** That is, those which can be determined as constants by lexical means alone, in the same way that a run of digits not starting perhaps with a minus sign but not a 0 can be recognised as a literal number.

---

§**3. Physical descriptions.** These are general descriptions of categories of physical things or places. A detail currently left vague, to avoid doubling the size of the grammar, is that in some contexts a physical description is allowed to consist of adjectives without a noun (as in the grammar written below, which is used most often), whereas in a few other cases a noun is required. (I construe this as a semantic rather than syntactic requirement, and so give the more general form of the grammar.)

---

§**4. Type expressions.** This is a difficult concept to coin a name for. These are not constant in value, but constant in identification. In particular, we can use these for deciding specificity. For instance, the literal object "Spanish saddle" is a type expression because it can be determined at compile time to be more specific than, say, the kind "equestrian kit": but the global variable "latest horse show prize" is not a type expression, since we can't judge its specificity without knowing the current value, and that's no use because it is undetermined at compile time.

§5. Values and conditions.

§**6. List punctuation.** Note the optional serial comma, always permitted.

---

§**7. Action patterns.** The most important and unconventional form of condition in I7: a description matching a range of possible actions, which is deemed true if the current action fits.

---

§**8. Source file layout.** There are two kinds of source file: the main source text, and an extension.

---

§**9.** Extension files have a similar make-up, but use different titling, and end differently too: with an explicit closing sentence and then, optionally, some documentation and examples.

---

§**10. Paragraphs.** Paragraphs are composed of one or more sentences of any kind in any order, with two exceptional forms: headings and tables.

§11. Whole-paragraph sentences.

§**12. Rules and phrase definitions.**

---

§**13. Assertion sentences.** This is the general form that almost all declarative sentences in I7 take: though there are a number of miscellaneous exceptional sentences in the next section, they only turn up occasionally.

---

§**14. Exceptional declarative sentences.** These look superficially like assertions, but are in fact parsed as exceptional cases. The net effect is that I7 behaves as if it has certain reserved verbs and prepositions, though in fact for various reasons this is not how such sentences are implemented.

§**15. Imperative sentences.** A miscellany of ways to instruct I7 this way and that, using an imperative verb rather than an SVO sentence.

1 Definitions

`1/basic`: *Basic Definitions.w* To make basic definitions used throughout NI.

`1/plat`: *Platform-Specific Definitions.w* To provide any adaptations needed for NI to work on different "platforms", or operating systems: for instance, styles of writing filenames specific to Linux, Windows or Mac OS X are specified here. The only platform dependencies in the whole of NI are contained in this section.

Basic Definitions 1/basic

Purpose

To make basic definitions used throughout NI.

1/basic.¶5 File Input/Output; ¶6 Encodings and formats; ¶7-0 Output file handles

---

Definitions

¶1. We first include the standard C library, using most of the basic repertoire of routines for handling simple data. `stdarg.h`, perhaps the least-used of these, is needed because a variable-argument function (an NI-specific form of `printf`), will be defined in Chapter 2 to handle the debugging log.

```
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

¶2. As in the Inform 6 source code, the following conventional definitions are used for values whenever integer variables are used as flags. (We only use `NOT_APPLICABLE` very occasionally.)

```
define TRUE 1
define FALSE 0
define NOT_APPLICABLE 2
```

¶3. These flags are set by command-line parameters. `for_release` will be set when NI is used in a run started by clicking on the Release button in the application. `fix_rng_at_start_of_play` is not used by the application, but the `intest` program makes use of this feature to make repeated tests of the Z-machine story file produce identical sequences of random numbers: without this, we would have difficulty comparing a transcript of text produced by the story file on one compilation from another.

`story_filename_extension` is also set as a result of information passed from the application via the command line to NI. In order for NI to write good releasing instructions, it needs to know the story file format (".z5", ".z8", etc.) of the finally produced story file. But since NI compiles only to Inform 6 code, and does not run I6 itself, it has no way of telling what the application intends to do on this. So the application is required to give NI advance notice of this via a command-line option.

```
int for_release = FALSE;                    Omit sections of source text marked not for release
int existing_story_file = FALSE;            Ignore source text to blorb existing story file?
int fix_rng_at_start_of_play = FALSE;       Compile I6 code which seeds the RNG
int rng_seed_at_start_of_play = 0;          The seed value, or 0 if not seeded
int census_mode = FALSE;                    NI running only to update extension documentation
char *story_filename_extension = NULL;      What story file we will eventually have
int show_progress_indicator = TRUE;         Produce percentage of progress messages
```

¶4. Broadly speaking, what NI does can be divided into two halves: in the first half, it reads all the assertions, makes all the objects and global variables and constructs the model world; in the second half, it compiles the phrases and grammar to go with it. `model_world_constructed` records which of these halves we are currently in: `FALSE` in the first half, `TRUE` in the second.

If there were a third stage, it would be indexing, and during that period `indexing_stage` is `TRUE`. But by that time the compilation of Inform 6 code is complete.

```
int text_loaded_from_source = FALSE;                    Lexical scanning is done
int model_world_constructed = FALSE;                    World model is constructed
int indexing_stage = FALSE;                             Everything is done except indexing
```

¶5. File Input/Output. This seems a good point to note that NI follows Unix Standard I/O conventions only up to a point. Text printed to `stdout` is descriptive of the output, but is not the primary output itself (i.e., is not the I6 translated code). Nor is the natural language original read from `stdin`. But we do follow convention by writing errors to `stderr`. Standard I/O is a good idea for tools to be used in pipelines, but NI is not a good candidate for such use because of the number of auxiliary files involved in any translation – the index, the debugging log, etc. Indeed, a true picture of file input/output by NI is more like the following: note the absence of `stdin`.

Primary source text		Standard Rules
UUID File		*Other extensions
		Template code (`.i6t` files)
		Extensions dictionary (in)
	...go into NI and out come...	
Inform 6 code	`stdout` console text	Extensions dictionary (out)
Release blurb	*`stderr` console errors	Extensions documentation (HTML files)
iFiction record	*EPS map file	
Headings file	*Trace log	
Debugging log		
Problems file		
Index (HTML files)		

Here files in the left-hand column belong to an Inform 7 "project": for a project called Kensington, they would all be files inside the `Kensington.inform` folder. Files in the middle column are stored in standard places on the user's machine: for instance, on Mac OS X the trace log and EPS file are placed on the Desktop. Files in the right-hand column are the private property of NI and live either inside the Inform 7 application, or else in an area of the user's file system which is maintained by NI without the user's direct involvement. Lastly, files marked with an asterisk are not accessed on every run of NI.

¶6. Encodings and formats. NI uses several different encodings of text, which coincide on the encoding of the standard range of ASCII characters between `0x0020` and `0x007E`.

File	Usage	Format	Encoding
Primary source text	in	Text	Unicode UTF-8 with or without BOM
Standard Rules	in	Text	Unicode UTF-8 with or without BOM
Other extensions	in	Text	Unicode UTF-8 with or without BOM
UUID File	in	Text	ASCII
Template code	in	I6 source	ISO Latin-1
Extensions dictionary	in/out	Text	ISO Latin-1
Inform 6 code	out	I6 source	ISO Latin-1
`stdout` console text	out	Text	ASCII
*`stderr` console errors	out	Text	ASCII
Extensions documentation	out	HTML	Unicode UTF-8 without BOM
Release blurb	out	Blurb file	ISO Latin-1
iFiction record	out	XML	Unicode UTF-8 without BOM
Headings file	out	XML	Unicode UTF-8 without BOM
Pictures manifest	out	XML	Unicode UTF-8 without BOM
Debugging log	out	Text	ISO Latin-1
*Trace log	out	Text	ISO Latin-1
Problems file	out	HTML	Unicode UTF-8 without BOM
Index (HTML files)	out	HTML	Unicode UTF-8 without BOM
*EPS map file	out	EPS	ISO Latin-1

The primary source text and extension files may have any of the standard line delimiters, including the little-implemented Unicode line divider. I6 files may have any of the non-Unicode line delimiters. Files written by NI use `'\n'` as line delimiter, the meaning of which varies from platform to platform.

Strings inside NI encode text as ISO Latin-1. Unicode characters beyond `0x00FF` are stored using escapes in the form `"[unicode 654]"`, where the character number is decimal.

It's convenient to define constants to refer to our two text encodings.

```
define UTF8_ENC 1                    Write as UTF-8 without BOM
define ISO_ENC 2                     Write as ISO Latin-1 (i.e., no conversion needed)
```

¶7. Output file handles. By convention, the file handles to which Inform's auxiliary outputs are written – any errors, debugging information, and so on – have short names: `dl`, for example. But note that these are variables and NI does vary them: for instance, it copies Problem messages both to `stderr` and to the debugging log, and it does this by setting `problems_file` to each in turn and printing the message twice. (Well, in fact it formats the message differently depending on the destination, but this is the idea.) The definitive file handles are in longer-named variables such as `debug_log_file`.

There is no short-named file handle for the I6 code output because it is instead passed as a function argument through most of the output routines, and invariably called `of`.

The "debugging log" gives a richly verbose account of what NI is doing. This (ISO Latin-1) plain text file exists to debug NI, *not* to help designers debug their own games. Almost every data structure has a corresponding routine to pretty-print a representation of it to the debug log. This is a plain text file. The "trace log" is a much simpler version, used for extracts only and only on request.

NI also creates a report which *is* for the eyes of designers: its report of Problems, which despite the name often simply says that all is well. Many parts of NI also check the error count to see if things are going well or badly, so we'll define that here too.

The index contains several files, each of which is written in turn: the file handle `ifl` is the currently open one, and is null until we reach indexing (if we do).

The file handle `inform6_file` is NI's output, which is a program written in Inform 6. We define this here so that error handling can close it if the program exits prematurely.

```
STREAM debug_log_file_struct;                          The actual debugging log file
STREAM inform6_file_struct;                            The actual I6 code file
STREAM trace_log_struct;                               The actual trace log file (if created)
STREAM problems_file_struct;                           The actual report of Problems file
STREAM index_file_struct;                              The current index file being written
STREAM telemetry_file_struct;                          The actual telemetry file (if created)

STREAM *debug_log_file = &debug_log_file_struct;       The actual debugging log file
STREAM *inform6_file = &inform6_file_struct;           The actual I6 code file
STREAM *trace_log = &trace_log_struct;                 The actual trace log file (if created)
STREAM *problems_file = &problems_file_struct;         The actual report of Problems file
STREAM *telemetry_file = &telemetry_file_struct;       The actual debugging log file

STREAM *dl = NULL;                                     Current destination of debugging text
STREAM *trl = NULL;                                    Current destination of trace log text
STREAM *ifl = NULL;                                    Current destination of index text
STREAM *probl = NULL;                                  Current destination of problem message text
STREAM *telmy = NULL;                                  Current destination of telemetry text

int problem_count = 0;                                 The number of Problem messages so far produced
int it_is_not_worth_adding = FALSE;                    To suppress the "It may be worth adding..."
int trace_sentences = FALSE;                           Currently debugging text to trace rather than log
int logging_to_I6_text = FALSE;                        Used for internal test cases
```

¶8. The top-level control of NI, telling it what sequence of tasks to perform, is done by a metalanguage interpreter which works through a series of template files with the `.i6t` filename extension (see Chapter 14). Setting the following flag causes this interpreter to shut down gracefully and exit NI.

```
int abort_I6T_interpreter = FALSE;
```

---

§**1.** This section has no body, and exists only to make definitions, so there is no code here.

Platform-Specific Definitions 1/plat

Purpose

To provide any adaptations needed for NI to work on different "platforms", or operating systems: for instance, styles of writing filenames specific to Linux, Windows or Mac OS X are specified here. The only platform dependencies in the whole of NI are contained in this section.

Definitions

¶1. We have, at present, three sets of platform definitions: to compile NI, you should either modify the text below so that the default platform is yours, or else predefine one of `PLATFORM_WINDOWS`, `PLATFORM_UNIX` or `PLATFORM_MACOSX`. (We allow for that predefinition so that `autoconf`-like tools can be used to build NI on Unix platforms.) Similarly, predefinition can be used to set the `CPU_WORDSIZE_MULTIPLIER`: this is CPU's word size as a number of bits divided by 32, so in other words, define this as 2 to compile a form of NI which works on 64-bit processors. (NI does not run on 16-bit processors.)

```
#ifndef PLATFORM_MACOSX
#ifndef PLATFORM_WINDOWS
#ifndef PLATFORM_UNIX
#define PLATFORM_MACOSX                          the original home for NI, and still the default
#endif
#endif
#endif

#ifndef CPU_WORDSIZE_MULTIPLIER
#define CPU_WORDSIZE_MULTIPLIER 1                the CPU word size is by default 32 bits
#endif
```

---

§**1.** We now come to the blocks of definitions chosen by the above. The principal differences lie in (i) where NI expects to find user-installed extensions, etc.; (ii) where to write the handful of files output during compilation but not stored inside a project; (iii) how to tweak HTML to look better on this platform; and (iv) how the filing system works and how to scan directories inside it.

Note that we use the C preprocessor's `#define` below, and also make use of `#ifdef`, rather than the higher-level `inweb` definition function `@d`: this is because `inweb` does not allow conditional definitions, and looks down on them as hacky expediencies.

§**2. Mac OS X.** These settings are suitable for using NI with Andrew Hunter's Inform 7 application for OS X. NI reads and writes extension-related material, etc., to an `Inform` folder inside `Library` in the user's home folder. (As we shall see, NI creates this if need be.) The trace and EPS files are written to the desktop.

```
#ifdef PLATFORM_MACOSX
#define FOLDER_SEPARATOR '/'
#define INFORM_FOLDER_RELATIVE_TO_HOME "Library/"
#define HOME_LIBRARY "/Library/Inform/Extensions"
#define HOME_DOCS "/Library/Inform/Documentation"
#define HOME_TEMPLATES "/Library/Inform/Templates"
#define TRACEFILE "/Library/Inform/Inform_Trace.txt"
#define TELEMETRYFILE "/Library/Inform/Telemetry"
#define OPTIONSFILE "/Library/Inform/Options.txt"
#define ICON_EXT "tif"
#define DEFAULT_HTML_FONT \
    "face=\"lucida grande,geneva,arial,tahoma,verdana,helvetica,helv\" size=2"
#define POSIX_DIRECTORY_HANDLING
#define EPS_FILE_DESTINATION "/Desktop/Inform Map.eps"
void set_platform_specific_eps_filename(char *to) {
    char *home = (char *) (getenv("HOME"));
    if (home == NULL) {
        strcpy(to, "/Inform Map.eps");
    } else {
        strcpy(to, home);
        strcat(to, EPS_FILE_DESTINATION);
    }
}
#define EXTENSIONS_MODEL_HTML "ExtensionsModel.html"
#define EXTENSION_FILE_MODEL_HTML "ExtensionFileModel.html"
#define JAVASCRIPT_MODEL 1
typedef long int pointer_sized_int;
#endif
```

The function set_platform_specific_eps_filename is called from 2/files.

§**3. Microsoft Windows.** The Windows Inform 7 application, by David Kinder, prefers to control the appearance of HTML in its panels using its own CSS arrangements, and so NI does not assert the font to be used in this case. NI reads and writes extension-related material, etc., to an `Inform` folder in the user's My Documents folder. (As we shall see, NI creates this if need be.) The trace and EPS files are written to the desktop.

```
#ifdef PLATFORM_WINDOWS
#define LOCALE_IS_ISO
#define FOLDER_SEPARATOR '\\'
#define INFORM_FOLDER_RELATIVE_TO_HOME ""
#define HOME_LIBRARY "\\Inform\\Extensions"
#define HOME_DOCS "\\Inform\\Documentation"
#define HOME_TEMPLATES "\\Inform\\Templates"
#define TRACEFILE "\\Inform\\Inform_Trace.txt"
#define TELEMETRYFILE "\\Inform\\Telemetry"
#define OPTIONSFILE "\\Inform\\Options.txt"
#define DEFAULT_HTML_FONT "size=2"
#define WINDOWS_DIRECTORY_HANDLING
#define EPS_FILE_DESTINATION "\\Inform Map.eps"
```

```
void set_platform_specific_eps_filename(char *to) {
    char *home = (char *) (getenv("HOME"));
    char *desktop = (char *) (getenv("DESKTOP"));
    if (desktop == NULL) desktop = home;
    if (desktop == NULL) desktop = "";
    strcpy(to, desktop);
    strcat(to, EPS_FILE_DESTINATION);
}
#define EXTENSIONS_MODEL_HTML "WinExtensionsModel.html"
#define EXTENSION_FILE_MODEL_HTML "WinExtensionFileModel.html"
#define JAVASCRIPT_MODEL 2
typedef long int pointer_sized_int;
#endif
```

The function set_platform_specific_eps_filename is called from 2/files.

§4. **A Windows-safe form of `isdigit`.** Annoyingly, the C specification allows the implementation to have `char` either signed or unsigned. On Windows it's generally signed. Now, consider what happens with a character value of acute-e. This has an `unsigned char` value of 233. When stored in a `char` on Windows, this becomes a value of `-23`. When this is passed to `isdigit()`, we need to consider the prototype for `isdigit()`:

```
int isdigit(int);
```

So, when casting to int we get `-23`, not `233`. Unfortunately the return value from `isdigit()` is only defined by the C specification for values in the range 0 to 255 (and also EOF), so the return value for `-23` is undefined. Unfortunately with Windows GCC, `isdigit(-23)` returns a non-zero value.

```
#ifdef PLATFORM_WINDOWS
int isdigit(int c) {
    return ((c >= '0') && (c <= '9')) ? 1 : 0;
}
#endif
```

§5. **Generic Unix.** These settings are used both for the Linux versions of NI (both command-line, by Adam Thornton, and for Ubuntu, Fedora, Debian and so forth, by Philip Chimento) and also for Solaris variants: they can probably be used for any Unix-based system.

```
#ifdef PLATFORM_UNIX
#include <strings.h>
#include <math.h>
#define FOLDER_SEPARATOR '/'
#define INFORM_FOLDER_RELATIVE_TO_HOME ""
#define HOME_LIBRARY "/Inform/Extensions"
#define HOME_DOCS "/Inform/Documentation"
#define HOME_TEMPLATES "/Inform/Templates"
#define TRACEFILE "/Inform/Inform_Trace.txt"
#define TELEMETRYFILE "/Inform/Telemetry"
#define OPTIONSFILE "/Inform/Options.txt"
#define DEFAULT_HTML_FONT \
    "face=\"lucida grande,geneva,arial,tahoma,verdana,helvetica,helv\" size=2"
#define POSIX_DIRECTORY_HANDLING
#define EPS_FILE_DESTINATION "/Inform Map.eps"
void set_platform_specific_eps_filename(char *to) {
    char *home = (char *) (getenv("HOME"));
    char *desktop = (char *) (getenv("DESKTOP"));
```

```
    if (desktop == NULL) desktop = home;
    strcpy(to, desktop);
    strcat(to, EPS_FILE_DESTINATION);
}
#define EXTENSIONS_MODEL_HTML "ExtensionsModel.html"
#define EXTENSION_FILE_MODEL_HTML "ExtensionFileModel.html"
#define JAVASCRIPT_MODEL 0
typedef long int pointer_sized_int;
#endif
```

The function set_platform_specific_eps_filename is called from 2/files.

§**6.** The "icon extension" is used for only two icons, the orange jump to source icon and the grey magnifying glass. These are rendered inside the Inform 7 application in both TIFF and PNG formats: TIFF looks better on OS X, but other platform browsers mostly cannot render TIFFs.

```
#ifndef ICON_EXT
#define ICON_EXT "png"                    File extension of the icons in the HTML index
#endif
```

§**7. Directory handling.** NI would ideally handle all directory activities – scanning directories to see what files are there, creating new directories, etc. – using the standard POSIX environment, which is supposed to make such things platform-independent. In practice Windows provides POSIX-like facilities but with sufficient differences that we have instead written the necessary routines twice: and in the rest of NI we treat the pointer to a directory structure as a `void *` to avoid making any assumptions about what the structure behind it actually is. First: "pure" POSIX.

```
#ifdef POSIX_DIRECTORY_HANDLING
#include <sys/stat.h>
#include <sys/types.h>
#include <errno.h>
#include <dirent.h>

int platform_specific_mkdir(char *path_to_folder) {
    int rv;
    errno = 0;
    rv = mkdir(path_to_folder, S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
    if (rv == 0) return TRUE;
    if (errno == EEXIST) return TRUE;
    STREAM_WRITE(STDERR, "Failed to create folder <%s>\n", path_to_folder);
    return FALSE;
}

void *platform_specific_opendir(char *path_to_folder) {
    DIR *dirp = opendir(path_to_folder);
    return (void *) dirp;
}

int platform_specific_readdir(void *folder, char *path_to_folder,
    char *leafname) {
    char path_to[2*MAX_FILENAME_LENGTH+2];
    struct stat file_status;
    int rv;
    DIR *dirp = (DIR *) folder;
    struct dirent *dp;
```

```
    if ((dp = readdir(dirp)) == NULL) return FALSE;
    sprintf(path_to, "%s%c%s", path_to_folder, FOLDER_SEPARATOR, dp->d_name);
    rv = stat(path_to, &file_status);
    if (rv != 0) {
        STREAM_WRITE(STDERR, "Stat failed on %s\n", path_to);
        return FALSE;
    }
    if (S_ISDIR(file_status.st_mode)) sprintf(leafname, "%s/", dp->d_name);
    else strcpy(leafname, dp->d_name);
    return TRUE;
}
void platform_specific_closedir(void *folder) {
    DIR *dirp = (DIR *) folder;
    closedir(dirp);
}
#endif
```

The function platform_specific_mkdir is called from 2/files and 10/bib.
The function platform_specific_opendir is called from 2/files.
The function platform_specific_readdir is called from 2/files.
The function platform_specific_closedir is called from 2/files.

§**8.** Now for the Windows versions of these same functions, though they are not very different.

```
#ifdef WINDOWS_DIRECTORY_HANDLING
#include <sys/stat.h>
#include <dirent.h>
#include <errno.h>
#include <io.h>
int platform_specific_mkdir(char *path_to_folder) {
    int rv;
    errno = 0;
    rv = _mkdir(path_to_folder);
    if (rv == 0) return TRUE;
    if (errno == EEXIST) return TRUE;
    STREAM_WRITE(STDERR, "Failed to create folder <%s>\n", path_to_folder);
    return FALSE;
}
void *platform_specific_opendir(char *path_to_folder) {
    DIR *dirp = opendir(path_to_folder);
    return (void *) dirp;
}
int platform_specific_readdir(void *folder, char *path_to_folder,
    char *leafname) {
    char path_to[2*MAX_FILENAME_LENGTH+2];
    struct _stat file_status;
    int rv;
    DIR *dirp = (DIR *) folder;
    struct dirent *dp;
    if ((dp = readdir(dirp)) == NULL) return FALSE;
    sprintf(path_to, "%s%c%s", path_to_folder, FOLDER_SEPARATOR, dp->d_name);
    rv = _stat(path_to, &file_status);
```

```
    if (rv != 0) {
        STREAM_WRITE(STDERR, "Stat failed on %s\n", path_to);
        return FALSE;
    }
    if (S_ISDIR(file_status.st_mode)) sprintf(leafname, "%s/", dp->d_name);
    else strcpy(leafname, dp->d_name);
    return TRUE;
}

void platform_specific_closedir(void *folder) {
    DIR *dirp = (DIR *) folder;
    closedir(dirp);
}

#endif
```

The function platform_specific_mkdir is called from 2/files and 10/bib.

The function platform_specific_opendir is called from 2/files.

The function platform_specific_readdir is called from 2/files.

The function platform_specific_closedir is called from 2/files.

§**9. Locale issues.** The following is intended to handle possible differences of text encoding in filenames, which (for Unix-based systems) depend on the current locale. The default is UTF-8 since Mac OS X, and probably most other modern Unixes, use this.

```
#ifndef LOCALE_IS_ISO
#ifndef LOCALE_IS_UTF8
#define LOCALE_IS_UTF8 1
#endif
#endif
```

§**10.** The function `iso_fopen` should behave exactly like `fopen`, but encoding the filename as ISO Latin-1 text. Thus, it can actually be just `fopen` on platforms whose locale uses ISO as its filename encoding; but on platforms with UTF-8 encoding of filenames, transcoding is needed.

The function `iso_fopen_caseless` is used when Inform wants to cope well if the casing (upper vs. lower case, that is) in the filename might not be right. On a caseless-read filing system such as that of Mac OS X (default) or Windows, there is no issue. But for a cased filing system, we may need to provide something.

```
FILE *iso_fopen(char *pathname, char *usage) {
    char transcoded_pathname[2*MAX_FILENAME_LENGTH];
    transcode_ISO_string_to_locale(pathname, transcoded_pathname);
    return fopen(transcoded_pathname, usage);
}

FILE *iso_fopen_caseless(char *pathname, char *usage) {
    char transcoded_pathname[2*MAX_FILENAME_LENGTH];
    transcode_ISO_string_to_locale(pathname, transcoded_pathname);
    return ci_fopen(transcoded_pathname, usage);
}
```

The function iso_fopen is called from 2/strm, 2/dl, 3/read, 4/ext, 4/excen, 4/edict, 4/edoc, 10/ifid, 10/bib, 10/fig, 10/sfx and 14/i6t.

The function iso_fopen_caseless is called from 3/read and 4/excen.

§**11.** And the locale-dependent part is:

```
#ifdef LOCALE_IS_ISO
void transcode_ISO_string_to_locale(char *from, char *to) {
    strcpy(to, from);
}
#endif
#ifdef LOCALE_IS_UTF8
void transcode_ISO_string_to_locale(char *from, char *to) {
    transcode_ISO_string_to_UTF8(from, to);
}
#endif
```

The function transcode_ISO_string_to_locale is called from 2/files and 4/excen.

§**12.** Here we read a line from a file whose encoding is that of filenames in the locale. If the locale uses ISO, it's sufficient to call NI's standard `truncated_iso_fgets` instead: but for UTF-8 locales, we need the following.

These matters are beyond evil. In testing, I discovered, for instance, that under Mac OS X the leafname in a `dirent` would encode c-cedilla not as `E7` ("Latin Small Letter C With Cedilla") but as the sequence `63 0327` (lower case c, followed by "Combining Cedilla"); while this pointless digraph fed back into `fopen` would fail to match the original again, and this caused extensions whose names contained a c-cedilla to fail to be documented properly.

```
#ifdef LOCALE_IS_ISO
int truncated_locale_fgets(FILE *F, char *buffer, int limit) {
    return truncated_iso_fgets(F, buffer, limit);
}
#endif
#ifdef LOCALE_IS_UTF8
int truncated_locale_fgets(FILE *F, char *buffer, int limit) {
    int len=0, c;
    if (buffer == NULL) return -1;
    if (feof(F)) return -1;
    while ((c = utf8_fgetc(F, FALSE)) != EOF) {
        if (len >= limit) break;
        if ((c == '\x0a') || (c == '\x0d')) break;
        if ((c >= 0x0300) && (c <= 0x036F) && (len > 0))
            c = combining_accent(c, buffer[--len]);
        buffer[len++] = c;
    }
    buffer[len++] = 0;
    return len;
}
#endif
```

The function truncated_locale_fgets is called from 4/excen.

§**13.** That concludes the platform-specific material, and from here on NI contains no conditional compilations.

2 Memory, Files, Problems and Logs

`2/mem`: *Memory.w* To allocate memory suitable for the dynamic creation of objects of different sizes, placing some larger objects automatically into doubly linked lists and assigning each a unique allocation ID number.

`2/strm`: *Streams.w* Support for writing structured textual output, perhaps to the screen, to a file, or to a flexible-sized area of memory.

`2/files`: *Filenames.w* Given the different environments in which we might be running, and the large range of files we may need from within a project folder, it's useful to have a section of code simply to deduce filenames.

`2/cifn`: *Case-Insensitive Filenames.w* On some of the Unix-derived file systems on which Inform runs, filenames are case-sensitive, so that FISH and fish might be different files. This makes extension files, installed by the user, prone to being missed. The code in this section provides a routine to carry out file opening as if filenames are case-insensitive, and is used only for extensions.

`2/image`: *Image Dimensions.w* These utility routines, contributed by Toby Nelson, look at the headers of JPEG and PNG files to find the pixel dimensions of any images supplied by the user for cover art and figures.

`2/sounds`: *Sound Durations.w* These utility routines, again by Toby Nelson, look at the headers of AIFF and OGG files to find the durations, and verify that they are what they purport to be.

`2/isn`: *Make Inform 6 Names.w* To construct valid Inform 6 identifiers, dictionary word constants and double-quoted string constants as needed.

`2/lns`: *Inform 6 Labels.w* I7 is has no Dijkstra-like conscience about compiling code which is full of I6 `jump` statements, and these require labels to jump to. This section provides those labels, and other related unique-ID-number counters.

`2/html`: *HTML Files.w* To provide utilities for writing HTML files such as the problems report, the extension documentation, the index files and so forth.

`2/java`: *Javascript Pastes.w* To write valid HTML for a paste icon which, when clicked, calls a Javascript function which will paste Inform source text into the Source panel of the application.

`2/hdoc`: *HTML Documentation.w* To translate a passage of source text into HTML-format documentation, for use in the automatically generated documentation pages on each installed extension.

`2/index`: *Index File Services.w* To provide routines to help build the various HTML index files, none of which are actually created in this section.

`2/lexi`: *Lexicon Index.w* To construct the Lexicon portion of the Phrasebook page of the Index, which gives brief definitions and references for nouns, adjectives and verbs used in source text for the current project.

`2/prog`: *Progress Percentages.w* This tiny section, the Lichtenstein of Inform, prints percentage of completion estimates onto `stderr` so that the host application can intercept them and update its graphical progress bar.

`2/prob0`: *Problems, Level 0.w* To handle fatal errors.

`2/prob1`: *Problems, Level 1.w* To render problem messages either as plain text or HTML, and write them out to files.

`2/prob2`: *Problems, Level 2.w* To assemble and format problem messages within the problem buffer.

`2/prob3`: *Problems, Level 3.w* Here we provide some convenient semi-standardised problem messages, which also serve as examples of how to use the Level 2 problem message routines.

`2/dl`: *Debugging Log.w* To write to the debugging log, a plain text file which traces what NI is doing, in order to assist those lost souls debugging it.

Memory 2/mem

Purpose

To allocate memory suitable for the dynamic creation of objects of different sizes, placing some larger objects automatically into doubly linked lists and assigning each a unique allocation ID number.

Template interpreter commands

```
2   {-callv:start_memory}
10   {-callv:free_memory}
34   {-callv:debug_memory_statistics}
```

Definitions

¶1. Statistics about Inform's memory usage change a little with each build, but in February 2009 it needed around 23 MB of memory to compile the medium-sized finished work "Bronze", which has a slightly over 40,000-word source text. Of that memory, about one-third was used by a small number of large buffers allocated as needed – what Inform calls "simple memory allocations". (Almost all of that space was needed by the lexer, to store the source text itself and its analysis; the other simple memory allocations were tiny by comparison.) The remaining two-thirds of memory consisted of some 357,285 objects of 112 different types and sizes. Like the chemical elements, these have very different natural abundances – the three most abundant being `specification`, `phrase` and `invocation`, each accounting for about 9% of Inform's memory, while the rarest 56 types accounted for less than 0.1% of memory each.

Our memory system is convenient and also, with luck, reduces errors due to wrongly implementing standard things like object creation and forming linked lists: but there is a price to pay. Safety measures such as the firebreaks of 0s between objects, together with the granularity of block sizes allocated, make an overhead of about 11%, that is, the total number of bytes allocated is 11% higher because of them. Similarly, profiling suggests that the processor spends about 5% of its time in the routines below. I think this is an acceptable bargain. (It's hard to know what is considered good practice, because nowadays memory allocation is often automatic so that programmers wrongly think of it as instantaneous, but for comparison, Knuth notes at section 114 of *TEX: the Program* that TEX spends 9% of its time allocating memory, and 6% deallocating it.)

¶2. The objects we allocate in memory come in two forms: managed and unmanaged. Managed structures, generally the larger ones with longer-term importance, are automatically placed into a doubly-linked list for each type, and are given unique "allocation ID numbers" (within that type) counting upwards from 0. Macros are provided for efficient looping forwards or backwards through the objects in existence.

Unmanaged structures, generally smaller ones or with shorter-term importance, are simply provided on request. They can't be looped over and have no IDs.

As is inevitable, managed structures take up slightly more memory, and take slightly more time to create. Every structure needing to be managed will have to have extra elements holding the necessary links and ID number. We define these elements with a macro (concealing its meaning from all other sections):

```
define MEMORY_MANAGEMENT
    int allocation_id;                    Numbered from 0 upwards in creation order
    void *next_structure;                 Next object in double-linked list
    void *prev_structure;                 Previous object in double-linked list
```

¶3. As an immediate example, we will hold details of our "simple memory allocations" within `simple_memory_claim` objects. We do this in order to make sure that anything we request from the operating system using `malloc` and `calloc` is eventually given back with `free`. The book-keeping for that requires the following tiny structure:

```
typedef struct simple_memory_claim {
    void *memory_claimed;                    position in memory of the array allocated
    int extent_of_claim;                     total number of bytes making it up
    MEMORY_MANAGEMENT
} simple_memory_claim;
```

The structure simple_memory_claim is private to this section.

¶4. Now to enumerate the "memory types" of objects, each corresponding to a different C structure defined by a `typedef` somewhere in Inform's source code. There is no significance to the order in which structures are registered with the memory system, but `NO_MEMORY_TYPES` must be 1 more than the highest MT number, so *do not add to this list without incrementing it*. There can in principle be up to 1000 memory types.

If a structure – say, `whatsit` – is to be managed, we should define the constant `whatsit_MT`, and if not we should define `whatsit_array_MT`.

```
define parse_node_MT 0
define vocabulary_entry_array_MT 1
define excerpt_meaning_MT 2
define heading_MT 3
define phrase_MT 4
define action_name_MT 5
define action_pattern_array_MT 6
define named_action_pattern_MT 7
define inference_array_MT 8
define source_file_MT 9
define grammar_verb_MT 10
define grammar_line_MT 11
define test_scenario_MT 12
define property_name_MT 13
define property_permission_MT 14
define extension_file_MT 15
define rulebook_MT 16
define booked_rule_MT 17
define phrase_option_array_MT 18
define quantity_MT 19
define table_MT 20
define table_column_MT 21
define world_object_MT 22
define specification_array_MT 23
define literal_text_MT 24
define text_routine_MT 25
define invocation_array_MT 26
define action_name_list_array_MT 27
define map_component_MT 28
define implication_MT 29
define activity_MT 30
define activity_list_array_MT 31
define use_option_MT 32
define i6_memory_setting_MT 33
```

```
define documentation_ref_MT 34
define adjectival_phrase_MT 35
define definition_MT 36
define lexicon_entry_MT 37
define plural_dictionary_entry_MT 38
define meaning_list_MT 39
define verb_usage_MT 40
define preposition_usage_MT 41
define adjective_list_entry_array_MT 42
define binary_predicate_MT 43
define pcalc_prop_array_MT 44
define pcalc_func_array_MT 45
define pcalc_prop_deferral_MT 46
define scene_MT 47
define scene_connector_array_MT 48
define literal_pattern_MT 49
define generalisation_MT 50
define command_index_entry_MT 51
define data_type_comparison_schema_array_MT 52
define auxiliary_file_MT 53
define extension_census_datum_MT 54
define extension_dictionary_entry_MT 55
define known_extension_clash_MT 56
define i6_schema_array_MT 57
define list_together_routine_MT 58
define runtime_ambiguity_MT 59
define past_tense_condition_record_MT 60
define past_tense_action_record_MT 61
define blorb_figure_MT 62
define named_rulebook_outcome_MT 63
define loop_over_scope_MT 64
define noun_filter_token_MT 65
define label_namespace_MT 66
define data_type_id_rules_MT 67
define dimensional_rule_array_MT 68
define determiner_MT 69
define type_id_rules_array_MT 70
define parse_node_annotation_array_MT 71
define stacked_variable_array_MT 72
define stacked_variable_list_array_MT 73
define stacked_variable_owner_array_MT 74
define stacked_variable_owner_list_array_MT 75
define ap_optional_clause_array_MT 76
define blorb_sound_MT 77
define external_file_MT 78
define pointer_allocation_MT 79
define data_type_casting_rule_array_MT 80
define ph_stack_frame_box_MT 81
define i6_inclusion_matter_MT 82
define kind_of_value_array_MT 83
define stored_block_value_MT 84
define literal_list_MT 85
define extension_identifier_database_entry_array_MT 86
```

```
define I6T_intervention_MT 87
define control_structure_phrase_MT 88
define quantifier_MT 89
define adjective_meaning_MT 90
define measurement_definition_MT 91
define reserved_command_verb_MT 92
define internal_test_case_MT 93
define type_template_definition_MT 94
define type_macro_definition_MT 95
define type_template_obligation_MT 96
define unit_sequence_array_MT 97
define literal_pattern_name_MT 98
define equation_MT 99
define equation_symbol_MT 100
define equation_node_MT 101
define invocation_list_array_MT 102
define invocation_list_entry_array_MT 103
define description_docket_array_MT 104
define placement_affecting_array_MT 105
define activity_crossref_array_MT 106
define VM_usage_note_MT 107
define time_period_array_MT 108
define invocation_token_list_array_MT 109
define invocation_options_array_MT 110
define simple_memory_claim_MT 111
define inv_token_problem_token_MT 112
define single_integer_array_MT 113
define NO_MEMORY_TYPES 114                    must be 1 more than the highest _MT constant above
```

---

§**1. Allocation status.** For each type of object to be allocated, a single structure of the following design is maintained. Managed types have `no_allocated_together` set to 1, and the doubly linked list is of the objects themselves. Unmanaged types are allocated in small arrays, typically of 100 objects at a time: in that case `no_allocated_together` is set to the number of objects in each completed array (so, typically 100) and the doubly linked list is of the arrays. (Inform handles an unmanaged structure `whatsit` by handling a managed structure of its own devising, `whatsit_array`, which consists of arrays of `whatsit`.)

```
typedef struct allocation_status_structure {
    actually needed for allocation purposes:
    int objects_allocated;                    total number of objects (or arrays) ever allocated
    void *first_in_memory;                    head of doubly linked list
    void *last_in_memory;                     tail of doubly linked list
    used only to provide statistics for the debugging log:
    char *name_of_type;                       e.g., "lexicon_entry_MT"
    int bytes_allocated;                      total allocation for this type of object, not counting overhead
    int objects_count;                        total number currently in existence (i.e., undeleted)
    int no_allocated_together;                number of objects in each array of this type of object
} allocation_status_structure;
```

The structure allocation_status_structure is private to this section.

§**2.** The memory allocator itself needs some memory, but only a fixed-size and fairly small array of the structures defined above. The allocator can safely begin as soon as this is initialised.

```
allocation_status_structure alloc_status[NO_MEMORY_TYPES];

void start_memory(void) {
    int i;
    for (i=0; i<NO_MEMORY_TYPES; i++) {
        alloc_status[i].first_in_memory = NULL;
        alloc_status[i].last_in_memory = NULL;
        alloc_status[i].objects_allocated = 0;
        alloc_status[i].objects_count = 0;
        alloc_status[i].bytes_allocated = 0;
        alloc_status[i].no_allocated_together = 1;
        alloc_status[i].name_of_type = "unused";
    }
}
```

The function start_memory is invoked by a command in a `.i6t` template file.

§**3. Architecture.** The memory manager is built in three levels, with its interface to the rest of Inform being entirely at level 3 (except that when Inform shuts down it calls a level 1 routine to free everything). Each level uses the one below it.

(3) Managing linked lists of large objects, within which objects can be created at any point, and from which objects can be deleted; and providing a way to create new small objects of any given type.
(2) Allocating some thousands of memory frames, each holding one large object or an array of small objects.
(1) Allocating and freeing a few dozen large blocks of contiguous memory.

§**4. Level 1: memory blocks.** Memory is allocated in blocks, within which objects are allocated as needed. The "safety margin" is the number of spare bytes left blank at the end of each object: this is done because we want to be paranoid about compilers on different architectures aligning structures to different boundaries (multiples of 4, 8, 16, etc.). Each block also ends with a firebreak of zeroes, which ought never to be touched: we want to minimise the chance of a mistake causing a memory exception which crashes the compiler, because if that happens it will be difficult to recover the circumstances from the debugging log.

```
define SAFETY_MARGIN 32
define BLANK_END_SIZE 128
```

§**5.** At present `MEMORY_GRANULARITY` is 200K on 32-bit CPUs, or 400K on 64-bit ones. This is the quantity of memory allocated by each individual `malloc` call.

After `MAX_BLOCKS_ALLOWED` blocks, we throw in the towel: Inform must have fallen into an endless loop which creates endless new objects somewhere. (If this ever happens, it would be a bug in Inform, I hasten to say: the point of this mechanism is to be able to recover. Without this safety measure, OS X in particular would grind slowly to a halt, never refusing a `malloc`, until the user was unable to get the GUI responsive enough to kill the Inform process.) The threshold is just under 2GB for 32-bit platforms, since more than that is probably impossible to handle in a 32-bit address space in any event. The actual record memory usage ever recorded for a genuine Inform project is 250MB, but that was before reworking of Inform's data structures reduced memory consumption by a factor of three. So the threshold is high enough that it won't be reached. In February 2009, "Bronze" used 81 blocks.

```
define MAX_BLOCKS_ALLOWED 10000
define MEMORY_GRANULARITY 200*1024*CPU_WORDSIZE_MULTIPLIER        which must be divisible by 1024

int no_blocks_allocated = 0;
int no_frames_allocated = 0;                                      a much larger number, used only for the debugging log
```

§**6.** Memory blocks are stored in a linked list, and we keep track of the size of the current block: that is, the block at the tail of the list. Each memory block consists of a header structure, followed by `SAFETY_MARGIN` null bytes, followed by actual data.

```
typedef struct memblock_header {
    int block_number;
    struct memblock_header *next;
    char *the_memory;
} memblock_header;
memblock_header *first_memblock_header = NULL;                    head of list of memory blocks
memblock_header *current_memblock_header = NULL;                  tail of list of memory blocks
int used_in_current_memblock = 0;                  number of bytes so far used in the tail memory block
```

The structure memblock_header is private to this section.

§**7.** The actual allocation and deallocation is performed by the following pair of routines.

```
void allocate_another_block(void) {
    unsigned char *cp;
    memblock_header *mh;
    ⟨Allocate and zero out a block of memory, making cp point to it 8⟩;
    mh = (memblock_header *) cp;
    used_in_current_memblock = sizeof(memblock_header) + SAFETY_MARGIN;
    mh->the_memory = (void *) (cp + used_in_current_memblock);
    ⟨Add new block to the tail of the list of memory blocks 9⟩;
    LOGIF(MEMORY_ALLOCATION, "AOB->the_memory: %08x\n", (pointer_sized_int) mh->the_memory);
}
```

§**8.** Note that `cp` and `mh` are set to the same value: they merely have different pointer types as far as the C compiler is concerned.

⟨Allocate and zero out a block of memory, making cp point to it 8⟩ ≡

```
    int i;
    if (no_blocks_allocated++ >= MAX_BLOCKS_ALLOWED)
        internal_error(
            "the memory manager has halted Inform, which seems to be generating "
            "endless structures. Presumably it is trapped in a loop");
    check_memory_integrity();
    LOGIF(MEMORY_ALLOCATION, "Allocating memory block %d (total objects now %d)\n",
        no_blocks_allocated, no_frames_allocated);
    cp = (unsigned char *) (malloc(MEMORY_GRANULARITY));
    if (cp == NULL) fatal_error("Run out of memory: malloc failed");
    LOGIF(MEMORY_ALLOCATION, "AOB: %08x to %08x\n",
        (pointer_sized_int) cp, (pointer_sized_int) cp+MEMORY_GRANULARITY);
    for (i=0; i<MEMORY_GRANULARITY; i++) cp[i] = 0;
```

This code is used in §7.

§**9.** As can be seen, memory block numbers count upwards from 0 in order of their allocation.

⟨Add new block to the tail of the list of memory blocks 9⟩ ≡
```
    if (current_memblock_header == NULL) {
        mh->block_number = 0;
        first_memblock_header = mh;
    } else {
        mh->block_number = current_memblock_header->block_number + 1;
        current_memblock_header->next = mh;
    }
    current_memblock_header = mh;
```

This code is used in §7.

§**10.** Freeing all this memory again is just a matter of freeing each block in turn, but of course being careful to avoid following links in a just-freed block.

```
void free_memory(void) {
    I7_free_all_remaining();
    memblock_header *mh = first_memblock_header;
    while (mh != NULL) {
        memblock_header *next_mh = mh->next;
        void *p = (void *) mh;
        free(p);
        mh = next_mh;
    }
}
```

The function free_memory is invoked by a command in a `.i6t` template file.

§**11. Level 2: memory frames and integrity checking.** Within these extensive blocks of contiguous memory, we place the actual objects in between "memory frames", which are only used at present to police the integrity of memory: again, finding obscure and irritating memory-corruption bugs is more important to us than saving bytes. Each memory frame wraps either a single large object, or a single array of small objects.

In February 2009, "Bronze" used around 48,000 memory frames.

```
define INTEGRITY_NUMBER 0x12345678                    a value unlikely to be in memory just by chance

typedef struct memory_frame {
    int integrity_check;                              this should always contain the INTEGRITY_NUMBER
    struct memory_frame *next_frame;                  next frame in the list of memory frames
    int mem_type;                                     type of object stored in this frame
    int allocation_id;                                allocation ID number of object stored in this frame
} memory_frame;
```

The structure memory_frame is private to this section.

§**12.** There is a single linked list of all the memory frames, perhaps of about 10000 entries in length, beginning here. (These frames live in different memory blocks, but we don't need to worry about that.)

```
memory_frame *first_memory_frame = NULL;              earliest memory frame ever allocated
memory_frame *last_memory_frame = NULL;               most recent memory frame allocated
```

§**13.** If the integrity numbers of every frame are still intact, then it is pretty unlikely that any bug has caused memory to overwrite one frame into another. `check_memory_integrity` might on very large runs be run often, if we didn't prevent this: since the number of calls would be roughly proportional to memory usage, we would implicitly have an $O(n^2)$ running time in the amount of storage n allocated.

```
int calls_to_cmi = 0;
void check_memory_integrity(void) {
    int c = calls_to_cmi++;
    if (!((c<10) || (c == 100) || (c == 1000) || (c == 10000))) return;
    LOGIF(MEMORY_ALLOCATION, "Beginning memory integrity check: ");
    memory_frame *mf;
    for (c = 0, mf = first_memory_frame; mf; c++, mf = mf->next_frame)
        if (mf->integrity_check != INTEGRITY_NUMBER) {
            LOG("*** Memory integrity check failed. Recent frames: ***\n");
            debug_memory_frames(c-20, c-1);
            internal_error("Memory manager failed integrity check");
        }
    LOGIF(MEMORY_ALLOCATION, "*** Succeeded ***\n");
}
void debug_memory_frames(int from, int to) {
    int c;
    memory_frame *mf;
    for (c = 0, mf = first_memory_frame; (mf) && (c <= to); c++, mf = mf->next_frame)
        if (c >= from) {
            char *desc = "corrupt";
            if (mf->integrity_check == INTEGRITY_NUMBER)
                desc = alloc_status[mf->mem_type].name_of_type;
            LOG("Frame at %08x: '%s'/%d\n",
                (pointer_sized_int) mf, desc, mf->allocation_id);
        }
}
```

§**14.** We have seen how memory is allocated in large blocks, and that a linked list of memory frames will live inside those blocks; we have seen how the list is checked for integrity; but we not seen how it is built. Every memory frame is created by the following function:

```
void *allocate_mem(int mem_type, int extent) {
    unsigned char *cp;
    memory_frame *mf;
    int bytes_free_in_current_memblock, extent_without_overheads = extent;
    extent += sizeof(memory_frame);                    each allocation is preceded by a memory frame
    extent += SAFETY_MARGIN;                           each allocation is followed by SAFETY_MARGIN null bytes
    ⟨Ensure that the current memory block has room for this many bytes 15⟩;
    cp = ((unsigned char *) (current_memblock_header->the_memory)) + used_in_current_memblock;
    used_in_current_memblock += extent;
    mf = (memory_frame *) cp;                                                the new memory frame,
    cp = cp + sizeof(memory_frame);                                  following which is the actual allocated data
    mf->integrity_check = INTEGRITY_NUMBER;
    mf->allocation_id = alloc_status[mem_type].objects_allocated;
    mf->mem_type = mem_type;
```

```
    ⟨Add the new memory frame to the big linked list of all frames 16⟩;
    ⟨Update the allocation status for this type of object 17⟩;
    no_frames_allocated++;
    return (void *) cp;
}
```

§**15.** The granularity error below will be triggered the first time a particular object type is allocated: if the Inform test suite passes on a given platform, then the granularity error can never occur. So this is not a potential time-bomb just waiting for a user with a particularly long and involved source text to discover. (It has only exploded once in Inform's development: when Inform was ported to 64-bit processors in November 2007, which greatly increased `sizeof` for many structures. This is why the granularity is now defined with a factor allowing for larger word-size CPUs.)

⟨Ensure that the current memory block has room for this many bytes 15⟩ ≡

```
    if (current_memblock_header == NULL) allocate_another_block();
    bytes_free_in_current_memblock = MEMORY_GRANULARITY - (used_in_current_memblock + extent);
    if (bytes_free_in_current_memblock < BLANK_END_SIZE) {
        allocate_another_block();
        if (extent+BLANK_END_SIZE >= MEMORY_GRANULARITY)
            fatal_error("Memory manager failed because granularity too low");
    }
```

This code is used in §14.

§**16.** New memory frames are added to the tail of the list:

⟨Add the new memory frame to the big linked list of all frames 16⟩ ≡

```
    mf->next_frame = NULL;
    if (first_memory_frame == NULL) first_memory_frame = mf;
    else last_memory_frame->next_frame = mf;
    last_memory_frame = mf;
```

This code is used in §14.

§**17.** See the definition of `alloc_status` above.

⟨Update the allocation status for this type of object 17⟩ ≡

```
    if (alloc_status[mem_type].first_in_memory == NULL)
        alloc_status[mem_type].first_in_memory = (void *) cp;
    alloc_status[mem_type].last_in_memory = (void *) cp;
    alloc_status[mem_type].objects_allocated++;
    alloc_status[mem_type].bytes_allocated += extent_without_overheads;
```

This code is used in §14.

§**18. Level 3: managing linked lists of allocated objects.** We define macros which look as if they are functions, but for which one argument is the name of a type: expanding these macros provides suitable C functions to handle each possible type. These macros provide the interface through which all other sections of Inform allocate and leaf through memory.

Note that `inweb` allows multi-line macro definitions without backslashes to continue them, unlike ordinary C. Otherwise these are "standard" macros, though this was my first brush with the `##` concatenation operator: basically `CREATE(thing)` expands into `(allocate_thing())` because of the `##`. (See Kernighan and Ritchie, section 4.11.2.)

```
define CREATE(type_name) (allocate_##type_name())
define CREATE_BEFORE(existing, type_name) (allocate_##type_name##_before(existing))
define DESTROY(this, type_name) (deallocate_##type_name(this))
define FIRST_OBJECT(type_name) ((type_name *) alloc_status[type_name##_MT].first_in_memory)
define LAST_OBJECT(type_name) ((type_name *) alloc_status[type_name##_MT].last_in_memory)
define NEXT_OBJECT(this, type_name) ((type_name *) (this->next_structure))
define PREV_OBJECT(this, type_name) ((type_name *) (this->prev_structure))
define NUMBER_CREATED(type_name) (alloc_status[type_name##_MT].objects_count)
```

§**19.** The following macros are widely used (well, the first one is, anyway) for looking through the double linked list of existing objects of a given type.

```
define LOOP_OVER(var, type_name)
    for (var=FIRST_OBJECT(type_name); var != NULL; var = NEXT_OBJECT(var, type_name))
define LOOP_OVER_FROM(var, type_name, start)
    for (var=(start)?start:(FIRST_OBJECT(type_name)); var != NULL; var = NEXT_OBJECT(var, type_name))
define LOOP_BACKWARDS_OVER(var, type_name)
    for (var=LAST_OBJECT(type_name); var != NULL; var = PREV_OBJECT(var, type_name))
```

§**20. Allocator functions created by macros.** The following macros generate a family of systematically named functions. For instance, we shall shortly expand `ALLOCATE_INDIVIDUALLY(parse_node)`, which will expand to three functions: `allocate_parse_node`, `deallocate_parse_node` and `allocate_parse_node_before`.

Quaintly, `#type_name` expands into the value of `type_name` put within double-quotes.

```
define NEW_OBJECT(type_name) ((type_name *) allocate_mem(type_name##_MT, sizeof(type_name)))
define ALLOCATE_INDIVIDUALLY(type_name)
MAKE_REFERENCE_ROUTINES(type_name, type_name##_MT)
type_name *allocate_##type_name(void) {
    alloc_status[type_name##_MT].name_of_type = #type_name;
    type_name *prev_obj = LAST_OBJECT(type_name);
    type_name *new_obj = NEW_OBJECT(type_name);
    new_obj->allocation_id = alloc_status[type_name##_MT].objects_allocated-1;
    new_obj->next_structure = NULL;
    if (prev_obj != NULL)
        prev_obj->next_structure = (void *) new_obj;
    new_obj->prev_structure = prev_obj;
    alloc_status[type_name##_MT].objects_count++;
    return new_obj;
}
void deallocate_##type_name(type_name *kill_me) {
    type_name *prev_obj = PREV_OBJECT(kill_me, type_name);
    type_name *next_obj = NEXT_OBJECT(kill_me, type_name);
    if (prev_obj == NULL) {
```

```
        alloc_status[type_name##_MT].first_in_memory = next_obj;
    } else {
        prev_obj->next_structure = next_obj;
    }
    if (next_obj == NULL) {
        alloc_status[type_name##_MT].last_in_memory = prev_obj;
    } else {
        next_obj->prev_structure = prev_obj;
    }
    alloc_status[type_name##_MT].objects_count--;
}
type_name *allocate_##type_name##_before(type_name *existing) {
    type_name *new_obj = allocate_##type_name();
    deallocate_##type_name(new_obj);
    new_obj->prev_structure = existing->prev_structure;
    if (existing->prev_structure != NULL)
        ((type_name *) existing->prev_structure)->next_structure = new_obj;
    else alloc_status[type_name##_MT].first_in_memory = (void *) new_obj;
    new_obj->next_structure = existing;
    existing->prev_structure = new_obj;
    alloc_status[type_name##_MT].objects_count++;
    return new_obj;
}
```

§**21.** `ALLOCATE_IN_ARRAYS` is still more obfuscated. When we `ALLOCATE_IN_ARRAYS(X, 100)`, the result will be definitions of a new type `X_block` and functions `allocate_X`, `allocate_X_block`, `deallocate_X_block` and `allocate_X_block_before` (though the last is not destined ever to be used). Note that we are not provided with the means to deallocate individual objects this time: that's the trade-off for allocating in blocks.

```
define ALLOCATE_IN_ARRAYS(type_name, NO_TO_ALLOCATE_TOGETHER)
MAKE_REFERENCE_ROUTINES(type_name, type_name##_array_MT)
typedef struct type_name##_array {
    int used;
    struct type_name array[NO_TO_ALLOCATE_TOGETHER];
    MEMORY_MANAGEMENT
} type_name##_array;
ALLOCATE_INDIVIDUALLY(type_name##_array)
type_name##_array *next_##type_name##_array = NULL;
struct type_name *allocate_##type_name(void) {
    if ((next_##type_name##_array == NULL) ||
        (next_##type_name##_array->used >= NO_TO_ALLOCATE_TOGETHER)) {
        alloc_status[type_name##_array_MT].no_allocated_together = NO_TO_ALLOCATE_TOGETHER;
        next_##type_name##_array = allocate_##type_name##_array();
        next_##type_name##_array->used = 0;
    }
    return &(next_##type_name##_array->array[
        next_##type_name##_array->used++]);
}
```

The structure type_name##_array is private to this section.

§**22. Expanding many macros.** Each given structure must have a typedef name, say `marvel`, and can be used in one of two ways. Either way, we can obtain a new one with the macro `CREATE(marvel)`.

Either (a) it will be individually allocated. In this case `marvel_MT` should be defined with a new MT (memory type) number, and the macro `ALLOCATE_INDIVIDUALLY(marvel)` should be expanded. The first and last objects created will be `FIRST_OBJECT(marvel)` and `LAST_OBJECT(marvel)`, and we can proceed either way through a double linked list of them with `PREV_OBJECT(mv, marvel)` and `NEXT_OBJECT(mv, marvel)`. For convenience, we can loop through marvels, in creation order, using `LOOP_OVER(var, marvel)`, which expands to a `for` loop in which the variable `var` runs through each created marvel in turn; or equally we can run backwards through using `LOOP_BACKWARDS_OVER(var, marvel)`. In addition, there are corruption checks to protect the memory from overrunning accidents, and the structure can be used as a value in the symbols table.

Or (b) it will be allocated in arrays. Once again we can obtain new marvels with `CREATE(marvel)`. This is more efficient both in speed and memory usage, but we lose the ability to loop through the objects. For this arrangement, define `marvel_array_MT` with a new MT number and expand the macro `ALLOCATE_IN_ARRAYS(marvel, 100)`, where 100 (or what may you) is the number of objects allocated jointly as a block.

Deep breath, then: the following macros define several hundred functions.

```
ALLOCATE_INDIVIDUALLY(action_name)
ALLOCATE_INDIVIDUALLY(activity)
ALLOCATE_INDIVIDUALLY(adjectival_phrase)
ALLOCATE_INDIVIDUALLY(adjective_meaning)
ALLOCATE_INDIVIDUALLY(auxiliary_file)
ALLOCATE_INDIVIDUALLY(binary_predicate)
ALLOCATE_INDIVIDUALLY(blorb_figure)
ALLOCATE_INDIVIDUALLY(blorb_sound)
ALLOCATE_INDIVIDUALLY(booked_rule)
ALLOCATE_INDIVIDUALLY(command_index_entry)
ALLOCATE_INDIVIDUALLY(control_structure_phrase)
ALLOCATE_INDIVIDUALLY(data_type_id_rules)
ALLOCATE_INDIVIDUALLY(definition)
ALLOCATE_INDIVIDUALLY(determiner)
ALLOCATE_INDIVIDUALLY(documentation_ref)
ALLOCATE_INDIVIDUALLY(equation_node)
ALLOCATE_INDIVIDUALLY(equation_symbol)
ALLOCATE_INDIVIDUALLY(equation)
ALLOCATE_INDIVIDUALLY(excerpt_meaning)
ALLOCATE_INDIVIDUALLY(extension_census_datum)
ALLOCATE_INDIVIDUALLY(extension_dictionary_entry)
ALLOCATE_INDIVIDUALLY(extension_file)
ALLOCATE_INDIVIDUALLY(external_file)
ALLOCATE_INDIVIDUALLY(generalisation)
ALLOCATE_INDIVIDUALLY(grammar_line)
ALLOCATE_INDIVIDUALLY(grammar_verb)
ALLOCATE_INDIVIDUALLY(heading)
ALLOCATE_INDIVIDUALLY(i6_inclusion_matter)
ALLOCATE_INDIVIDUALLY(i6_memory_setting)
ALLOCATE_INDIVIDUALLY(I6T_intervention)
ALLOCATE_INDIVIDUALLY(implication)
ALLOCATE_INDIVIDUALLY(internal_test_case)
ALLOCATE_INDIVIDUALLY(inv_token_problem_token)
ALLOCATE_INDIVIDUALLY(known_extension_clash)
ALLOCATE_INDIVIDUALLY(label_namespace)
ALLOCATE_INDIVIDUALLY(lexicon_entry)
ALLOCATE_INDIVIDUALLY(list_together_routine)
```

```
ALLOCATE_INDIVIDUALLY(literal_list)
ALLOCATE_INDIVIDUALLY(literal_pattern_name)
ALLOCATE_INDIVIDUALLY(literal_pattern)
ALLOCATE_INDIVIDUALLY(literal_text)
ALLOCATE_INDIVIDUALLY(loop_over_scope)
ALLOCATE_INDIVIDUALLY(map_component)
ALLOCATE_INDIVIDUALLY(meaning_list)
ALLOCATE_INDIVIDUALLY(measurement_definition)
ALLOCATE_INDIVIDUALLY(named_action_pattern)
ALLOCATE_INDIVIDUALLY(named_rulebook_outcome)
ALLOCATE_INDIVIDUALLY(noun_filter_token)
ALLOCATE_INDIVIDUALLY(parse_node)
ALLOCATE_INDIVIDUALLY(past_tense_action_record)
ALLOCATE_INDIVIDUALLY(past_tense_condition_record)
ALLOCATE_INDIVIDUALLY(pcalc_prop_deferral)
ALLOCATE_INDIVIDUALLY(ph_stack_frame_box)
ALLOCATE_INDIVIDUALLY(phrase)
ALLOCATE_INDIVIDUALLY(plural_dictionary_entry)
ALLOCATE_INDIVIDUALLY(pointer_allocation)
ALLOCATE_INDIVIDUALLY(preposition_usage)
ALLOCATE_INDIVIDUALLY(property_name)
ALLOCATE_INDIVIDUALLY(property_permission)
ALLOCATE_INDIVIDUALLY(quantifier)
ALLOCATE_INDIVIDUALLY(quantity)
ALLOCATE_INDIVIDUALLY(reserved_command_verb)
ALLOCATE_INDIVIDUALLY(rulebook)
ALLOCATE_INDIVIDUALLY(runtime_ambiguity)
ALLOCATE_INDIVIDUALLY(scene)
ALLOCATE_INDIVIDUALLY(simple_memory_claim)
ALLOCATE_INDIVIDUALLY(source_file)
ALLOCATE_INDIVIDUALLY(stored_block_value)
ALLOCATE_INDIVIDUALLY(table_column)
ALLOCATE_INDIVIDUALLY(table)
ALLOCATE_INDIVIDUALLY(test_scenario)
ALLOCATE_INDIVIDUALLY(text_routine)
ALLOCATE_INDIVIDUALLY(type_macro_definition)
ALLOCATE_INDIVIDUALLY(type_template_definition)
ALLOCATE_INDIVIDUALLY(type_template_obligation)
ALLOCATE_INDIVIDUALLY(use_option)
ALLOCATE_INDIVIDUALLY(verb_usage)
ALLOCATE_INDIVIDUALLY(VM_usage_note)
ALLOCATE_INDIVIDUALLY(world_object)
```

§**23.** So much for the managed structures: now for the unmanaged structures.

```
ALLOCATE_IN_ARRAYS(action_name_list, 1000)
ALLOCATE_IN_ARRAYS(action_pattern, 100)
ALLOCATE_IN_ARRAYS(activity_crossref, 100)
ALLOCATE_IN_ARRAYS(activity_list, 1000)
ALLOCATE_IN_ARRAYS(adjective_list_entry, 1000)
ALLOCATE_IN_ARRAYS(ap_optional_clause, 400)
ALLOCATE_IN_ARRAYS(data_type_casting_rule, 100)
ALLOCATE_IN_ARRAYS(data_type_comparison_schema, 100)
ALLOCATE_IN_ARRAYS(description_docket, 100)
ALLOCATE_IN_ARRAYS(dimensional_rule, 100)
ALLOCATE_IN_ARRAYS(extension_identifier_database_entry, 100)
ALLOCATE_IN_ARRAYS(i6_schema, 100)
ALLOCATE_IN_ARRAYS(inference, 100)
ALLOCATE_IN_ARRAYS(invocation_list_entry, 100)
ALLOCATE_IN_ARRAYS(invocation_list, 100)
ALLOCATE_IN_ARRAYS(invocation_options, 100)
ALLOCATE_IN_ARRAYS(invocation_token_list, 500)
ALLOCATE_IN_ARRAYS(invocation, 100)
ALLOCATE_IN_ARRAYS(kind_of_value, 1000)
ALLOCATE_IN_ARRAYS(parse_node_annotation, 500)
ALLOCATE_IN_ARRAYS(pcalc_func, 1000)
ALLOCATE_IN_ARRAYS(pcalc_prop, 1000)
ALLOCATE_IN_ARRAYS(phrase_option, 100)
ALLOCATE_IN_ARRAYS(placement_affecting, 100)
ALLOCATE_IN_ARRAYS(scene_connector, 1000)
ALLOCATE_IN_ARRAYS(single_integer, 100)
ALLOCATE_IN_ARRAYS(specification, 100)
ALLOCATE_IN_ARRAYS(stacked_variable_list, 100)
ALLOCATE_IN_ARRAYS(stacked_variable_owner_list, 100)
ALLOCATE_IN_ARRAYS(stacked_variable_owner, 100)
ALLOCATE_IN_ARRAYS(stacked_variable, 100)
ALLOCATE_IN_ARRAYS(time_period, 100)
ALLOCATE_IN_ARRAYS(type_id_rules, 100)
ALLOCATE_IN_ARRAYS(unit_sequence, 50)
ALLOCATE_IN_ARRAYS(vocabulary_entry, 100)
```

§**24. Simple memory allocations.** Not all of our memory will be claimed in the form of structures: now and then we need to use the equivalent of traditional `malloc` and `calloc` routines.

```
define EXTENSION_DICTIONARY_MREASON 0
define INDEX_SORTING_MREASON 1
define INSTANCE_COUNTING_MREASON 2
define LEXER_TEXT_MREASON 3
define LEXER_WORDS_MREASON 4
define MAP_INDEX_MREASON 5
define PARTITION_MREASON 6
define STREAM_MREASON 7
define TYPE_TABLES_MREASON 8
define INV_LIST_MREASON 9
define NUMBER_OF_MREASONS 10

char *memory_needs[NUMBER_OF_MREASONS] = {
    "extension dictionary",
    "index sorting",
    "instance-of-kind counting",
    "source text",
    "source text details",
    "map in the World index",
    "initial state for relations in groups",
    "internal text-handling",
    "tables of details of the kinds of values",
    "lists for type-checking invocations"
};
```

§**25.** We keep some statistics on this. The value for "memory claimed" is the net amount of memory currently owned, which is increased when we allocate it and decreased when we free it. Whether the host OS is able to make efficient use of the memory we free, we can't know, but it probably is, and therefore the best estimate of how well we're doing is the "maximum memory claimed" – the highest recorded net usage count over Inform's run.

```
int max_memory_at_once_for_each_need[NUMBER_OF_MREASONS],
    memory_claimed_for_each_need[NUMBER_OF_MREASONS],
    number_of_claims_for_each_need[NUMBER_OF_MREASONS];
int total_claimed_simply = 0;
```

§**26.** Our allocation routines behave just like the standard C library's `malloc` and `calloc`, but where a third argument supplies a reason why the memory is needed, and where any failure to allocate memory is tidily dealt with. Inform exits on any such failure, so that the caller can be certain that the return values of these functions are always non-`NULL` pointers.

```
void *I7_calloc(int how_many, int size_in_bytes, int reason) {
    return I7_alloc(how_many, size_in_bytes, reason);
}
void *I7_malloc(int size_in_bytes, int reason) {
    return I7_alloc(-1, size_in_bytes, reason);
}
```

The function I7_calloc is called from 3/lex, 4/edict, 4/vm, 5/rel, 7/tids, 7/inv, 9/cot, 9/map, 9/tmap and 11/ina.
The function I7_malloc is called from 2/strm and 3/lex.

§**27.** And this, then, is the joint routine implementing both.

```
void *I7_alloc(int N, int S, int R) {
    void *pointer;
    int bytes_needed;
    if ((R < 0) || (R >= NUMBER_OF_MREASONS)) internal_error("no such memory reason");
    if (total_claimed_simply == 0) ⟨Zero out the statistics on simple memory allocations 29⟩;
    ⟨Claim the memory using malloc or calloc as appropriate 28⟩;
    ⟨Update the statistics on simple memory allocations 30⟩;
    simple_memory_claim *smc = CREATE(simple_memory_claim);
    smc->memory_claimed = pointer;
    smc->extent_of_claim = bytes_needed;
    return pointer;
}
```

§**28.** I am nervous about assuming that `calloc(0, X)` returns a non-`NULL` pointer in all implementations of the standard C library, so the case when `N` is zero allocates a tiny but positive amount of memory, just to be safe.

⟨Claim the memory using malloc or calloc as appropriate 28⟩ ≡

```
    if (N > 0) {
        pointer = calloc(N, S);
        bytes_needed = N*S;
    } else {
        pointer = malloc(S);
        bytes_needed = S;
    }
    if (pointer == NULL)
        memory_allocation_problem(_P_(BelievedImpossible),                    i.e., not reliably testable
            memory_needs[R]);
```

This code is used in §27.

§**29.** These statistics have no function except to improve the diagnostics in the debugging log, but they are very cheap to keep, since `I7_alloc` is called only rarely and to allocate large blocks of memory.

⟨Zero out the statistics on simple memory allocations 29⟩ ≡

```
    int i;
    for (i=0; i<NUMBER_OF_MREASONS; i++) {
        max_memory_at_once_for_each_need[i] = 0;
        memory_claimed_for_each_need[i] = 0;
        number_of_claims_for_each_need[i] = 0;
    }
```

This code is used in §27.

§**30.**

⟨Update the statistics on simple memory allocations 30⟩ ≡

```
memory_claimed_for_each_need[R] += bytes_needed;
total_claimed_simply += bytes_needed;
number_of_claims_for_each_need[R]++;
if (memory_claimed_for_each_need[R] > max_memory_at_once_for_each_need[R])
    max_memory_at_once_for_each_need[R] = memory_claimed_for_each_need[R];
```

This code is used in §27.

§**31.** We also provide our own wrapper for `free`. This looks as if it might be a little slow, since it has to loop through all of the SMC objects to find the right one. But note that SMC objects are removed from the SMC linked list using `DESTROY` when the memory in question is freed; that means there are typically fewer than 10 in the domain of `LOOP_OVER` at any one time.

```
void I7_free(void *pointer, int R) {
    if ((R < 0) || (R >= NUMBER_OF_MREASONS)) internal_error("no such memory reason");
    if (pointer == NULL) internal_error("can't free NULL memory");
    simple_memory_claim *smc;
    LOOP_OVER(smc, simple_memory_claim)
        if (smc->memory_claimed == pointer) {
            free(pointer);
            memory_claimed_for_each_need[R] -= smc->extent_of_claim;
            total_claimed_simply -= smc->extent_of_claim;
            DESTROY(smc, simple_memory_claim);
            return;
        }
    internal_error("can't free this memory, which is not now allocated");
}
```

The function I7_free is called from 2/strm, 3/lex, 4/edict, 4/vm, 7/tids, 9/tmap and 11/ina.

§**32.** Some parts of Inform have no convenient way to deallocate their own memory – notably the lexer, when it calls for extensions to its workspace. At the end of the run, then, we call the following, which frees whatever has not already been freed. We don't bother to `DESTROY` them, since the SMC object list will never be used again.

```
void I7_free_all_remaining(void) {
    simple_memory_claim *smc;
    LOOP_OVER(smc, simple_memory_claim) {
        free (smc->memory_claimed);
        total_claimed_simply -= smc->extent_of_claim;
    }
    if (total_claimed_simply != 0)
        internal_error("memory not freed exactly correctly");
}
```

§**33.** And the following provides statistics, and a mini-report, for the memory report in the debugging log (for which, see below).

```
int log_I7_alloc_usage(int total) {
    if (total_claimed_simply == 0) return 0;
    int i, t = 0;
    for (i=0; i<NUMBER_OF_MREASONS; i++) {
        t += max_memory_at_once_for_each_need[i];
        if (total > 0)
            LOG("0.%03d: %s - %d bytes in %d claim(s)\n",
                memory_proportion(max_memory_at_once_for_each_need[i], total),
                memory_needs[i],
                max_memory_at_once_for_each_need[i],
                number_of_claims_for_each_need[i]);
    }
    return t;
}
```

§**34. Memory usage report.** A small utility routine to help keep track of Inform's unquestioned profligacy.

```
void debug_memory_statistics(void) {
    int total_for_objects = MEMORY_GRANULARITY*no_blocks_allocated;                    usage in bytes
    int total_for_SMAs = log_I7_alloc_usage(0);                                        usage in bytes
    int sorted_usage[NO_MEMORY_TYPES];                                  memory type numbers, in usage order
    int total = (total_for_objects + total_for_SMAs)/1024;                          total memory usage in KB
    ⟨Sort the table of memory type usages into decreasing size order 36⟩;
    int total_for_objects_used = 0;                          out of the total_for_objects, the bytes used
    int total_objects = 0;
    ⟨Calculate the memory usage for objects 35⟩;
    int overhead_for_objects = total_for_objects - total_for_objects_used;                bytes wasted
    ⟨Print the report to the debugging log 37⟩;
}
```

The function debug_memory_statistics is invoked by a command in a `.i6t` template file.

§**35.**

⟨Calculate the memory usage for objects 35⟩ ≡

```
    int i, j;
    for (j=0; j<NO_MEMORY_TYPES; j++) {
        i = sorted_usage[j];
        if (alloc_status[i].objects_allocated != 0) {
            if (alloc_status[i].no_allocated_together == 1)
                total_objects += alloc_status[i].objects_allocated;
            else
                total_objects += alloc_status[i].objects_allocated*
                                 alloc_status[i].no_allocated_together;
            total_for_objects_used += alloc_status[i].bytes_allocated;
        }
    }
```

This code is used in §34.

§**36.** This is the criterion for sorting memory types in the report: descending order of total number of bytes allocated.

⟨Sort the table of memory type usages into decreasing size order 36⟩ ≡

```
int i;
for (i=0; i<NO_MEMORY_TYPES; i++) sorted_usage[i] = i;
qsort(sorted_usage, NO_MEMORY_TYPES, sizeof(int), compare_usage_entries);
```

This code is used in §34.

§**37.** And here is the actual report:

⟨Print the report to the debugging log 37⟩ ≡

```
LOG("\nReport by memory manager:\n\n");
LOG("Total consumption was %dK = %dMB, divided up in the following proportions:\n",
    total, (total+512)/1024);
LOG("0.%03d: %d objects in %d frames in %d memory blocks (of %dK each):\n",
    memory_proportion(total_for_objects, total),
    total_objects, no_frames_allocated, no_blocks_allocated, MEMORY_GRANULARITY/1024);
LOG("    0.%03d: memory manager overhead - %d bytes\n",
    memory_proportion(overhead_for_objects, total), overhead_for_objects);
int i, j;
for (j=0; j<NO_MEMORY_TYPES; j++) {
    i = sorted_usage[j];
    if (alloc_status[i].objects_allocated != 0) {
        LOG("    0.%03d: %s  -  ",
            memory_proportion(alloc_status[i].bytes_allocated, total),
            alloc_status[i].name_of_type);
        if (alloc_status[i].no_allocated_together == 1) {
            LOG("%d ", alloc_status[i].objects_count);
            if (alloc_status[i].objects_count != alloc_status[i].objects_allocated)
                LOG("(+%d deleted) ",
                    alloc_status[i].objects_allocated - alloc_status[i].objects_count);
        } else LOG("%d blocks of %d = %d ",
            alloc_status[i].objects_allocated, alloc_status[i].no_allocated_together,
            alloc_status[i].objects_allocated*alloc_status[i].no_allocated_together);
        LOG("objects, %d bytes\n", alloc_status[i].bytes_allocated);
    }
}
log_I7_alloc_usage(total);
LOG("\nNOTES:\nSpecifications: %d created, %d copies made\n",
    new_spec_count, copied_spec_count);
LOG("Permanent MLs: %d; Ephemeral MLs: %d; GAP movements: %d\n",
    no_permanent_MLs, no_ephemeral_MLs, GAP_movements);
LOG("Created %d kovas, %d kovkos, %d kovivs, %d kovcons\n",
    no_kovas_created, no_kovkos_created, no_kovivs_created, no_kovcons_created);
```

This code is used in §34.

§**38.**

```
int compare_usage_entries(const void *ent1, const void *ent2) {
    int ix1 = *((const int *) ent1);
    int ix2 = *((const int *) ent2);
    return alloc_status[ix2].bytes_allocated - alloc_status[ix1].bytes_allocated;
}
```

§**39.** Finally, a little routine to compute the proportions of memory for each usage. Recall that `bytes` is measured in bytes, but `total` in kilobytes.

```
int memory_proportion(int bytes, int total) {
    float B = (float) bytes, T = (float) total;
    float P = (1000*B)/(1024*T);
    return (int) P;
}
```

§**40. Run-time pointer type checking.** In several places Inform needs to store pointers of type `void *`, that is, pointers which have no indication of what type of data they point to. This is not type-safe and therefore offers plenty of opportunity for blunders. The following provides run-time type checking to ensure that each time we dereference a typeless pointer, it does indeed point to a structure of the type we think it should.

The structure `general_pointer` holds a `void *` pointer to any one of the following:

(a) `NULL`, to which we assign ID number -1;
(b) `char`, to which we assign ID number 1000;
(c) any individually allocated structure of the types listed above, to which we assign the ID numbers used above: for instance, `blorb_figure_MT` is the ID number for a `general_pointer` which points to a `blorb_figure` structure.

define `NULL_GENERAL_POINTER (STORE_POINTER_NULL())`

```
typedef struct general_pointer {
    void *pointer_to_data;
    int run_time_type_code;
} general_pointer;

general_pointer STORE_POINTER_NULL(void) {
    general_pointer gp;
    gp.pointer_to_data = NULL;
    gp.run_time_type_code = -1;                    guaranteed to differ from all _MT values
    return gp;
}
int GENERAL_POINTER_IS_NULL(general_pointer gp) {
    if (gp.run_time_type_code == -1) return TRUE;
    return FALSE;
}
```

The function GENERAL_POINTER_IS_NULL is called from 6/tcpr.
The structure general_pointer is private to this section.

§**41.** The symbols tables need to look at pointer values directly without knowing their types, but only to test equality, so we abstract that thus. And the debugging log also shows actual hexadecimal addresses to distinguish nameless objects and to help with interpreting output from GDB, so we abstract that too.

```
define COMPARE_GENERAL_POINTERS(gp1, gp2)
    (gp1.pointer_to_data == gp2.pointer_to_data)
define GENERAL_POINTER_AS_INT(gp)
    ((pointer_sized_int) gp.pointer_to_data)
```

§**42.** If we have a pointer to `circus` (say) then `g=STORE_POINTER_circus(p)` returns a `general_pointer` with `p` as the actual pointer, but will not compile unless `p` is indeed of type `circus *`. When we later `RETRIEVE_POINTER_circus(g)`, an internal error is thrown if `g` contains a pointer which is other than `void *`, or which has never been referenced.

```
define MAKE_REFERENCE_ROUTINES(type_name, id_code)
general_pointer STORE_POINTER_##type_name(type_name *data) {
    general_pointer gp;
    gp.pointer_to_data = (void *) data;
    gp.run_time_type_code = id_code;
    return gp;
}
type_name *RETRIEVE_POINTER_##type_name(general_pointer gp) {
    if (gp.run_time_type_code != id_code) {
        LOG("Wanted ID code %d, found %d\n", id_code, gp.run_time_type_code);
        internal_error("attempt to retrieve wrong pointer type as " #type_name);
    }
    return (type_name *) gp.pointer_to_data;
}
general_pointer PASS_POINTER_##type_name(general_pointer gp) {
    if (gp.run_time_type_code != id_code) {
        LOG("Wanted ID code %d, found %d\n", id_code, gp.run_time_type_code);
        internal_error("attempt to pass wrong pointer type as " #type_name);
    }
    return gp;
}
int VALID_POINTER_##type_name(general_pointer gp) {
    if (gp.run_time_type_code == id_code) return TRUE;
    return FALSE;
}
```

§**43.** Suitable `MAKE_REFERENCE_ROUTINES` were expanded for all of the memory allocated objects above; so that leaves only humble `char *` pointers:

```
MAKE_REFERENCE_ROUTINES(char, 1000)
```

§**44. The general pointer attached to a specification.** As we shall see in Chapter 7, the SP structure has a "structure" field which is a general pointer – SPs can represent a wide range of different Inform concepts, and the field usually points to the internal data structure in question. For instance, the text "Table of Gotham City Hideouts" might parse to a SP representing a constant value of kind "table" (i.e., `kova(TABLE_TY)`), and its structure field would be a pointer to the `table` structure created when Inform parsed the Table of Gotham City Hideouts elsewhere in the source text.

It greatly simplifies the code to provide convenient routines to convert pointers to structures to and from the specifications representing them. We'll call these "conversion routines".

```
define CONVERSION_ROUTINES(SPECNAME, STRUCTNAME)
STRUCTNAME *SPECNAME##_spec_to_##STRUCTNAME(specification *spec) {
    if (spec == NULL) internal_error("tried to find " #STRUCTNAME " within NULL type");
    if (!(spec_is_actual_CONSTANT_of_kova(spec, SPECNAME##_TY))) {
        LOG("Bad spec is: $S\n", spec);
        internal_error("tried to find " #STRUCTNAME " inside SP of wrong type");
    }
    STRUCTNAME *str = RETRIEVE_POINTER_##STRUCTNAME(spec_get_structure_field(spec));
    if (str == NULL) internal_error("found NULL " #STRUCTNAME " within " #SPECNAME " type");
    return str;
}
specification *STRUCTNAME##_to_##SPECNAME##_spec(STRUCTNAME *str) {
    if (str == NULL) internal_error("tried to make null " #STRUCTNAME " into " #SPECNAME " type");
    specification *spec = new_actual_CONSTANT_spec(kova(SPECNAME##_TY));
    spec_set_structure_field(spec, STORE_POINTER_##STRUCTNAME(str));
    return spec;
}
define RETRIEVE_FROM_SPEC(spec, structure)
    RETRIEVE_POINTER_##structure(spec_get_structure_field(spec))
define ATTACH_TO_SPEC(spec, structure, pointer)
    spec_set_structure_field(spec, STORE_POINTER_##structure(pointer))
```

§**45.** And here are the cases we actually need:

```
CONVERSION_ROUTINES(ACTION_NAME, action_name)
CONVERSION_ROUTINES(ACTIVITY, activity)
CONVERSION_ROUTINES(EQUATION, equation)
CONVERSION_ROUTINES(OBJECT, world_object)
CONVERSION_ROUTINES(PROPERTY, property_name)
CONVERSION_ROUTINES(RELATION, binary_predicate)
CONVERSION_ROUTINES(RULE, booked_rule)
CONVERSION_ROUTINES(RULEBOOK, rulebook)
CONVERSION_ROUTINES(RULEBOOK_OUTCOME, named_rulebook_outcome)
CONVERSION_ROUTINES(SCENE, scene)
CONVERSION_ROUTINES(TABLE, table)
CONVERSION_ROUTINES(TABLE_COLUMN, table_column)
CONVERSION_ROUTINES(UNDERSTANDING, grammar_verb)
CONVERSION_ROUTINES(USEOPTION, use_option)
```

Streams 2/strm

Purpose

Support for writing structured textual output, perhaps to the screen, to a file, or to a flexible-sized area of memory.

Definitions

¶1. Inform produces textual output in many formats (HTML, EPS, plain text, Inform 6 code, XML, and so on), writing to a variety of files, and often needs to juggle and rearrange partially written segments. These texts tend to be structured with running indentation, and may eventually need to be written to a disc file with either ISO Latin-1 or UTF-8 encodings. "Streams" are an abstraction to make it easy to handle all of this. The writer to a stream never needs to know where the text will come out, or how it should be indented or encoded.

¶2. The `STREAM` data type began as a generalisation of the standard C library's `FILE`, and it is used in mostly similar ways. The user – the whole program outside of this section – deals only with `STREAM *` pointers to represent streams in use.

All stream handling is defined via macros. While many operations could be handled by ordinary functions, others cannot. `STREAM` cannot have exactly the semantics of `FILE` since we cannot rely on the host operating system to allocate and deallocate the structures behind `STREAM *` pointers; and we cannot use our own memory system, either, since we need stream handling to work both before the memory allocator starts and after it has finished. Our macros allow us to hide all this from the rest of Inform. Besides that, a macro approach makes it easier to retrofit new implementations as necessary. (The present implementation is the second stab at it.)

¶3. The main purpose of many functions in Inform is to write textual material to some file. Such functions almost always have a special argument in their prototypes: `OUTPUT_STREAM`. This tells them where to pipe their output, which is always to a "current stream" called `OUT`. What this leads to, and who will see that it's properly opened and closed, are not their concern.

define `OUTPUT_STREAM STREAM *OUT` *used only as a function prototype argument*

¶4. `WRITE` is essentially `printf` but prints to the current stream instead of `stdout`, while `STREAM_WRITE` does the same but to any named stream. `INDEX` writes to the currently open index file; another commonly-used macro, `LOG`, does eventually write to a stream but has a more elaborate definition in "Debugging Log".

They are defined as what the C standard calls "variadic macros", meaning they have a variable-number-of-arguments token. They are written in the old-fashioned pre-C99 way, for compatibility with old copies of GCC, and avoid the need for comma deletion around empty tokens, as that is a point of incompatibility between implementations of the C preprocessor `cpp`. All the same, if you're porting this code, you may need to rewrite the macro with `...` in place of `args...` in the header, and then `__VA_ARGS__` in place of `args` in the definition: that being the modern way, apparently.

```
#define WRITE(args...) stream_printf(OUT, args)
#define INDEX(args...) stream_printf(ifl, args)
#define STREAM_WRITE(stream, args...) stream_printf(stream, args)
```

¶5. PUT and STREAM_PUT similarly print single characters, which are specified as unsigned integer values. In practice, STREAM_WRITE and STREAM_PUT are seldom needed because there is almost always only one stream of interest at a time – OUT, the current stream.

```
define PUT(c) stream_putc(c, OUT)
define STREAM_PUT(stream, c) stream_putc(c, stream)
```

¶6. Each stream has a current indentation level, initially 0. Lines of text will be indented by one tab stop for each level; it's an error for the level to become negative.

```
define INDENT stream_indent(OUT);
define STREAM_INDENT(x) stream_indent(x);
define OUTDENT stream_outdent(OUT);
define STREAM_OUTDENT(x) stream_outdent(x);
```

¶7. Three other output streams are always open. One is NULL, that is, its value as a STREAM * pointer is NULL, the generic C null pointer. This represents an oubliette: it is entirely valid to use it, but output sent to NULL will never be seen again.

The others are STDOUT and STDERR. As the names suggest these are wrappers for stdout and stderr, the standard console output and error messages "files" provided by the C library.

Inform always uses STREAM_WRITE(STDOUT, ...) instead of printf(...) for console output, so there are no uses of printf anywhere in the program.

```
define STDOUT stream_get_stdout()
define STDERR stream_get_stderr()
```

¶8. Other streams only exist when explicitly created, or "opened". A function is only allowed to open a new stream if it can be proved that this stream will always subsequently be "closed". (Except for the possibility of Inform halting with an internal error, and therefore an exit(1), while the stream is still open.) A stream can be opened and closed only once, and outside that time its state is undefined: it must not be used at all.

The simplest way is to make a temporary stream, which can be used as a sort of clipboard. For instance, suppose we have to compile X before Y, but have to ensure Y comes before X in the eventual output. We create a temporary stream, compile X into it, then compile Y to OUT, then copy the temporary stream into OUT and dispose of it.

Temporary streams are always created in memory, held in C's local stack frame rather than allocated and freed via malloc and free. It must always be possible to prove that execution passes from TEMPORARY_STREAM to CLOSE_TEMPORARY_STREAM. The stream exists only between those macros, only in that one function, and is called TEMP. We can legitimately create a temporary stream many times in one function (for instance inside a loop body) because each time TEMP is created as a new stream, overwriting the old one. TEMP is a different stream each time it is created, so it does not violate the rule that every stream is opened and closed once only.

```
define TEMPORARY_STREAM
    char dest[2048];
    STREAM TEMP_stream_structure = stream_new_buffer(2048, dest);
    STREAM *TEMP = &TEMP_stream_structure;
define CLOSE_TEMPORARY_STREAM
    STREAM_CLOSE(TEMP);
```

¶9. Otherwise we can create new globally existing streams, provided we take on the responsibility for seeing that they are properly closed. There are two choices: a stream in memory, allocated via `malloc` and freed by `free` when the stream is closed; or a file written to disc, opened via `fopen` and later closed by `fclose`. Files are always written in text mode, that is, `"w"` not `"wb"`, for those platforms where this makes a difference.

Inform uses streams to handle all of its text file output, so there are no calls to `fprintf` anywhere in the program except in the streams implementation below.

```
define STREAM_OPEN_TO_FILE(new, fn, enc) stream_open_to_file(new, fn, enc)
define STREAM_OPEN_TO_FILE_APPEND(new, fn, enc) stream_open_to_file_append(new, fn, enc)
define STREAM_OPEN_IN_MEMORY(new) stream_open_to_memory(new, 20480)
define STREAM_CLOSE(stream) stream_close(stream)
```

¶10. The following operation is equivalent to `fflush` and makes it more likely (I put it no higher) that the text written to a stream has all actually been copied onto the disc, rather than sitting in some operating system buffer. This helps ensure that the debugging log is up to the minute, in case of a crash, but its absence wouldn't hurt Inform's normal function. Flushing a memory stream is legal but does nothing.

```
define STREAM_FLUSH(stream) stream_flush(stream)
```

¶11. A piece of information we can read for any stream is the number of characters written to it: its "extent". In fact, UTF-8 multi-byte encoding schemes, together with differing platform interpretations of C's `'\n'`, mean that this extent is not necessarily either the final file size or bytes or the actual number of human-readable characters. We will only actually use it to detect whether text has, or has not, been written to a stream between two points in time, by seeing whether or not it has increased.

```
define STREAM_EXTENT(x) stream_get_position(x)
```

¶12. Any stream can have the following flag set or cleared. When this is set, the XML (and HTML) escapes of `&` for ampersand, and `<` and `>` for angle brackets, will be used automatically on writing. By default this flag is clear, that is, no conversion is made.

```
define STREAM_USE_XML_ESCAPES(x, state) if (x) x->use_xml_escapes = state;
```

¶13. A theological question: what is the text encoding for the null stream?

```
define STREAM_ENCODING(x) ((x)?(x->encoding_to_write_to_file):ISO_ENC)
```

¶14. The remaining operations are available only for streams in memory (well, and for `NULL`, but of course they do nothing when applied to that). While they could be provided for file streams, this would be so inefficient that we will pretend it is impossible. Any Inform function which might need to use one of these operations should open with the following sentinel macro:

```
define STREAM_MUST_BE_IN_MEMORY(x)
    if ((x != NULL) && (x->write_to_memory == NULL))
        internal_error("STREAM not in memory");
```

¶15. First, we can erase one or more recently written characters:

```
define STREAM_BACKSPACE(x) stream_set_position(x, stream_get_position(x) - 1)
define STREAM_ERASE_BACK_TO(start_position) stream_set_position(OUT, start_position)
```

¶16. Second, we can look at the text written. The minimal form is to look at just the most recent character, but we can also copy one entire memory stream into another stream (where the target can be either a memory or file stream).

We can also access a null-terminated C string containing the contents of a memory stream up as far as, at least, the first 255 characters. `STREAM_TEXT` is permitted to return a truncated string if a memory stream exceeds that.

```
define STREAM_MIN_ACCESSIBLE_LENGTH 255
define STREAM_MOST_RECENT_CHAR(x) stream_latest(x)
define STREAM_COPY(to, from) stream_copy(to, from)
define STREAM_TEXT(x) stream_get_text(x)
```

¶17. So much for the definition; now the implementation. Here is the `STREAM` structure. Open memory streams are represented by structures where `write_to_memory` is valid, open file streams by those where `write_to_file` is valid. That counts every open stream except `NULL`, which of course doesn't point to a `STREAM` structure at all.

```
typedef struct STREAM {
    FILE *write_to_file;                          for an open stream, exactly one of these is NULL
    char *write_to_memory;
    int encoding_to_write_to_file;                                relevant only for file streams
    int use_xml_escapes;                                                          see above
    int allocated_by_malloc;                      was the write_to_memory pointer claimed by malloc?
    int chars_written;                                    number of characters sent, counting \n as 1
    int chars_capacity;              maximum number the stream can accept without claiming more resources
    struct STREAM *stream_continues;                  if one memory stream is extended by another
    int indentation_level;                                    number of tab stops in from the left margin
    int indentation_pending;                    we have just ended a line, so further text should indent
} STREAM;
```

The structure STREAM is private to this section.

¶18. When text is stored at `write_to_memory`, it is kept as a zero-terminated C string and encoded as ISO Latin-1, with one byte per character. It turns out to be efficient to preserve a small margin of clear space at the end of the space, so out of the `chars_capacity` space, the following amount will be kept clear:

```
define SPACE_AT_END_OF_STREAM 100
```

¶19. A statistic we keep, since it costs little:

```
int total_file_writes = 0;                        number of text files opened for writing during the run
```

§**1. Initialising the stream structure.** Note that the following fills in sensible defaults for every field, but the result is not a valid open stream; it's a blank form ready to be filled in.

By default the upper limit on file size is 2 GB. It's very hard to see this ever being approached for text files whose size is proportionate to the result of human writing. The only output file with a sorceror's-apprentice-like ability to grow and grow is the debugging file, and if it should reach 2 GB then it *deserves* to be truncated and we will shed no tears.

```
void stream_initialise(STREAM *stream) {
    if (stream == NULL) internal_error("tried to initialise NULL stream");
    stream->write_to_file = NULL;
    stream->write_to_memory = NULL;
    stream->allocated_by_malloc = FALSE;
    stream->chars_written = 0;
    stream->chars_capacity = 2147483647;
    stream->stream_continues = NULL;
    stream->encoding_to_write_to_file = ISO_ENC;
    stream->use_xml_escapes = FALSE;
    stream->indentation_level = 0;
    stream->indentation_pending = FALSE;
}
```

§**2. Standard I/O wrappers.** The first call to `stream_get_stdout()` creates a suitable wrapper for `stdout` and returns a `STREAM *` pointer to it; subsequent calls just return this wrapper.

```
STREAM STDOUT_struct; int stdout_wrapper_initialised = FALSE;
STREAM *stream_get_stdout(void) {
    if (stdout_wrapper_initialised == FALSE) {
        stream_initialise(&STDOUT_struct); STDOUT_struct.write_to_file = stdout;
        stdout_wrapper_initialised = TRUE;
    }
    return &STDOUT_struct;
}
```

§**3.** And similarly for the standard error file.

```
STREAM STDERR_struct; int stderr_wrapper_initialised = FALSE;
STREAM *stream_get_stderr(void) {
    if (stderr_wrapper_initialised == FALSE) {
        stream_initialise(&STDERR_struct); STDERR_struct.write_to_file = stderr;
        stderr_wrapper_initialised = TRUE;
    }
    return &STDERR_struct;
}
```

§**4. Creating file streams.** Note that this can fail, if the host filing system refuses to open the file, so we return `TRUE` if and only if successful.

```
int stream_open_to_file(STREAM *stream, char *filename, int encoding) {
    if (stream == NULL) internal_error("tried to open NULL stream");
    if (filename == NULL) internal_error("stream_open_to_file on null filename");
    stream_initialise(stream);
    stream->write_to_file = iso_fopen(filename, "w");
    if (stream->write_to_file == NULL) return FALSE;
    stream->encoding_to_write_to_file = encoding;
    total_file_writes++;
    return TRUE;
}
```

§**5.** Similarly for appending:

```
int stream_open_to_file_append(STREAM *stream, char *filename, int encoding) {
    if (stream == NULL) internal_error("tried to open NULL stream");
    if (filename == NULL) internal_error("stream_open_to_file on null filename");
    stream_initialise(stream);
    stream->write_to_file = iso_fopen(filename, "a");
    if (stream->write_to_file == NULL) return FALSE;
    stream->encoding_to_write_to_file = encoding;
    total_file_writes++;
    return TRUE;
}
```

§**6. Creating memory streams.** Here we have a choice. One option is to use `malloc` to allocate memory to hold the text of the stream; this too can fail for host platform reasons, so again we return a success code.

```
int stream_open_to_memory(STREAM *stream, int capacity) {
    if (stream == NULL) internal_error("tried to open NULL stream");
    if (capacity < STREAM_MIN_ACCESSIBLE_LENGTH + SPACE_AT_END_OF_STREAM)
        internal_error("memory stream too small");
    stream_initialise(stream);
    stream->write_to_memory = I7_malloc(capacity, STREAM_MREASON);
    if (stream->write_to_memory == NULL) return FALSE;
    (stream->write_to_memory)[0] = 0;
    stream->allocated_by_malloc = TRUE;
    stream->chars_capacity = capacity;
    return TRUE;
}
```

§**7.** The other option avoids `malloc` by using specific storage already available. (That avoids gratuitously frequent use of `malloc` and `free`, since this incurs speed costs on some platforms.) If called validly, this cannot fail.

```
STREAM stream_new_buffer(int capacity, char *at) {
    if (at == NULL) internal_error("tried to make stream wrapper for NULL string");
    if (capacity < STREAM_MIN_ACCESSIBLE_LENGTH + SPACE_AT_END_OF_STREAM)
        internal_error("memory stream too small");
    STREAM stream;
    stream_initialise(&stream);
    stream.write_to_memory = at;
    (stream.write_to_memory)[0] = 0;
    stream.chars_capacity = capacity;
    return stream;
}
```

§**8. Flush and close.** Note that flush is an operation which can be performed on any stream, including `NULL`:

```
void stream_flush(STREAM *stream) {
    if (stream == NULL) return;
    if (stream->write_to_file) fflush(stream->write_to_file);
}
```

§**9.** But closing is not allowed for `NULL` or the standard I/O wrappers:

```
void stream_close(STREAM *stream) {
    if (stream == NULL) internal_error("tried to close NULL stream");
    if (stream == &STDOUT_struct) internal_error("tried to close STDOUT stream");
    if (stream == &STDERR_struct) internal_error("tried to close STDERR stream");
    if (stream->chars_capacity == -1) internal_error("stream closed twice");
    if (stream->stream_continues) {
        stream_close(stream->stream_continues);
        stream->stream_continues = NULL;
    }
    stream->chars_capacity = -1;                                    mark as closed
    if (stream->write_to_file) ⟨Take suitable action to close the file stream 10⟩;
    if (stream->write_to_memory) ⟨Take suitable action to close the memory stream 11⟩;
}
```

§**10.** Note that we need do nothing to close a memory stream when the storage was supplied by our client; it only needs freeing if we were the ones who allocated it.

Inscrutably, `fclose` returns `EOF` to report any failure.

⟨Take suitable action to close the file stream 10⟩ ≡

```
    if ((ferror(stream->write_to_file)) || (fclose(stream->write_to_file) == EOF))
        fatal_error("The host computer reported an error trying to write a text file");
    stream->write_to_file = NULL;
```

This code is used in §9.

§**11.** Note that we need do nothing to close a memory stream when the storage was supplied by our client; it only needs freeing if we were the ones who allocated it. `free` is a void function; in theory it cannot fail, if supplied a valid argument.

We have to be very careful once we have called `free`, because that memory may well contain the `STREAM` structure to which `stream` points – see how continuations are made, below.

⟨Take suitable action to close the memory stream 11⟩ ≡
```
if (stream->allocated_by_malloc) {
    I7_free(stream->write_to_memory, STREAM_MREASON); stream = NULL;
}
```
This code is used in §9.

§**12. Writing.** The following function is modelled on the "minimum `printf`" function used as an example in Kernighan and Ritchie, Chapter 7. All text in the formatting string is treated as raw except for `%` escapes, which are handled by ordinary `fprintf`, and new `$` escapes, which are signals to pretty-print NI data structures.

```
void stream_printf(STREAM *stream, char *fmt, ...) {
    va_list ap;                                          the variable argument list signified by the dots
    char *p;
    if (stream == NULL) return;
    STREAM_FLUSH(dl);
    va_start(ap, fmt);                                   macro to begin variable argument processing
    for (p = fmt; *p; p++) {
        while (stream->stream_continues) stream = stream->stream_continues;
        switch (*p) {
            case '%':
                ⟨Ensure there is room to expand the escape sequence into 17⟩;
                ⟨Insert indentation if this is pending 16⟩;
                ⟨Recognise traditional printf escape sequences 13⟩; break;
            default: stream_putc(*p, stream); break;
        }
    }
    va_end(ap);                                          macro to end variable argument processing
}
```

§**13.** We don't trouble to check that correct `printf` escapes have been used: instead, we pass anything in the form of a percentage sign, followed by up to four nonalphabetic modifying characters, followed by an alphabetic category character for numerical printing, straight through to `sprintf` or `fprintf`.

Thus an escape like `%04d` is handled by the standard C library, but not `%s`, which we handle directly. That's for two reasons: first, we want to be careful to prevent overruns of memory streams; second, we need to ensure that the correct encoding is used when writing to disc. The numerical escapes involve only characters whose representation is the same in all our file encodings, but expanding `%s` does not.

⟨Recognise traditional printf escape sequences 13⟩ ≡
```
int ival; double dval; char *sval;
char format_string[8], category = ' ';
int i = 1;
format_string[0] = '%';
p++;
while (*p) {
    format_string[i++] = *p;
```

```
        if ((islower(*p)) || (isupper(*p)) || (*p == '%')) category = *p;
        p++;
        if ((category != ' ') || (i==6)) break;
    }
    format_string[i] = 0; p--;
    switch (category) {
        case 'c': case 'd': case 'i': case 'x':            char is promoted to int in variable arguments
            ival = va_arg(ap, int);
            if (stream->write_to_file) fprintf(stream->write_to_file, format_string, ival);
            if (stream->write_to_memory) {
                sprintf(stream->write_to_memory + stream->chars_written, format_string, ival);
                stream->chars_written += strlen(stream->write_to_memory + stream->chars_written);
            }
            break;
        case 'f':
            dval = va_arg(ap, double);
            if (stream->write_to_file) fprintf(stream->write_to_file, format_string, dval);
            if (stream->write_to_memory) {
                sprintf(stream->write_to_memory + stream->chars_written, format_string, dval);
                stream->chars_written += strlen(stream->write_to_memory + stream->chars_written);
            }
            break;
        case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
                stream_putc(*sval, stream);
            break;
        case '%': stream_putc('%', stream); break;
        default:
            fprintf(stderr, "*** Bad escape: <%s> ***\n", format_string);
            internal_error("Unknown % string escape in stream_printf");
    }
```

This code is used in §12.

§**14.** Our equivalent of `fputc` reads:

```
void stream_putc(int c_int, STREAM *stream) {
    unsigned int c;
    if (c_int >= 0) c = c_int; else c = c_int + 256;
    if (stream == NULL) return;
    if (c != '\n') ⟨Insert indentation if this is pending 16⟩;
    if (stream->use_xml_escapes) {
        switch(c) {
            case NEWLINE_IN_STRING: stream_literal(stream, "<br>"); return;
            case '&': stream_literal(stream, "&amp;"); return;
            case '<': stream_literal(stream, "&lt;"); return;
            case '>': stream_literal(stream, "&gt;"); return;
        }
    }
    while (stream->stream_continues) stream = stream->stream_continues;
    ⟨Ensure there is room to expand the escape sequence into 17⟩;
    if (stream->write_to_file) {
        switch(stream->encoding_to_write_to_file) {
            case UTF8_ENC: ⟨Put a UTF8-encoded character into the underlying file 15⟩; break;
```

```
            case ISO_ENC: fputc(c, stream->write_to_file); break;
            default: internal_error("stream has unknown text encoding");
        }
    } else if (stream->write_to_memory) {
        (stream->write_to_memory)[stream->chars_written] = c;
        (stream->write_to_memory)[stream->chars_written+1] = 0;
    }
    if (c == '\n') stream->indentation_pending = TRUE;
    stream->chars_written++;
}
```

§**15.** Where we pack large character values as follows. The following works only if $0 \leq c < 2048$, but since Inform uses only ISO Latin-1 strings internally, it only needs to work for $0 \leq c < 256$.

⟨Put a UTF8-encoded character into the underlying file 15⟩ ≡

```
    if (c >= 128) {
        fputc(0xC0 + (c >> 6), stream->write_to_file);
        fputc(0x80 + (c & 0x3f), stream->write_to_file);
    } else fputc(c, stream->write_to_file);
```

This code is used in §14.

§**16.**

⟨Insert indentation if this is pending 16⟩ ≡

```
    if (stream->indentation_pending) {
        stream->indentation_pending = FALSE;
        int i;
        for (i=0; i<stream->indentation_level; i++) {
            stream_putc(' ', stream); stream_putc(' ', stream);
            stream_putc(' ', stream); stream_putc(' ', stream);
        }
    }
```

This code is used in §12,14,12,14,12,14.

§**17.** The following is checked before any numerical `printf`-style escape is expanded into the stream, or before any single character is written. Thus we cannot overrun our buffers unless the expansion of a numerical escape exceeds `SPACE_AT_END_OF_STREAM` plus 1 in size. Since no outside influence gets to choose what formatting escapes we use (so that `%3000d`, say, can't occur), we can be pretty confident.

The interesting case occurs when we run out of memory in a memory stream. We make a continuation to a fresh `STREAM` structure, which points to twice as much memory as the original, allocated via `malloc`. (We will actually need a little more memory than that because we also have to make room for the `STREAM` structure itself.) We then set `stream` to the `continuation`. Given that `malloc` was successful – and it must have been or we would have stopped with a fatal error – the continuation is guaranteed to be large enough, since it's twice the size of the original, which itself was large enough to hold any escape sequence when opened.

⟨Ensure there is room to expand the escape sequence into 17⟩ ≡

```
    if (stream->chars_written + SPACE_AT_END_OF_STREAM >= stream->chars_capacity) {
        if (stream->write_to_file) return;                    write nothing further
        if (stream->write_to_memory) {
            size_t offset = 32 + 2*stream->chars_capacity;
            size_t needed = offset + sizeof(STREAM) + 32;
            void *further_allocation = I7_malloc(needed, STREAM_MREASON);
```

```
            if (further_allocation == NULL) fatal_error("Out of memory");
            STREAM *continuation = (STREAM *) (further_allocation + offset);
            stream_initialise(continuation);
            continuation->write_to_memory = further_allocation;
            continuation->chars_capacity = 2*stream->chars_capacity;
            (continuation->write_to_memory)[0] = 0;
            stream->stream_continues = continuation;
            stream = continuation;
        }
    }
```

This code is used in §12,14,12,14,12,14.

§**18.** Literal printing is just printing with XML escapes switched off:

```
void stream_literal(STREAM *stream, char *p) {
    if (stream == NULL) return;
    int i, x = stream->use_xml_escapes;
    stream->use_xml_escapes = FALSE;
    for (i=0; p[i]; i++) stream_putc((unsigned int) (p[i]), stream);
    stream->use_xml_escapes = x;
}
```

§**19.** Shifting indentation. For every indent there must be an equal and opposite outdent, but error conditions can cause some compilation routines to issue problem messages and then leave what they're doing incomplete, so we will be a little cautious about assuming that a mismatch means an error in Inform's program.

```
void stream_indent(STREAM *stream) {
    if (stream == NULL) return;
    stream->indentation_level++;
}
void stream_outdent(STREAM *stream) {
    if (stream == NULL) return;
    stream->indentation_level--;
    if ((stream->indentation_level < 0) && (problem_count == 0))
        internal_error("stream indentation negative");
}
```

§**20.** We can read the position for any stream, including `NULL`, but no matter how much is written to `NULL` this position never budges.

Because of continuations, this is not as simple as returning the `chars_written` field.

```
int stream_get_position(STREAM *stream) {
    int t = 0;
    while (stream) {
        t += stream->chars_written;
        stream = stream->stream_continues;
    }
    return t;
}
```

§**21. Memory-stream-only functions.** While it would be easy enough to implement this for file streams too, there's no point, since it is used only in concert with backspacing.

```
int stream_latest(STREAM *stream) {
    if (stream == NULL) return 0;
    if (stream->write_to_file) internal_error("stream_latest on file stream");
    while ((stream->stream_continues) && (stream->stream_continues->chars_written > 0))
        stream = stream->stream_continues;
    if (stream->write_to_memory) {
        if (stream->chars_written > 0)
            return (stream->write_to_memory)[stream->chars_written - 1];
    }
    return 0;
}
```

§**22.** Now for what is the trickiest function, because the position may be moved back so that later continuations fall away. This will very rarely happen, so we won't worry about the inefficiency of freeing up the memory saved by closing such continuation blocks (which would be inefficient if we immediately had to open similar ones again).

```
void stream_set_position(STREAM *stream, int position) {
    if (stream == NULL) return;
    if (position < 0) position = 0;                          to simplify the implementation of backspacing
    if (stream->write_to_file) internal_error("stream_set_position on file stream");
    if (stream->write_to_memory) {
        while (position > stream->chars_written) {
            position = position - stream->chars_written;
            stream = stream->stream_continues;
            if (stream == NULL) internal_error("can't set position forwards");
        }
        stream->chars_written = position;
        (stream->write_to_memory)[stream->chars_written] = 0;
        if (stream->stream_continues) {
            stream_close(stream->stream_continues);
            stream->stream_continues = NULL;
        }
    }
}
```

§**23.** We must return a valid C string which contains at least the first `STREAM_MIN_ACCESSIBLE_LENGTH` characters in the stream. The complete text of the stream may be cut up into a number of C strings because of continuations, but we know that the first has at least the length we need, because we don't allow memory streams to be opened with capacities smaller than that.

```
char *stream_get_text(STREAM *stream) {
    if (stream == NULL) return NULL;
    if (stream->write_to_file) internal_error("stream_get_text on file stream");
    return stream->write_to_memory;
}
```

§**24.** Lastly, our copying function, where `from` has to be a memory stream (or `NULL`) but `to` can be anything (including `NULL`).

```
void stream_copy(STREAM *to, STREAM *from) {
    if ((from == NULL) || (to == NULL)) return;
    if (from->write_to_file) internal_error("stream_copy from file stream");
    if (from->write_to_memory) {
        int i;
        for (i=0; i<from->chars_written; i++) {
            int c = (from->write_to_memory)[i];
            stream_putc(c, to);
        }
        if (from->stream_continues) stream_copy(to, from->stream_continues);
    }
}
```

Filenames 2/files

Purpose

Given the different environments in which we might be running, and the large range of files we may need from within a project folder, it's useful to have a section of code simply to deduce filenames.

2/files. ¶2-0 Filing system assumptions; §1 Location of extensions; §2-8 Project bundles; §9-11 Directory handling utilities; §12 Reading a line from a text file

---

Definitions

¶1. We begin with the leafnames of the main output files, which are the same on all platforms. The following list excludes the HTML index files: those have leafnames following their main headings (thus the Kinds index page is stored as `Kinds.html`, and so on).

```
define I6_OUTPUT_LEAFNAME "auto.inf"
define DEBUG_LOG_LEAFNAME "Debug log.txt"
define PROBLEM_LOG_LEAFNAME "Problems.html"
define XML_HEADINGS_LEAFNAME "Headings.xml"
define METADATA_LEAFNAME "Metadata.iFiction"
define UUID_LEAFNAME "uuid.txt"
define BLURB_LEAFNAME "Release.blurb"
define MANIFEST_LEAFNAME "manifest.plist"
define EPSMAP_LEAFNAME "Map.eps"
```

¶2. Filing system assumptions. It is assumed that our host filing system can manage at least 30-character filenames, that space is legal as a character in a filename, and that trailing extensions can be longer than 3 characters (in particular, that `.html` is allowed).

```
define MAX_FILENAME_LENGTH 522
```
Assuming no leafname exceeds 255 characters

---

§**1. Location of extensions.** NI looks for extensions in two places: a primary source, the "built in" set which comes as part of the resources of the Inform 7 application, and a secondary source, intended as a place where the designer can store downloaded extensions, new extensions being written, etc. Under Mac OS X, the secondary location is inside the user's Library, and to get its pathname we need to evaluate an environment variable which holds the current user's username. This has comparable results under generic Unix and Windows, but we take precautions against a null result anyway.

NI cannot know where the Inform 7 application is living in the filing system, nor where it keeps its built-in extensions (since this will depend on the platform being used). NI therefore expects to be told this information with the `-rules` command-line switch. The applications always call NI with `-rules` supplied, but in case NI is being invoked directly from the command line we do provide a default if `-rules` is omitted: which is simply to use the same library folder as for user-supplied extensions. See also the `main` routine, where this overriding is done.

```
char filename_of_tracefile[MAX_FILENAME_LENGTH];
char filename_of_telemetry[MAX_FILENAME_LENGTH];
char filename_of_options[MAX_FILENAME_LENGTH];
char filename_of_epsfile[MAX_FILENAME_LENGTH];
char pathname_of_extensions[MAX_FILENAME_LENGTH];
char pathname_of_templates[MAX_FILENAME_LENGTH];
```

```
char pathname_of_extension_docs[MAX_FILENAME_LENGTH];
char pathname_of_built_in_extensions[MAX_FILENAME_LENGTH];
void make_pathname_of_extensions(void) {
    char *home = (char *) (getenv("HOME"));
    if (home == NULL) home = "";
    strcpy(pathname_of_extensions, home); strcat(pathname_of_extensions, HOME_LIBRARY);
    strcpy(pathname_of_extension_docs, home); strcat(pathname_of_extension_docs, HOME_DOCS);
    strcpy(pathname_of_built_in_extensions, pathname_of_extensions);
    strcpy(filename_of_tracefile, home); strcat(filename_of_tracefile, TRACEFILE);
    strcpy(pathname_of_templates, home); strcat(pathname_of_templates, HOME_TEMPLATES);
    strcpy(filename_of_options, home); strcat(filename_of_options, OPTIONSFILE);
    strcpy(filename_of_telemetry, home); strcat(filename_of_telemetry, TELEMETRYFILE);
    int this_month = the_present->tm_mon + 1;
    int this_day = the_present->tm_mday;
    int this_year = the_present->tm_year + 1900;
    sprintf(filename_of_telemetry + strlen(filename_of_telemetry),
        " %04d-%02d-%02d.txt", this_year, this_month, this_day);
    set_platform_specific_eps_filename(filename_of_epsfile);
}
```

The function make_pathname_of_extensions is called from 14/main.

§**2. Project bundles.** For any given project, NI has only one fresh input file (the source text), but it generates a swathe of output in different formats, together with some temporary files along the way. It is convenient therefore to bundle all this material together – and essential if NI is being used with a port of the Inform 7 application, which needs to know where to put things and where to get them back again. If we are using this standard organisation, we say NI is running in "bundle mode"; it can also run less tidily as a command-line tool in which most of what it produces heaps up in the current working directory, but when NI is being used in the Inform 7 application, it will always be running in bundle mode.

The command line arguments to NI must specify either a bundle name, in which case `source_text_file` stays null, or an explicit source text filename, in which case `bundle_name` stays null: but not both.

```
char *bundle_name = NULL;
char *source_text_file = NULL;
```

§**3.** The layout of a bundle is as follows. Let us suppose the bundle is called, for want of a better name, Voltaire. (This does not have to be the same as the name of the project it makes: like any file, an Inform project can be renamed without changing its contents.) Then we have:

- `Voltaire.inform`
  - `Build`
    - `auto.inf` (the I6 code compiled by NI)
    - `Debug log.txt`
    - `gameinfo.dbg` (subsequently created by I6)
    - `Map.eps` (the EPS file created by NI)
    - `output.z5` (subsequently created by I6)
    - `Problems.html` (the report generated by NI)
    - `temporary file.inf` (used temporarily by NI)
    - `temporary file 2.inf` (used temporarily by NI)
  - `Index`
    - `Actions.html`
    - `Contents.html`
    - `Headings.xml` (not visible to the user)
    - `Kinds.html`
    - `Phrasebook.html`
    - `Rules.html`
    - `Scenes.html`
    - `World.html`
    - `Details` (a subfolder created as needed by NI)
      - `13_A.html` (and so forth: about 90 pages about actions)
  - `manifest.plist` (list of images and sounds used, made by NI)
  - `Metadata.iFiction` (iFiction record made by NI)
  - `Release.blurb` (releasing instructions for `cblorb`)
  - `Settings.plist` (used by the Inform application, not NI)
  - `Skein.skein` (used by the Inform application and `cblorb` but not NI)
  - `Source`
    - `story.ni` (the source text)
  - `uuid.txt` (UUID created by application and read by NI)

The application is free to store other resources in the bundle. (For instance, Inform for OS X also includes a `notes.rtf` file of rough notes made by the author: but NI does not notice this.)

§**4.** NI is not capable of creating a new project bundle: that is the responsibility of the Inform application, or, in extremis, the user. The following minimal structure must exist for NI to be able to work:

- `StEvremond.inform`
  - `Build` (empty subfolder)
  - `Index` (empty subfolder)
  - `Skein.skein` (only needs to exist for `cblorb` to release along with source)
  - `Source`
    - `story.ni` (the source text)
  - `uuid.txt` (the UUID)

It is essential that the application create a fresh UUID for each new project: see the Interactive Fiction ID section in chapter 10 below.

§**5.** Now for some dull but worthy routines to turn leafnames into filenames, splicing them together into a temporary buffer. (Only one of these routines can be used at a time, since otherwise the buffer is overwritten by one filename before the other has been finished with.)

```
char local_filename_buffer[MAX_FILENAME_LENGTH];
char *top_level_filename(char *leafname) {
    if (bundle_name == NULL) return leafname;                    outside bundle mode, use current directory
    sprintf(local_filename_buffer, "%s%c%s",
        bundle_name, FOLDER_SEPARATOR, leafname);
    return local_filename_buffer;
}
```

The function top_level_filename is called from 10/ifid and 10/bib.

§**6.** Filenames within the `Build` subfolder. The opening part may be a surprise: In extension census mode, NI is running not to compile something but to extract details of all the extensions installed. But it still needs somewhere to write its temporary and debugging files, and there is no project bundle to write into. To get round this, we use the user's installed extensions folder as if it were a project bundle.

```
char *build_filename(char *leafname) {
    if (census_mode) {
        if (verify_installed_extensions_tree() == FALSE) return leafname;
        sprintf(local_filename_buffer, "%s%c%s",
            pathname_of_extension_docs, FOLDER_SEPARATOR, leafname);
        return local_filename_buffer;
    }
    if (bundle_name == NULL) return leafname;
    sprintf(local_filename_buffer, "%s%cBuild%c%s",
        bundle_name, FOLDER_SEPARATOR, FOLDER_SEPARATOR, leafname);
    return local_filename_buffer;
}
```

The function build_filename is called from 2/prob3, 2/dl, 9/map, 10/bib and 14/i6t.

§**7.** Filenames within the `Source` subfolder: two versions here, as it is sometimes convenient to have the pathname from the top of the bundle, rather than an absolute pathname.

```
char *source_filename(char *leafname) {
    if (bundle_name == NULL) return leafname;
    sprintf(local_filename_buffer, "%s%cSource%c%s",
        bundle_name, FOLDER_SEPARATOR, FOLDER_SEPARATOR, leafname);
    return local_filename_buffer;
}
char *source_filename_relative_to_bundle(char *leafname) {
    if (bundle_name == NULL) return leafname;
    sprintf(local_filename_buffer, "Source%c%s",
        FOLDER_SEPARATOR, leafname);
    return local_filename_buffer;
}
```

The function source_filename_relative_to_bundle is called from 3/read.

§**8.** And filenames within the `Index` subfolder, where the further `Details` subfolder will be created if necessary. Filenames in `Details` have the form `N_S` where `N` is the integer supplied and `S` the leafname; for instance, `21_A.html` provides details page number 21 about actions, derived from the leafname `A.html`.

```
int details_subfolder_made = FALSE;
char *index_filename(char *leafname, int sub) {
    if (bundle_name == NULL) return leafname;
    sprintf(local_filename_buffer, "%s%cIndex%c",
        bundle_name, FOLDER_SEPARATOR, FOLDER_SEPARATOR);
    if (sub >= 0) {
        strcat(local_filename_buffer, "Details");
        if (details_subfolder_made == FALSE) {
            platform_specific_mkdir(local_filename_buffer);
            details_subfolder_made = TRUE;
        }
        sprintf(local_filename_buffer + strlen(local_filename_buffer),
            "%c%d_", FOLDER_SEPARATOR, sub);
    }
    sprintf(local_filename_buffer + strlen(local_filename_buffer), "%s", leafname);
    return local_filename_buffer;
}
```

The function index_filename is called from 2/index and 4/head.

§**9. Directory handling utilities.** The following routines handle directories, using routines defined in Platform-Specific Definitions.

The first creates a folder `new` at the given path in the Inform external materials area (e.g. the user's Library folder for OS X). This routine is always called in the right order, e.g., if Inform wants to make

```
Inform/Extensions/Frederick Bloggs
```

then it makes the following calls:

```
verify_library_folder(NULL, NULL, NULL, "Inform");
verify_library_folder("Inform", NULL, NULL, "Extensions");
verify_library_folder("Inform", "Extensions", NULL, "Frederick Bloggs");
```

```
int verify_library_folder(char *p1, char *p2, char *p3, char *new) {
    char path[MAX_FILENAME_LENGTH];
    char transcoded_path[MAX_FILENAME_LENGTH];
    char *home = (char *) (getenv("HOME"));
    int rv;
    path[0] = 0;
    if (home) sprintf(path, "%s%c", home, FOLDER_SEPARATOR);
    sprintf(path + strlen(path), "%s", INFORM_FOLDER_RELATIVE_TO_HOME);
    if (p1) sprintf(path + strlen(path), "%s%c", p1, FOLDER_SEPARATOR);
    if (p2) sprintf(path + strlen(path), "%s%c", p2, FOLDER_SEPARATOR);
    if (p3) sprintf(path + strlen(path), "%s%c", p3, FOLDER_SEPARATOR);
    if (new) sprintf(path + strlen(path), "%s", new);
    transcode_ISO_string_to_locale(path, transcoded_path);
    rv = platform_specific_mkdir(transcoded_path);
    if (rv == FALSE) LOG("verify_library_folder failed on path %s\ntranscoded %s\n", path, transcoded_path);
    return rv;
}
```

The function verify_library_folder is called from 4/edict and 4/edoc.

§**10.** In particular, here we make sure that we can definitely access the minimal library structure of directories for the "external area" which holds the user's installed extensions:

```
Inform
    Documentation
        Extensions
    Extensions
        Reserved
```

```
int verify_installed_extensions_tree(void) {
    if (verify_library_folder(NULL, NULL, NULL, "Inform") == 0) return FALSE;
    if (verify_library_folder("Inform", NULL, NULL, "Documentation") == 0) return FALSE;
    if (verify_library_folder("Inform", "Documentation", NULL, "Extensions") == 0) return FALSE;
    if (verify_library_folder("Inform", NULL, NULL, "Extensions") == 0) return FALSE;
    if (verify_library_folder("Inform", "Extensions", NULL, "Reserved") == 0) return FALSE;
    return TRUE;
}
```

The function verify_installed_extensions_tree is called from 4/excen and 4/edict.

§**11.** This routine writes the contents of the given folder to a plain text file, one item per line: e.g.

```
Emily Short/
Fried eggs.txt
Fred Quimby/
```

They can be written in any order. Any item which is itself a folder, rather than a file, should be terminated / (even if that is not the folder separator character on the current platform). The encoding of this file will be whatever the locale encoding is for filenames, something which will oblige us to take care later on.

```
int write_folder_contents_to_file(char *pathname, char *writeto) {
    STREAM CONTS_struct; STREAM *CONTS = &CONTS_struct;
    void *FOLDER = platform_specific_opendir(pathname);
    char leafname[MAX_FILENAME_LENGTH+1];
    if (FOLDER == NULL) return FALSE;
    if (STREAM_OPEN_TO_FILE(CONTS, writeto, ISO_ENC) == FALSE) return FALSE;
    while (platform_specific_readdir(FOLDER, pathname, leafname)) {
        if (leafname[0] == '.') continue;
        STREAM_WRITE(CONTS, "%s\n", leafname);
    }
    STREAM_CLOSE(CONTS);
    platform_specific_closedir(FOLDER);
    return TRUE;
}
```

The function write_folder_contents_to_file is called from 4/excen.

§**12. Reading a line from a text file.** The following utility might as well be given here as anywhere else: it's simply a form of `fgets` for reading a single line from an ISO Latin-1 text file up to a limited number of characters, regarding the first instance of any of the termination sequences `0A`, `0D`, `0A0D` or `0D0A` as a line ending.

```
int truncated_iso_fgets(FILE *F, char *buffer, int limit) {
    int len=0, c;
    if (buffer == NULL) return -1;
    if (feof(F)) return -1;
    while ((c = fgetc(F)) != EOF) {
        if (len >= limit) break;
        if (c == '\x0a') break;
        if (c == '\x0d') break;
        buffer[len++] = c;
    }
    buffer[len++] = 0;
    return len;
}
```

The function truncated_iso_fgets is called from 1/plat, 2/dl, 4/ext, 4/excen, 4/edict and 4/edoc.

Case-Insensitive Filenames 2/cifn

Purpose

On some of the Unix-derived file systems on which Inform runs, filenames are case-sensitive, so that FISH and fish might be different files. This makes extension files, installed by the user, prone to being missed. The code in this section provides a routine to carry out file opening as if filenames are case-insensitive, and is used only for extensions.

Definitions

¶1. This section contains a single utility routine, contributed by Adam Thornton: a specialised, case-insensitive form of `fopen()` called `ci_fopen()`. It is specialised in that it is designed for opening extensions, where the file path will be case-correct up to the last two components of the path (the leafname and the immediately containing directory), but where the casing may be wrong in those last two components.

¶2. If the exact filename or extension directory (case-correct) exists, `ci_fopen()` will choose it to open. If it does not, `ci_fopen` will use `strcasecmp()` to find a file or directory with the same name but differing in case and use it instead. If it finds exactly one candidate file, it will then attempt to `fopen()` it and return the result.

If `ci_fopen()` succeeds, it returns a `FILE *` (passed back to it from the underlying `fopen()`). If `ci_fopen()` fails, it returns `NULL`, and `errno` is set accordingly:

(a) If no suitable file was found, `errno` is set to `ENOENT`.
(b) If more than one possibility was found, but none of them exactly match the supplied case, `errno` is set to `EBADF`.
(c) Note that if multiple directories which match case-insensitively are found, but none is an exact match, `EBADF` will be set regardless of the contents of the directories.
(d) If `ci_fopen()` fails during its allocation of space to hold its intermediate strings for comparison, or for its various data structures, `errno` is set to `ENOMEM`.
(e) If an unambiguous filename is found but the `fopen()` fails, `errno` is left at whatever value the underlying `fopen()` set it to.

---

§**1. Use of POSIX.** The routine is available only on platforms where `POSIX_DIRECTORY_HANDLING` is defined (see "Platform-Specific Definitions"). In practice this means everywhere except Windows, but all Windows file systems are case-preserving and case-insensitive in any case.

```
#ifdef POSIX_DIRECTORY_HANDLING
```

§**2. The routine.** Briefly, we try to get the extension directory name right first, by looking for the given casing, then if that fails, for a unique alternative with different casing; and then repeat within that directory for the extension file itself.

```
FILE *ci_fopen(const char *path, const char *mode) {
    char *topdirpath = NULL, *ciextdirpath = NULL, *cistring = NULL, *ciextname = NULL;
    char *workstring = NULL, *workstring2 = NULL;
    DIR *topdir = NULL, *extdir = NULL; FILE *handle;
    size_t length;
    LOGIF(CI_FOPEN, "ci_fopen called on '%s' with mode '%s'\n", path, mode);
    for efficiency's sake, though it's logically equivalent, we try...
    handle = fopen(path, mode); if (handle) ⟨Happy ending to ci-fopen 6⟩;
    ⟨Find the length of the path, giving an error if it is empty or NULL 9⟩;
    ⟨Allocate memory for strings large enough to hold any subpath of the path 5⟩;
    ⟨Parse the path to break it into topdir path, extension directory and leafname 10⟩;
    topdir = opendir(topdirpath);                    whose pathname is assumed case-correct...
    if (topdir == NULL) ⟨Sad ending to ci-fopen 7⟩;          ...so that failure is fatal; errno is set by opendir
    sprintf(workstring, "%s%c%s", topdirpath, FOLDER_SEPARATOR, ciextdirpath);
    LOGIF(CI_FOPEN, "supplied dir = %s\n", workstring);
    extdir = opendir(workstring);                    try with supplied extension directory name
    if (extdir == NULL) ⟨Try to find a unique insensitively matching directory name in topdir 3⟩
    else strcpy(cistring, workstring);
    sprintf(workstring, "%s%c%s", cistring, FOLDER_SEPARATOR, ciextname);
    LOGIF(CI_FOPEN, "supplied name = %s\n", workstring);
    handle = fopen(workstring, mode);                try with supplied name
    if (handle) ⟨Happy ending to ci-fopen 6⟩;
    ⟨Try to find a unique insensitively matching entry in extdir 4⟩;
}
```

The function ci_fopen is called from 1/plat.

§**3. Looking for case-insensitive matches instead.** We emerge from the following only in the happy case where a unique matching directory name can be found.

⟨Try to find a unique insensitively matching directory name in topdir 3⟩ ≡

```
int rc = count_matches_within_directory(topdir, ciextdirpath, workstring);
switch (rc) {
    case 0:
        errno = ENOENT; ⟨Sad ending to ci-fopen 7⟩;
    case 1:
        sprintf(cistring, "%s%c%s", topdirpath, FOLDER_SEPARATOR, workstring);
        LOGIF(CI_FOPEN, "real dirname = %s\n", cistring);
        extdir = opendir(cistring);
        if (extdir == NULL) {
            LOGIF(CI_FOPEN, "%s not found\n", cistring);
            errno = ENOENT; ⟨Sad ending to ci-fopen 7⟩;
        }
        break;
    default:
        errno = EBADF; ⟨Sad ending to ci-fopen 7⟩;
}
```

This code is used in §2.

§**4.** More or less the same, but we never emerge at all: all cases of the switch return from the function.

⟨Try to find a unique insensitively matching entry in extdir 4⟩ ≡
```
int rc = count_matches_within_directory(extdir, ciextname, workstring);
switch (rc) {
    case 0:
        errno = ENOENT; ⟨Sad ending to ci-fopen 7⟩;
    case 1:
        sprintf(workstring2, "%s%c%s", cistring, FOLDER_SEPARATOR, workstring);
        workstring2[length] = 0;
        LOGIF(CI_FOPEN, "real filename = %s\n", workstring2);
        handle = fopen(workstring2, mode);
        if (handle) ⟨Happy ending to ci-fopen 6⟩;
        LOGIF(CI_FOPEN, "Couldn't open %s\n", workstring2);
        errno = ENOENT; ⟨Sad ending to ci-fopen 7⟩;
    default:
        errno = EBADF; ⟨Sad ending to ci-fopen 7⟩;
}
```

This code is used in §2.

§**5. Allocation and deallocation.** We use six strings to hold full or partial pathnames.

⟨Allocate memory for strings large enough to hold any subpath of the path 5⟩ ≡
```
workstring = calloc(length+1, sizeof(char));
if (workstring == NULL) { errno = ENOMEM; ⟨Sad ending to ci-fopen 7⟩; }
workstring2 = calloc(length+1, sizeof(char));
if (workstring2 == NULL) { errno = ENOMEM; ⟨Sad ending to ci-fopen 7⟩; }
topdirpath = calloc(length+1, sizeof(char));
if (topdirpath == NULL) { errno = ENOMEM; ⟨Sad ending to ci-fopen 7⟩; }
ciextdirpath = calloc(length+1, sizeof(char));
if (ciextdirpath == NULL) { errno = ENOMEM; ⟨Sad ending to ci-fopen 7⟩; }
cistring = calloc(length+1, sizeof(char));
if (cistring == NULL) { errno = ENOMEM; ⟨Sad ending to ci-fopen 7⟩; }
ciextname = calloc(length+1, sizeof(char));
if (ciextname == NULL) { errno = ENOMEM; ⟨Sad ending to ci-fopen 7⟩; }
```

This code is used in §2.

§**6.** If we are successful, we return a valid file handle...

⟨Happy ending to ci-fopen 6⟩ ≡
```
LOGIF(CI_FOPEN, "ci-fopen: succeeded\n");
⟨Prepare to exit ci-fopen cleanly 8⟩;
return handle;
```

This code is used in §2,4,2,4,2,4.

§**7.** ...and otherwise `NULL`, having already set `errno` with the reason why.

⟨Sad ending to ci-fopen 7⟩ ≡
```
LOGIF(CI_FOPEN, "ci-fopen: failed with errno = %d\n", errno);
⟨Prepare to exit ci-fopen cleanly 8⟩;
return NULL;
```

This code is used in §2,3,4,5,2,3,4,5,2,3,4,5.

§**8.**

⟨Prepare to exit ci-fopen cleanly 8⟩ ≡

```
if (workstring) free(workstring);
if (workstring2) free(workstring2);
if (topdirpath) free(topdirpath);
if (ciextdirpath) free(ciextdirpath);
if (cistring) free(cistring);
if (ciextname) free(ciextname);
if (topdir) closedir(topdir);
if (extdir) closedir(extdir);
```

This code is used in §6,7,6,7,6,7.

§9. Pathname hacking.

⟨Find the length of the path, giving an error if it is empty or NULL 9⟩ ≡

```
length = 0;
if (path) length = strlen(path);
if (length < 1) { errno = ENOENT; return NULL; }
```

This code is used in §2.

§**10.** And here we break up a pathname like

`/Users/bobama/Library/Inform/Extensions/Hillary Clinton/Health Care.i7x`

into three components:

`topdirpath` is `/Users/bobama/Library/Inform/Extensions`, and its casing is correct
`ciextdirpath` is `Hillary Clinton`, but its casing may not be correct
`ciextname` is `Health Care.i7x`, but its casing may not be correct

The contents of `workstring` are not significant afterwards.

⟨Parse the path to break it into topdir path, extension directory and leafname 10⟩ ≡

```
char *p;
size_t extdirindex, extindex, namelen, dirlen;

p = strrchr(path, FOLDER_SEPARATOR);
extindex = (size_t) (p - path);
namelen = length - extindex - 1;
strncpy(ciextname, path + extindex + 1, namelen);
ciextname[namelen] = 0;

strncpy(workstring, path, (extindex-1));
workstring[extindex-1] = 0;
p = strrchr(workstring, FOLDER_SEPARATOR);
extdirindex = (size_t) (p - workstring);
strncpy(topdirpath, path, extdirindex);
topdirpath[extdirindex] = 0;

dirlen = extindex - extdirindex - 1;
strncpy(ciextdirpath, path + extdirindex + 1, dirlen);
ciextdirpath[dirlen] = 0;

LOGIF(CI_FOPEN, "topdirpath = %s\n", topdirpath);
LOGIF(CI_FOPEN, "ciextdirpath = %s\n", ciextdirpath);
LOGIF(CI_FOPEN, "ciextname = %s\n", ciextname);
```

This code is used in §2.

§**11. Counting matches.** We count the number of names within the directory which case-insensitively match against `name`, and copy the last which matches into `last_match`. This must be at least as long as `name`. (We ought to be just a little careful in case of improbable cases where the matched name contains a different number of characters from `name`, for instance because on a strict reading of Unicode "SS" is casing-equivalent to the eszet, but it's unlikely that many contemporary implementations of `strcasecmp` are aware of this, and in any case the code above contains much larger buffers than needed.)

```
int count_matches_within_directory(void *vd, char *name, char *last_match) {
    DIR *d = (DIR *) vd;
    struct dirent *dirp;
    int rc = 0;
    last_match[0] = 0;
    while ((dirp = readdir(d)) != NULL) {
        LOGIF(CI_FOPEN, "testing: name = %s; entry = %s\n", name, dirp->d_name);
        if (strcasecmp(name, dirp->d_name) == 0) {
            LOGIF(CI_FOPEN, "accepted as match %d\n", rc);
            rc++;
            strcpy(last_match, dirp->d_name);
        }
    }
    LOGIF(CI_FOPEN, "count_matches_within_directory rc = %d\n", rc);
    return rc;
}
```

§**12. Non-POSIX tail.** On platforms without POSIX directory handling, we revert to regular `fopen`.

```
#else
FILE *ci_fopen(const char *path, const char *mode) {
    return fopen(path, mode);
}
#endif
```

The function ci_fopen is called from 1/plat.

Image Dimensions 2/image

Purpose

These utility routines, contributed by Toby Nelson, look at the headers of JPEG and PNG files to find the pixel dimensions of any images supplied by the user for cover art and figures.

§**1.** Image and sound files are the only binary files read by NI: we begin with a set of utilities to read them. To begin with, integers of 8, 16, 32 and 64 bit widths respectively, arranged with most significant byte (MSB) first.

```
int read_int8(FILE *binary_file, unsigned int *result) {
    int c1 = getc(binary_file);
    if (c1 == EOF) return FALSE;
    *result = (unsigned int) c1;
    return TRUE;
}
int read_int16(FILE *binary_file, unsigned int *result) {
    int c1, c2;
    c1 = getc(binary_file);
    c2 = getc(binary_file);
    if (c1 == EOF || c2 == EOF) return FALSE;
    *result = (((unsigned int) c1) << 8) + ((unsigned int) c2);
    return TRUE;
}
int read_int32(FILE *binary_file, unsigned int *result) {
    int c1, c2, c3, c4;
    c1 = getc(binary_file);
    c2 = getc(binary_file);
    c3 = getc(binary_file);
    c4 = getc(binary_file);
    if (c1 == EOF || c2 == EOF || c3 == EOF || c4 == EOF) return FALSE;
    *result = (((unsigned int) c1) << 24) +
                (((unsigned int) c2) << 16) +
                (((unsigned int) c3) << 8) + ((unsigned int) c4);
    return TRUE;
}
int read_int64(FILE *binary_file, unsigned long long *result) {
    int c1, c2, c3, c4, c5, c6, c7, c8;
    c1 = getc(binary_file);
    c2 = getc(binary_file);
    c3 = getc(binary_file);
    c4 = getc(binary_file);
    c5 = getc(binary_file);
    c6 = getc(binary_file);
    c7 = getc(binary_file);
    c8 = getc(binary_file);
    if (c1 == EOF || c2 == EOF || c3 == EOF || c4 == EOF || c5 == EOF
```

```
        || c6 == EOF || c7 == EOF || c8 == EOF) return FALSE;
    *result = (((unsigned long long) c1) << 56) +
            (((unsigned long long) c2) << 48) +
            (((unsigned long long) c3) << 40) +
            (((unsigned long long) c4) << 32) +
            (((unsigned long long) c5) << 24) +
            (((unsigned long long) c6) << 16) +
            (((unsigned long long) c7) << 8) +
                ((unsigned long long) c8);
    return TRUE;
}
```

The function read_int8 is called from 2/sounds.
The function read_int16 is called from 2/sounds.
The function read_int32 is called from 2/sounds.
The function read_int64 is called from 2/sounds.

§**2.** We will sometimes need to toggle between MSB and LSB representation of integers 32 or 64 bits wide:

```
void swap_bytes32(unsigned int *value) {
    unsigned int result = (((*value & 0xff) << 24) +
                           ((*value & 0xff00) << 8) +
                           ((*value & 0xff0000) >> 8) +
                           ((*value & 0xff000000) >> 24 ) );
    *value = result;
}
void swap_bytes64(unsigned long long *value) {
    unsigned long long result = (((*value & 0xff) << 56) +
                                 ((*value & 0xff00) << 40) +
                                 ((*value & 0xff0000) << 24) +
                                 ((*value & 0xff000000) << 8) +
                                 ((*value >> 8)  & 0xff000000) +
                                 ((*value >> 24) & 0xff0000) +
                                 ((*value >> 40) & 0xff00) +
                                 ((*value >> 56) & 0xff) );
    *value = result;
}
```

The function swap_bytes32 is called from 2/sounds.
The function swap_bytes64 is called from 2/sounds.

§**3.** Some file formats also have variable-sized integers, as a sequence of bytes (most significant first) in which each byte consists of seven bits of data plus a most significant bit which marks that a continuation byte follows:

```
int read_variable_length_integer(FILE *binary_file, unsigned int *result) {
    int c;
    *result = 0;
    do {
        c = getc(binary_file);
        if (c == EOF) return FALSE;
        *result = (*result << 7) + (((unsigned char) c) & 0x7F);
    } while  (((unsigned char) c) & 0x80);
    return TRUE;
}
```

The function read_variable_length_integer is called from 2/sounds.

§**4.** Here we read just the mantissa of a particular representation of floating-point numbers:

```
int read_float80(FILE *binary_file, unsigned int *result) {
    int c1, c2, exp;
    unsigned int prev = 0, mantissa;
    c1 = getc(binary_file);
    c2 = getc(binary_file);
    if (c1 == EOF || c2 == EOF) return FALSE;
    if (!read_int32(binary_file, &mantissa)) return FALSE;
    exp = 30 - c2;
    while  (exp--) {
        prev = mantissa;
        mantissa >>= 1;
    }
    if (prev & 1) mantissa++;
    *result = (unsigned int) mantissa;
    return TRUE;
}
```

The function read_float80 is called from 2/sounds.

§**5.** And lastly we read a string of a supplied length from the file, and then null terminate it to make it valid C string. (`string` must therefore be at least `length` plus 1 bytes long.)

```
int read_string(FILE *binary_file, char *string, unsigned int length) {
    if (length > 0) {
        if (fread(string, 1, length, binary_file) != length) return FALSE;
    }
    string[length] = 0;
    return TRUE;
}
```

The function read_string is called from 2/sounds.

§**6. JPEG files.** The following either finds the pixel width and height of a given JPEG file and returns `TRUE` or, if it can't read the file or doesn't recognise the header as having JPEG format, returns `FALSE`.

JPEG is properly speaking not a file format but a compression technique: the routine below works with either JIF (JPEG Interchange Format) or its simpler cousin JFIF (JPEG File Interchange Format).

We scan the file looking for "markers", each of which begins with an `0xFF` byte and is followed by a marker-type byte which is neither `0x00` nor `0xFF`. The compulsory marker SOI must appear at the start of the file, providing one way to detect probable JPEGs by looking at the first two bytes. There must also eventually be a start of frame marker, for the actual image: this can have many forms, but in all cases tells us the height and width.

```
int get_jpg_dimensions(FILE *JPEG_file, unsigned int *width, unsigned int *height) {
    unsigned int sig, length;
    int marker;
    if (!read_int16(JPEG_file, &sig)) return FALSE;
    if (sig != 0xFFD8) return FALSE;                                    0xFF (marker) then 0xD8 (SOI)
    do {
        do {
            marker = getc(JPEG_file);
            if (marker == EOF) return FALSE;
        } while (marker != 0xff);                                       skip to next 0xFF byte
        do {
            marker = getc(JPEG_file);
        } while (marker == 0xff);                                       skip to next non FF byte
        if (!read_int16(JPEG_file, &length)) return FALSE;              length of marker
        switch(marker) {
            all variant forms of "start of frame": e.g., 0xC0 is a baseline DCT image
            case 0xc0:
            case 0xc1: case 0xc2: case 0xc3:
            case 0xc5: case 0xc6: case 0xc7:
            case 0xc9: case 0xca: case 0xcb:
            case 0xcd: case 0xce: case 0xcf: {
                fortunately these markers all then open with the same format
                if (getc(JPEG_file) == EOF) return FALSE;               skip 1 byte of data precision
                if (!read_int16(JPEG_file, height)) return FALSE;
                if (!read_int16(JPEG_file, width)) return FALSE;
                return TRUE;
            }
            default:
                if (fseek(JPEG_file, length - 2, SEEK_CUR) != 0) return FALSE;      skip rest of marker
        }
    }
    while (marker != EOF);
    return FALSE;
}
```

The function get_jpg_dimensions is called from 10/bib and 10/fig.

§**7. PNG files.** The PNG file must start with a signature which indicates that the remainder contains a single PNG image, consisting of a series of chunks beginning with an IHDR chunk and ending with an IEND chunk (*Portable Network Graphics (PNG) Specification*, 2nd edition, section 5.2). We only need to scan the IHDR chunk, of which the pixel width and height are the first two words (section 11.2.2).

```
int get_png_dimensions(FILE *PNG_file, unsigned int *width, unsigned int *height) {
    unsigned int sig1, sig2, length, type;
    Check PNG signature
    if (!read_int32(PNG_file, &sig1)) return FALSE;
    if (!read_int32(PNG_file, &sig2)) return FALSE;
    if ((sig1 != 0x89504e47) || (sig2 != 0x0d0a1a0a)) return FALSE;
    Read first chunk
    if (!read_int32(PNG_file, &length)) return FALSE;
    if (!read_int32(PNG_file, &type)) return FALSE;
    First chunk must be IHDR
    if (type != 0x49484452) return FALSE;
    Width and height follow
    if (!read_int32(PNG_file, width)) return FALSE;
    if (!read_int32(PNG_file, height)) return FALSE;
    return TRUE;
}
```

The function get_png_dimensions is called from 10/bib and 10/fig.

Sound Durations 2/sounds

Purpose

These utility routines, again by Toby Nelson, look at the headers of AIFF and OGG files to find the durations, and verify that they are what they purport to be.

§**1.** To explicate the following, see the specifications for AIFF and OGG headers. Durations are measured in centiseconds.

§2. AIFF files.

```
int get_aiff_duration(FILE *pFile, unsigned int *pDuration,
    unsigned int *pBitsPerSecond, unsigned int *pChannels, unsigned int *pSampleRate) {
    unsigned int sig;
    unsigned int chunkID;
    unsigned int chunkLength;
    unsigned int numSampleFrames;
    unsigned int sampleSize;
    if (!read_int32(pFile, &sig)) return FALSE;
    if (sig != 0x464F524D) return FALSE;                          "FORM" indicating an IFF file
    if (!read_int32(pFile, &sig)) return FALSE;
    if (!read_int32(pFile, &sig)) return FALSE;
    if (sig != 0x41494646) return FALSE;                          "AIFF" indicating an AIFF file
    Read chunks, skipping over those we are not interested in
    while (TRUE) {
        if (!read_int32(pFile, &chunkID)) return FALSE;
        if (!read_int32(pFile, &chunkLength)) return FALSE;
        if (chunkID == 0x434F4D4D) {                              "COMM" indicates common AIFF data
            if (chunkLength < 18) return FALSE;                     Check we have enough data to read
            if (!read_int16(pFile, pChannels))          return FALSE;
            if (!read_int32(pFile, &numSampleFrames))  return FALSE;
            if (!read_int16(pFile, &sampleSize))       return FALSE;
            if (!read_float80(pFile, pSampleRate))      return FALSE;
            if (*pSampleRate == 0) return FALSE;                  Sanity check to avoid a divide by zero
            Result is in centiseconds
            *pDuration = ((unsigned long long) numSampleFrames * 100) / *pSampleRate;
            *pBitsPerSecond = *pSampleRate * *pChannels * sampleSize;
            return TRUE;
        } else {
            Skip unwanted chunk
            if (fseek(pFile, chunkLength, SEEK_CUR) != 0) return FALSE;
        }
    }
    *pDuration = 0;
    return FALSE;
}
```

The function get_aiff_duration is called from 10/sfx.

§3. OGG Vorbis files.

```
int get_ogg_duration(FILE *pFile, unsigned int *pDuration,
    unsigned int *pBitsPerSecond, unsigned int *pChannels, unsigned int *pSampleRate) {
    unsigned int sig;
    unsigned int version;
    unsigned int numSegments;
    unsigned int packetType;
    unsigned int vorbisSig1;
    unsigned int vorbisSig2;
    unsigned int seekPos;
    unsigned int fileLength, bytesToRead, lastSig, index;
    unsigned long long granulePosition;
    unsigned char buffer[256];
    if (!read_int32(pFile, &sig)) return FALSE;
    if (sig != 0x4F676753) return FALSE;                          "OggS" indicating an OGG file
    Check OGG version is zero
    if (!read_int8(pFile, &version)) return FALSE;
    if (version != 0) return FALSE;
    Skip header type, granule position, serial number, page sequence and CRC
    if (fseek(pFile, 21, SEEK_CUR) != 0) return FALSE;
    Read number of page segments
    if (!read_int8(pFile, &numSegments)) return FALSE;
    Skip segment table
    if (fseek(pFile, numSegments, SEEK_CUR) != 0) return FALSE;
    Vorbis Identification header
    if (!read_int8(pFile, &packetType)) return FALSE;
    if (packetType != 1) return FALSE;
    if (!read_int32(pFile, &vorbisSig1)) return FALSE;
    if (vorbisSig1 != 0x766F7262) return FALSE;                                  "VORB"
    if (!read_int16(pFile, &vorbisSig2)) return FALSE;
    if (vorbisSig2 != 0x6973) return FALSE;                                       "IS"
    Check Vorbis version is zero
    if (!read_int32(pFile, &version)) return FALSE;
    if (version != 0) return FALSE;
    Read number of channels
    if (!read_int8(pFile, pChannels)) return FALSE;
    Read sample rate
    if (!read_int32(pFile, pSampleRate)) return FALSE;
    swap_bytes32(pSampleRate);                                    Ogg Vorbis uses LSB first
    Skip bitrate maximum
    if (fseek(pFile, 4, SEEK_CUR) != 0) return FALSE;
    Read Nominal Bitrate
    if (!read_int32(pFile, pBitsPerSecond)) return FALSE;
    swap_bytes32(pBitsPerSecond);                                 Ogg Vorbis uses LSB first
    Encoders can be unhelpful and give no bitrate in the header
    if (pBitsPerSecond == 0) return FALSE;
    Search for the final Ogg page (near the end of the file) to read duration,
    i.e., read the last 4K of the file and look for the final "OggS" sig
    if (fseek(pFile, 0, SEEK_END) != 0) return FALSE;
```

```
    fileLength = ftell(pFile);
    seekPos = fileLength - 4096;
    if (seekPos < 0) seekPos = 0;
    lastSig = -1;
    while (seekPos < fileLength) {
        if (fseek(pFile, seekPos, SEEK_SET) != 0) return FALSE;
        bytesToRead = fileLength - seekPos;
        if (bytesToRead > 256) bytesToRead = 256;
        if (fread(buffer, 1, bytesToRead, pFile) != bytesToRead) return FALSE;
        for(index = 0; index < bytesToRead; index++) {
            if ((buffer[index] == 0x4F) &&
                (buffer[index + 1] == 0x67) &&
                (buffer[index + 2] == 0x67) &&
                (buffer[index + 3] == 0x53)) {
                lastSig = seekPos + index;
            }
        }
        Next place to read from is 256 bytes further on, but to catch
        sigs that span between these blocks, read the last four bytes again
        seekPos += 256 - 4;
    }
    if (lastSig == -1) return FALSE;
    if (fseek(pFile, lastSig, SEEK_SET) != 0) return FALSE;
    if (!read_int32(pFile, &sig)) return FALSE;
    if (sig != 0x4F676753) return FALSE;                    "OggS" indicating an OGG file
    Check OGG version is zero
    if (!read_int8(pFile, &version)) return FALSE;
    if (version != 0) return FALSE;
    Skip header Type
    if (fseek(pFile, 1, SEEK_CUR) != 0) return FALSE;
    if (!read_int64(pFile, &granulePosition)) return FALSE;
    swap_bytes64(&granulePosition);
    *pDuration = (unsigned int) ((granulePosition * 100) /
                    (unsigned long long) *pSampleRate);
    return TRUE;
}
```

The function get_ogg_duration is called from 10/sfx.

§4. MIDI files. At one time it was proposed that Inform 7 should allow a third sound file format: MIDI. This provoked considerable debate in July 2007 and enough doubts were raised that the implementation below was never in fact officially used. It is preserved here in case we ever revive the issue.

Inform is not really able to decide this for itself, in any case, since it can only usefully provide sound files which the virtual machines it compiles for will allow. At present, the Glulx virtual machine does not officially support MIDI, which makes the question moot.

```
int get_midi_file_information(FILE *pFile, unsigned int *pType,
    unsigned int *pNumTracks) {
    unsigned int sig;
    unsigned int length;
    unsigned int pulses;
    unsigned int frames_per_second;
    unsigned int subframes_per_frame;
    unsigned int clocks_per_second;
    unsigned int start_of_chunk_data;
    unsigned int status;
    unsigned int clocks;
    unsigned int sysex_length;
    unsigned int non_midi_event_length;
    unsigned int start_of_non_midi_data;
    unsigned int non_midi_event;
    if (!read_int32(pFile, &sig)) return FALSE;
    "RIFF" indicating a RIFF file
    if (sig == 0x52494646) {
        Skip the filesize and typeID
        if (fseek(pFile, 8, SEEK_CUR) != 0) return FALSE;
        now read the real MIDI sig
        if (!read_int32(pFile, &sig)) return FALSE;
    }
    "MThd" indicating a MIDI file
    if (sig != 0x4D546864) return FALSE;
    Read length of chunk
    if (!read_int32(pFile, &length)) return FALSE;
    Make sure we have enough data to read
    if (length < 6) return FALSE;
    Read the MIDI type: 0,1 or 2
    0 means one track containing up to 16 channels to make a single tune
    1 means one or more tracks, commonly each with a single channel, making up a single tune
    2 means one or more tracks, where each is a separate tune in it's own right
    if (!read_int16(pFile, pType)) return FALSE;
    Read the number of tracks
    if (!read_int16(pFile, pNumTracks)) return FALSE;
    Read "Pulses Per Quarter Note" (PPQN)
    if (!read_int16(pFile, &pulses)) return FALSE;
    if top bit set, then number of subframes per second can be deduced
    if (pulses >= 0x8000) {
        First byte is a negative number for the frames per second
        Second byte is the number of subframes in each frame
        frames_per_second    = (256 - (pulses & 0xff));
        subframes_per_frame  = (pulses >> 8);
```

```
    clocks_per_second    = frames_per_second * subframes_per_frame;
    LOG("frames_per_second   = %d\n",   frames_per_second);
    LOG("subframes_per_frame = %d\n", subframes_per_frame);
    LOG("clocks_per_second   = %d\n",   clocks_per_second);
    Number of pulses per quarter note unknown
    pulses = 0;
} else {
    unknown values
    frames_per_second    = 0;
    subframes_per_frame  = 0;
    clocks_per_second    = 0;
    LOG("pulses per quarter note = %d\n",   pulses);
}
Skip any remaining bytes in the MThd chunk
if (fseek(pFile, length - 6, SEEK_CUR) != 0) return FALSE;
Keep reading chunks, looking for "MTrk"
do {
    Read chunk signature and length
    if (!read_int32(pFile, &sig)) {
        if (feof(pFile)) return TRUE;
        return FALSE;
    }
    if (!read_int32(pFile, &length)) return FALSE;
    start_of_chunk_data = ftell(pFile);
    if (sig == 0x4D54726B) {                                                    "MTrk"
        LOG("track starts\n");
        Read each event, looking for information before the real tune starts, e.g., tempo
        do {
            Read the number of clocks since the previous event
            if (!read_variable_length_integer(pFile, &clocks))
                return FALSE;
            We bail out when the track starts
            if (clocks > 0) break;
            Read the MIDI Status byte
            if (!read_int8(pFile, &status)) return FALSE;
            Start or continuation of system exclusive data
            if ((status == 0xF0) || (status == 0xF7)) {
                Read length of system exclusive event data
                if (!read_variable_length_integer(pFile, &sysex_length)) return FALSE;
                Skip sysex event
                if (fseek(pFile, sysex_length, SEEK_CUR) != 0) return FALSE;
            } else if (status == 0xFF) {                                    Non-MIDI event
                Read the Non-MIDI event type and length
                if (!read_int8(pFile, &non_midi_event)) return FALSE;
                if (!read_variable_length_integer(pFile, &non_midi_event_length))
                    return FALSE;
                start_of_non_midi_data = ftell(pFile);
                switch(non_midi_event) {
                    case 0x01:                                              Comment text
                    case 0x02:                                              Copyright text
                    case 0x03:                                              Track name
```

```
                        case 0x04: {                                        Instrument name
                            char text[257];
                            if (!read_string(pFile, text, non_midi_event_length))
                                return FALSE;
                            LOG("%d: %s\n", non_midi_event, text);
                            break;
                        }
                        case 0x51:                                          Tempo change
                        case 0x58:                                          Time signature
                        case 0x59:                                          Key signature
                            break;
                    }
                    Skip non-midi event
                    if (fseek(pFile,
                        start_of_non_midi_data + non_midi_event_length, SEEK_SET) != 0)
                        return FALSE;
                } else {
                    Real MIDI data found: we've read all we can so bail out at this point
                    break;
                }
            }
            while (TRUE);
        }
        Seek to start of next chunk
        if (fseek(pFile, start_of_chunk_data + length, SEEK_SET) != 0) return FALSE;
        Reached end of file
        if (feof(pFile)) return TRUE;
        Did we try to seek beyond the end of the file?
        if (ftell(pFile) < (start_of_chunk_data + length)) return TRUE;
    }
    while (TRUE);
    return TRUE;
}
```

The function get_midi_file_information is called from 10/sfx.

Make Inform 6 Names 2/isn

Purpose

To construct valid Inform 6 identifiers, dictionary word constants and double-quoted string constants as needed.

2/isn.§1 Automatically composed identifiers; §2-5 Explicitly translated identifiers; §6 I6 dictionary words; §7-8 I6 quoted text

---

§**1. Automatically composed identifiers.** A legal I6 identifier is a sequence of 1 to 31 characters, which must be alphanumeric or else underscores, except that the leading character must not be a 0.

For the sake of legibility of the I6 generated by NI, we adapt natural language descriptions to this format as best we can, in a standardised pattern which uses an identifying letter (e.g., A for Action), a unique ID number (preventing I6 name-clashes) and then a truncated alphanumeric-safe form of the words used in the textual description, if any. For example, an object called "apple crumble" might have I6 identifier `O100_apple_crumble`. Any other object also called "apple crumble" would have a different identifier since the number parts would be different.

Beginning with the identifying letter ensures that we do not open with a 0 digit.

We truncate to 28 characters in length so that other routines can concatenate our identifier with up to 3 further characters, if they choose.

```
void isn_compose_identifier(char *ledger,
    char nature_character, int id_number, int w1, int w2) {
    int j;
    char identifier[64];
    sprintf(identifier, "%c%d", nature_character, id_number);
    if ((w1>=0) && (w2>=w1)) {
        for (j=w1; j<=w2; j++) {
            identifier is at this point 32 chars or fewer in length: add at most 30 more
            if (strlen(lw_array[j].lw_text) > 30) sprintf(identifier + strlen(identifier), " etc");
            else sprintf(identifier + strlen(identifier), " %s", lw_array[j].lw_text);
            if (strlen(identifier) > 32) break;
        }
    }
    identifier[28] = 0;                          it was at worst 62 chars in size, but is now truncated to 28
    for (j=0; identifier[j]; j++) {
        int x = identifier[j];
        if (!(((x >= '0') && (x <= '9')) ||
            ((x >= 'a') && (x <= 'z')) || ((x >= 'A') && (x <= 'Z')) || (x == '_')))
            identifier[j] = '_';
    }
    strcpy(ledger, identifier);
}
```

The function isn_compose_identifier is called from 7/data, 9/qty, 9/prop, 9/model, 10/tab, 10/eqns, 10/exf, 11/act, 11/av and 12/rb.

§**2. Explicitly translated identifiers.** I7 provides the ability for the user to specify exactly what identifier name to use as the I6 image of something, overriding the automatically composed name above, in some cases. The following routine parses, tidies up and acts on "... translates into I6 as ..." sentences; it gives their sentence nodes an annotation marking what sort of thing is being translated.

```
define INVALID_I6TR 0
define PROPERTY_I6TR 1                                  "The open property translates into I6 as "open"."
define WORLD_OBJECT_I6TR 2                              "The north object translates into I6 as "n_obj"."
define RULE_I6TR 3                                      "The baffling rule translates into I6 as "BAFFLING_R"."
define VARIABLE_I6TR 4                                  "The sludge count variable translates into I6 as "sldgc".
define ACTION_I6TR 5                                    "The taking action translates into I6 as "Take".
define GRAMMAR_TOKEN_I6TR 6          "The grammar token "[whatever]" translates into I6 as "WHATEVER".
```

```
sentence_handler TRANSLATES_SH_handler =
    { SENTENCE_NT, TRANSLATES_VB, 0, translates_into_I6_as };

void translates_into_I6_as(parse_node *pn) {
    parse_node *p1 = pn->down->next;
    parse_node *p2 = pn->down->next->next;
    int category = INVALID_I6TR, w1, w2;
    if (traverse == 1) {
        pn_annotate_int(pn, category_of_I6_translation_ANNOT, INVALID_I6TR);
        ⟨Ensure that we are translating to a quoted I6 identifier 4⟩;
        [[w1, w2 <-- p1]];
        [[w1, w2 == the ... --> w1, w2]]; p1->word_ref1 = w1;
        if [[w1, w2 == ... property]] { p1->word_ref2--; category = PROPERTY_I6TR; }
        else if [[w1, w2 == ... object/kind]] { p1->word_ref2--; category = WORLD_OBJECT_I6TR; }
        else if [[w1, w2 == ... rule]] { category = RULE_I6TR; }
        else if [[w1, w2 == ... variable]] { p1->word_ref2--; category = VARIABLE_I6TR; }
        else if [[w1, w2 == ... action]] { p1->word_ref2--; category = ACTION_I6TR; }
        else if [[w1, w2 == understand token ...]] { p1->word_ref1 += 2; category = GRAMMAR_TOKEN_I6TR;
}
        else {
            sentence_problem(_P_(C2TranslatedUnknownCategory),
                "that isn't one of the things which can be translated to I6",
                "and should be '... property', '... object', '... kind', '... rule', or "
                "'... action'. For instance, 'The yourself object translates into "
                "I6 as \"selfobj\".'");
            return;
        }
        pn_annotate_int(pn, category_of_I6_translation_ANNOT, category);
    } else category = pn_int_annotation(pn, category_of_I6_translation_ANNOT);
    ⟨Take immediate action on the translation where possible 3⟩;
}
```

§**3.** In some cases, we might as well act now; but in others we will act later, traversing the parse tree to look for translation sentences of the right sort.

⟨Take immediate action on the translation where possible 3⟩ ≡
```
switch(category) {
    case PROPERTY_I6TR:
        property_translates(pn);
        pn_annotate_int(pn, category_of_I6_translation_ANNOT, INVALID_I6TR); break;
    case WORLD_OBJECT_I6TR: break;
    case RULE_I6TR: if (traverse == 2) handle_library_rule(pn); break;
    case VARIABLE_I6TR: if (traverse == 2) qty_translates(pn); break;
    case ACTION_I6TR: if (traverse == 2) an_translates(pn); break;
    case GRAMMAR_TOKEN_I6TR: if (traverse == 2) gv_translates(pn); break;
}
```

This code is used in §2.

§**4.** The word "as" is still loose here because "translates into I6 as" was too much for the sentence-breaker to swallow, so it broke at "translates into I6" and we therefore have to detect the "as" here ourselves.

⟨Ensure that we are translating to a quoted I6 identifier 4⟩ ≡
```
int valid = TRUE, w1, w2;
[[w1, w2 <-- p2]];
if ((![[w1, w2 == as ###]]) || (vocab_test_flags(w2, TEXT_MC) == 0)) valid = FALSE;
if (valid) ⟨Dequote it and see if it's valid 5⟩;
if (valid == FALSE) {
    sentence_problem(_P_(C2TranslatedToNonIdentifier),
        "Inform 7 constructions can only translate into quoted I6 identifiers",
        "which must be strings of 1 to 31 characters drawn from 1, 2, ..., 9, "
        "a or A, b or B, ..., z or Z, or underscore '_', except that the "
        "first character is not allowed to be a digit.");
    return;
}
```

This code is used in §2.

§**5.** If it turns out not to be, we simply set `valid` to false.

⟨Dequote it and see if it's valid 5⟩ ≡
```
int i;
char *p;
p2->word_ref1 = w2;
dequote_word(w2);
p = lw_array[w2].lw_text;
if ((strlen(p) == 0) || (strlen(p) > 31)) valid = FALSE;
for (i=0; p[i]; i++)
    if ((isdigit(p[i]) == 0) && (isalpha(p[i]) == 0) && (p[i] != '_'))
        valid = FALSE;
if (isdigit(p[0])) valid = FALSE;
```

This code is used in §4.

§6. I6 dictionary words. We now take an ISO Latin-1 string and compile an I6 dictionary word constant to lodge the same text into the virtual machine's parsing dictionary.

A legal I6 dictionary word can take several forms: it can be in single quotes, `'thus'`, but only if it is more than one character long, since `'t'` would be the character value of lower-case T instead. (Or it can be double-quoted `"so"`, but only in grammar or properties; this usage is deprecated and we avoid it.) Within the dictionary word, `^` is an escape character meaning a literal single quote, and the notation `@{xx}` is an escape meaning the character with hexadecimal value `xx`.

Optionally, a dictionary word can end with a pair of slashes and then, optionally again, markers to indicate that the word is (for instance) a plural: thus `'newts//p'`. Using no markers, as in `'toads//'`, makes a word equivalent to that without a marker, but avoids the single-letter problem – so the preferred modern way to write a single-character I6 dictionary word is `'t//'`, and this is what the following routine does. (Note the exceptional case where the word consists only of a `'/'`: here we cannot write `'///'` because I6 reads this as `//` plus an invalid marker `/`, and throws an error. We escape the single `/` to avoid this. In all other cases there's no need to escape a `/`.)

Dictionary words with a literal `~` in are, as it happens, not parsable by the Z-machine, but the code below – employing the `@{7E}` escape – is in principle legal, and it does work on Glulx.

```
void isn_compile_dictionary_word(OUTPUT_STREAM, char *p, int pluralise) {
    int c;
    WRITE("'");
    for (c=0; p[c] != 0; c++) {
        switch(p[c]) {
            case '/': if (p[1] == 0) WRITE("@{2F}"); else WRITE("/"); break;
            case '\'': WRITE("^"); break;
            case '^': WRITE("@{5E}"); break;
            case '~': WRITE("@{7E}"); break;
            case '@': WRITE("@{40}"); break;
            default: WRITE("%c", p[c]);
        }
    }
    if (pluralise) WRITE("//p");
    else if (strlen(p) == 1) WRITE("//");
    WRITE("'");
}
```

The function isn_compile_dictionary_word is called from 5/litp, 9/cot, 13/gv, 13/gl, 13/gtok, 13/gprv and 13/gpr.

§7. I6 quoted text. We now take an ISO Latin-1 string and compile a double-quoted I6 string constant which will print out the same content, or initialise a string array.

A subtly different set of escape characters are used here. It's all trickier than it might be because of some unfortunate design choices in I6. For instance, an `@` character can't be escaped as `@{40}` because that would be expanded too early in the string-reading process, so that it would still be read as an escape-character `@`, not a literal one. We therefore have to use `@@64`. This unfortunately goes wrong, however, when the immediately following character is a decimal digit, because then we might construct, e.g., `@@647`, and I6 will throw an error: no valid character has ZSCII code 647. We therefore use the `@{...}` escape to represent any digit following an `@@` escape. Since no digit is itself an escape character with side-effects, a vicious circle is avoided.

Initialisation of I6 string arrays has different conventions again: these behave more like dictionary words and the `@@` escape is not allowed.

define `ISN_CAPITALISE 1` *capitalise first letter of text*
define `ISN_EXPAND_APOSTROPHES 2` *sometimes regard ' as "*
define `ISN_RECOGNISE_APOSTROPHE_SUBSTITUTION 4` *recognise ['] as a literal '*

define ISN_DEQUOTE 8 *ignore initial and terminal " pair, e.g., render* `"fish"` *as* `fish`
define ISN_FOR_ARRAY 16 *force use of* `@{xx}` *form not* `@@ddd`
define ISN_FLATTEN_NEWLINES 32 *convert any* `'\n'` *to a simple space*

```
void isn_compile_string(OUTPUT_STREAM, char *p, int options) {
    int i, from = 0, to = strlen(p), esc_digit = FALSE;
    if ((options & ISN_DEQUOTE) && (p[0] == '"') && (p[to-1] == '"')) {
        from++; to--;
    }
    for (i=from; i<to; i++) {
        switch(p[i]) {
            case '\n':
                if (options & ISN_FLATTEN_NEWLINES) WRITE(" "); else WRITE("\n");
                break;
            case NEWLINE_IN_STRING: WRITE("^"); break;
            case '"': WRITE("~"); break;
            case '@':
                if (options & ISN_FOR_ARRAY) WRITE("@{40}");
                else { WRITE("@@64"); esc_digit = TRUE; continue; }
                break;
            case '^':
                if (options & ISN_FOR_ARRAY) WRITE("@{5E}");
                else { WRITE("@@94"); esc_digit = TRUE; continue; }
                break;
            case '~':
                if (options & ISN_FOR_ARRAY) WRITE("@{7E}");
                else { WRITE("@@126"); esc_digit = TRUE; continue; }
                break;
            case '\\': WRITE("@{5C}"); break;
            case '\'':
                if (options & ISN_EXPAND_APOSTROPHES)
                    ⟨Apply Inform 7's convention on interpreting single quotation marks 8⟩
                else WRITE("'");
                break;
            case '[':
                if ((options & ISN_RECOGNISE_APOSTROPHE_SUBSTITUTION) &&
                    (p[i+1] == '\'') && (p[i+2] == ']')) { i += 2; WRITE("'"); }
                else WRITE("[");
                break;
            default:
                if ((i==0) && (options & ISN_CAPITALISE))
                    WRITE("%c", toupper(p[i]));
                else if ((esc_digit) && (isdigit(p[i])))
                    WRITE("@{%02x}", p[i]);
                else
                    WRITE("%c", p[i]);
                break;
        }
        esc_digit = FALSE;
    }
}
```

The function isn_compile_string is called from 5/litp, 9/rsdt, 9/cot, 10/str, 12/cph and 13/test.

§**8.** This is where Inform's convention on expanding single quotation marks to double, provided they appear to be quoting text rather than used as apostrophes in contractions such as "don't", is implemented. Note the exceptional case.

⟨Apply Inform 7's convention on interpreting single quotation marks 8⟩ ≡

```
if ((i==from) && (p[i+1] == 's') && ((to == 3) || (p[i+2] == ' ')))
    WRITE("'");                                  allow apostrophe if appending e.g. "'s nose" to "Jane"
else if ((i>0) && (p[i+1]) && (isalpha(p[i-1])) && (isalpha(p[i+1])))
    WRITE("'");                                  allow apostrophe sandwiched between two letters
else WRITE("~");                                 and otherwise convert to double-quote
```

This code is used in §7.

Purpose

I7 is has no Dijkstra-like conscience about compiling code which is full of I6 `jump` statements, and these require labels to jump to. This section provides those labels, and other related unique-ID-number counters.

Template interpreter commands

```
4   {-callv:compile_allocated_counter_storage}
```

Definitions

¶1. For clarity we give each label in the whole I6 program its own unique name (even though this is not strictly necessary since labels have only local scope to their routines), and this means allowing for sets of labels with a unique ID number providing guaranteed-previously-unused new labels in every set.

So: each set of labels is identified with a name, and the labels written take the form `L_NameNumber`. For instance, `L_Marble17` is the 18th label in namespace `Marble`. Every label namespace's name must differ from every other. It is legal for a namespace's name to be the empty string, which generates labels `L_0`, `L_1`, ...

```
define MAX_NAMESPACE_PREFIX_LENGTH 20          when L_ and a number are added, we are within 31 chars

typedef struct label_namespace {
    char label_prefix[MAX_NAMESPACE_PREFIX_LENGTH + 1];
    int label_counter;                                  next free ID number for this label namespace
    int allocate_storage;                          number of words of memory to reserve for each label
    MEMORY_MANAGEMENT
} label_namespace;
```

The structure label_namespace is private to this section.

---

§**1.** The creator for new label namespaces. Note that, by default, a label namespace reserves no memory.

```
label_namespace *lns_new(char *name) {
    label_namespace *lns;
    int i;
    if (strlen(name) > MAX_NAMESPACE_PREFIX_LENGTH)
        sentence_problem(_P_(C2LabelNamespaceTooLong),
            "a label namespace prefix is too long",
            "and should be shortened to a few alphabetic characters.");
    lns = CREATE(label_namespace);
    for (i=0; (name[i]) && (i<MAX_NAMESPACE_PREFIX_LENGTH); i++)
        lns->label_prefix[i] = name[i];
    lns->label_prefix[i] = 0;
    lns->label_counter = 0;
    lns->allocate_storage = 0;
    return lns;
}
```

§**2.** The rest of NI tends not to store pointers to namespaces: instead it must access them by searching on the name. This is inefficient, but there are few namespaces and it happens fairly seldom, so there is no point in optimising.

```
label_namespace *lns_by_prefix(char *name) {
    label_namespace *lns;
    LOOP_OVER(lns, label_namespace)
        if (strcmp(name, lns->label_prefix) == 0)
            return lns;
    return NULL;
}
label_namespace *lns_read_or_create(char *name) {
    label_namespace *lns = lns_by_prefix(name);
    if (lns == NULL) lns = lns_new(name);
    return lns;
}
```

§**3.** The rest of NI is allowed only to call for a label in a given namespace, advancing the counter or not as it pleases; or to call for the current counter value.

```
void lns_zero_counter(char *namespace) {
    label_namespace *lns = lns_read_or_create(namespace);
    lns->label_counter = 0;
}
int lns_read_counter(char *namespace, int advance_flag) {
    label_namespace *lns = lns_read_or_create(namespace);
    int c = lns->label_counter;
    if (advance_flag) lns->label_counter++;
    return c;
}
void lns_write(char *namespace, OUTPUT_STREAM, int advance_flag) {
    label_namespace *lns = lns_read_or_create(namespace);
    WRITE("L_%s%d", lns->label_prefix, lns->label_counter);
    if (advance_flag) lns->label_counter++;
}
```

The function lns_zero_counter is called from 12/cinv.

The function lns_read_counter is called from 12/cinv and 14/i6t.

The function lns_write is called from 12/cinv.

§**4.** It is possible to mark a namespace as requiring 1 or more words of storage. If so, the namespace `Whatsit` makes a word array called `I7_ST_Whatsit` which contains enough words for each label actually allocated to have that many words of storage. (And we add 2 words, to provide a safety margin, and because in the event of a namespace for which no labels are created, I6 would otherwise throw an error at being asked to make an array with the specification `--> 0`.)

```
void lns_allocate_storage(char *namespace, int multiplier) {
    label_namespace *lns = lns_read_or_create(namespace);
    lns->allocate_storage = multiplier;
}
void compile_allocated_counter_storage(OUTPUT_STREAM) {
    label_namespace *lns;
    LOOP_OVER(lns, label_namespace)
        if (lns->allocate_storage > 0)
            WRITE("Array I7_ST_%s --> %d;\n",
                lns->label_prefix, (lns->allocate_storage)*(lns->label_counter)+2);
}
```

The function lns_allocate_storage is called from 12/cinv.

The function compile_allocated_counter_storage is invoked by a command in a `.i6t` template file.

HTML Files 2/html

Purpose

To provide utilities for writing HTML files such as the problems report, the extension documentation, the index files and so forth.

---

§**1.** Inform documentation – its HTML text and the images, etc., used within it – is stored in two areas: "built-in" and "external". The built-in area is expected to be within the Inform 7 application itself. For instance, on OS X, this is at:

`...wherever.../Inform.app/Contents/Resources/` and/or

`...wherever.../Inform.app/Contents/Resources/English.lproj/`

(The duplication is a complication to do with localisation which we can ignore here.) The material stored in this built-in area is fixed: the Inform application needs to work even if stored on a read-only disc, or where the user has insufficient permissions to alter it. NI itself neither reads from, nor writes to, any file in the built-in documentation area.

Documentation for the installed extensions does, however, change: it is written by NI as and when necessary. This is the material making up the "external" area, and it needs to be somewhere which the user certainly has the necessary permissions to write to. For instance:

`~/Library/Inform/Documentation/` (OS X)

`My Documents\Inform\Documentation\` (Windows)

Pages in these two areas, built-in and external, need to link to each other by `<a href=...>` links: in addition, pages in the external area need access to images stored in the built-in area.

The other HTML files written by NI are stored within the relevant project's bundle: these are the report of Problems (if any) and the Index. They, too, need access to images stored in the built-in area.

The problem we face is that these three mini-websites – the built-in documentation, the external documentation, and the project-specific pages – are written by tools which cannot know the correct file URLs. (For instance, it would not even help for the application to tell NI where the built-in area is: because the HTML written by NI would then cease to work if the user moved the application elsewhere in the filing system after NI had run.)

§**2. The "inform:" URL scheme.** We solve this by requiring that the Inform 7 application must support a new URL scheme.

(a) `<inform://...>` is interpreted as a file in the built-in documentation area, except that

(b) `<inform://Extensions/...whatever...>` should be fetched by first checking for "...whatever..." in the external area, and then – if that fails – also checking for "...whatever..." in the `ExtnDocs` subfolder of the built-in area.

For instance, Inform 7 for OS X would look for `inform://Extensions/magic.png` at the following locations:

(i) `~/Library/Inform/Documentation/magic.png`

(ii) `.../Inform.app/Contents/Resources/ExtnDocs/magic.png`

If no file was found in either place, the link should simply do nothing: the application is required not to produce a 404 error page, or to blank out the page currently showing.

§3. The "source:" URL scheme. The other non-standard Inform URL scheme is "source:", which is used for a link which, when clicked, opens the Source panel with the given line made visible.

For instance, line 21 of file `Bits and Pieces/marbles.txt` has URL

`source:Bits and Pieces/marbles.txt#line14`

Filenames are given relative to the current project bundle. However, if only a leafname is supplied, then this is read as a file within the `Source` subfolder of the project bundle. (Thus it is not possible to have a source link to a source file at the root of the project bundle: but this is no loss, since source is not allowed to be kept there.) For instance, line 14 of file `Source/story.ni` has URL

`source:story.ni#line14`

The following routine writes the clickable source-reference icon, and is the only place in NI where "source:" is used.

Source which is generated internally to NI cannot be opened in the Source panel, for obvious reasons, so we produce nothing if the location is internal.

```
void html_source_link(OUTPUT_STREAM, source_location sl) {
    if (sl.file_of_origin) {
        char *filename = sf_get_filename(sl.file_of_origin);
        if (bundle_name && (strncmp(filename, bundle_name, strlen(bundle_name)) == 0))
            filename += strlen(bundle_name) + 1;
        if ((strncmp(filename, "Source", 6)==0) && (filename[6]==FOLDER_SEPARATOR))
            filename += 7;
        WRITE(" <a href=\"source:%s#line%d\"><img border=0 src=inform:/Reveal.%s></a>",
            filename, sl.line_number, ICON_EXT);
    }
}
```

The function html_source_link is called from 2/index.

§4. Icons with and without tooltips. Tooltips are the evanescent pop-up windows which appear, a little behind the mouse arrow, when it is poised waiting over the icon. (We make heavy use of these in the World index, for instance, to clarify what abbreviations mean.)

```
void html_icon_with_tooltip(OUTPUT_STREAM, char *icon_name, char *tip, char *tip2) {
    WRITE("<img border=0 src=inform:/doc_images/%s ", icon_name);
    if (tip) {
        WRITE("title=\"%s", tip); if (tip2) WRITE(" %s", tip2); WRITE("\"");
    }
    WRITE(">");
}
```

The function html_icon_with_tooltip is called from 12/br.

§**5. Outcome images.** These are the two images used on the Problems page to visually indicate success or failure. We also use special images on special occasions.

The date of Easter depends on who and where you are. Inform's notional home is England. Following League of Nations advice in 1926, Easter is legally celebrated in England on the Sunday after the second Saturday in April, but a let-out clause allowed the churches to continue the old methods as an interim measure. The interim shows no sign of ending, so we have to turn to the church, and this is where it becomes complicated. There are five main algorithms ordained by major Christian churches: Catholic, continental European Protestant, Church of England, Eastern and Russian Orthodox. The first three always agree on the date, but usually disagree with the last two. The two eastern algorithms only disagree with each other once or twice a century, but the usual result has been riots with significant loss of life.

We will ignore the clumsy *Book of Common Prayer* method adopted by the Church of England as a political manoeuvre during the reign of George II and instead use the algorithm of J.-M. Oudin (*Étude sur la date de Pâques*, Bulletin astronomique, 1940), in a form now published by the US Naval Observatory: Oudin corrected a small mistake in the calculation by Gauss (1800) of the *Allgemeiner Reichskalender* (1776) which reconciled Lutheran Easter with Gregorian, which in turn followed the calendrical reforms of Clavius et al. (1582), which in turn... and so on. For a bewilderingly involved account of the classical history, see Leofranc Holford-Strevens, *The History of Time* (Oxford, 2005).

In principle we calculate the first Sunday after the first ecclesiastical moon that occurs on or after March 21. An "ecclesiastical moon" is one as seen from a longitude near Rome, but the ratios used to adjust lunar and solar calendars are not quite right. The result is also tampered with to stop Easter from coinciding with the pagan anniversary of the founding of Rome (for the convenience of people living in the Vatican) and also to stop it from coinciding with the Jewish Passover (a change motivated purely by anti-Semitism, playing down that the event commemorated the arrest of a Jew at the time of Passover). They botched this tampering, too.

Knuth remarks that calculating Easter was almost the only algorithmic research in the West for many centuries. Nevertheless the result is practically a random-number generator. The one thing to be said in its favour is that it can be computed accurately with integer arithmetic using fairly low numbers, and this we now do.

```
void html_outcome_image(OUTPUT_STREAM, char *image) {
    char *vn = "";
    int c, y, k, i, n, j, l, m, d;
    int this_month = the_present->tm_mon + 1;
    int this_day = the_present->tm_mday;
    int this_year = the_present->tm_year + 1900;

    y = this_year;

    c = y/100;
    n = y-19*(y/19);
    k = (c-17)/25;
    i = c-c/4-(c-k)/3+19*n+15;
    i = i-30*(i/30);
    i = i-(i/28)*(1-(i/28)*(29/(i+1))*((21-n)/11));
    j = y+y/4+i+2-c+c/4;
    j = j-7*(j/7);
    l = i-j;
    m = 3+(l+40)/44;
    d = l+28-31*(m/4);

    if ((this_month == m) && (this_day >= d-2) && (this_day <= d+1))
        vn = "_3";                                   that is, Good Friday to Easter Monday
    if ((this_year == 2018) && (this_month == 3) && (this_day >= 30))
        vn = "_3";                                   Easter Sunday falls on 1 April in 2018
```

```
    if ((this_month == 12) && (this_day >= 25))
        vn = "_2";                                    that is, Christmas Day to New Year's Eve
    WRITE("<p><center><img src=inform:/%s%s.png border=0></center><p>", image, vn);
}
```

The function html_outcome_image is called from 2/prob2.

§**6. Header and footer.** Cascading style sheets provide indentation levels of 0, 1, 2, 3 or 4 tabs deep, the tab stops being 25 pixels apart, and hanging indents. Rather than providing a `.css` file, we define this small amount of CSS in the HTML header.

The default font varies from platform to platform: for instance, on OS X we use the Lucida font.

```
void html_header(OUTPUT_STREAM, char *title, char *thumbnail, char *caption) {
    WRITE("<html><head><meta http-equiv=\"content-type\" ");
    WRITE("content=\"text/html; charset=UTF-8\">\n"); INDENT;
    WRITE("<style type=\"text/css\">\n");
    WRITE("<!--\n"); INDENT;
    WRITE("p.hang {\n"); INDENT;
    WRITE("padding-left: 25px;\n");
    WRITE("text-indent: -25px;\n");
    WRITE("margin-top: 0px;\n");
    WRITE("margin-bottom: 0px;\n");
    OUTDENT; WRITE("}\n");
    int margin;
    for (margin = 1; margin<=9; margin++) {
        WRITE("p.in%d {\n", margin); INDENT;
        WRITE("padding-left: %dpx;\n", (margin-1)*25);
        OUTDENT; WRITE("}\n");
        WRITE("p.tightin%d {\n", margin); INDENT;
        WRITE("padding-left: %dpx;\n", (margin-1)*25);
        WRITE("margin-top: 2px;\n");
        WRITE("margin-bottom: 2px;\n");
        OUTDENT; WRITE("}\n");
        WRITE("p.hangingin%d {\n", margin); INDENT;
        WRITE("padding-left: %dpx;\n", (margin+1)*25);
        WRITE("text-indent: -50px;\n");
        WRITE("margin-top: 0px;\n");
        WRITE("margin-bottom: 1px;\n");
        OUTDENT; WRITE("}\n");
    }
    WRITE("div.hr {\n"); INDENT;
    WRITE("border: 0;\n");
    WRITE("width: 100%%;\n");
    WRITE("color: #707070;\n");
    WRITE("background-color: #707070;\n");
    WRITE("height: 5px;\n");
    OUTDENT; WRITE("}\n");
    OUTDENT; WRITE("-->\n");
    OUTDENT; WRITE("</style>\n");
    WRITE("</head>\n");
    WRITE("<body>");
    if (JAVASCRIPT_MODEL == 1) {
        WRITE("<script language=\"JavaScript\">");
```

```
        WRITE("function pasteCode(code) { ");
        WRITE("var myProject = window.Project; ");
        WRITE("myProject.selectView('source'); ");
        WRITE("myProject.pasteCode(code); ");
        WRITE("}");
        WRITE("</script>\n");
    }
    WRITE("<font %s>", DEFAULT_HTML_FONT);
    if (thumbnail) {
        open_coloured_box(OUT, "e0e0e0");
        begin_html_table(OUT, NULL, TRUE, 0, 0, 0, 0, 0);
        first_html_column(OUT, 72);
        WRITE("<img border=1 src=inform:/doc_images/%s title=\"%s\"> ", thumbnail, caption);
        next_html_column(OUT, 0);
    }
    if (title[0]=='<') {                                    used for a details page beneath the page titled thus
        WRITE("<h3><b>%s</b>: Detail view  "
            "<a href=../%s.html><font color=\"#002060\"><i>Back to full view</i></font></a></h3><p>",
            title+1, title+1);
    } else WRITE("<h3><b>%s</b></h3>", title);
}
void html_header_complete(OUTPUT_STREAM, char *title, char *thumbnail) {
    if (thumbnail) {
        end_html_row(OUT); end_html_table(OUT);
        close_coloured_box(OUT, "e0e0e0");
        WRITE("</p><p>");
    }
}
void html_footer(OUTPUT_STREAM) {
    WRITE("</font></body></html>");
}
```

The function html_header is called from 2/index and 2/prob3.
The function html_header_complete is called from 2/index.
The function html_footer is called from 2/index and 2/dl.

§7. HTML paragraphs with indentation.

```
void open_html_paragraph(OUTPUT_STREAM, int depth, char *class) {
    int margin = depth;
    if (margin < 1) internal_error("minimal HTML indentation is 1");
    if (margin > 9) margin = 9;
    WRITE("<p class=\"%sin%d\">", class, margin);
    while (depth > 9) { depth--; WRITE("    "); }
}
```

The function open_html_paragraph is called from 2/index, 2/prob3, 4/head, 9/inpw, 9/tmap, 11/ina, 11/av, 12/br, 12/rb and 13/gl.

§**8.** A little testing routine, not otherwise used.

```
void test_html_paragraphs(void) {
    int k;
    char *text = "Jackdaws love my big sphinx of quartz, "
        "jinxed wizards pluck ivy from my quilt.";
    for (k=1; k<4; k++) {
        open_html_paragraph(ifl, k, ""); INDEX(text); INDEX("</p>");
    }
    for (k=1; k<4; k++) {
        open_html_paragraph(ifl, k, "tight"); INDEX(text); INDEX("</p>");
    }
    for (k=1; k<4; k++) {
        open_html_paragraph(ifl, k, "hanging"); INDEX(text); INDEX("</p>");
    }
}
```

§**9. Writing HTML characters.** The following routine is a low-level filter which takes ISO Latin-1 characters one at a time, feeding them out to the given stream with any unsafe characters converted to suitable HTML elements. (The stream writer will transcode to UTF-8 encoding, since all HTML file streams written by Inform are declared as having the UTF-8 character encoding.)

A special escape sequence, marked by starting and finishing with a character which otherwise can never occur, is expanded to a source code link. If we used an asterisk to denote this, then a source reference is fed into `html_char_out` as the following stream of characters:

```
*source text*Source/story.ni*14*
```

When we notice the trigger character, we cease to output HTML and instead buffer up the reference until we reach the terminating trigger character: we then parse a little, tidy up and send it to `html_source_link` to be turned into a `source:` link.

But of course we don't use an asterisk as trigger – we use character `F0`. Arguably this is dodgy, since the character is legal in ISO Latin-1. But it is a lower-case Icelandic eth, which is not allowed in unquoted Inform 7 source text (because of being absent from the ZSCII character set).

```
define SOURCE_REF_CHAR '\xf0'
define FORCE_NEW_PARA_CHAR '\xd0'
define MAX_SOURCE_REF_LENGTH (MAX_FILENAME_LENGTH + 20)

char source_ref_paraphrase[MAX_SOURCE_REF_LENGTH];                    field 1
char source_ref_filename[MAX_SOURCE_REF_LENGTH];                      field 2
char source_ref_line[MAX_SOURCE_REF_LENGTH];                          field 3
int source_ref_field = 0;                                  which field we are buffering
int source_ref_marker = -1;                    write position in buffer, or -1 if not buffering

void html_char_out(OUTPUT_STREAM, char c) {
    int charcode = (int) ((unsigned char) c);
    if (source_ref_field >= 1) {
        char *buffer = NULL;
        switch (source_ref_field) {
            case 1: buffer = source_ref_paraphrase; break;
            case 2: buffer = source_ref_filename; break;
            case 3: buffer = source_ref_line; break;
        }
        if (source_ref_marker >= MAX_SOURCE_REF_LENGTH)
            internal_error("filenames too long in paths to source text");
```

```
        if (c == SOURCE_REF_CHAR) {
            buffer[source_ref_marker] = 0;
            source_ref_marker = -1;
        } else {
            buffer[source_ref_marker++] = c;
            return;
        }
    }
    switch(c) {
        case '"': WRITE("&quot;"); return;
        case '<': WRITE("&lt;"); return;
        case '>': WRITE("&gt;"); return;
        case '&': WRITE("&amp;"); break;
        case NEWLINE_IN_STRING: WRITE("<br>"); return;
        case FORCE_NEW_PARA_CHAR: WRITE("</p><p class=\"in2\">");
            html_icon_with_tooltip(OUT, "ornament_flower.png", NULL, NULL);
            WRITE(" "); return;
        case SOURCE_REF_CHAR:
            source_ref_field++; source_ref_marker = 0;
            if (source_ref_field == 4) {
                source_ref_field = 0; source_ref_marker = -1;
                source_location sl;
                sl.file_of_origin = filename_to_source_file(source_ref_filename);
                sl.line_number = atoi(source_ref_line);
                html_source_link(OUT, sl);
            }
            return;
        default:
            PUT(charcode);
            return;
    }
}
```

The function html_char_out is called from 2/hdoc, 2/prob1, 5/litp and 9/map.

§**10. Writing HTML tables.** Opening a generic bland table with reasonable column spacing:

```
void begin_plain_html_table(OUTPUT_STREAM) {
    begin_html_table(OUT, NULL, FALSE, 0, 0, 0, 0, 0);
}
```

The function begin_plain_html_table is called from 2/hdoc, 4/vm, 5/rel, 7/dim, 7/kix, 9/map and 10/tab.

§**11.** And now the general table-writing code used throughout Inform, with only a couple of exceptions where fiddly spacing effects are being used (such as on the World index map).

```
void begin_html_table(OUTPUT_STREAM, char *colour, int full_width,
    int border, int cellspacing, int cellpadding, int height, int width) {
    WRITE("<table border=\"%d\" cellspacing=\"%d\" cellpadding=\"%d\"",
        border, cellspacing, cellpadding);
    if (colour) WRITE(" bgcolor=\"%s\"", colour);
    if (full_width) WRITE(" width=100%%");
    if (width > 0) WRITE(" width=\"%d\"", width);
    if (height > 0) WRITE(" height=\"%d\"", height);
    WRITE(">");
}
void first_html_column(OUTPUT_STREAM, int width) {
    WRITE("<tr><td align=\"left\" valign=\"top\"");
    if (width > 0) WRITE(" width=\"%d\"", width);
    WRITE("><font %s>", DEFAULT_HTML_FONT);
}
void next_html_column(OUTPUT_STREAM, int width) {
    WRITE("    </font></td>"
        "<td align=\"left\" valign=\"top\"");
    if (width > 0) WRITE(" width=\"%d\"", width);
    WRITE("><font %s>", DEFAULT_HTML_FONT);
}
void next_html_column_right_justified(OUTPUT_STREAM, int width) {
    WRITE("</font></td><td align=\"right\" valign=\"top\"");
    if (width > 0) WRITE(" width=\"%d\"", width);
    WRITE("><font %s>", DEFAULT_HTML_FONT);
}
void end_html_row(OUTPUT_STREAM) {
    WRITE("</font></td></tr>");
}
void end_html_table(OUTPUT_STREAM) {
    WRITE("</table>");
}
```

The function begin_html_table is called from 2/hdoc, 2/lexi, 9/map, 10/bib, 10/fig, 10/sfx, 10/exf, 11/ina and 12/rb.

The function first_html_column is called from 2/hdoc, 2/lexi, 4/vm, 5/rel, 7/dim, 7/kix, 9/map, 10/tab, 10/bib, 10/fig, 10/sfx, 10/exf, 11/ina and 12/rb.

The function next_html_column is called from 2/hdoc, 2/lexi, 4/vm, 5/rel, 7/dim, 7/kix, 9/map, 10/tab, 10/bib, 10/fig, 10/sfx, 10/exf and 11/ina.

The function next_html_column_right_justified is called from 12/rb.

The function end_html_row is called from 2/hdoc, 2/lexi, 4/vm, 5/rel, 7/dim, 7/kix, 9/map, 10/tab, 10/bib, 10/fig, 10/sfx, 10/exf, 11/ina and 12/rb.

The function end_html_table is called from 2/hdoc, 2/lexi, 4/vm, 5/rel, 7/dim, 7/kix, 9/map, 10/tab, 10/bib, 10/fig, 10/sfx, 10/exf, 11/ina and 12/rb.

§**12.** For boxes with rounded corners:

define CORNER_SIZE 8 *measured in pixels*

```
void open_coloured_box(OUTPUT_STREAM, char *html_colour) {
    WRITE("<table width=\"100%%\" cellpadding=\"0\" cellspacing=\"0\" border=\"0\" ");
    WRITE("style=\"background-color: #%s\">", html_colour);
    WRITE("<tr><td width=\"%d\">", CORNER_SIZE);
    box_corner(OUT, html_colour, "tl");
    WRITE("</td><td></td>");
    WRITE("<td width=\"%d\">", CORNER_SIZE);
    box_corner(OUT, html_colour, "tr");
    WRITE("</td></tr>", html_colour);
    WRITE("<tr><td width=\"%d\"></td><td>", CORNER_SIZE);
    WRITE("<font %s>", DEFAULT_HTML_FONT);
}
void close_coloured_box(OUTPUT_STREAM, char *html_colour) {
    WRITE("</font></td><td width=\"%d\"></td></tr>", CORNER_SIZE);
    WRITE("<tr><td width=\"%d\">", CORNER_SIZE);
    box_corner(OUT, html_colour, "bl");
    WRITE("</td><td></td>");
    WRITE("<td width=\"%d\">", CORNER_SIZE);
    box_corner(OUT, html_colour, "br");
    WRITE("</td></tr>", html_colour);
    WRITE("</table>");
}
void box_corner(OUTPUT_STREAM, char *html_colour, char *corner) {
    WRITE("<img src=\"inform:/%s_corner_%s.gif\" ", corner, html_colour);
    WRITE("width=\"%d\" height=\"%d\" border=\"0\" alt=\"...\" />",
        CORNER_SIZE, CORNER_SIZE);
}
```

The function open_coloured_box is called from 12/rb.
The function close_coloured_box is called from 12/rb.

§**13. Writing ISO Latin-1 strings as XML-escaped text.** We do occasionally use this on arbitrary Unicode text, so must not assume that we aren't touching Icelandic archaisms.

```
void write_xml_safe_text(OUTPUT_STREAM, char *txt) {
    int i, charcode;
    for (i=0; txt[i]; i++) {
        switch(txt[i]) {
            case '&': WRITE("&amp;"); break;
            case '<': WRITE("&lt;"); break;
            case '>': WRITE("&gt;"); break;
            default:
                charcode = (int) ((unsigned char) txt[i]);
                PUT(charcode);
                break;
        }
    }
}
```

The function write_xml_safe_text is called from 10/bib.

§14. Bibliographic text. "Bibliographic text" is text used in bibliographic data about the work of IF compiled: for instance, in the iFiction record, or in the Library Card section of the HTML index. Note that the exact output format depends on global variables, which allow the bibliographic text writing code to configure NI for its current purposes. On non-empty strings this routine therefore splits into one of three independent methods.

```
void compile_bibliographic_text(OUTPUT_STREAM, char *p) {
    STREAM_MUST_BE_IN_MEMORY(OUT);
    if (p == NULL) return;
    if (TEST_COMPILATION_MODE(COMPILE_TEXT_TO_XML_CMODE))
        ⟨Compile bibliographic text as XML respecting Treaty of Babel rules 15⟩;
    if (TEST_COMPILATION_MODE(TRUNCATE_TEXT_CMODE))
        ⟨Compile bibliographic text as a truncated filename 17⟩;
    ⟨Compile bibliographic text as HTML 16⟩;
}
```

The function compile_bibliographic_text is called from 10/str.

§15. This looks like a standard routine for converting ISO Latin-1 to UTF-8 with XML escapes, but there are a few conventions on whitespace, too, in order to comply with a strict reading of the Treaty of Babel. (This is intended for fields in iFiction records.)

⟨Compile bibliographic text as XML respecting Treaty of Babel rules 15⟩ ≡

```
    int i = 0, i2 = strlen(p)-1, snl, wsc, charcode;
    if ((p[0] == '"') && (p[i2] == '"')) { i++; i2--; }                omit surrounding double-quotes
    while (is_babel_whitespace(p[i])) i++;                             omit leading whitespace
    while ((i2>=0) && (is_babel_whitespace(p[i2]))) i2--;              omit trailing whitespace
    for (snl = FALSE, wsc = 0; i<=i2; i++) {
        switch(p[i]) {
            case ' ': case '\x0a': case '\x0d': case '\t':
                snl = FALSE;
                wsc++;
                if (wsc == 1) WRITE(" ");
                break;
            case NEWLINE_IN_STRING:
                if (snl) break;
                if (STREAM_MOST_RECENT_CHAR(OUT) == ' ') STREAM_BACKSPACE(OUT);
                WRITE("<br/>");
                snl = TRUE; wsc = 1; break;
            case '[':
                if ((p[i+1] == '\'') && (p[i+2] == ']')) {
                    i += 2;
                    WRITE("'"); break;
                }                                                      and otherwise fall through to the default case
            default:
                snl = FALSE;
                wsc = 0;
                switch(p[i]) {
                    case '&': WRITE("&amp;"); break;
                    case '<': WRITE("&lt;"); break;
                    case '>': WRITE("&gt;"); break;
                    default:
                        charcode = (int) (((unsigned char *)p)[i]);
                        PUT(charcode);
```

```
                    break;
                }
                break;
        }
    }
    return;
```

This code is used in §14.

§**16.** In the HTML version, we want to respect the forcing of newlines, and also the `[']` escape to obtain a literal single quotation mark.

⟨Compile bibliographic text as HTML 16⟩ ≡

```
    int i, whitespace_count=0, charcode;
    if (p[0] == '"') p++;
    for (i=0; p[i]; i++) {
        if ((p[i] == '"') && (p[i+1] == 0)) break;
        switch(p[i]) {
            case ' ': case '\x0a': case '\x0d': case '\t':
                whitespace_count++;
                if (whitespace_count == 1) PUT(' ');
                break;
            case NEWLINE_IN_STRING:
                while (p[i+1] == NEWLINE_IN_STRING) i++;
                PUT('<');
                PUT('p');
                PUT('>');
                whitespace_count = 1;
                break;
            case '[':
                if ((p[i+1] == '\'') && (p[i+2] == ']')) {
                    i += 2;
                    PUT('\''); break;
                }                                    and otherwise fall through to the default case
            default:
                charcode = (int) (((unsigned char *)p)[i]);
                whitespace_count = 0;
                PUT(charcode);
                break;
        }
    }
    return;
```

This code is used in §14.

§**17.** This code is used to work out a good filename for something given a name inside NI. For instance, if a project is called

"St. Bartholemew's Fair: Étude for a Push-Me/Pull-You Machine"

then what would be a good filename for its released story file?

In the filename version we must forcibly truncate the text to ensure that it does not exceed a certain length, and must also make it filename-safe, omitting characters used as folder separators on various platforms and (for good measure) removing accents from accented letters, so that we can arrive at a sequence of ASCII characters. Each run of whitespace is also converted to a single space. If this would result in an empty text or only a single space, we return the text "story" instead.

Our example (if not truncated) then emerges as:

```
St- Bartholemew's Fair- Etude for a Push-Me-Pull-You Machine
```

Note that we do not write any filename extension (e.g., `.z5`) here.

⟨Compile bibliographic text as a truncated filename 17⟩ ≡

```
int i, pos = STREAM_EXTENT(OUT), whitespace_count=0, black_chars_written = 0, charcode;
if (p[0] == '"') p++;
for (i=0; p[i]; i++) {
    if (STREAM_EXTENT(OUT) - pos >= BIBLIOGRAPHIC_TEXT_TRUNCATION) break;
    if ((p[i] == '"') && (p[i+1] == 0)) break;
    switch(p[i]) {
        case ' ': case '\x0a': case '\x0d': case '\t': case NEWLINE_IN_STRING:
            whitespace_count++;
            if (whitespace_count == 1) PUT(' ');
            break;
        default:
            charcode = (int) (((unsigned char *)p)[i]);
            ⟨Render the character code filename-safe if possible 18⟩;
            whitespace_count = 0;
            if (charcode < 128) {
                PUT(charcode); black_chars_written++;
            }
            break;
    }
}
if (black_chars_written == 0) WRITE("story");
return;
```

This code is used in §14.

§**18.** We change possible filename separators or extension indicators to hyphens, and remove accents from each possible ISO Latin-1 accented letter. This does still mean that the OE and AE digraphs will simply be omitted, while the German eszet will be barbarously shortened to a single "s", but life is just too short to care overmuch about this.

⟨Render the character code filename-safe if possible 18⟩ ≡

```
charcode = iso_remove_accents(charcode);
```

This code is used in §17.

§**19.** This is isolated as a routine since it is also convenient for cleaning the output to `stdout`.

```
int iso_remove_accents(int charcode) {
    if (charcode<0) charcode += 256;                                    in case char is signed
    switch (charcode) {
        case '/': case '\\': case ':': case '.': charcode = '-'; break;
        case 0xC0: case 0xC1: case 0xC2: case 0xC3:
        case 0xC4: case 0xC5: charcode = 'A'; break;
        case 0xE0: case 0xE1: case 0xE2: case 0xE3:
        case 0xE4: case 0xE5: charcode = 'a'; break;
        case 0xC8: case 0xC9: case 0xCA: case 0xCB: charcode = 'E'; break;
        case 0xE8: case 0xE9: case 0xEA: case 0xEB: charcode = 'e'; break;
        case 0xCC: case 0xCD: case 0xCE: case 0xCF: charcode = 'I'; break;
        case 0xEC: case 0xED: case 0xEE: case 0xEF: charcode = 'i'; break;
        case 0xD2: case 0xD3: case 0xD4: case 0xD5:
        case 0xD6: case 0xD8: charcode = 'O'; break;
        case 0xF2: case 0xF3: case 0xF4: case 0xF5:
        case 0xF6: case 0xF8: charcode = 'o'; break;
        case 0xD9: case 0xDA: case 0xDB: case 0xDC: charcode = 'U'; break;
        case 0xF9: case 0xFA: case 0xFB: case 0xFC: charcode = 'u'; break;
        case 0xDD: charcode = 'Y'; break;
        case 0xFD: charcode = 'y'; break;
        case 0xD1: charcode = 'N'; break;
        case 0xF1: charcode = 'n'; break;
        case 0xC7: charcode = 'C'; break;
        case 0xE7: charcode = 'c'; break;
        case 0xDF: charcode = 's'; break;
    }
    if (charcode >= 128) charcode = '-';
    return charcode;
}
```

The function iso_remove_accents is called from 4/iext.

§**20.** An irksome companion piece to this: a routine which converts the Unicode combining accents with letters, sufficient correctly to handle all characters in the above collection.

```
int combining_accent(int accent, int letter) {
    switch(accent) {
        case 0x0300:                                                    Unicode combining grave
            switch(letter) {
                case 'a': return 0xE0; case 'e': return 0xE8; case 'i': return 0xEC;
                case 'o': return 0xF2; case 'u': return 0xF9;
                case 'A': return 0xC0; case 'E': return 0xC8; case 'I': return 0xCC;
                case 'O': return 0xD2; case 'U': return 0xD9;
            }
            break;
        case 0x0301:                                                    Unicode combining acute
            switch(letter) {
                case 'a': return 0xE1; case 'e': return 0xE9; case 'i': return 0xED;
                case 'o': return 0xF3; case 'u': return 0xFA; case 'y': return 0xFF;
                case 'A': return 0xC1; case 'E': return 0xC9; case 'I': return 0xCD;
                case 'O': return 0xD3; case 'U': return 0xDA;
            }
```

```
            break;
        case 0x0302:                                        Unicode combining circumflex
            switch(letter) {
                case 'a': return 0xE2; case 'e': return 0xEA; case 'i': return 0xEE;
                case 'o': return 0xF4; case 'u': return 0xFB;
                case 'A': return 0xC2; case 'E': return 0xCA; case 'I': return 0xCE;
                case 'O': return 0xD4; case 'U': return 0xDB;
            }
            break;
        case 0x0303:                                        Unicode combining tilde
            switch(letter) {
                case 'a': return 0xE3; case 'n': return 0xF1; case 'o': return 0xF5;
                case 'A': return 0xC3; case 'N': return 0xD1; case 'O': return 0xD5;
            }
            break;
        case 0x0308:                                        Unicode combining diaeresis
            switch(letter) {
                case 'a': return 0xE4; case 'e': return 0xEB; case 'u': return 0xFC;
                case 'A': return 0xC4; case 'E': return 0xCB; case 'U': return 0xDC;
            }
            break;
        case 0x0327:                                        Unicode combining cedilla
            switch(letter) {
                case 'c': return 0xE7; case 'C': return 0xC7;
            }
            break;
    }
    return '?';
}
```

The function combining_accent is called from 1/plat.

§**21.** These are all the characters which would come out as whitespace in the sense of the Treaty of Babel rules on leading and trailing spaces in iFiction records.

```
int is_babel_whitespace(char c) {
    if ((c == ' ') || (c == '\t') || (c == '\x0a')
        || (c == '\x0d') || (c == NEWLINE_IN_STRING)) return TRUE;
    return FALSE;
}
```

Javascript Pastes 2/java

Purpose

To write valid HTML for a paste icon which, when clicked, calls a Javascript function which will paste Inform source text into the Source panel of the application.

2/java.§7 Individual characters

Definitions

¶1. The application is required to provide a Javascript function to copy text into the source window. Broadly speaking, the application needs to support Javascript in the following form:

```
var myProject = external.Project;
myProject.selectView('source');
myProject.pasteCode('Trying Taking Manhattan');
```

This for Windows: for OS X, the same code but `window.Project` rather than `external.Project`.

As this implies, the details unfortunately differ on different platforms. The constant `JAVASCRIPT_MODEL`, defined in Platform-Specific Definitions, can be any of:

(a) Model 0 - no Javascript pastes are available (so no icon will be set).
(b) Model 1 - paste in OS X style, directly within the HREF of a link.
(c) Model 2 - paste in Windows style, defining a function and calling that.

In model 2 we define a Javascript function for each individual paste because this protects against long paste texts overflowing what Windows considers the maximum permitted length of a link: the WebKit rendering engine in OS X has no such limit, apparently. This means that for Windows we define numerous copies of the Javascript code above. In model 1, we never need to compile fresh Javascript functions because the template file `ExtensionFileModel.html` for OS X contains a definition of the single Javascript function:

```
<script language="JavaScript">
function pasteCode(code) {
 var myProject = window.Project;
 myProject.selectView('source');
 myProject.pasteCode(code);
}
</script>
```

and we can simply call `href="javascript:pasteCode(...)"` from any link.

The text pasted may in some cases be quite long (say, 5K or more) and the code below should work whatever its length. It will of course be UTF-8 encoded, since all HTML produced by NI is.

¶2. We have found that different Javascript implementations handle escape characters in quoted text differently. (For instance, some allow a double-quote " to appear as a literal in single-quoted text, others require `&quot;` to be used, others still do not recognise HTML entities like `&quot;` and treat them as literal text.) To avoid these tiresome platform dependencies a single new escape-character syntax was added in November 2007. This puts obligations both on NI (and `indoc`, which also generates HTML with Javascript pastes), to make use of the escape syntax, and also on the application, to understand and act on it.

The application must implement `myProject.pasteCode(code)` such that every instance of `[=0xHHHH=]` is replaced with the Unicode character whose hexadecimal code is `HHHH`. There will always be four digits, with leading zeros as needed, and `A` to `F` will be written in upper case. The only Unicode characters with codes below `0x0020` which must be handled are newline, `0x000A`, and tab, `0x0009`.

The generator (NI or `indoc`) must always escape every instance of the following characters:

(a) every tab is escaped to `[=0x0009=]`;
(b) every newline is escaped to `[=0x000A=]`;
(c) every double quotation mark is escaped to `[=0x0022=]`;
(d) every ampersand is escaped to `[=0x0026=]`;
(e) every single quotation mark is escaped to `[=0x0027=]`;
(f) every less than sign is escaped to `[=0x003C=]`;
(g) every greater than sign is escaped to `[=0x003E=]`;
(h) every backslash is escaped to `[=0x005C=]`.

It may also choose to escape other character codes, as it prefers. Other characters are generated as literal UTF-8. In no case will any character with code below `0x0020` be passed as a literal.

---

§**1.** At the top level, the form of link used depends on the `JAVASCRIPT_MODEL`. Note that model 0 results in no material at all being output. The actual text to be passed is all set via `javascript_string_out` below, and which does not depend on the model.

```
int javascript_fn_counter = 1000;
void write_javascript_paste(OUTPUT_STREAM, int from, int to, char *alt_text) {
    switch(JAVASCRIPT_MODEL) {
        case 1:                                   OS X style, with long function arguments allowed in links
            WRITE("<a href=\"javascript:pasteCode(");
            javascript_string_out(OUT, from, to, alt_text);
            WRITE(")\"><img border=0 src=inform:/doc_images/paste.png></a>");
            break;
        case 2:                                   Windows style, with long function arguments in links unreliable
            WRITE("<script language=\"JavaScript\">\n");
            WRITE("function pasteCode%d(code) {\n", javascript_fn_counter); INDENT;
            WRITE("var myProject = external.Project;\n\n");
            WRITE("myProject.selectView('source');\n");
            WRITE("myProject.pasteCode(");
            OUTDENT; javascript_string_out(OUT, from, to, alt_text);
            WRITE(");\n");
            WRITE("}\n");
            WRITE("</script>\n");
            WRITE("<a href=\"javascript:pasteCode%d()\">"
                "<img border=0 src=inform:/doc_images/paste.png></a>",
                javascript_fn_counter++);
            break;
    }
}
```

The function write_javascript_paste is called from 2/hdoc, 4/edoc, 12/br and 12/rb.

§**2.** In the following, the source of the text can be either a range of words from the lexer (as for instance when a portion of an extension is being typeset as documentation, with an example that can be pasted), or can be a C string: if the latter, then its encoding must be ISO Latin-1. The conversion to UTF-8 is performed in `javascript_char_out` below.

```
void javascript_string_out(OUTPUT_STREAM, int from, int to, char *C_string) {
    WRITE("'");
    if (C_string) ⟨Write C string as Javascript string 3⟩;
    if (from >= 0) ⟨Write word range as Javascript string 4⟩;
    WRITE("'");
}
```

§**3.** The art of leadership is delegation.

⟨Write C string as Javascript string 3⟩ ≡

```
    int i;
    for (i=0; C_string[i]; i++)
        javascript_char_out(OUT, C_string[i]);
```

This code is used in §2.

§**4.** Writing a word range is much harder. In effect, we have to provide an inverse function for the lexer, which converted raw source text to nicely packaged up words.

See Lexer for details of how words are stored, and in particular for the `lw_break` character, which is `'\t'` when the word followed a tab, but is `'1'` to `'9'` when it followed a newline plus that many tabs. We need this because lexing has otherwise removed whitespace from the source, and we need it back again if we're to paste a faithful Javascript representation: otherwise the tabs used as column-dividers in tables will not come through, for instance. Moreover, indentation from the left margin is used to make prettier pastes (which respect the layout of the original examples from which the paste has been made), and for that we need the `'1'` to `'9'` possibilities.

Note that we expect the material pasted to be indented at 1 tab stop from the margin already, because it will almost always be a source text within an example, where any matter unindented will be commentary rather than source text. Thus a single tab after a newline is not significant, and we only need to supply extra Javascript tabs when the indentation is 2 tab stops or more.

⟨Write word range as Javascript string 4⟩ ≡

```
    int i, suppress_space = FALSE, follows_paragraph_break = FALSE;
    for (i=from; i<=to; i++) {
        int indentation = 0, j;
        char *p = lw_array[i].lw_rawtext;
        if [[word i == STROKE]] {                                   marker for a paragraph break
            javascript_char_out(OUT, '\n');
            javascript_char_out(OUT, '\n');
            suppress_space = TRUE;
            follows_paragraph_break = TRUE;
            while [[word i == STROKE]] i++; i--;                    elide multiple breaks
            continue;
        }
        if (isdigit(lw_array[i].lw_break)) indentation = lw_array[i].lw_break - '0';
        if (indentation > 0) {                          number of tab stops of indentation on this para
            javascript_char_out(OUT, '\n');
            for (j=0; j<indentation-1; j++) javascript_char_out(OUT, '\t');
            suppress_space = TRUE;
```

```
        }
        if ((lw_array[i].lw_break == '\t') && (follows_paragraph_break == FALSE)) {
            javascript_char_out(OUT, '\t');
            suppress_space = TRUE;
        }
        follows_paragraph_break = FALSE;
        if (suppress_space==FALSE)
            ⟨Restore inter-word spaces unless this would be unnatural 5⟩;
        suppress_space = FALSE;
        for (j=0; p[j]; j++) javascript_char_out(OUT, p[j]);
        ⟨Insert a close-literal-I6 escape sequence if necessary 6⟩;
    }
    javascript_char_out(OUT, '\n');
    javascript_char_out(OUT, '\n');
```

This code is used in §2.

§**5.** The lexer also broke words around punctuation marks, so that, for instance, "fish, finger" would have been lexed as `fish , finger` – three words. But we want to restore the more natural spacing.

⟨Restore inter-word spaces unless this would be unnatural 5⟩ ≡

```
    if ((i>from)
        && ((p[1] != 0) || (is_punctuation(p[0]) == FALSE) ||
            (p[0] == '(') || (p[0] == '{') || (p[0] == '}'))
        && (compare_word(i-1, OPENBRACKET_V)==FALSE))
        javascript_char_out(OUT, ' ');
```

This code is used in §4.

§**6.** Finally, the lexer rendered a literal I6 inclusion in the form

(- deadflag=2; -)

as a sequence of two lexical words: `(-` and then `deadflag=2;`. In order to paste back safely, we must supplement this with the closure "-)" once again:

⟨Insert a close-literal-I6 escape sequence if necessary 6⟩ ≡

```
    if (vocab_test_flags(i, I6_MC)) {
        javascript_char_out(OUT, '-');
        javascript_char_out(OUT, ')');
        javascript_char_out(OUT, ' ');
    }
```

This code is used in §4.

§**7. Individual characters.** Note that every character within the single quotes of a Javascript string is produced through the following routine. It combines the steps of escaping awkward characters and converting from ISO Latin-1 to UTF-8.

```
void javascript_char_out(OUTPUT_STREAM, char c) {
    switch(c) {
        case '\t': WRITE("[=0x0009=]"); return;
        case '\n': WRITE("[=0x000A=]"); return;
        case '"': WRITE("[=0x0022=]"); return;
        case '&': WRITE("[=0x0026=]"); return;
        case '\'': WRITE("[=0x0027=]"); return;
        case '<': WRITE("[=0x003C=]"); return;
        case '>': WRITE("[=0x003E=]"); return;
        case '\\': WRITE("[=0x005C=]"); return;
        default: {
            int charcode = (int) ((unsigned char) c);
            if (charcode < 32) {
                LOG("Illegal character %d found\n", charcode);
                internal_error("illegal character in Javascript paste");
            }
            if (charcode >= 128) {
                PUT(0xC0 + (charcode >> 6));
                PUT(0x80 + (charcode & 0x3f));
            } else PUT(charcode);
            return;
        }
    }
}
```

HTML Documentation 2/hdoc

Purpose

To translate a passage of source text into HTML-format documentation, for use in the automatically generated documentation pages on each installed extension.

---

§**1.** This is a port of a simplified version of the Inform documentation tool `indoc`, and like all Perl scripts ported to C, it bears a few scars.

Documentation is extracted from extensions by lexing them (ignoring all of the actual source text and picking up only after the divider line) and then running the code below on the word range for the lexed documentation. Our task is therefore to print out this documentation in an HTML format which matches the look and feel of pages produced by `indoc` for the manuals.

§**2. Links and leafnames.** Matters are complicated because an extension typically has not only a run of source text, but also up to 26 examples: suppose there are X of these. The extension then needs to produce $X+1$ pages of HTML: the primary one, which just has the body text, and then X variants which duplicate the primary one except that one of the examples is opened up to reveal its content. Each of these pages will have X anchor points named `#eg1` up to `#egX`, for the positions of the examples.

The pages will typically be filenamed with the extension title, followed by `-eg1`, `-eg2`, ..., in the case of the example variants, and then `.html`.

The following routine prints the leafname part of an HTML reference to the extension documentation, at anchor point `to_example_anchor` (or if 0 then at the top) of the version with example `to_example_variant` opened (or if 0 then the original with all examples closed). What complicates it is that the base leafname might be any of the above variant filenames, so we may need to strip off an existing ending. For instance, if we are in example 2 and want to link to anchor 5 on example 4, the base leafname might be `Gusher-eg2` and we need to remove the `-eg2` and replace with `-eg4` before we can add the `#eg5`.

This will fail if anyone's extension has a title ending in `-eg` followed by a number. I believe I can live with the guilt.

```
void href_of_example(OUTPUT_STREAM, char *base_leafname,
    int to_example_variant, int to_example_anchor) {
    int i;
    for (i=0; base_leafname[i]; i++) {
        if ((base_leafname[i] == '-') && (base_leafname[i+1] == 'e')
            && (base_leafname[i+2] == 'g') && (isdigit(base_leafname[i+3]))) break;
        WRITE("%c", base_leafname[i]);
    }
    if (to_example_variant > 0) WRITE("-eg%d", to_example_variant);
    WRITE(".html");
    if (to_example_anchor > 0) WRITE("#eg%d", to_example_anchor);
}
```

§**3. Structural blocks of extension documentation.** The extension documentation text can optionally include section and chapter headings, and also examples. Here we parse the opening of a paragraph to see if it might be a heading. For instance, a paragraph consisting of

Section: Black Gold

matches successfully and sets the level to 2 and the name to the word range "Black Gold".

```
int extension_documentation_heading(int w1, int w2,
    int *level, int *hn1, int *hn2) {
    int lev, end;
    if (w1+10 >= w2) return FALSE;                    not enough space: this runs into the end-of-file padding
    if [[w1, w2 == chapter COLON ...]] lev = 1;
    else if [[w1, w2 == section COLON ...]] lev = 2;
    else return FALSE;
    *level = lev;
    w1 += 2;
    end = w1; while ((end<=w2) && ([[word end == STROKE]] == FALSE)) end++; end--;
    *hn1 = w1; *hn2 = end;
    return TRUE;
}
```

§**4.** And here we do the same to identify an example, which has to satisfy a more exacting specification: a paragraph in the shape

Example: *** Gelignite Anderson - A Tale of the Texas Oilmen

which would result in the name being set to the range "Gelignite Anderson", an asterisk count of 3, and the rubric being "A Tale of the Texas Oilmen".

```
int extension_documentation_example(int w1, int w2,
    int *ast, int *egn1, int *egn2, int *egr1, int *egr2) {
    if (w1+10 >= w2) return FALSE;                    not enough space: this runs into the end-of-file padding
    if [[w1, w2 == example COLON ...]] {
        int aw = w1+2, n1, n2, r1, r2, asterisks = 0;
        after the colon, an optional row of unspaced asterisks:
        if [[word aw == ASTERISK]] asterisks = 1;
        if [[word aw == TWOASTERISKS]] asterisks = 2;
        if [[word aw == THREEASTERISKS]] asterisks = 3;
        if [[word aw == FOURASTERISKS]] asterisks = 4;
        n1 = aw; if (asterisks > 0) n1 = aw + 1;
        the example name now extends up as far as the next freestanding hyphen:
        n2 = n1;
        while ((n2 <= w2) && ([[word n2 == HYPHEN]] == FALSE)) n2++;
        if (n2 >= w2) return FALSE; n2--;
        on the other side of the hyphen, the rubric extends to the next paragraph break
        r1 = n2 + 2; r2 = r1;
        while ((r2 <= w2) && ([[word r2 == STROKE]] == FALSE)) r2++;
        if (r2 >= w2) return FALSE; r2--;
        a successful match has now been made
        *ast = asterisks; *egn1 = n1; *egn2 = n2; *egr1 = r1; *egr2 = r2;
        return TRUE;
    }
    return FALSE;
}
```

§5. The table of contents. The user sees chapters as A subheadings, numbered upwards from 1, and sees sections as B subheadings, numbered from 1 within each chapter. It is legal to have only A subheadings; only B subheadings; or a mixture of the two.

If a scan can find any headings at all then we will wish to typeset a table of contents up front. The following routine looks for what material might go into a TOC, and sets one if it finds anything: otherwise, it sets nothing and has no effect. Because of the compulsory paragraph break following the divider line in the extension, we can safely assume that every headng will follow a paragraph break word, even one right at the top of the extension's documentation.

(Examples are included in the table of contents only if they occur after the first heading, which I think is reasonable enough: there can be at most 26 per extension, enabling them to be lettered as Example A to Example Z.)

```
void edoc_set_table_of_contents(int w1, int w2, OUTPUT_STREAM, char *base_leafname) {
    int i;
    int heading_count = 0, chapter_count = 0, section_count = 0, example_count = 0;
    for (i=w1; i<=w2; i++) {
        int edhl, name_w1, name_w2, eg_rubric_w1, eg_rubric_w2, asterisks;
        if [[word i == STROKE]] {                          the lexer records this to mean a paragraph break
            while [[word i == STROKE]] i++; if (i>w2) break;
            if (extension_documentation_heading(i, w2, &edhl, &name_w1, &name_w2)) {
                heading_count++;
                if (heading_count == 1) WRITE("<p><hr><p>");                ruled line at top of TOC
                if (edhl == 1) {
                    chapter_count++; section_count = 0;
                    if (chapter_count > 1) WRITE("<br>");          skip a line between chapters in TOC
                }
                if (edhl == 2) section_count++;
                ⟨Typeset the table of contents entry for this heading 6⟩;
                i = name_w2; continue;
            }
            if ((heading_count > 0) && (example_count < 26) &&
                (extension_documentation_example(i, w2,
                    &asterisks, &name_w1, &name_w2, &eg_rubric_w1, &eg_rubric_w2))) {
                if (++example_count == 1) WRITE("<br><b>Examples</b><br>");
                ⟨Typeset the table of contents entry for this example 7⟩;
                i = eg_rubric_w2; continue;
            }
        }
    }
    if (heading_count > 0)
        WRITE("<p><hr><p>");                          ruled line at foot of TOC, if there is one
}
```

The function edoc_set_table_of_contents is called from 4/edoc.

§**6.** Internally, we are numbering all headings independently upwards from 1, and we set anchor points in the documentation called `#docsec1`, `#docsec2`, and so on: some of these will be chapter headings, some section headings. These are the destinations of links from heading lines in the TOC.

```
define EDOC_TOC_LINK_STYLE "STYLE=\"text-decoration: none\""
define EDOC_TOC_LINK_FONT "<font color=\"#000000\">"                    this must open a <font> tag
```

⟨Typeset the table of contents entry for this heading 6⟩ ≡

```
    char *html_to_close = NULL;
    switch (edhl) {
        case 1:
            WRITE("<b><a %s href=#docsec%d>%sChapter %d: ",
                EDOC_TOC_LINK_STYLE, heading_count, EDOC_TOC_LINK_FONT, chapter_count);
            html_to_close = "</font></a></b>";
            break;
        case 2:
            if (chapter_count > 0)                                  if there are chapters as well as sections...
                WRITE("   ");                          ...then set an indentation before entry
            WRITE("<a %s href=#docsec%d>%sSection ",
                EDOC_TOC_LINK_STYLE, heading_count, EDOC_TOC_LINK_FONT);
            if (chapter_count > 0)                                  if there are chapters as well as sections...
                WRITE("%d.%d: ", chapter_count, section_count);                    quote in form S.C
            else
                WRITE("%d: ", section_count);                         otherwise quote section number only
            html_to_close = "</font></a>";
            break;
        default: internal_error("unable to set this heading level in extension TOC");
    }
    edoc_set_body_text(name_w1, name_w2, OUT, EDOC_FRAGMENT_ONLY, NULL);
    WRITE("%s<br>\n", html_to_close);
```

This code is used in §5.

§**7.** The TOC entries for examples are similar. Here the link is to the variant page in the current family which has the given example open, and moreover, to the anchor in that page corresponding to the top of the example: thus as far as the user is concerned it opens the example and goes there.

⟨Typeset the table of contents entry for this example 7⟩ ≡

```
    WRITE("   ");                                always indent TOC entries for examples
    WRITE("<a %s href=\"", EDOC_TOC_LINK_STYLE);
    href_of_example(OUT, base_leafname, example_count, example_count);
    WRITE("\">%s%c: ", EDOC_TOC_LINK_FONT, 'A'+example_count-1);                    the letter A to Z
    edoc_set_body_text(name_w1, name_w2, OUT, EDOC_FRAGMENT_ONLY, NULL);
    WRITE("</font></a><br>");
```

This code is used in §5.

§8. Setting the body text.

define EDOC_ALL_EXAMPLES_CLOSED -1 *do not change this without also changing Extensions*
define EDOC_FRAGMENT_ONLY -2 *must differ from this and from all example variant numbers*

```
int edoc_set_body_text(int w1, int w2, OUTPUT_STREAM,
    int example_which_is_open, char *base_leafname) {
    int heading_count = 0, chapter_count = 0, section_count = 0, example_count = 0;
    int mid_example = FALSE, skipping_text_of_an_example = FALSE,
        start_table_next_line = FALSE, mid_I7_table = FALSE, row_of_table_is_empty = FALSE,
        mid_displayed_source_text = FALSE, indentation = 0;
    int i;
    for (i=w1; i<=w2; i++) {
        int edhl, name_w1, name_w2, eg_rubric_w1, eg_rubric_w2, asterisks;
        if [[word i == STROKE]] {                    the lexer records this to mean a paragraph break
            ⟨Handle a paragraph break 9⟩;
            while [[word i == STROKE]] i++; if (i>w2) break;    treat multiple paragraph breaks as one
            ⟨Determine indentation of new paragraph 10⟩;
            if (extension_documentation_heading(i, w2, &edhl, &name_w1, &name_w2)) {
                heading_count++;
                if (edhl == 1) {
                    chapter_count++; section_count = 0;
                    if (chapter_count > 1) WRITE("<p><hr>");          rule a line between chapters
                }
                if (edhl == 2) section_count++;
                ⟨Typeset the heading of this chapter or section 15⟩;
                i = name_w2; continue;
            }
            if ((example_count < 26) && (extension_documentation_example(i, w2,
                    &asterisks, &name_w1, &name_w2, &eg_rubric_w1, &eg_rubric_w2))) {
                skipping_text_of_an_example = FALSE;
                if (mid_example) ⟨Close the previous example's text 25⟩;
                mid_example = FALSE;
                example_count++;
                ⟨Typeset the heading of this example 16⟩;
                if (example_count == example_which_is_open) {
                    ⟨Open the new example's text 24⟩;
                    mid_example = TRUE;
                } else skipping_text_of_an_example = TRUE;
                i = eg_rubric_w2; continue;
            }
        }
        if (skipping_text_of_an_example) continue;
        ⟨Handle a line or column break, if there is one 11⟩;
        ⟨Transcribe an ordinary word of the documentation 12⟩;
    }
    if (mid_example) ⟨Close the previous example's text 25⟩;
    if (example_which_is_open != EDOC_FRAGMENT_ONLY) ⟨Handle a paragraph break 9⟩;
    return example_count;
}
```

The function edoc_set_body_text is called from 4/edoc.

§**9. Typesetting the standard matter.** A paragraph break might mean the end of displayed matter (and if so, then also the end of any table being displayed). Otherwise, it just means a paragraph break, and a chance to restore our tired variables.

⟨Handle a paragraph break 9⟩ ≡
```
if (mid_displayed_source_text)  {
    WRITE("</font>");
    if (mid_I7_table) ⟨End I7 table in extension documentation 23⟩;
    WRITE("</blockquote>");
}
WRITE("\n<p>\n");
mid_displayed_source_text = FALSE; mid_I7_table = FALSE;
```

This code is used in §8.

§**10.** The indentation setting is made here because a tab anywhere else does not mean a paragraph has been indented. Here `i` is at the number of the first word after the paragraph break; the break character corresponding to it is the one before that word, so describes the kind of whitespace between the paragraph break and the first nonwhitespace of the new paragraph.

⟨Determine indentation of new paragraph 10⟩ ≡
```
indentation = 0; if (lw_array[i].lw_break == '\t') indentation = 1;
```

This code is used in §8.

§**11.** Two lower-level sorts of breaks can also occur in the middle of a paragraph: line breaks, indicated by newlines plus some tabs, and column breaks inside I7 source tables, indicated by tabs. We have to deal with those before we can move on to the subsequent word.

⟨Handle a line or column break, if there is one 11⟩ ≡
```
if (isdigit(lw_array[i].lw_break)) indentation = lw_array[i].lw_break - '0';
if (indentation > 0) ⟨Handle the start of a line which is indented 14⟩;
if [[i, w2 == ASTERISK COLON ...]] {
    ⟨Incorporate an icon linking to a Javascript function to paste the text which follows 13⟩;
    i++; continue;
}
indentation = 0;
if ((mid_I7_table) && ((lw_array[i].lw_break == '\t') || (lw_array[i].lw_break == '1'))) {
    if (row_of_table_is_empty == FALSE)
        ⟨End table cell for I7 table in extension documentation 20⟩;
    ⟨Begin table cell for I7 table in extension documentation 21⟩;
    row_of_table_is_empty = FALSE;
}
```

This code is used in §8.

§**12.** See Javascript Pastes for further explanation of the general method here.

⟨Transcribe an ordinary word of the documentation 12⟩ ≡

```
char *p = lw_array[i].lw_rawtext; int j;
if ((i>w1)
    && ((p[1] != 0) || (is_punctuation(p[0]) == FALSE)
        || (p[0] == '(') || (p[0] == '{') || (p[0] == '}'))
    && (compare_word(i-1, OPENBRACKET_V)==FALSE))
    WRITE(" ");                                    restore normal spacing around punctuation
for (j=0; p[j]; j++) html_char_out(OUT, p[j]);                          set the actual word
if ((![[word i == OPENI6]]) &&
    (vocab_test_flags(i, I6_MC))) WRITE("-) ");                  ensure I6 literals are closed
```

This code is used in §8.

§**13.** A paste causes the same material to be set twice: once in the argument to the Javascript paste function (which is passed to the application when the user clicks on the paste icon, and thus ends up in the Source panel), and once also in the HTML documentation. That's why the code here ranges forward to see how far it should go (to the next paragraph break which is not followed by further tabbed matter, or in other words, to the end of the display), but does not advance `i` commensurately.

⟨Incorporate an icon linking to a Javascript function to paste the text which follows 13⟩ ≡

```
int from = i+2, to = w2, j;
for (j=from; j<=to; j++)                                  first find the end of the quoted passage
    if [[word j == STROKE]] {
        int possible_end = j-1;
        while [[word j == STROKE]] j++;
        if ((j<to) && ((lw_array[j].lw_break == '\t') || (isdigit(lw_array[j].lw_break)))) continue;
        to = possible_end; break;
    }
write_javascript_paste(OUT, from, to, NULL);
```

This code is used in §11.

§**14.** The first step of indentation is handled using the `<blockquote>` tag; within that, further tab stops are simulated by printing a row of four non-breaking spaces for each indentation level above 1. A paragraph of indented (i.e., display matter) beginning with the word "Table" is taken to be an I7 table, and we remember that the next line break will take us past the titling line and into the table entries, which we will need to achieve with an HTML `<table>`.

⟨Handle the start of a line which is indented 14⟩ ≡

```
int j;
if (mid_displayed_source_text) {
    if (start_table_next_line) {
        start_table_next_line = FALSE;
        mid_I7_table = TRUE;
        ⟨Begin I7 table in extension documentation 19⟩;
    } else {
        if (mid_I7_table) ⟨Begin new row of I7 table in extension documentation 22⟩
        else WRITE("<br>");
    }
    if (mid_I7_table) row_of_table_is_empty = TRUE;
} else {
    WRITE("<blockquote><font color=\"#000080\">");
    mid_displayed_source_text = TRUE;
```

```
        if [[word i == table]] start_table_next_line = TRUE;
    }
    indentation--;
    for (j=0; j<indentation; j++) WRITE("    ");
```

This code is used in §11.

§**15. Typesetting the headings.** That is thankfully all for the tormented logic of all those changes of state: from here to the rest of the section, all we do is to generate pretty HTML, and without altering any variables or causing any side-effects at all. First, the headings. Recall that heading number `N` is required to be at anchor `#docsecN`.

```
define EDOC_TOC_C_HEAD_FONT "<font color=\"#800000\">"          this must open a <font> tag
define EDOC_TOC_S_HEAD_FONT "<font color=\"#000000\">"          this must open a <font> tag
```

⟨Typeset the heading of this chapter or section 15⟩ ≡

```
    switch (edhl) {
        case 1:
            WRITE("<p><a name=docsec%d><b>%sChapter %d: ",
                heading_count, EDOC_TOC_C_HEAD_FONT, chapter_count);
            break;
        case 2:
            WRITE("<p><a name=docsec%d><b>%sSection ",
                heading_count, EDOC_TOC_S_HEAD_FONT);
            if (chapter_count > 0) WRITE("%d.", chapter_count);
            WRITE("%d: ", section_count);
            break;
    }
    edoc_set_body_text(name_w1, name_w2, OUT, EDOC_FRAGMENT_ONLY, NULL);
    WRITE("</b>");
```

This code is used in §8.

§**16.** An example is set with a two-table header, and followed optionally by a table of its inset copy, shaded to distinguish it from the rest of the page. The heading is constructed with a main table of one row of two cells, in the following section. The left-hand cell then contains a further table, in the next section.

⟨Typeset the heading of this example 16⟩ ≡

```
    WRITE("<hr><p>");                                   rule a line before the example heading
    WRITE("<a name=eg%d>", example_count);              provide the anchor point
    begin_plain_html_table(OUT);
    WRITE("<tr>");
    Left hand cell: the oval icon
    WRITE("<td halign=\"left\" valign=\"top\" cellpadding=0 cellspacing=0 width=38px>");
    ⟨Typeset the lettered oval example icon 17⟩;
    WRITE("</td>");
    Right hand cell: the asterisks and title, with rubric underneath
    WRITE("<td cellpadding=0 cellspacing=0 halign=\"left\" valign=\"top\"><font %s>",
        DEFAULT_HTML_FONT);
    ⟨Incorporate link to the example opened up 18⟩;
    while (asterisks-- > 0)
        WRITE("<img border=\"0\" src='inform:/doc_images/asterisk.png'>");
    WRITE("<b><font color=\"#505050\"> Example </font> <font color=\"#000000\">");
    edoc_set_body_text(name_w1, name_w2, OUT, EDOC_FRAGMENT_ONLY, base_leafname);
    WRITE("</font></b></a>");                                                  end the textual link
```

```
    WRITE("<br>");
    edoc_set_body_text(eg_rubric_w1, eg_rubric_w2, OUT, EDOC_FRAGMENT_ONLY, base_leafname);
    WRITE("<p>");
    WRITE("</td></tr>");
    end_html_table(OUT);
    WRITE("<p>");
```

This code is used in §8.

§**17.** The little oval icon with its superimposed boldface letter is much harder to get right on all browsers than it looks, and the following is the result of some pretty grim experimentation. Basically, we make a tight, borderless, one-cell-in-one-row table, use CSS to make a transparent PNG image of an oval the background image for the table, then put a boldface letter in the centre of its one and only cell. (Things were even worse when IE6 for Windows still had its infamous PNG transparency bug.)

⟨Typeset the lettered oval example icon 17⟩ ≡

```
    begin_plain_html_table(OUT);
    WRITE("<tr class=\"oval\"><td width=38px height=30px align=\"left\" valign=\"center\">");
    ⟨Incorporate link to the example opened up 18⟩;
    WRITE("<div class=\"paragraph Body\" style=\"line-height: 1px; margin-bottom: 0px; "
        "margin-top: 4px; padding-bottom: 0pt; padding-top: 4px; text-align: center; "
        "color: #202020; font-size: 14px; line-height: 1px;\"><b>%c</b></div>",
        'A' + example_count - 1);
    WRITE("</a></td></tr>");
    end_html_table(OUT);
```

This code is used in §16.

§**18.** Clicking on the example banner opens it up, if it's currently closed, or closes it up, if it's currently open.

⟨Incorporate link to the example opened up 18⟩ ≡

```
    WRITE("<a href=\"");
    if (example_count == example_which_is_open)                    this example currently open
        href_of_example(OUT, base_leafname, EDOC_ALL_EXAMPLES_CLOSED, example_count);
    else                                                           this example not yet open
        href_of_example(OUT, base_leafname, example_count, example_count);
    WRITE("\" STYLE=\"text-decoration: none\">");
```

This code is used in §16,17,16,17,16,17.

§**19. Typesetting I7 tables in displayed source text.** Unsurprisingly, I7 tables are set (after their titling lines) as HTML tables, and this is fiddly but elementary in the usual way of HTML tables:

⟨Begin I7 table in extension documentation 19⟩ ≡

```
    WRITE("</font><br>");
    begin_plain_html_table(OUT);
    first_html_column(OUT, 0);
```

This code is used in §14.

§**20.** Drinka.

⟨End table cell for I7 table in extension documentation 20⟩ ≡
```
WRITE("</font>");
next_html_column(OUT, 0);
```
This code is used in §11.

§**21.** Pinta.

⟨Begin table cell for I7 table in extension documentation 21⟩ ≡
```
WRITE("<font color=\"#000080\">");
```
This code is used in §11.

§**22.** Milka.

⟨Begin new row of I7 table in extension documentation 22⟩ ≡
```
end_html_row(OUT);
first_html_column(OUT, 0);
```
This code is used in §14.

§**23.** Day.

⟨End I7 table in extension documentation 23⟩ ≡
```
end_html_row(OUT);
end_html_table(OUT);
```
This code is used in §9.

§**24. Typesetting the body of an example.** This is done just the way all other extension documentation material is handled, except that it is inside an inset box: which is provided by a shaded HTML table, containing just one row, which contains just one cell. Here the inset table begins:

⟨Open the new example's text 24⟩ ≡
```
begin_html_table(OUT, "#f0f0f0", TRUE, 0, 0, 0, 0, 0);
first_html_column(OUT, 0);
WRITE("<p>");
```
This code is used in §8.

§**25.** And here the inset table ends:

⟨Close the previous example's text 25⟩ ≡
```
end_html_row(OUT);
end_html_table(OUT);
WRITE("<p>");
```
This code is used in §8.

Index File Services 2/index

Purpose

To provide routines to help build the various HTML index files, none of which are actually created in this section.

2/index.§3-5 Links to source and extension footnotes; §6 Links to detail pages; §7-11 Links to documentation; §12 "See below" links; §13-15 Miscellaneous utilities

---

Template interpreter commands

```
2   {-callv:complete_index}
```

Definitions

¶1. If only we had an index file, we could look it up under "index file"...

The Index of a project is a set of HTML files describing its milieu: the definitions it makes, the world model resulting, and the rules in force, which arise as a combination of the source text and the extensions (in particular, the Standard Rules).

¶2. Documentation is arranged in a series of HTML pages identified by section number 0, 1, 2, ..., and the index contains little blue help icons which link into this. In order to give these links the correct destinations, NI needs to know which section number contains what: but section numbering moves around a lot as the documentation is written.

To avoid needlessly recompiling NI when documentation changes, we give certain sections aliases called "symbols" which are rather more lasting than the section numbering. For instance, the sentence

> Document KINDSVALUE at doc64.

(in the Standard Rules, the only source file allowed to define symbols) creates the symbol `KINDSVALUE` and says that it currently corresponds to `doc64.html` in the documentation. Such sentences are automatically amended by `indoc` to keep the Standard Rules in line with the Inform documentation.

```
typedef struct documentation_ref {
    int symbol;                          Word number in source where the symbol is
    int section;                         HTML page number
    int used_already;                    Has this been used in a problem message already?
    char *chapter_reference;             Or NULL if no chapter name supplied
    char *section_reference;             Or NULL if no section name supplied
    MEMORY_MANAGEMENT
} documentation_ref;
```

The structure documentation_ref is private to this section.

---

§**1.** The index is written at the end of each successful compilation. During indexing, only one index file is ever open for output at a time: this always has the file handle `ifl`. The following routine is called to open a new index file for output.

```
char current_thumbnail_image[MAX_FILENAME_LENGTH];
char current_thumbnail_caption[MAX_FILENAME_LENGTH];
int cti_set = FALSE;
void set_thumbnail_image(char *image, char *caption) {
    cti_set = TRUE;
    strcpy(current_thumbnail_image, image);
    strcpy(current_thumbnail_caption, caption);
}
void open_index_file(char *index_leaf, char *title, int sub,
    char *explanation) {
    char *fn = index_filename(index_leaf, sub);
    if (ifl) close_index_file();
    if (STREAM_OPEN_TO_FILE(&index_file_struct, fn, UTF8_ENC) == FALSE)
        fatal_error2("Can't open index file", fn);
    ifl = &index_file_struct;
    html_header(ifl, title, (cti_set)?current_thumbnail_image:NULL, (cti_set)?current_thumbnail_caption:NULL);
    if (explanation) {
        char temp[MAX_FILENAME_LENGTH];
        strcpy(temp, explanation);
        int i, len = strlen(temp), italics_open = FALSE;
        open_html_paragraph(ifl, 1, "tight");
        for (i=0; i<len; i++) {
            switch (temp[i]) {
                case '|': INDEX("<br><i>"); italics_open = TRUE; break;
                case '<': {
                    int j = i+1;
                    INDEX(" ");
                    while ((j<len) && (temp[j] != '>')) j++;
                    temp[j] = 0;
                    index_doc_link(temp+i+1);
                    i = j;
                    break;
                }
                case '[': {
                    int j = i+1;
                    while ((j<len) && (temp[j] != ']')) j++;
                    temp[j] = 0;
                    index_below_link(temp+i+1);
                    INDEX(" ");
                    i = j;
                    break;
                }
                default: INDEX("%c", temp[i]); break;
            }
        }
        if (italics_open) INDEX("</i>");
        INDEX("</p>\n");
    }
    html_header_complete(ifl, title, (cti_set)?current_thumbnail_image:NULL);
```

```
    ⟨Start keeping track of references to source text in extensions 3⟩;
    indexing_stage = TRUE;
}
```

The function set_thumbnail_image is called from 14/i6t.
The function open_index_file is called from 11/ina, 12/rb and 14/i6t.

§**2.** Written index files are closed either when the next one is opened (see above), or when the `.i6t` interpreter signals the end of indexing by calling `complete_index` below.

```
void complete_index(void) {
    if (ifl) close_index_file();
}
void close_index_file(void) {
    if (ifl == NULL) return;
    ⟨Add footnotes for references to source text in extensions 4⟩;
    html_footer(ifl);
    STREAM_CLOSE(ifl); ifl = NULL;
}
```

The function complete_index is invoked by a command in a `.i6t` template file.

§**3. Links to source and extension footnotes.** When index files need to reference source text material, they normally do so by means of orange back-arrow icons which are linked to positions in the source as typed by the user. But source text also comes from extensions. We don't want to provide source links to those, because they can't easily be opened in the Inform application (on some platforms, anyway), and in any case, can't easily be modified (or should not be, anyway). Instead, we produce footnotes. Each time we start an index file, we wipe the slate clean:

⟨Start keeping track of references to source text in extensions 3⟩ ≡

```
    ef_clear_extension_footnotes();
```

This code is used in §1.

§**4.** And at the foot of each index file, we provide anchors `#fn_1`, `#fn_2`, ..., to footnotes explaining which extension was being referred to:

⟨Add footnotes for references to source text in extensions 4⟩ ≡

```
    extension_file *ef;
    int fnc = 0, n;
    INDEX("<p>");
    for (n=1; n<=ef_no_of_footnoted_extensions(); n++) {
        ef = ef_by_footnote(n);
        if (ef) {
            if (++fnc == 1) INDEX("<hr><p>");
            INDEX("<a name=fn_%d><img border=0 src=inform:/Revealext.png>. Defined in ",
                n, n);
            /* INDEX("<a name=fn_%d><font color=\"#800000\">[E%d]</font>. Defined in ",
                n, n); */
            ef_write_full_title_to_file(ef, ifl);
            INDEX("<br>");
        }
    }
```

This code is used in §2.

§**5.** So, then, source links are omitted if the reference is to a location in the Standard Rules; if it is to an extension other than that, the link is replaced by a square-bracketed footnote about this; and otherwise we make a link to the source text in the application.

```
void index_link(source_location sl) {
    extension_file *ef = sf_get_extension_corresponding(sl.file_of_origin);
    if (ef) {
        if (ef != standard_rules_extension) {
            /* int fn = ef_get_footnote(ef);
            INDEX(" <small><a href=#fn_%d><font color=\"#800000\">", fn);
            INDEX("[E%d]</font></a></small>", fn); */
            INDEX(" ");
            begin_extension_link(ifl, ef_get_eid(ef), NULL);
            INDEX("<img border=0 src=inform:/Revealext.png>");
            end_extension_link(ifl, ef_get_eid(ef));
        }
        return;
    }
    html_source_link(ifl, sl);
}
```

The function index_link is called from 2/lexi, 4/ext, 4/vm, 4/head, 5/rel, 5/aph, 7/dim, 7/kix, 9/qty, 9/inpw, 9/tmap, 10/tab, 10/eqns, 10/fig, 10/sfx, 10/exf, 11/act, 11/nap, 11/av, 12/phin, 12/tiph, 12/br, 12/rb and 13/gl.

§**6. Links to detail pages.** The "Beneath" icon is used for links to details pages seen as beneath the current index page: for instance, for the link from the Actions page to the page about the taking action.

```
void index_detail_link(char *stub, int sub, int down) {
    INDEX(" <a href=%s%d_%s.html><img border=0 src=inform:/Beneath.png></a>",
        (down)?"Details/":"", sub, stub);
}
```

The function index_detail_link is called from 11/act, 11/ina, 12/rb and 13/gl.

§**7. Links to documentation.** The blue query icons link to pages in the documentation, as described above. Documentation references are used to match the documentation text against the NI compiler so that each can be changed independently of the other. First, we handle the Standard-Rules-only sentences which specify documentation references:

```
sentence_handler DOC_SH_handler =
    { SENTENCE_NT, DOC_VB, 1, dref_new };
void dref_new(parse_node *p) {
    int sym1, sym2, ref1, ref2;
    [[sym1, sym2 <-- p->down->next]];
    [[ref1, ref2 <-- p->down->next->next]];
    documentation_ref *dr = CREATE(documentation_ref);
    dr->symbol = sym1+1;
    dr->section = ref1;
    dr->used_already = FALSE;
    dr->chapter_reference = NULL;
    dr->section_reference = NULL;
    if ((ref2 >= ref1+1) && (vocab_test_flags(ref1+1, TEXT_MC))) {
        dequote_word(ref1+1);
        dr->chapter_reference = lw_array[ref1+1].lw_text;
```

```
    }
    if ((ref2 >= ref1+2) && (vocab_test_flags(ref1+2, TEXT_MC))) {
        dequote_word(ref1+2);
        dr->section_reference = lw_array[ref1+2].lw_text;
    }
}
```

§**8.** The following routine is used to verify that a given text is, or is not, a valid documentation reference symbol. (For instance, we might look up `kind_vehicle` to see if any section of documentation has been flagged as giving information on vehicles.) If our speculative link symbol exists, we return the leafname for this documentation page, without filename extension (say `doc24`); if it does not exist, we return NULL.

```
char *dref_validate_if_possible(char *temp) {
    documentation_ref *dr;
    LOOP_OVER(dr, documentation_ref)
        if (compare_word_by_strcmp(dr->symbol, temp))
            return lw_array[dr->symbol].lw_text;
    return NULL;
}
```

The function dref_validate_if_possible is called from 9/inpw.

§**9.** And similarly, though case-sensitively this time, returning the page we link to:

```
char *dref_link_if_possible_once(char *temp, char **chap, char **sec) {
    documentation_ref *dr;
    LOOP_OVER(dr, documentation_ref)
        if (compare_raw_word_by_strcmp(dr->symbol, temp))
            if (dr->used_already == FALSE) {
                char *leaf = lw_array[dr->section].lw_text;
                *chap = dr->chapter_reference;
                *sec = dr->section_reference;
                LOOP_OVER(dr, documentation_ref)
                    if (strcmp(leaf, lw_array[dr->section].lw_text) == 0)
                        dr->used_already = TRUE;
                return leaf;
            }
    return NULL;
}
```

The function dref_link_if_possible_once is called from 2/prob3.

§**10.** In the Standard Rules, a number of phrases (and other constructs) are defined along with markers to sections in the documentation: here we parse these markers, returning either the word number of the documentation symbol in question, or −1 if there is none.

```
int dref_position_of_symbol(int *w1, int *w2, int front) {
    int x1 = *w1, x2 = *w2;
    if (x2 < x1+4) return -1;
    if (front == FALSE) x1 = x2-4;
    if [[x1, x2 == OPENBRACKET documented at ### CLOSEBRACKET]] {
        if (front) *w1 += 5; else *w2 -= 5;
        return x1+3;
    }
    if [[x1, x2 == DASHDASH documented at ### DASHDASH]] {
        if (front) *w1 += 5; else *w2 -= 5;
        return x1+3;
    }
    return -1;
}
```

The function dref_position_of_symbol is called from 8/creat, 9/qty, 11/av and 12/ph.

§**11.** Finally, the blue "see relevant help page" icon links are placed by the following routine. The internal error will, of course, never occur, but the reader will draw his own conclusion from the way it is so very communicative if it does.

```
void index_doc_link(char *fn) {
    documentation_ref *dr;
    int c = 0;
    LOOP_OVER(dr, documentation_ref) {
        c++;
        if (compare_raw_word_by_strcmp(dr->symbol, fn)) {
            INDEX(" <a href=inform:/%s.html><img border=0 src=inform:/help.png></a>",
                lw_array[dr->section].lw_rawtext);
            return;
        }
    }
    LOG("Bad ref was <%s>. %d known references are:\n", fn, c);
    LOOP_OVER(dr, documentation_ref) {
        LOG("<%s> = %s\n", lw_array[dr->symbol].lw_rawtext, lw_array[dr->section].lw_rawtext);
    }
    internal_error("Bad index documentation reference");
}
```

The function index_doc_link is called from 2/lexi, 4/ext, 7/kix, 9/qty, 9/inpw, 10/tab, 10/eqns, 10/bib, 10/fig, 10/sfx, 10/exf, 11/nap, 12/phin and 12/rb.

§**12. "See below" links.** These are the grey magnifying glass icons. The links are done by internal href links to anchors lower down the same HTML page. These can be identified either by number, or by name: whichever is more convenient for the indexing code.

```
void index_below_link(char *p) {
    INDEX(" <a href=#%s><img border=0 src=inform:/Below.%s></a>",
        p, ICON_EXT);
}
void index_anchor(char *p) {
    INDEX("<a name=%s></a>", p);
}
void index_below_link_numbered(int n) {
    INDEX(" <a href=#A%d><img border=0 src=inform:/Below.%s></a>", n,
        ICON_EXT);
}
void index_anchor_numbered(int n) {
    INDEX("<a name=A%d></a>", n);
}
```

The function index_below_link is called from 9/inpw, 12/phin and 13/gv.

The function index_anchor is called from 2/lexi, 4/ext, 4/vm, 9/qty, 9/inpw, 9/tmap, 10/bib, 11/act, 11/nap, 11/ina, 12/phin and 13/gl.

The function index_below_link_numbered is called from 2/lexi, 7/kix, 9/tmap and 12/phin.

The function index_anchor_numbered is called from 2/lexi, 7/kix, 9/tmap and 12/phin.

§**13. Miscellaneous utilities.** First: to print a double-quoted word into the index, without its surrounding quotes.

```
void index_dequote(char *p) {
    int i = 1;
    if ((p[0] == 0) || (p[1] == 0)) return;
    for (i=1; p[i+1]; i++) {
        char c = p[i];
        switch(c) {
            case '"': INDEX("&quot;"); break;
            default: INDEX("%c", c); break;
        }
    }
}
```

The function index_dequote is called from 9/inpw.

§**14.** Next: indexing the name of a world object, supplementing it with a link to the sentence in the source text which created it.

```
void index_name_object(world_object *wo, char *before, char *after) {
    INDEX("%s", before);
    if (wo->word_ref1 < 0) INDEX("(nameless)");
    else print_raw_text_to_file(wo->word_ref1, wo->word_ref2, ifl);
    INDEX("%s", after);
    if (wo->creating_sentence != NULL)
        index_link(lw_array[wo->creating_sentence->word_ref1].lw_source);
}
```

The function index_name_object is called from 7/kix and 9/inpw.

§**15.** Finally: three routines for indexing kinds of value and data types.

```
void index_kov(kind_of_value *kov) {
    int w1, w2;
    world_object *wo = kovko_get_kind(kov);
    if (wo) {
        print_raw_text_to_file(wo->word_ref1, wo->word_ref2, ifl);
        return;
    }
    kov_get_name(kov, &w1, &w2, FALSE);
    if (w1 >= 0) print_raw_text_to_file(w1, w2, ifl);
    else INDEX("%s", kov_get_description(kov));
}
void index_lowercase_dtid(int vs) {
    int w1, w2;
    type_ID_copy_name(vs, &w1, &w2);
    if (w1 >= 0) print_text_to_file(w1, w2, ifl);
    else INDEX("%s", type_ID_get_description(vs));
}
void log_lowercase_dtid(int vs) {
    int w1, w2;
    type_ID_copy_name(vs, &w1, &w2);
    if (w1 >= 0) print_text_to_file(w1, w2, dl);
    else INDEX("%s", type_ID_get_description(vs));
}
void index_lowercase_kov(kind_of_value *kov) {
    int w1, w2;
    world_object *wo = kovko_get_kind(kov);
    if (wo) {
        print_text_to_file(wo->word_ref1, wo->word_ref2, ifl);
        return;
    }
    kov_get_name(kov, &w1, &w2, FALSE);
    if (w1 >= 0) print_text_to_file(w1, w2, ifl);
    else INDEX("%s", kov_get_description(kov));
}
```

The function index_kov is called from 9/inpw.

The function index_lowercase_dtid is called from 7/kix and 12/phin.

The function log_lowercase_dtid is called from 7/tids.

The function index_lowercase_kov is called from 9/qty and 12/phtd.

Lexicon Index 2/lexi

Purpose

To construct the Lexicon portion of the Phrasebook page of the Index, which gives brief definitions and references for nouns, adjectives and verbs used in source text for the current project.

2/lexi.§4-5 Producing the lexicon; §6-9 Stocking the lexicon; §10-11 Processing the lexicon; §12-19 Printing the lexicon out in HTML format; §20-21 The table of verbs

---

Definitions

¶1. The lexicon is the part of the Index which gives an alphabetised list of adjectives, nouns, verbs and other words which can be used in descriptions of things: it's the nearest thing to an index of the meanings inside NI. This is in one sense quite an elaborate indexing mechanism, since it brings together meanings relating to various different NI structures under a single umbrella, the "lexicon entry" structure:

```
define NOUN_LEXE 1                          a kind
define PROPER_NOUN_LEXE 2                   a world object which is not a kind
define ADJECTIVAL_PHRASE_LEXE 3             the subject of a "Definition:"
define ENUMERATED_QUANTITY_LEXE 4           e.g., "green" if colour is a kind of value and green a colour
define VERB_LEXE 5                          and ordinary verb
define ABLE_VERB_LEXE 6                     a "to be able to..." verb
define MISCELLANEOUS_LEXE 7                 a connective, article or determiner
```

¶2. We can set entries either to excerpts of words from the source, or to any collation of up to 5 vocabulary entries.

```
typedef struct lexicon_entry {
    int w1, w2;                             either the text of the entry, or -1,-1, in which case...
    struct vocabulary_entry *ve1;           ...the first word is here
    struct vocabulary_entry *ve2;           second word, or NULL
    struct vocabulary_entry *ve3;           third word, or NULL
    struct vocabulary_entry *ve4;           fourth word, or NULL
    struct vocabulary_entry *ve5;           fifth word, or NULL
    int part_of_speech;                     one of those above
    char *category;                         textual description of said, e.g., "adjective"
    struct general_pointer entry_refers_to; depending on which part of speech
    struct parse_node *verb_defined_at;     sentence where defined (verbs only)
    char *gloss_note;                       gloss on the definition, or NULL if none is provided
    char reduced_to_lower_case[32];         text converted to lower case for sorting
    struct lexicon_entry *sorted_next;      next in lexicographic order
    MEMORY_MANAGEMENT
} lexicon_entry;

lexicon_entry *sorted_lexicon = NULL;       head of list in lexicographic order
lexicon_entry *current_main_verb = NULL;    when parsing verb declarations
```

The structure lexicon_entry is private to this section.

---

§**1.** Lexicon entries are created by the following routine:

```
lexicon_entry *lexicon_new_entry(int w1, int w2) {
    lexicon_entry *lex = CREATE(lexicon_entry);
    lex->w1 = w1; lex->w2 = w2;
    lex->ve1 = NULL; lex->ve2 = NULL; lex->ve3 = NULL; lex->ve4 = NULL; lex->ve5 = NULL;
    lex->part_of_speech = MISCELLANEOUS_LEXE;
    lex->entry_refers_to = NULL_GENERAL_POINTER;
    lex->category = NULL; lex->gloss_note = NULL; lex->verb_defined_at = NULL;
    return lex;
}
```

§**2.** The next two routines provide higher-level creators for lexicon entries. The `current_main_verb` setting is used to ensure that inflected forms of the same verb are grouped together in the verbs table.

```
lexicon_entry *lexicon_new_entry_with_details(int w1, int w2, int pos,
    vocabulary_entry *ve1, vocabulary_entry *ve2, char *category, char *gloss) {
    lexicon_entry *lex = lexicon_new_entry(w1, w2);
    lex->part_of_speech = pos;
    lex->ve1 = ve1; lex->ve2 = ve2;
    lex->category = category; lex->gloss_note = gloss;
    return lex;
}
lexicon_entry *lexicon_new_main_verb(vocabulary_entry *infinitive, int part) {
    lexicon_entry *lex = lexicon_new_entry(-1, -1);
    lex->ve1 = infinitive;
    lex->part_of_speech = part;
    lex->category = "verb";
    lex->verb_defined_at = current_sentence;
    current_main_verb = lex;
    return lex;
}
```

The function lexicon_new_entry_with_details is called from 5/det.
The function lexicon_new_main_verb is called from 5/conj.

§**3.** As we've seen, a lexicon entry's text can be either a word range or a collection of vocabulary words, and it's therefore convenient to have a utility routine which extracts the name in plain text from either source.

define `MAX_TEXT_FOR_LEXICON_ENTRY (5*MAX_WORD_LENGTH+5)` *the theoretical worst-case length, plus 1*

```
void lexicon_copy_to_string(lexicon_entry *lex, char *str) {
    if (lex->w1 >= 0) print_raw_text_to_string(lex->w1, lex->w2, str);
    else {
        str[0] = 0;
        if (lex->ve1) sprintf(str+strlen(str), "%s", vocab_get_exemplar(lex->ve1, FALSE));
        if (lex->ve2) sprintf(str+strlen(str), " %s", vocab_get_exemplar(lex->ve2, FALSE));
        if (lex->ve3) sprintf(str+strlen(str), " %s", vocab_get_exemplar(lex->ve3, FALSE));
        if (lex->ve4) sprintf(str+strlen(str), " %s", vocab_get_exemplar(lex->ve4, FALSE));
        if (lex->ve5) sprintf(str+strlen(str), " %s", vocab_get_exemplar(lex->ve5, FALSE));
    }
}
```

§4. **Producing the lexicon.** The lexicon is by no means empty when the following routine is called: lexicon entries have already been created for verbs and determiners. But it doesn't yet contain nouns or adjectives.

```
void index_lexicon(void) {
    ⟨Stock the lexicon with nouns from names of kinds of object 6⟩;
    ⟨Stock the lexicon with adjectives from names of adjectival phrases 7⟩;
    ⟨Stock the lexicon with nouns from names of quantities 8⟩;
    ⟨Stock the lexicon with miscellaneous bits and pieces 9⟩;
    ⟨Create lower-case forms of all lexicon entries 10⟩;
    ⟨Sort the lexicon into alphabetical order 11⟩;
    int common_nouns_only = FALSE;
    index_anchor("LEXICON");
    INDEX("<b>Lexicon of words used in descriptions</b><p>\n");
    ⟨Explanatory head-note at the top of the lexicon 12⟩;
    ⟨Main body of the lexicon 13⟩;
    INDEX("<p><hr><p>");
    index_anchor("RELTABLE");
    INDEX("<b>Relations</b><p>\n");
    index_table_of_relations();
    INDEX("<p><hr><p>");
    index_anchor("VERBTABLE");
    INDEX("<b>Verbs used in descriptions</b><p>\n");
    ⟨Table of parts of verbs at the foot of the lexicon 20⟩;
}
```

The function index_lexicon is called from 12/phin.

§5. And here is a cut-down version which prints a lexicon of common nouns only, for the foot of the World index.

```
void index_common_nouns(void) {
    int common_nouns_only = TRUE;
    index_anchor("LEXICON");
    INDEX("<p><hr><p><b>A to Z</b><p>\n");
    ⟨Main body of the lexicon 13⟩;
}
```

The function index_common_nouns is called from 9/inpw.

§**6. Stocking the lexicon.** Despite the implication of the over-cautious code below, kinds do always have creation nodes – i.e., their names always derive from the source text.

⟨Stock the lexicon with nouns from names of kinds of object 6⟩ ≡

```
lexicon_entry *lex;
world_object *wo;
LOOP_OVER(wo, world_object)
    if (wo->word_ref1 >= 0) {
        lex = lexicon_new_entry(wo->word_ref1, wo->word_ref2);
        if (wo->kind_flag) lex->part_of_speech = NOUN_LEXE;
        else lex->part_of_speech = PROPER_NOUN_LEXE;
        lex->category = "noun";
        lex->entry_refers_to = STORE_POINTER_world_object(wo);
    }
```

This code is used in §4.

§**7.** These are adjectives set up by "Definition:".

⟨Stock the lexicon with adjectives from names of adjectival phrases 7⟩ ≡

```
lexicon_entry *lex;
adjectival_phrase *adj;
LOOP_OVER(adj, adjectival_phrase) {
    lex = lexicon_new_entry(adj->word_ref1, adj->word_ref2);
    lex->part_of_speech = ADJECTIVAL_PHRASE_LEXE;
    lex->category = "adjective";
    lex->entry_refers_to = STORE_POINTER_adjectival_phrase(adj);
}
```

This code is used in §4.

§**8.** The idea here is that if a new kind of value such as "colour" is created, then its values should be indexed as nouns – "red", "blue" and so on. (Sometimes these will also be listed separately with an adjectival sense.)

⟨Stock the lexicon with nouns from names of quantities 8⟩ ≡

```
lexicon_entry *lex;
quantity *qn;
LOOP_OVER(qn, quantity)
    if ((qty_is_a_variable(qn) == FALSE) &&
        (kov_is_an_enumeration(qty_kind_of_value(qn)))) {
        property_name *prn = kov_get_coinciding_property(qty_kind_of_value(qn));
        if ((prn) && (prn_condition_of_which_object(prn))) continue;
        lex = lexicon_new_entry(qn->word_ref1, qn->word_ref2);
        lex->part_of_speech = ENUMERATED_QUANTITY_LEXE;
        lex->category = "noun";
        lex->entry_refers_to = STORE_POINTER_quantity(qn);
    }
```

This code is used in §4.

§**9.** It seems unfitting for a dictionary to omit “a”, “an”, “the”, “some”, “which” or “who”.

⟨Stock the lexicon with miscellaneous bits and pieces 9⟩ ≡

```
lexicon_entry *lex;
lex = lexicon_new_entry(-1, -1);
lex->ve1 = which_V;
lex->part_of_speech = MISCELLANEOUS_LEXE; lex->category = "connective";
lex->gloss_note = "used to place a further condition on a description: "
    "'A which is B', or 'A which carries B', for instance.";
lex = lexicon_new_entry(-1, -1);
lex->ve1 = who_V;
lex->part_of_speech = MISCELLANEOUS_LEXE; lex->category = "connective";
lex->gloss_note = "- see </i>which<i>.";
lex = lexicon_new_entry(-1, -1);
lex->ve1 = a_V;
lex->part_of_speech = MISCELLANEOUS_LEXE; lex->category = "indefinite article";
lex = lexicon_new_entry(-1, -1);
lex->ve1 = an_V;
lex->part_of_speech = MISCELLANEOUS_LEXE; lex->category = "indefinite article";
lex->gloss_note = "- see </i>a<i>";
lex = lexicon_new_entry(-1, -1);
lex->ve1 = some_V;
lex->part_of_speech = MISCELLANEOUS_LEXE; lex->category = "indefinite article";
lex = lexicon_new_entry(-1, -1);
lex->ve1 = the_V;
lex->part_of_speech = MISCELLANEOUS_LEXE; lex->category = "definite article";
```

This code is used in §4.

§**10. Processing the lexicon.** Before we can sort the lexicon, we need to turn its disparate forms of name into a single, canonical, lower-case representation. We truncate this to at most 31 characters to keep the storage requirements reasonable: so in fact sorting of the lexicon will fail correctly to sequence name variations after the 31st character, but this seems an acceptable trade-off.

⟨Create lower-case forms of all lexicon entries 10⟩ ≡

```
lexicon_entry *lex;
LOOP_OVER(lex, lexicon_entry) {
    int i;
    char entry_text[MAX_TEXT_FOR_LEXICON_ENTRY];
    lexicon_copy_to_string(lex, entry_text); entry_text[31] = 0;
    for (i=0; entry_text[i]; i++) entry_text[i] = tolower(entry_text[i]);
    strcpy(lex->reduced_to_lower_case, entry_text);
}
```

This code is used in §4.

§**11.** The lexicon is sorted by insertion sort, which is not ideally fast, but which is convenient when dealing with linked lists: there are unlikely to be more than 1000 or so entries, so the speed penalty for insertion rather than (say) quicksort is not great.

⟨Sort the lexicon into alphabetical order 11⟩ ≡

```
lexicon_entry *lex;
LOOP_OVER(lex, lexicon_entry) {
    lexicon_entry *lex2, *last_lex;
    if (sorted_lexicon == NULL) {
        sorted_lexicon = lex; lex->sorted_next = NULL; continue;
    }
    for (last_lex = NULL, lex2 = sorted_lexicon; lex2;
        last_lex = lex2, lex2 = lex2->sorted_next)
        if (strcmp(lex->reduced_to_lower_case, lex2->reduced_to_lower_case) < 0) {
            if (last_lex == NULL) sorted_lexicon = lex;
            else last_lex->sorted_next = lex;
            lex->sorted_next = lex2; goto Inserted;
        }
    last_lex->sorted_next = lex; lex->sorted_next = NULL;
    Inserted: ;
}
```

This code is used in §4.

§12. Printing the lexicon out in HTML format.

⟨Explanatory head-note at the top of the lexicon 12⟩ ≡

```
INDEX("<small>For instance, the description 'an unlocked door' is made "
    "up from the adjective 'unlocked' and the noun 'door', both of which "
    "can be found below. Property adjectives, like 'open', can be used "
    "when creating things - 'In the Ballroom is an open container' is "
    "allowed because 'open' is a property - but those with complicated "
    "definitions, like 'empty', can only be tested during play, e.g. "
    "with rules like 'Instead of taking an empty container, ...'.</small><p>");
```

This code is used in §4.

§**13.** Now for the bulk of the work. Entries appear in CSS paragraphs with hanging indentation and no interparagraph spacing, so we need to insert regular `<p>` paragraphs between the As and the Bs, then between the Bs and the Cs, and so on. Each entry consists of the wording, then maybe some icons, then an explanation of what it is: for instance,

player's holdall [icon] *noun, a kind of* container

In a few cases, there is a further textual gloss to add.

⟨Main body of the lexicon 13⟩ ≡

```
lexicon_entry *lex;
char current_initial_letter = '?';
int verb_count = 0, entry_count = 0, c;
for (lex = sorted_lexicon; lex; lex = lex->sorted_next)
    if (lex->part_of_speech == PROPER_NOUN_LEXE)
        entry_count++;
if (common_nouns_only) {
    begin_html_table(ifl, NULL, TRUE, 0, 0, 0, 0, 0);
    first_html_column(ifl, 0);
```

```
}
for (c = 0, lex = sorted_lexicon; lex; lex = lex->sorted_next) {
    if (common_nouns_only) { if (lex->part_of_speech != PROPER_NOUN_LEXE) continue; }
    else { if (lex->part_of_speech == PROPER_NOUN_LEXE) continue; }
    if ((common_nouns_only) && (c++ == entry_count/2)) next_html_column(ifl, 0);
    if (current_initial_letter != lex->reduced_to_lower_case[0]) {
        INDEX("<p>");
        current_initial_letter = lex->reduced_to_lower_case[0];
    }
    INDEX("<p class=\"hang\">");
    ⟨Text of the actual lexicon entry 14⟩;
    ⟨Icon with link to documentation, source or verb table, if any 15⟩;
    INDEX("  <i>");
    if ((common_nouns_only == FALSE) && (lex->category))
        INDEX("%s", lex->category);
    switch(lex->part_of_speech) {
        case ADJECTIVAL_PHRASE_LEXE:
            ⟨Description of adjectival phrase entry 18⟩; break;
        case ENUMERATED_QUANTITY_LEXE:
            ⟨Description of enumerated quantity entry 19⟩; break;
        case PROPER_NOUN_LEXE:
            ⟨Description of proper noun entry 17⟩; break;
        case NOUN_LEXE:
            ⟨Description of noun entry 16⟩; break;
        default: INDEX("</i>"); break;
    }
    if (lex->gloss_note) INDEX(" <i>%s</i>", lex->gloss_note);
    INDEX("</p>");
}
if (common_nouns_only) { end_html_row(ifl); end_html_table(ifl); }
```

This code is used in §4,5,4,5,4,5.

§**14.** In traditional dictionary fashion, we present the text in what may not be the most normal ordering, in order to place the alphabetically important part first: thus "see, to be able to" rather than "to be able to see". (Compare "Gallifreyan High Council, continual incidences of madness and treachery amongst the" in *Doctor Who: The Completely Useless Encyclopaedia*, eds. Howarth and Lyons (1996).)

⟨Text of the actual lexicon entry 14⟩ ≡

```
char entry_text[MAX_TEXT_FOR_LEXICON_ENTRY];
lexicon_copy_to_string(lex, entry_text);
print_literal_string_to_file(ifl, entry_text);
if (lex->part_of_speech == ABLE_VERB_LEXE) INDEX(", to be able to");
```

This code is used in §13.

§**15.** Main lexicon entries to do with verbs link further down the index page to the corresponding entries in the verb table. We want to use numbered anchors for these links, but we want to avoid colliding with numbered anchors already used for other purposes higher up on the Phrasebook index page. So we use a set of anchors numbered 10000 and up, which is guaranteed not to coincide with any of those.

We omit source links to an adjectival phrase because these are polymorphic, that is, the phrase may have multiple definitions in different parts of the source text: so any single link would be potentially misleading.

⟨Icon with link to documentation, source or verb table, if any 15⟩ ≡

```
switch(lex->part_of_speech) {
    case NOUN_LEXE: {
        world_object *wo = RETRIEVE_POINTER_world_object(lex->entry_refers_to);
        if (wo->wo_documentation_symbol != NULL)
            index_doc_link(wo->wo_documentation_symbol);
        break;
    }
    case VERB_LEXE:
    case ABLE_VERB_LEXE:
        index_below_link_numbered(10000+verb_count++);
        break;
}
if ((lex->part_of_speech != ADJECTIVAL_PHRASE_LEXE) && (lex->w1 >= 0))
    index_link(lw_array[lex->w1].lw_source);
```

This code is used in §13.

§**16.** Recall that the `kind` of a kind points either to the kind it is a subkind of, or else to `kind_kind` if it is a kind of object and therefore at the top of the hierarchy.

⟨Description of noun entry 16⟩ ≡

```
world_object *wo = RETRIEVE_POINTER_world_object(lex->entry_refers_to);
if (wo->kind != kind_kind) {
    INDEX(", a kind of </i>");
    print_raw_text_to_file(wo->kind->word_ref1, wo->kind->word_ref2, ifl);
} else INDEX(", a kind</i>");
```

This code is used in §13.

§**17.** Simply the name of a world object.

⟨Description of proper noun entry 17⟩ ≡

```
world_object *wo = RETRIEVE_POINTER_world_object(lex->entry_refers_to);
if (wo->kind != kind_thing)
    print_raw_text_to_file(wo->kind->word_ref1, wo->kind->word_ref2, ifl);
INDEX("</i>");
```

This code is used in §13.

§**18.** As mentioned above, an adjectival phrase can be multiply defined in different contexts. We want to quote all of those.

⟨Description of adjectival phrase entry 18⟩ ≡
```
int ac = 0, nc;
adjective_meaning *am;
adjectival_phrase *aph =
    RETRIEVE_POINTER_adjectival_phrase(lex->entry_refers_to);
INDEX(": ");
LOOP_OVER_SORTED_MEANINGS(aph, am) ac++; nc = ac;
LOOP_OVER_SORTED_MEANINGS(aph, am) {
    ac--;
    if (nc > 1) INDEX("<br>%d. ", nc-ac);
    am_print_to_index(am);
    if (ac >= 1) INDEX("; ");
}
INDEX("</i>");
```
This code is used in §13.

§**19.** Lastly and most easily, the name of an enumerated value of some kind of value.

⟨Description of enumerated quantity entry 19⟩ ≡
```
int w1, w2;
quantity *qn = RETRIEVE_POINTER_quantity(lex->entry_refers_to);
kind_of_value *kov = qty_kind_of_value(qn);
INDEX(", value of </i>");
kov_get_name(kov, &w1, &w2, FALSE);
print_raw_text_to_file(w1, w2, ifl);
```
This code is used in §13.

§**20. The table of verbs.** This is used in two different ways: firstly, at the foot of the lexicon –

⟨Table of parts of verbs at the foot of the lexicon 20⟩ ≡
```
lexicon_entry *lex = sorted_lexicon;
int verb_count = 0;
for (lex = sorted_lexicon; lex; lex = lex->sorted_next)
    if ((lex->part_of_speech == VERB_LEXE) || (lex->part_of_speech == ABLE_VERB_LEXE)) {
        char entry_text[MAX_TEXT_FOR_LEXICON_ENTRY];
        INDEX("<p class=\"hang\">");
        index_anchor_numbered(10000+verb_count++);                    anchors from 10000: see above
        lexicon_copy_to_string(lex, entry_text);
        if (lex->part_of_speech == VERB_LEXE) INDEX("To <b>%s</b>", entry_text);
        else INDEX("To be able to <b>%s</b>", entry_text);
        if (lex->w1 >= 0) index_link(lw_array[lex->w1].lw_source);
        INDEX("</p>");
        tabulate_verb(lex, IS_TENSE, "present");
        tabulate_verb(lex, WAS_TENSE, "past");
        tabulate_verb(lex, HASBEEN_TENSE, "present perfect");
        tabulate_verb(lex, HADBEEN_TENSE, "past perfect");
    }
```
This code is used in §4.

§**21.** – and secondly, in the documentation for extensions, where we want to be able to print out a table of just those verbs created in that extension.

```
void lexicon_list_verbs_in_file(OUTPUT_STREAM, source_file *sf, extension_file *ef) {
    int verb_count = 0;
    lexicon_entry *lex;
    LOOP_OVER(lex, lexicon_entry)
        if (((lex->part_of_speech == VERB_LEXE) || (lex->part_of_speech == ABLE_VERB_LEXE))
            && (lex->verb_defined_at)
            && (lw_array[lex->verb_defined_at->word_ref1].lw_source.file_of_origin == sf)) {
            char entry_text[MAX_TEXT_FOR_LEXICON_ENTRY];
            lexicon_copy_to_string(lex, entry_text);
            if (verb_count++ == 0) WRITE("Verbs: "); else WRITE(", ");
            if (lex->part_of_speech == VERB_LEXE) WRITE("to <b>%s</b>", entry_text);
            else WRITE("to be able to <b>%s</b>", entry_text);
            new_dictionary_entry_str("verb", ef, entry_text);
        }
    if (verb_count > 0) WRITE("<p>");
}
```

The function lexicon_list_verbs_in_file is called from 4/edoc.

Progress Percentages 2/prog

Purpose

This tiny section, the Lichtenstein of Inform, prints percentage of completion estimates onto `stderr` so that the host application can intercept them and update its graphical progress bar.

---

§**1.** Clearly we can only estimate how far Inform has progressed. While we could in principle measure the number of CPU-seconds it has run for, we don't know how many more it will need. Instead, we rely on experience to suggest that its run can be broken down into the following five stages, which a given percentage of time spent in each stage: thus, for instance, semantic analysis takes up about ten percent of every run in which problems do not cause an early halt. Within each stage, we have a reasonable measure of how far we have got: what proportion of the phrases have been compiled, for instance, tells us how far "generating code" has got. The result is that (if the relevant command line setting has been set, so that `show_progress_indicator` is true) Inform prints about thirty lines like this one to `stderr`:

```
++ 32% (Binding rulebooks)
```

The Inform application can intercept and parse these lines to display a progress bar with a rubric beneath it.

```
int last_progress_pc = -100;
int progress_stage_from[] = { 0, 5, 15, 20, 40, 100 };
char *progress_stage_name[] = { "Lexical analysis", "Semantic analysis",
    "Drawing inferences", "Binding rulebooks", "Generating code" };

void progress_bar(int stage, float proportion) {
    int r1 = progress_stage_from[stage], r2 = progress_stage_from[stage+1];
    int pc = r1 + ((int) (proportion*(r2-r1)));
    if (show_progress_indicator == FALSE) return;
    if (pc-last_progress_pc < 3) return;
    STREAM_WRITE(STDERR, "++ %d%% (%s)\n", pc, progress_stage_name[stage]);
    STREAM_FLUSH(STDERR);
    last_progress_pc = pc;
}
```

The function progress_bar is called from 12/cs, 12/ph and 14/i6t.

Problems, Level 0 2/prob0

Purpose
To handle fatal errors.

---

§**1.** In my beginning is my end: this lowest level of the error-handling system deals with systemic collapses.

```
void fatal_error(char *message) {
    STREAM_WRITE(STDERR, message);
    STREAM_WRITE(STDERR, "\n");
    STREAM_FLUSH(STDERR);
    exit(2);
}
void fatal_error2(char *message, char *fn) {
    STREAM_WRITE(STDERR, message);
    STREAM_WRITE(STDERR, "\nOffending filename: <%s>\n", fn);
    STREAM_FLUSH(STDERR);
    exit(2);
}
```

The function fatal_error is called from 2/mem, 2/strm, 2/dl, 3/lex, 4/ext, 4/excen, 12/phtd, 12/cinv, 12/cph, 14/main and 14/i6t.
The function fatal_error2 is called from 2/index, 2/prob3, 2/dl, 3/read, 4/ext, 4/excen, 4/edict, 4/edoc, 4/head, 9/map, 10/bib and 14/i6t.

§**2.** Fatal errors are not necessarily a bad thing. When tracking down why NI issues certain problem messages (especially internal errors) it can be useful to provoke a deliberate crash of the application, in order to get a stack backtrace into the GNU debugger `gdb` (and/or onto the system console logs). We can force this using the following variables (which main sets with the command-line switch "-gdb").

```
int crash_on_all_errors = FALSE;
int crash_on_internal_errors = FALSE;
void force_crash(void) {
    STREAM_FLUSH(STDOUT);
    STREAM_FLUSH(dl);
    STREAM_WRITE(STDERR,
        "*** Intentionally crashing to force stack backtrace to console logs ***\n");
    STREAM_FLUSH(STDERR);
    parse_node *PN = NULL; log_subtree(PN->next, 1);
    exit(1);
}
```

The function force_crash is called from 2/prob2 and 2/prob3.

Problems, Level 1 2/prob1

Purpose

To render problem messages either as plain text or HTML, and write them out to files.

---

§**1.** Only one error text needs to be stored in memory at any one time, so we keep it in a single string array called the `problem_buffer`. Horrible things will happen if the bounds of this buffer are ever broken, so it has been made very large.

```
define PROBLEM_BUFFER_LENGTH 10000
define PBUFF problem_buffer+strlen(problem_buffer)

char problem_buffer[PROBLEM_BUFFER_LENGTH];
```

§**2.** Roughly speaking, text for error messages comes from two possible sources: fairly short standard texts inside NI, and quotations (direct or indirect) from the source text. The latter are inserted by the routines in this section, and they are the ones we should be wary of, since bizarre input might cause dangerously long quotations to be made. A quotation involves copying text from word `w1` to `w2`, so there are two dangers: copying a single very long word, or copying too many of them. We protect against the first by using the `print_modest_sized_text_to_string` routine, which truncates long literals. We protect against the second overflow hazard by limiting the amount of text we are prepared to quote from any sentence in one go:

```
define QUOTATION_TOLERANCE_LIMIT 100

void copy_text_into_problem_buffer(int w1, int w2) {
    if (w2 > w1 + QUOTATION_TOLERANCE_LIMIT) w2 = w1 + QUOTATION_TOLERANCE_LIMIT - 1;
    print_modest_sized_text_to_string(w1, w2, PBUFF);
}
```

The function copy_text_into_problem_buffer is called from 2/prob2.

§**3.** References to source code need careful handling because they may need to be transcribed both as HTML links (in the `Problems.html` file) and as plain text (to `stderr` and the debugging log). See "HTML Files" above.

```
void copy_source_reference_into_problem_buffer(int w1, int w2) {
    char *filename, *paraphrase;
    if (w1 < 0) { sprintf(PBUFF, "<no text>"); return; }
    source_file *referred = lw_array[w1].lw_source.file_of_origin;
    if (referred) {
        filename = sf_get_filename(referred);
        if (bundle_name && (strncmp(filename,bundle_name,strlen(bundle_name)) == 0))
            filename += strlen(bundle_name) + 1;
    } else filename = "(no file)";
    sprintf(PBUFF, "'");
    copy_text_into_problem_buffer(w1, w2);
    paraphrase = "source text";
    extension_file *ef = sf_get_extension_corresponding(referred);
    if (ef) {
        extension_identifier *eid = ef_get_eid(ef);
        paraphrase = filename;
        if (eid) {
```

```
            if (eid_is_standard_rules(eid)) paraphrase = "the Standard Rules";
        }
    }
    sprintf(PBUFF, "' %c%s%c%s%c%d%c",
        SOURCE_REF_CHAR, paraphrase,
        SOURCE_REF_CHAR, filename,
        SOURCE_REF_CHAR, lw_array[w1].lw_source.line_number,
        SOURCE_REF_CHAR);
}
```

The function copy_source_reference_into_problem_buffer is called from 2/prob2.

§**4.** We allow for three whitespace characters in the error buffer, though in fact we will then handle any sequence of them as if it were a single space character:

```
int is_problem_buffer_whitespace(char c) {
    if ((c == ' ') || (c == '\t') || (c == '\n')) return TRUE;
    return FALSE;
}
```

§**5.** Once the error message is fully constructed, we will want to output it to a file: in fact, by default it will go in three directions, to `stderr`, to the debugging log and of course to the error log. The main thing is to word-wrap it, since it is likely to be a paragraph-sized chunk of text, not a single line. The unprintable `SOURCE_REF_CHAR` and `FORCE_NEW_PARA_CHAR` are simply filtered out for plain text output: for HTML, they are dealt with elsewhere.

```
void output_problem_buffer_to(OUTPUT_STREAM, int indentation) {
    int i, k, line_width = 0, html_flag = FALSE;
    if (OUT == problems_file) html_flag = TRUE;
    for (k=0; k<indentation; k++) { WRITE("  "); line_width+=2; }
    for (i=0; problem_buffer[i] != 0; i++) {
        char c = problem_buffer[i];
        ⟨In HTML mode, convert drawing-your-attention arrows 6⟩;
        ⟨In plain text mode, remove bold and italic HTML tags 7⟩;
        if ((html_flag == FALSE) && (c == SOURCE_REF_CHAR))
            ⟨Issue plain text paraphrase of source reference 8⟩
        else ⟨Output single character of problem message 9⟩;
    }
    if (html_flag) WRITE("<p>");
    WRITE("\n");
}
```

§**6.** The plain text "may I draw your attention to the following paragraph" marker,

```
>--> Which looks like this.
```

is converted into a suitable CSS-styled HTML paragraph with hanging indentation. And similarly for `>++>`, used to mark continuations.

⟨In HTML mode, convert drawing-your-attention arrows 6⟩ ≡

```
if ((html_flag) &&
    (problem_buffer[i] == '>') &&
    (problem_buffer[i+1] == '-') &&
    (problem_buffer[i+2] == '-') &&
    (problem_buffer[i+3] == '>')) {
    if (problem_count > 1) WRITE("</p><hr>");
    WRITE("<p class=\"hang\"><b>Problem.</b> ");
    i+=3; continue;
}
if ((problem_buffer[i] == '>') &&
    (problem_buffer[i+1] == '+') &&
    (problem_buffer[i+2] == '+') &&
    (problem_buffer[i+3] == '>')) {
    if (html_flag) WRITE("<p class=\"tightin2\">"); else WRITE("  ");
    i+=3; continue;
}
```

This code is used in §5.

§**7.** The problem messages are put together (by Level 2 below) in a plain text way, but with a little formatting included: in particular, they contain HTML-style `<i>` and `<b>` tags, which the following code strips out when writing to plain text format.

⟨In plain text mode, remove bold and italic HTML tags 7⟩ ≡

```
if ((problem_buffer[i] == '<') &&
    ((problem_buffer[i+1] == 'i') || (problem_buffer[i+1] == 'b')) &&
    (problem_buffer[i+2] == '>')) {
    if (html_flag) WRITE("<%c>", problem_buffer[i+1]);
    i+=2; continue;
}
if ((problem_buffer[i] == '<') &&
    (problem_buffer[i+1] == '/') &&
    ((problem_buffer[i+2] == 'i') || (problem_buffer[i+2] == 'b')) &&
    (problem_buffer[i+3] == '>')) {
    if (html_flag) WRITE("</%c>", problem_buffer[i+2]);
    i+=3; continue;
}
```

This code is used in §5.

§**8.** Okay, so the format for a source reference here is:

`XparaphraseXfilenameXnumberX`

e.g., `Xmain textXsource/story.niX102`, where `X` is the unprintable `SOURCE_REF_CHAR`. The counter `i` is at the first `X`, and we must now convert this to something fit for printing to `stdout`, finishing up with `i` pointing to the last `X`.

We always use the paraphrase, not the filename, on `stdout` because (i) that's slightly easier to understand for the user, but more importantly (ii) it makes the output the same on all platforms when only main text and Standard Rules are referred to, and that simplifies `intest` and the Test Suite quite a bit, because we don't have to worry about trivial differences between OS X and Windows caused by the slashes going the wrong way, and so on.

⟨Issue plain text paraphrase of source reference 8⟩ ≡

```
WRITE("("); line_width++;
while (problem_buffer[++i] != SOURCE_REF_CHAR) ⟨Output single character of problem message 9⟩;
while (problem_buffer[++i] != SOURCE_REF_CHAR) ;
WRITE(", line "); line_width += 7;
while (problem_buffer[++i] != SOURCE_REF_CHAR) ⟨Output single character of problem message 9⟩;
WRITE(")"); line_width++;
```

This code is used in §5.

§**9.**

⟨Output single character of problem message 9⟩ ≡

```
c = problem_buffer[i];
if (is_problem_buffer_whitespace(c)) {                                  this starts a run of whitespace
    int k = i; while (is_problem_buffer_whitespace(problem_buffer[k])) k++;
    if (problem_buffer[k] == 0) break;                                  omit any trailing spaces
    i = k - 1;                                                          skip to final whitespace character of the run
    if (html_flag) html_char_out(OUT, ' ');
    else ⟨In plain text mode, wrap the line or print a space as necessary 10⟩;
} else {
    line_width++;
    if (html_flag) html_char_out(OUT, c);
    else if ((c != SOURCE_REF_CHAR) && (c != FORCE_NEW_PARA_CHAR)) WRITE("%c", c);
}
```

This code is used in §5,8,5,8,5,8.

§**10.** At this point, `k` is the position of the first non-whitespace character after the sequence of whitespace.

define `PROBLEM_WORD_WRAP_WIDTH 80`

⟨In plain text mode, wrap the line or print a space as necessary 10⟩ ≡

```
int word_width = 0;
while ((!is_problem_buffer_whitespace(problem_buffer[k])) && (problem_buffer[k] != 0)
    && (problem_buffer[k] != SOURCE_REF_CHAR))
    k++, word_width++;
if (line_width + word_width + 1 >= PROBLEM_WORD_WRAP_WIDTH) {
    WRITE("\n"); line_width = 0;
    for (k=0; k<indentation+1; k++) { line_width+=2; WRITE("  "); }
} else {
    WRITE(" "); line_width++;
}
```

This code is used in §9.

§**11.** The following handles the three-way distribution of problems, but also allows us to route individual messages to only one output of our choice by temporarily setting the `probl` variable: which is a convenience for informational messages such as appear in index files, for instance.

```
int telemetry_recording = FALSE;

void output_problem_buffer(int indentation) {
    if (probl == NULL) {
        output_problem_buffer_to(problems_file, indentation);
        STREAM_WRITE(problems_file, "\n");
        output_problem_buffer_to(STDOUT, indentation);
        STREAM_WRITE(dl, "\n");
        output_problem_buffer_to(dl, indentation);
        STREAM_WRITE(dl, "\n");
        if (telemetry_recording) {
            STREAM_WRITE(telmy, "\n");
            output_problem_buffer_to(telmy, indentation);
            STREAM_WRITE(telmy, "\n");
        }
    } else output_problem_buffer_to(probl, indentation);
}
```

The function output_problem_buffer is called from 2/prob2.

Problems, Level 2 2/prob2

Purpose

To assemble and format problem messages within the problem buffer.

2/prob2.§6-12 Problem quotations; §13-14 Short and long forms; §15 How problems begin and end; §16-20 Issuing a segment of a problem message; §21-24 Problems report and index

---

Template interpreter commands

```
1   {-callv:empty_all_headings}
21  {-callv:complete_problems_report}
```

---

§**1.** Problem messages begin with an indication of where in the source text the problem occurs, in terms of headings and subheadings written in the source text, and it is this indication that we consider first. For example,

> In Part the First, Chapter 1 - Attic Area:

There can be up to 10 levels in the hierarchy of headings and subheadings, with level 0 the top level and level 9 the lowest. Recall that for a `HEADING_NT` type node, the `flag` field contains this level number. The current hierarchical position can be stored in the following array:

```
parse_node *problem_headings[NO_HEADING_LEVELS];
void empty_all_headings(void) {
    empty_headings(0);
}
void empty_headings(int from_level) {
    int i;
    for (i=from_level; i<NO_HEADING_LEVELS; i++) problem_headings[i] = NULL;
}
```

The function empty_all_headings is invoked by a command in a `.i6t` template file.

The function empty_headings is called from 4/iext.

§**2.** When we need to issue a problem at sentence `S`, we work out what the current heading is (if any) at each of the 10 levels. We do this by trekking right through the whole linked list of sentences until we reach `S`, changing the current headings whenever we pass one. This sounds inefficient, but of course few problems are issued, and in any case we cannot optimise by simply cacheing the heading level from one problem to the next because it is not true that problems are always issued in source code order.

(We must not use the `TREE_START` and `TREE_NEXT` macros for the traverse below, as they would change the value of `current_sentence`.)

```
void find_headings_at(parse_node *sentence) {
    parse_node *p;
    empty_headings(0);
    if (sentence == NULL) return;
    for (p = tree_root->down; p != NULL; p = p->next) {
        if (pn_contains(p, sentence)) return;
        if (pn_get_node_type(p) == HEADING_NT) {
            problem_headings[pn_int_annotation(p, heading_level_ANNOT)] = p;
            empty_headings(pn_int_annotation(p, heading_level_ANNOT) + 1);
        }
        if (pn_get_node_type(p) == ENDHERE_NT) {
            empty_all_headings();
        }
    }
}
```

§**3.** A further refinement is that we remember the last set of headings used for an error message, and only mention what has changed about the location. Thus we might next print:

> In Chapter 2 - Cellar Area:

omitting to mention "Part the First" this time, since that part has not changed. (And we never print internally made level 0, File, headings.)

```
parse_node *last_problem_headings[NO_HEADING_LEVELS];
```

§**4.** Now for the actual displaying of the location text. The outcome image is a trickier thing to get right than might appear. By being in this routine, we know that a problem has been issued: the run will therefore not have been a success, and we can issue the "NI failed" outcome image. A success image cannot be placed until right at the end of the run, when all possibility of problems has been passed: so there's no single point in NI where a single line of code could choose between the two possibilities.

```
int do_not_locate_problems = FALSE;

void show_problem_location(void) {
    int i, f = FALSE;
    if (problem_count == 0) {
        html_outcome_image(problems_file, "ni_failed");
        for (i=0; i<NO_HEADING_LEVELS; i++) last_problem_headings[i] = NULL;
    }
    if (do_not_locate_problems) return;
    find_headings_at(current_sentence);
    for (i=0; i<NO_HEADING_LEVELS; i++) if (problem_headings[i] != NULL) f = TRUE;
    if (f)
        for (i=1; i<NO_HEADING_LEVELS; i++)
            if (last_problem_headings[i] != problem_headings[i]) {
```

```
                ⟨Print the heading position 5⟩;
                break;
            }
    for (i=0; i<NO_HEADING_LEVELS; i++) last_problem_headings[i] = problem_headings[i];
}
```

§**5.** We print only the part of the heading position which differs from that of the previous one quoted: `i` is at this point the highest level at which they differ.

⟨Print the heading position 5⟩ ≡

```
    sprintf(problem_buffer, "In<b>");
    for (f=FALSE; i<NO_HEADING_LEVELS; i++)
        if (problem_headings[i] != NULL) {
            if (f) strcat(problem_buffer, ", ");
            else strcat(problem_buffer, " ");
            f = TRUE;
            copy_text_into_problem_buffer(
                problem_headings[i]->word_ref1, problem_headings[i]->word_ref2);
        }
    strcat(problem_buffer, "</b>:");
    output_problem_buffer(0);
```

This code is used in §4.

§**6. Problem quotations.** Problem messages are printed using a printf-like formatting system. Unlike printf, though, where `%s` means a string and `%d` a number, here the escape codes do not indicate the type of the data: they are simply `%1`, `%2`, `%3`, ..., `%9`. This is to prevent horrendous crashes when type mismatches occur: using a pointer to a phrase when trying to print a source code reference, for instance.

The texts to be substituted in place of `%1`, `%2`, ..., are called the "quotations". The value is either a range of words in the source text, or else a pointer to some data structure, depending on the type. The type is a single character code. (This coding system is used only here, and could easily be changed, but there seems no reason to.)

The type codes are S(ource), W(ords), B(inary predicate), E(xtension), (p)H(rase), I(nvocation), N(umber), O(bject), P(roperty name), T(ext), (t)Y(pe specification).

```
typedef struct problem_quotation {
    char quotation_type;                                   one of the above
    void *structure_quoted;                  except for S and W, when this is null
    int range_quoted_w1, range_quoted_w2;                  for S and W only
} problem_quotation;

problem_quotation problem_quotations[10];
```

The structure problem_quotation is private to this section.

§**7.** When some higher-level part of NI wants to issue a formatted problem message, it first declares the contents of any quotations it will make. It does this using the routines `quote_object`, `quote_spec`, ... below. Thus `quote_spec(2, SP)` specifies that `%2` should be printed as the inference `SP`.

```
void problem_quote(int t, char type, void *v) {
    if ((t<0) || (t > 10)) internal_error("problem quotation number out of range");
    problem_quotations[t].structure_quoted = v;
    problem_quotations[t].quotation_type = type;
    problem_quotations[t].range_quoted_w1 = -1;
    problem_quotations[t].range_quoted_w2 = -1;
}
void problem_quote_textual(int t, char type, int w1, int w2) {
    if ((t<0) || (t > 10)) internal_error("problem quotation number out of range");
    problem_quotations[t].structure_quoted = NULL;
    problem_quotations[t].quotation_type = type;
    problem_quotations[t].range_quoted_w1 = w1;
    problem_quotations[t].range_quoted_w2 = w2;
}
```

§**8.** Here are the three public routines for quoting from text: either via a node in the parse tree, or with a literal word range.

```
void quote_source(int t, parse_node *p) {
    if (p == NULL) problem_quote_textual(t, 'S', -1, -1);
    else problem_quote_textual(t, 'S', p->word_ref1, p->word_ref2);
}
void quote_source_eliding_begin(int t, parse_node *p) {
    if (p == NULL) problem_quote_textual(t, 'S', -1, -1);
    else {
        int w1 = p->word_ref1, w2 = p->word_ref2;
        [[w1, w2 == ... begin --> w1, w2]];
        problem_quote_textual(t, 'S', w1, w2);
    }
}
void quote_words(int t, int w1, int w2) { problem_quote_textual(t, 'W', w1, w2); }
```

The function quote_source is called from 2/prob3, 2/dl, 4/sent, 4/ext, 4/iext, 4/rtree, 5/rel, 5/litp, 5/mlc, 6/treec, 6/asp, 6/defer, 7/data, 7/tc, 8/tass, 8/creat, 8/mass, 8/knowc, 8/knowp, 9/qty, 9/kind, 9/prop, 9/pp, 10/tab, 10/eqns, 10/bib, 11/act, 11/ap, 11/av, 12/phud, 12/phtd, 12/phod, 12/stv, 12/cinv, 12/rb, 12/rps, 13/gv, 13/gtok and 13/test.

The function quote_source_eliding_begin is called from 4/rtree.

The function quote_words is called from 2/prob3, 2/dl, 4/ext, 4/iext, 5/rel, 5/litp, 6/treec, 6/tcpr, 6/asp, 6/defer, 7/kov, 7/data, 7/ttts, 7/tc, 8/tass, 8/creat, 8/mass, 8/knowp, 9/qty, 9/prop, 9/pp, 10/str, 10/eqns, 11/act, 11/ap, 11/av, 12/ph, 12/phud, 12/phod, 12/stv, 12/cinv, 12/rb, 12/rps and 13/gtok.

§**9.** And here are the public routines for quoting structures in NI:

```
void quote_relation(int t, binary_predicate *bp) { problem_quote(t, 'B', (void *) bp); }
void quote_extension(int t, extension_file *ef) { problem_quote(t, 'E', (void *) ef); }
void quote_phrase(int t, phrase *p) { problem_quote(t, 'H', (void *) p); }
void quote_invocation(int t, invocation *inv) { problem_quote(t, 'I', (void *) inv); }
void quote_number(int t, int *num) { problem_quote(t, 'N', (void *) num); }
void quote_object(int t, world_object *wo) { problem_quote(t, 'O', (void *) wo); }
void quote_property(int t, property_name *p) { problem_quote(t, 'P', (void *) p); }
void quote_text(int t, char *message) { problem_quote(t, 'T', (void *) message); }
void quote_spec(int t, specification *spec) { problem_quote(t, 'Y', (void *) spec); }
```

The function quote_relation is called from 6/tcpr.
The function quote_extension is called from 2/prob3, 4/ext and 4/iext.
The function quote_phrase is called from 12/phod.
The function quote_invocation is called from 7/tc.
The function quote_number is called from 2/prob3, 5/litp and 7/tc.
The function quote_object is called from 2/prob3, 7/kov, 7/tc, 8/mass, 8/knowc, 9/kind, 9/model and 11/ap.
The function quote_property is called from 2/prob3, 6/treec, 7/tc, 8/knowp, 9/pp, 9/cmpbp, 9/vpbp and 9/model.
The function quote_text is called from 2/prob3, 5/rel, 5/litp, 6/defer, 7/kov, 7/tc, 8/mass, 9/pp, 10/eqns, 11/act, 11/ap, 12/phud, 12/cinv, 13/gv and 13/gtok.
The function quote_spec is called from 7/tc, 8/mass, 8/knowp, 9/pp and 10/eqns.

§**10.** We also provide refinements for various ways to talk about kinds of value (or data types) within problem messages. First, we quote not a literal constant but its kind of value, changing (say) the actual constant 15 to the generic constant "number":

```
void quote_type_of(int t, specification *spec) {
    if ((spec) && (species_is(spec, CONSTANT_SPC)))
        spec = new_generic_CONSTANT_type(spec_get_kind_of_value(spec));
    quote_spec(t, spec);
}
```

The function quote_type_of is called from 7/tc, 8/mass, 8/knowp, 9/prop, 10/tab and 13/gtok.

§**11.** Relatedly, and for the case of a SP which evaluates to a description of one or more objects:

```
void quote_kind_of(int t, specification *spec) {
    if (spec_get_described_kind(spec)) { quote_object(t, spec_get_described_kind(spec)); return; }
    if (spec_get_described_object(spec)) { quote_object(t, spec_get_described_object(spec)->kind); return;
}
    if (spec_is_CONSTANT_of_kova(spec, OBJECT_TY)) {
        world_object *wo = OBJECT_spec_to_world_object(spec);
        quote_object(t, wo->kind);
        return;
    }
    quote_type_of(t, spec);
}
```

The function quote_kind_of is called from 12/cinv.

§**12.** To quote a kind of value, we take care of the two special cases separately: firstly when it's the null KOV (meaning "of unknown type"), and secondly when it's a kind of object. There's no actual need to treat this second case separately, but it does mean that we can print a more helpful description: "kind of container", say, rather than "kind of object".

```
void quote_kov(int t, kind_of_value *kov) {
    if (kov == NULL) { quote_spec(t, new_UNKNOWN_spec()); return; }
    if (is_kova(kov, OBJECT_TY)) {
        world_object *k = kovko_get_kind(kov);
        if (k) { quote_object(t, k); return; }
    }
    quote_spec(t, new_generic_CONSTANT_type(kov));
}
```

The function quote_kov is called from 5/litp, 6/treec, 6/tcpr, 6/equal, 7/kov, 7/tc, 8/creat, 8/mass, 9/qty, 9/pp, 9/provr, 9/vpbp, 10/qnbp, 10/tab and 10/eqns.

§**13. Short and long forms.** Most of the error messages have a short form, giving the main factual information, and a long form which also has an explanation attached. Some of the text is common to both versions, but other parts are specific to either one or the other, and variables record the status of our current position in scanning and transcribing the problem message.

```
int shorten_problem_message;                    give short form of this previously-seen problem
int reading_text_specific_to_short_version;     modes during problem message writing
int reading_text_specific_to_long_version;
```

§**14.** We do not want to keep wittering on with the same old explanations, so we remember which ones we've given before (i.e., in previous problem messages during the same run of NI). Eventually our patience is exhausted and we give no further explanation in any circumstances.

```
define PATIENCE_EXHAUSTION_POINT 100

char *explanations[PATIENCE_EXHAUSTION_POINT];
int no_explanations = 0;

int explained_before(char *explanation) {
    int i;
    if (no_explanations == PATIENCE_EXHAUSTION_POINT) return TRUE;
    for (i=0; i<no_explanations; i++)
        if (explanation == explanations[i]) return TRUE;
    explanations[no_explanations++] = explanation;
    return FALSE;
}
```

§15. How problems begin and end. During the construction of a problem message, we will be running through a standard text, and at any point might be considering matter which should appear only in the long form, or only in the short form.

If the text of a message begins with an asterisk, then it is a continuation of a message already partly issued. Otherwise we can sensibly find out whether this is one we've seen before. Either way, we set `shorten_problem_message` to remember whether to use the short or long form.

```
void issue_problem_begin(char *message) {
    problem_buffer[0] = 0;
    if (strcmp(message, "*") == 0) {
        strcpy(problem_buffer, ">++>");
        shorten_problem_message = FALSE;
    } else if (strcmp(message, "**") == 0) {
        shorten_problem_message = FALSE;
    } else {
        show_problem_location();
        problem_count++;
        strcpy(problem_buffer, ">--> ");
        shorten_problem_message = explained_before(message);
    }
    reading_text_specific_to_short_version = FALSE;
    reading_text_specific_to_long_version = FALSE;
}
void issue_problem_end(void) {
    if (compiling_text_routines_mode) append_text_routine_proviso();
    output_problem_buffer(1);
    problem_documentation_links(problems_file);
    if (crash_on_all_errors) force_crash();
}
```

The function issue_problem_begin is called from 2/prob3 and 7/tc.

The function issue_problem_end is called from 2/prob3, 2/dl, 4/sent, 4/ext, 4/iext, 4/rtree, 5/rel, 5/litp, 5/mlc, 6/treec, 6/asp, 6/defer, 7/kov, 7/data, 7/tc, 8/tass, 8/creat, 8/mass, 8/knowc, 8/knowp, 9/qty, 9/kind, 9/prop, 9/pp, 9/model, 10/tab, 10/eqns, 10/bib, 11/act, 11/ap, 11/av, 12/phud, 12/phtd, 12/phod, 12/stv, 12/cinv, 12/rb, 12/rps, 13/gv, 13/gtok and 13/test.

§16. Issuing a segment of a problem message. Here is the routine which performs the substitution of quotations into problem messages and sends them on their way: which is called `issue_problem_segment` since it only appends a further piece of text, and may be used several times to build up complicated messages.

We have seen that, within problem message texts, the escapes `%1` to `%9` are to produce quotations. Four further escape codes switch between problem message versions, as follows: we have `%L`, which indicates that the text which follows should only be used in the long form of the error message; `%S`, the same for the short form; `%%`, which cancels either of these settings and restores the text to appearing in both forms, which is the default. A percentage sign followed by a vertical stroke acts more simply as a divider between the brief message and the explanation text in many simple problem messages which do not use the elaborations of the other three escapes.

```
void issue_problem_segment(char *message) {
    int i;
    for (i=0; i<strlen(message); i++) {
        if (message[i] == '%') {
            switch (message[i+1]) {
                case 'L': reading_text_specific_to_long_version = TRUE; i++; continue;
                case 'S': reading_text_specific_to_short_version = TRUE; i++; continue;
```

```
                case '%': reading_text_specific_to_short_version = FALSE;
                    reading_text_specific_to_long_version = FALSE; i++; continue;
                case '|': reading_text_specific_to_short_version = FALSE;
                    reading_text_specific_to_long_version = TRUE;
                    if (shorten_problem_message) strcat(problem_buffer, ".");
                    i++; continue;
            }
        }
        if ((reading_text_specific_to_short_version) &&
            (shorten_problem_message == FALSE)) continue;
        if ((reading_text_specific_to_long_version) &&
            (shorten_problem_message == TRUE)) continue;
        ⟨Act on the problem message text, since it is now contextually allowed 17⟩;
    }
}
```

The function issue_problem_segment is called from 2/prob3, 2/dl, 4/sent, 4/ext, 4/iext, 4/rtree, 5/rel, 5/litp, 5/mlc, 6/treec, 6/asp, 6/defer, 7/kov, 7/data, 7/tc, 8/tass, 8/creat, 8/mass, 8/knowc, 8/knowp, 9/qty, 9/kind, 9/prop, 9/pp, 9/model, 10/str, 10/tab, 10/eqns, 10/bib, 11/act, 11/ap, 11/av, 12/ph, 12/phud, 12/phtd, 12/phod, 12/stv, 12/cinv, 12/rb, 12/rps, 13/gv, 13/gtok and 13/test.

§**17.** Ordinarily we just append the new character, but we also act on the escapes `%P` and `%1` to `%9`. `%P` forces a paragraph break, or at any rate, it does in the eventual HTML version of the problem message. Note that these escapes are acted on only if they occur in a contextually allowed part of the problem message (e.g., if they occur in the short form only, they will only be acted on when the shortened form is the one being issued).

⟨Act on the problem message text, since it is now contextually allowed 17⟩ ≡

```
int len;
if (message[i] == '%') {
    switch (message[i+1]) {
        case 'P': sprintf(problem_buffer+strlen(problem_buffer), "%c",
            FORCE_NEW_PARA_CHAR);
            i++; continue;
    }
    if (isdigit(message[i+1])) {
        int t = ((int) (message[i+1]))-((int) '0'); i++;
        if ((t>=1) && (t<=9)) ⟨Expand numbered problem message escape 18⟩;
        continue;
    }
}
len = strlen(problem_buffer);
problem_buffer[len] = message[i]; problem_buffer[len+1] = 0;
```

This code is used in §16.

§**18.** This is where a quotation escape, such as %2, is expanded: by looking up its type, stored internally as a single character.

⟨Expand numbered problem message escape 18⟩ ≡

```
property_name *pr; phrase *ph; specification *spec; extension_file *ef; invocation *inv;
switch(problem_quotations[t].quotation_type) {
    Text range-based quotations
    case 'S': copy_source_reference_into_problem_buffer(
            problem_quotations[t].range_quoted_w1, problem_quotations[t].range_quoted_w2);
        break;
    case 'W': copy_text_into_problem_buffer(
            problem_quotations[t].range_quoted_w1, problem_quotations[t].range_quoted_w2);
        break;
    Binary structure-based quotations
    case 'B': ⟨Quote a binary predicate in a problem message 19⟩;
        break;
    case 'E':
        ef = (extension_file *) (problem_quotations[t].structure_quoted);
        ef_write_full_title_to_string(ef, problem_buffer+strlen(problem_buffer));
        break;
    case 'H':
        ph = (phrase *) (problem_quotations[t].structure_quoted);
        ph_write_HTML_representation(ph, problem_buffer+strlen(problem_buffer));
        break;
    case 'I':
        inv = (invocation *) (problem_quotations[t].structure_quoted);
        inv_write_HTML_representation(inv, problem_buffer+strlen(problem_buffer));
        break;
    case 'N':
        sprintf(problem_buffer+strlen(problem_buffer), "%d",
            *((int *) (problem_quotations[t].structure_quoted)));
        break;
    case 'O': ⟨Quote a world object in a problem message 20⟩;
        break;
    case 'P': pr = (property_name *) problem_quotations[t].structure_quoted;
        copy_text_into_problem_buffer(pr->word_ref1, pr->word_ref2);
        break;
    case 'T': strcat(problem_buffer, (char *) (problem_quotations[t].structure_quoted));
        break;
    case 'Y': spec = (specification *) problem_quotations[t].structure_quoted;
        write_specification_out_in_English(spec, problem_buffer+strlen(problem_buffer));
        break;
    default: internal_error("unknown error token type");
}
```

This code is used in §17.

§**19.** This one we delegate:

⟨Quote a binary predicate in a problem message 19⟩ ≡

```
binary_predicate *bp = (binary_predicate *) problem_quotations[t].structure_quoted;
bp_describe_for_problems(bp, problem_buffer);
```

This code is used in §18.

§**20.** Since numerous world objects are created without explicit and distinct names, for instance by sentences like

Four coins are in the box.

...it's prudent to quote objects without names carefully, and not to ignore this as some kind of marginal will-never-happen case.

⟨Quote a world object in a problem message 20⟩ ≡

```
world_object *wo = (world_object *) problem_quotations[t].structure_quoted;
if (wo == NULL) break;
if (wo->word_ref1 >= 0)
    copy_text_into_problem_buffer(wo->word_ref1, wo->word_ref2);
else {
    strcat(problem_buffer, "object I am looking at (");
    strcat(problem_buffer, "a nameless ");
    if ((wo->kind) && (wo->kind->word_ref1 >= 0))
        copy_text_into_problem_buffer(
            wo->kind->word_ref1, wo->kind->word_ref2);
    else strcat(problem_buffer, "thing");
    if (wo->creating_sentence != NULL) {
        strcat(problem_buffer, " created in the sentence ");
        copy_source_reference_into_problem_buffer(wo->creating_sentence->word_ref1,
            wo->creating_sentence->word_ref2);
    }
    strcat(problem_buffer, ")");
}
```

This code is used in §18.

§**21. Problems report and index.** That gives us enough infrastructure to produce the final report. Note the use of error redirection to `probl` in order to put pseudo-problem messages – actually informational – into the report. In the case where the run was successful and there we no Problem messages, we have to be careful to reset `problem_count` back down to 0 – it will have been increased by the issuing of these pseudo-problems, and we need it to remain 0 so that `main()` can finally return back to the operating system without an error code.

```
void complete_problems_report(void) {
    write_reports(FALSE);
}
void write_reports(int disaster_struck) {
    int total_words;
    crash_on_all_errors = FALSE;
    if (problem_count > 0) {
        probl = problems_file;
        issue_problem_begin("*");
        if (disaster_struck) ⟨Issue problem summary for an internal error 22⟩
        else ⟨Issue problem summary for a run with problem messages 23⟩;
        issue_problem_end();
        probl = NULL;
    } else {
        html_outcome_image(problems_file, "ni_succeeded");
        quote_number(1, &(kind_room->instance_count));
        if (kind_room->instance_count == 1) quote_text(2, "room");
        else quote_text(2, "rooms");
```

```
        quote_number(3, &(kind_thing->instance_count));
        if (kind_thing->instance_count == 1) quote_text(4, "thing");
        else quote_text(4, "things");
        total_words = sf_total_word_count(FIRST_OBJECT(source_file));
        quote_number(5, &total_words);
        ⟨Issue problem summaries for a run without problems 24⟩;
        problem_count = 0;
    }
}
```

The function complete_problems_report is invoked by a command in a `.i6t` template file.
The function write_reports is called from 2/prob3.

§**22.** One of the slightly annoying things about internal errors is that Inform's users persistently refer to them as "crashes" on bug report forms. I mean, the effort we go to! They are entirely clean exits from the program! The ingratitude of some – oh, all right.

⟨Issue problem summary for an internal error 22⟩ ≡

```
    issue_problem_segment(
        "What has happened here is that one of the checks Inform carries "
        "out internally, to see if it is working properly, has failed. "
        "There must be a bug in this copy of Inform. It may be worth "
        "checking whether you have the current, up-to-date version. "
        "If so, please report this problem using the Bug Report Form "
        "posted at www.inform-fiction.org. %P"
        "As for fixing your source text to avoid this bug, the last thing "
        "you changed is probably the cause, if there is a simple cause. "
        "Your source text might in fact be wrong, and the problem might be "
        "occurring because Inform has failed to find a good way to say so. "
        "But even if your source text looks correct, there are "
        "probably rephrasings which would achieve the same effect.");
```

This code is used in §21.

§**23.** Singular and plural versions of the same message, really:

⟨Issue problem summary for a run with problem messages 23⟩ ≡

```
    if (problem_count == 1)
        issue_problem_segment(
            "Because of this problem, the source could not be translated "
            "into a working game. (Correct the source text to "
            "remove the difficulty and click on Go once again.)");
    else
        issue_problem_segment(
            "Problems occurring in translation prevented the game "
            "from being properly created. (Correct the source text to "
            "remove these problems and click on Go once again.)");
```

This code is used in §21.

§**24.** The success message needs to take different forms in `stdout` and in the Problems log file. In the latter, we write as though the subsequent conversion of NI's output to a story file via Inform 6 had already been completed successfully – this is because the Problems log is intended to be viewed inside the Inform application, which will instead divert to an error page if I6 should fail. So although the Problems file contains an unwarranted claim, if not an actual falsehood, no human eye should see it unless and until it comes true.

We don't want to make similar claims on `stdout`, where the user – who might well not be running in the Inform application, but only on the command line – deserves the truth.

⟨Issue problem summaries for a run without problems 24⟩ ≡

```
probl = problems_file;
issue_problem_begin("**");
issue_problem_segment(
    "The %5-word source text has successfully been translated "
    "into a world with %1 %2 and %3 %4, and the index has been "
    "brought up to date.");
issue_problem_end();

if (telemetry_recording) {
    ensure_telemetry_file();
    probl = telmy;
    issue_problem_begin("**");
    issue_problem_segment(
        "The %5-word source text has successfully been translated "
        "into a world with %1 %2 and %3 %4, and the index has been "
        "brought up to date.");
    issue_problem_end();
    STREAM_WRITE(telmy, "\n");
}

probl = STDOUT;
STREAM_WRITE(STDOUT, "\n");
issue_problem_begin("**");
issue_problem_segment(
    "The %5-word source text has successfully been translated "
    "into an intermediate description which can be run through "
    "Inform 6 to complete compilation. There were %1 %2 and %3 %4.");
issue_problem_end();
STREAM_FLUSH(STDOUT);
probl = NULL;
```

This code is used in §21.

Problems, Level 3 2/prob3

Purpose

Here we provide some convenient semi-standardised problem messages, which also serve as examples of how to use the Level 2 problem message routines.

---

§**1.** The following internal errors should never occur, whatever the provocation: if they do, they are symptoms of incorrect code in NI.

The internal error "functions" used by the rest of NI are in fact macros, in order that they can supply the current filename and line number automatically to the actual internal error functions. The result is, for instance,

> Problem. An internal error has occurred: Unknown verb code. The current sentence is "A room is a kind"; the error was detected at line 133 of "Chapter 5/Traverse for Objects.w". This should never happen, and I am now halting in abject failure.

```
define internal_error(X) internal_error_fn(X, __FILE__, __LINE__)
define internal_error_tree_unsafe(X) internal_error_tu_fn(X, __FILE__, __LINE__)
define nodal_error(X, Y) nodal_error_fn(X, Y, __FILE__, __LINE__)
define internal_error_if_node_type_wrong(X, Y) nodal_check(X, Y, __FILE__, __LINE__)
define internal_error_on_node_type(X) internal_error_on_node_type_fn(X, __FILE__, __LINE__)
```

§**2.** Internal errors are generated much like any other problem message, except that we use a variant form of the "end" routine which salvages what it can from the wreckage, then either forces a crash (to make the stack backtrace visible in a debugger) or simply exits to the operating system with error code 1:

```
void internal_error_end(void) {
    issue_problem_end();
    log_mlc_backtrace();
    write_reports(TRUE);
    if (crash_on_internal_errors) force_crash();
    exit(1);
}
```

§**3.** And now for the functions which the above macros invoke. There are two versions: one which cites the current sentence, and another which doesn't, for use if either there is no current sentence (because NI wasn't traversing the parse tree at the time) or if the parse tree is unsafe – it's possible that the internal error occurred during parse tree construction, so we need to be cautious.

```
void internal_error_fn(char *p, char *filename, int linenum) {
    if (current_sentence == NULL) {
        internal_error_tu_fn(p, filename, linenum);
        return;
    }
    quote_source(1, current_sentence);
    quote_text(2, p);
    quote_text(3, filename);
    quote_number(4, &linenum);
    issue_problem_begin(p);
    issue_problem_segment(
        "An internal error has occurred: %2. The current sentence is %1; the "
        "error was detected at line %4 of \"%3\". This should never happen, "
        "and I am now halting in abject failure.");
    internal_error_end();
}
void internal_error_tu_fn(char *p, char *filename, int linenum) {
    quote_text(1, p);
    quote_text(2, filename);
    quote_number(3, &linenum);
    issue_problem_begin(p);
    issue_problem_segment(
        "An internal error has occurred: %1. The error was detected at "
        "line %3 of \"%2\". This should never happen, and I am now halting "
        "in abject failure.");
    internal_error_end();
}
```

§**4. Nodal errors.** Very many routines are designed to work only on nodes within the parse tree of a particular node type. If NI is in working order, then they will never be called at any other nodes; but it seems best to check this. Any failure of such an invariant produces a form of internal error called a "nodal error".

```
void nodal_error_fn(parse_node *pn, char *p, char *filename, int linenum) {
    LOG("Internal nodal error at:\n");
    log_subtree(pn, 1);
    internal_error_fn(p, filename, linenum);
}
```

§**5.** Here is a convenient function to check said invariant.

```
void nodal_check(parse_node *pn, int node_type_required, char *filename, int linenum) {
    char internal_message[128];
    if (pn == NULL) {
        sprintf(internal_message, "NULL node found where type %s expected",
            pnt_get_name(node_type_required));
        internal_error_tu_fn(internal_message, filename, linenum);
    }
    if (pn_get_node_type(pn) == node_type_required) return;
    if ((pn_get_node_type(pn) < 1) || (pn_get_node_type(pn) > HIGHEST_NODE_TYPE)) {
        sprintf(internal_message, "Node with corrupt type found where type %s expected",
            pnt_get_name(node_type_required));
        internal_error_tu_fn(internal_message, filename, linenum);
    }
    sprintf(internal_message, "Node of type %s found where type %s expected",
        pnt_get_name(pn_get_node_type(pn)),
        pnt_get_name(node_type_required));
    nodal_error_fn(pn, internal_message, filename, linenum);
}
```

§**6.** Nodal errors also turn up as the default clauses in switch statements which act on various selections of node types, and those use the `internal_error_on_node_type` macro, which invokes the following:

```
void internal_error_on_node_type_fn(parse_node *pn, char *filename,
    int linenum) {
    char internal_message[128];
    if (pn == NULL)
        internal_error_tu_fn("Unexpected NULL node found", filename, linenum);
    if ((pn_get_node_type(pn) < 1) || (pn_get_node_type(pn) > HIGHEST_NODE_TYPE))
        internal_error_tu_fn("Node found with corrupt type", filename, linenum);
    sprintf(internal_message, "Unexpectedly found node of type %s",
        pnt_get_name(pn_get_node_type(pn)));
    nodal_error_fn(pn, internal_message, filename, linenum);
}
```

§**7.** The following routines are relics of an era of horrific, primordial upheaval, when the meaning list conversion code was being debugged. (An S-subtree is a portion of a meaning list which represents a sentence.)

```
meaning_list *latest_s_subtree = NULL;
void s_subtree_error_set_position(meaning_list *ml) {
    latest_s_subtree = ml;
}
void s_subtree_error(char *mess) {
    char internal_message[128];
    sprintf(internal_message, "S-subtree error: %s", mess);
    LOG("%s", internal_message);
    if (latest_s_subtree) LOG("Applied to the subtree:\n$m", latest_s_subtree);
    internal_error(internal_message);
}
```

The function s_subtree_error_set_position is called from 5/mlc and 6/sconv.
The function s_subtree_error is called from 5/mlc and 6/sconv.

§**8. Sigils.** Every problem message in Inform is identified by a *sigil*, a short alphanumeric symbol. The `_P_` notation is used to write these; see almost every section in later chapters for examples. The naming rules for sigils are as follows:

(a) A problem which is thought never to be generated has the sigil `BelievedImpossible`. Inform is quite defensively coded, so there are several dozen of these – they are safety nets to catch cases we didn't think of.

(b) A problem which either cannot be tested by `intest`, or is just impracticable to do so, has the sigil `Untestable`.

(c) A problem which can be tested, but for which nobody has yet written a test case, has the sigil ... (these are gradually declining in number, and eventually, of course, will disappear altogether).

(d) Otherwise a problem should have a unique sigil beginning `C` and then the chapter number in which it is found: say, `C6NoSuchHieroglyph`. The sigil should have the same name as an `intest` test case which demonstrates the problem.

(e) A sigil which ends `-G` should be used for those few problems which appear only when the virtual machine is Glulx.

It would be easy for all this to fall out of sync, or for us just to lose track of odd cases, since there are more than 750 problem messages; so a shell script called `listproblems.sh` exists to verify that the above rules have been adhered to.

§**9.** As can be seen, `_P_` is a macro expanding to the sigil's name in double quotes followed by the source section and line number at which it is generated. This provides three function arguments matching the `SIGIL_ARGUMENTS` prototype, which appears as a pseudo-argument in all of the problem routines below.

Each such routine should either `ACT_ON_SIGIL` itself or else pass over to another problem routine, using `PASS_SIGIL` as the pseudo-argument.

```
define _P_(sigil) #sigil, __FILE__, __LINE__

define SIGIL_ARGUMENTS char *sigil, char *file, int line

define ACT_ON_SIGIL
    LOG("Problem %s issued from %s, line %d\n", sigil, file, line);
    if (telemetry_recording) {
        ensure_telemetry_file();
        STREAM_WRITE(telmy, "Problem %s issued from %s, line %d\n", sigil, file, line);
    }
    sigil_of_latest_problem = sigil;
    if (echo_problem_message_sigils) printf("Problem:: %s\n", sigil);

define PASS_SIGIL sigil, file, line

char *sigil_of_latest_problem = NULL;

void problem_documentation_links(OUTPUT_STREAM) {
    if (sigil_of_latest_problem == NULL) return;
    char *chap = NULL, *sec = NULL;
    char *leaf = dref_link_if_possible_once(sigil_of_latest_problem, &chap, &sec);
    if (leaf) {
        open_html_paragraph(OUT, 2, "tight");
        WRITE("<a href=inform:/%s.html><img border=0 src=inform:/help.png></a>", leaf);
        if ((chap) && (sec)) {
            WRITE(" <i>See the manual: %s &gt; %s</i></p>\n", chap, sec);
        } else {
            WRITE(" <i>See the manual.</i></p>\n");
        }
        if (telemetry_recording) {
```

```
            STREAM_WRITE(telmy, "See the manual: %s > %s\n\n", chap, sec);
        }
    }
    sigil_of_latest_problem = NULL;
}
```

The function problem_documentation_links is called from 2/prob2.

§**10.** The command-line switch `-sigils` causes the following flag to be set, which in turn causes the sigil of any problem to be echoed to standard output (i.e., printed). This is useful in testing, as it makes it easier to be sure that the test case `C6NoSuchHieroglyph.txt` does indeed generate the problem `C6NoSuchHieroglyph`, and so on.

```
int echo_problem_message_sigils = FALSE;
```

§**11. Handmade problems.** Those made without using the convenient shorthand forms below:

```
void handmade_problem(SIGIL_ARGUMENTS) {
    ACT_ON_SIGIL
    issue_problem_begin("");
}
```

The function handmade_problem is called from 2/dl, 4/sent, 4/ext, 4/iext, 4/rtree, 5/rel, 5/litp, 5/mlc, 6/treec, 6/asp, 6/defer, 7/kov, 7/data, 7/tc, 8/tass, 8/creat, 8/mass, 8/knowc, 8/knowp, 9/qty, 9/kind, 9/prop, 9/pp, 9/model, 10/tab, 10/eqns, 10/bib, 11/act, 11/ap, 11/av, 12/phud, 12/phtd, 12/phod, 12/stv, 12/cinv, 12/rb, 12/rps, 13/gv, 13/gtok and 13/test.

§**12. Limit problems.** Running out of memory, irretrievably: the politest kind of fatal error, though let's face it, fatal is fatal.

```
void limit_problem(SIGIL_ARGUMENTS, char *what_has_run_out, int how_many) {
    ACT_ON_SIGIL
    quote_text(1, what_has_run_out);
    quote_number(2, &how_many);
    issue_problem_begin("");
    issue_problem_segment(
        "I have run out of memory for %1 - there's room for %2, but no more. "
        "This is a 'hard limit', hard in the sense of deadlines, or luck: "
        "there is no getting around it. You will need to rewrite your source "
        "text so that it needs fewer %1.");
    issue_problem_end();
    write_reports(FALSE);
    exit(1);
}
void memory_allocation_problem(SIGIL_ARGUMENTS, char *what_has_run_out) {
    ACT_ON_SIGIL
    quote_text(1, what_has_run_out);
    issue_problem_begin("");
    issue_problem_segment(
        "I am unable to persuade this computer to let me have memory in "
        "which to store %1. This rarely happens on a modern desktop of laptop, "
        "but might occur on a small handheld device - if so, it may be a "
        "symptom that the device isn't powerful enough to run me. (See how "
        "I pass the blame?)");
```

```
    issue_problem_end();
    write_reports(FALSE);
    exit(1);
}
```

The function limit_problem is called from 4/ofs and 11/chron.
The function memory_allocation_problem is called from 2/mem.

§**13. Problem messages unlocated in the source text.** And now the regular problem messages, the ones which are *not my fault*. We begin with lexical problems happening when the run is hardly begun:

```
void lexical_problem(SIGIL_ARGUMENTS, char *message, char *concerning, char *exp) {
    ACT_ON_SIGIL
    char *lexical_explanation =
        "This is a low-level problem happening when I am still reading in the "
        "source. Such problems sometimes arise because I have been told to "
        "read a source file which is not text at all.";
    if (exp != NULL) lexical_explanation = exp;
    quote_text(1, message);
    quote_text(2, concerning);
    quote_text(3, lexical_explanation);
    issue_problem_begin(lexical_explanation);
    issue_problem_segment("%1: %2%L.%%%| %3");
    issue_problem_end();
}
```

The function lexical_problem is called from 3/lex.

§**14.** Clearly lexical problems cannot cite positions in the source text, and some other problems can't either, so:

```
void unlocated_problem(SIGIL_ARGUMENTS, char *message) {
    ACT_ON_SIGIL
    do_not_locate_problems = TRUE;
    issue_problem_begin(message);
    issue_problem_segment(message);
    issue_problem_end();
    do_not_locate_problems = FALSE;
}
```

The function unlocated_problem is called from 9/model and 14/i6t.

§**15. Problem messages keyed to positions in the source text.** The following routine is used to produce more than 300 different problem messages, making it the most prolific of all the problem routines: perhaps that isn't surprising, since it simply quotes the entire sentence at fault (which is always the current sentence) and issues a message.

```
void sentence_problem(SIGIL_ARGUMENTS, char *message, char *explanation) {
    ACT_ON_SIGIL
    quote_source(1, current_sentence);
    quote_text(2, message);
    quote_text(3, explanation);
    issue_problem_begin(explanation);
    issue_problem_segment("You wrote %1: %Sagain, %%%Lbut %%%2%|, %3");
    issue_problem_end();
}
```

The function sentence_problem is called from 2/isn, 2/lns, 3/read, 3/lwb, 4/iext, 4/head, 4/verb, 4/ofs, 4/rtree, 5/rel, 5/conj, 5/aph, 5/litp, 5/em, 5/unitr, 5/lit, 5/candp, 5/mlc, 6/sconv, 6/tcpr, 6/equal, 6/asp, 6/atoms, 6/defer, 6/cdefp, 7/dim, 7/ttts, 7/cmsp, 7/tc, 8/tass, 8/sob, 8/refpt, 8/creat, 8/mass, 8/assem, 8/knowp, 8/relv, 8/imp, 9/qty, 9/scene, 9/kind, 9/spabp, 9/mapbp, 9/exrel, 9/prop, 9/pp, 9/vpbp, 9/inf, 9/model, 9/cot, 10/str, 10/tab, 10/eqns, 10/bib, 10/fig, 10/sfx, 10/exf, 10/isin, 11/chron, 11/act, 11/ap, 11/av, 12/ph, 12/phud, 12/phrcd, 12/phtd, 12/phod, 12/phsf, 12/cinv, 12/def, 12/br, 12/rb, 13/tfg, 13/gv, 13/gl, 13/gty, 13/gtok, 13/test and 14/i6t.

§**16.** And a variant which adds a note in a subsequent paragraph.

```
void sentence_problem_with_note(SIGIL_ARGUMENTS,
        char *message, char *explanation, char *note) {
    ACT_ON_SIGIL
    quote_source(1, current_sentence);
    quote_text(2, message);
    quote_text(3, explanation);
    quote_text(4, note);
    issue_problem_begin(explanation);
    issue_problem_segment("You wrote %1: %Sagain, %%%Lbut %%%2%|, %3 %P%4");
    issue_problem_end();
}
```

The function sentence_problem_with_note is called from 5/litp and 7/tc.

§**17.** And this is a variant which draws particular attention to a word range which is part of the current sentence.

```
void sentence_in_detail_problem(SIGIL_ARGUMENTS,
        int w1, int w2, char *message, char *explanation) {
    ACT_ON_SIGIL
    quote_source(1, current_sentence);
    quote_text(2, message);
    quote_text(3, explanation);
    quote_words(4, w1, w2);
    issue_problem_begin(explanation);
    issue_problem_segment(
        "You wrote %1, and in particular '%4': %Sagain, %%%Lbut %%%2%|, %3");
    issue_problem_end();
}
```

The function sentence_in_detail_problem is called from 7/tc.

§**18.** A not always helpful problem message which is needed in several places, and therefore is kept here:

```
void negative_sentence_problem(SIGIL_ARGUMENTS) {
    sentence_problem(PASS_SIGIL,
        "assertions about the initial state of play must be positive, not negative",
        "so 'The cat is an animal' is fine but not 'The cat is not a container'. "
        "I have only very feeble powers of deduction - sometimes the implications "
        "of a negative statement are obvious to a human reader, but not to me.");
}
```

The function negative_sentence_problem is called from 4/verb and 8/mass.

§**19.** Similarly:

```
void kind_of_object_problem(SIGIL_ARGUMENTS) {
    sentence_problem(PASS_SIGIL,
        "you aren't allowed to create kinds of 'object'",
        "because that is too vague a description. If what you need is "
        "a kind for places or items to appear during play, then you should "
        "create a 'kind of thing', a 'kind of room', a 'kind of man', etc., "
        "as appropriate (see the Kinds index for a list). If you want values "
        "rather than physical things, make a 'kind of value'. (Technically, "
        "it is also possible to make something just a 'kind', but this is not "
        "at all recommended unless you know exactly what you are doing.)");
}
```

The function kind_of_object_problem is called from 8/refpt and 8/mass.

§**20.** This is a much more elaborate form of the standard `sentence_problem`, used when an assertion sentence has gone wrong. Experience from the early builds of the Public Beta showed that many people tried syntaxes which Inform did not recognise, and which cause Inform to misread the primary verb of the sentence. It would then issue a Problem – because the sentence would be peculiar – but this problem report would itself be odd, and make little sense to the user. So we look to see if the current sentence is an assertion with a primary verb: and if it is, we hunt through it for alternative verbs which might have been intended, and try to produce a message which diagnoses the problem rather better.

```
void assertion_problem(SIGIL_ARGUMENTS, char *message, char *explanation) {
    int i, w1, w2, rather_than_w1 = -1, rather_than_w2 = -1;
    ACT_ON_SIGIL
    if ((current_sentence == NULL) || (current_sentence->down == NULL) ||
        (pn_get_node_type(current_sentence->down) != VERB_NT)) {
        LOG("(Assertion error reverting to sentence error.)\n");
        sentence_problem(PASS_SIGIL, message, explanation);
        return;
    }
    w1 = current_sentence->word_ref1;
    w2 = current_sentence->word_ref2;
    LOG("(Assertion error: looking for alternative verbs in <$W>.)\n", w1, w2);
    for (i=w1+1; i<=w2-1; i++)
        if (i != current_sentence->down->word_ref1) {
            verb_usage *vu;
            LOOP_OVER(vu, verb_usage) {
                if ((vu_get_tense_used(vu) == IS_TENSE) &&
                    (unexpectedly_upper_case(i) == FALSE)) {
```

```
                    int j = vu_parse_text_against(i, w2, vu);
                    if (j >= 0) {
                        rather_than_w1 = i;
                        rather_than_w2 = j-1;
                        LOG("(Verb phrase <$W> found.)\n",
                            rather_than_w1, rather_than_w2);
                    }
                }
            }
        }
    quote_source(1, current_sentence);
    quote_text(2, message);
    quote_text(3, explanation);
    issue_problem_begin(explanation);
    issue_problem_segment("You wrote %1: %Sagain, %%%Lbut %%%2%|, %3");
    if (rather_than_w1 >= 0) {
        quote_words(4, current_sentence->down->word_ref1,
            current_sentence->down->word_ref2);
        quote_words(5, rather_than_w1, rather_than_w2);
        issue_problem_segment(                                    see also C2AmbiguousVerb
            " %P(It may help to know that I am reading the primary verb here "
            "as '%4', not '%5'.)");
    }
    issue_problem_end();
}
```

The function assertion_problem is called from 6/treec, 8/creat and 8/mass.

§**21. Definition problems.** Sentence problems are a nuisance for "Definition:" definitions, because those usually occur when the current sentence is rather unhelpfully just the word "Definition" alone. So we use this routine instead:

```
void definition_problem(SIGIL_ARGUMENTS, parse_node *q,
        char *message, char *explanation) {
    ACT_ON_SIGIL
    quote_source(1, q);
    quote_text(2, message);
    quote_text(3, explanation);
    issue_problem_begin(explanation);
    issue_problem_segment("You gave as a definition %1: %Sagain, %%%Lbut %%%2%|, %3");
    issue_problem_end();
}
void adjective_problem(SIGIL_ARGUMENTS, int ix1, int ix2, int d1, int d2,
        char *message, char *explanation) {
    ACT_ON_SIGIL
    quote_words(1, ix1, ix2);
    quote_words(2, d1, d2);
    quote_text(3, message);
    quote_text(4, explanation);
    issue_problem_begin(explanation);
    issue_problem_segment("You defined an adjective by '%1' intending that "
        "it would apply to '%2': %Sagain, %%%Lbut %%%3%|, %4");
    issue_problem_end();
}
```

The function definition_problem is called from 9/madj and 12/def.
The function adjective_problem is called from 5/aph.

§**22. Contradictions.** As soon as we combine information from two sentences, we are at risk of a contradiction of some kind:

```
void two_sentences_problem(SIGIL_ARGUMENTS, parse_node *other_sentence,
    char *message, char *explanation) {
    ACT_ON_SIGIL
    if (current_sentence == other_sentence) {
        sentence_problem(PASS_SIGIL, message, explanation);
        return;
    }
    quote_source(1, current_sentence);
    quote_source(2, other_sentence);
    quote_text(3, message);
    quote_text(4, explanation);
    issue_problem_begin(explanation);
    issue_problem_segment(
        "You wrote %1, but in another sentence %2: %Sagain, %%%Lbut %%%3%|, %4");
    issue_problem_end();
}
```

The function two_sentences_problem is called from 5/conj and 9/inf.

§**23.** Almost exactly the same thing, but happening at arbitrary positions in the parse tree, and concerning a world object:

```
void contradiction_problem(SIGIL_ARGUMENTS, parse_node *A, parse_node *B,
        world_object *wo, char *message, char *explanation) {
    ACT_ON_SIGIL
    quote_source(1, A);
    quote_source(2, B);
    quote_object(3, wo);
    quote_text(4, message);
    quote_text(5, explanation);
    issue_problem_begin(explanation);
    if ((B->word_ref1 != A->word_ref1) || (B->word_ref2 != A->word_ref2))
        issue_problem_segment("You wrote %1, but in another sentence %2: ");
    else issue_problem_segment("You wrote %1: ");
    issue_problem_segment("%Sagain, %%%3 %4%|, %5");
    issue_problem_end();
}
```

The function contradiction_problem is called from 8/sob and 9/model.

§**24. Table problems.** In principle we could treat these as sentence problems, but the "sentence" for a table can be enormous: so we need something which can show which table we are in, yet still only cite a small part of it –

```
void table_problem(SIGIL_ARGUMENTS, table *t, table_column *tc, parse_node *data,
    char *message, char *explanation) {
    ACT_ON_SIGIL
    int tw1, tw2;
    table_get_identifying_name(t, &tw1, &tw2);
    quote_words(1, tw1, tw2);
    if (tc != NULL) quote_words(2, tc->word_ref1, tc->word_ref2);
    if (data != NULL) quote_source(3, data);
    issue_problem_begin(explanation);
    issue_problem_segment(message);
    issue_problem_end();
}
```

The function table_problem is called from 10/tab.

§**25. Equation problems.** So this is where hopes are generically dashed about equations:

```
void equation_problem(SIGIL_ARGUMENTS, equation *eqn, char *p, char *text) {
    ACT_ON_SIGIL
    char plenty[1000];
    quote_source(1, current_sentence);
    quote_words(2, eqn->word_ref1, eqn->word_ref2);
    quote_text(3, p);
    strcpy(plenty, "In %1, you define an equation '%2': but ");
    truncated_strcpy(plenty + strlen(plenty), text, 900);
    issue_problem_begin("");
    issue_problem_segment(plenty);
    issue_problem_end();
}
void equation_symbol_problem(SIGIL_ARGUMENTS, equation *eqn, int w1, int w2, char *text) {
    ACT_ON_SIGIL
    quote_source(1, current_sentence);
    quote_words(2, w1, w2);
    quote_text(3, text);
    issue_problem_begin("");
    issue_problem_segment(
        "In %1, you define an equation which mentions the symbol '%2': but %3");
    issue_problem_end();
}
```

The function equation_problem is called from 10/eqns.
The function equation_symbol_problem is called from 10/eqns.

§**26. Proposition type-checking problems.** Are mostly issued thus:

```
void tcp_problem(SIGIL_ARGUMENTS, tc_problem_kit *tck, char *prototype) {
    if (tck->issue_error) {
        ACT_ON_SIGIL
        quote_source(1, current_sentence);
        quote_words(2, tck->ew1, tck->ew2);
        quote_text(3, tck->intention);
        issue_problem_begin("");
        issue_problem_segment("In the sentence %1, it looks as if you intend '%2' to %3, but ");
        issue_problem_segment(prototype);
        issue_problem_end();
    }
    tck->flag_problem = TRUE;
}
```

The function tcp_problem is called from 6/tcpr, 6/equal, 9/provr, 9/cmpbp, 9/vpbp and 10/qnbp.

§**27. World object problems.** Objects can be created rather indirectly (for instance, in the course of assemblies of other objects), and their properties are sometimes inferred from information in sentences far distant, so we can't always locate object problems at any particular sentence.

```
void object_problem(SIGIL_ARGUMENTS, world_object *wo,
        char *message, char *explanation) {
    ACT_ON_SIGIL
    quote_object(1, wo);
    quote_text(2, message);
    quote_text(3, explanation);
    issue_problem_begin(explanation);
    if (wo->kind_flag)
    issue_problem_segment("The kind %1 %2%|, %3");
    else
    issue_problem_segment("The %1 %2%|, %3");
    issue_problem_end();
    log_knowledge_about_object(wo);
}
void object_problem_at_sentence(SIGIL_ARGUMENTS, world_object *wo,
        char *message, char *explanation) {
    ACT_ON_SIGIL
    quote_source(1, current_sentence);
    quote_text(2, message);
    quote_text(3, explanation);
    quote_object(4, wo);
    issue_problem_begin(explanation);
    issue_problem_segment("You wrote %1, but the %4 %2%|, %3");
    issue_problem_end();
}
```

The function object_problem is called from 8/assem, 8/relv, 9/model and 9/cot.
The function object_problem_at_sentence is called from 8/knowc.

§**28.** When objects are created with bizarre names, this is usually a symptom of some other malaise, and so we issue the following message with a little more tact (and in particular, we don't assert very confidently that what we are dealing with is genuinely an object).

```
void object_creation_problem(SIGIL_ARGUMENTS, world_object *wo,
        char *message, char *explanation) {
    ACT_ON_SIGIL
    quote_object(1, wo);
    quote_text(2, message);
    quote_text(3, explanation);
    if (wo->word_ref1 >= 0) {
        parse_node *creator = new_nounphrase_raw(wo->word_ref1, wo->word_ref2);
        quote_source(4, creator);
        issue_problem_begin(explanation);
        issue_problem_segment(
            "I've made something called %4 but it %2%|, %3");
        issue_problem_end();
        log_knowledge_about_object(wo);
    } else {
        issue_problem_begin(explanation);
        issue_problem_segment(
            "I've made something called '%1' but it %2%|, %3");
        issue_problem_end();
        log_knowledge_about_object(wo);
    }
}
```

The function object_creation_problem is called from 9/model.

§**29.** Information about objects is assembled in a mass of inferences. When these contradict each other, that is usually the occasion for a contradiction problem (see above); the following routine is used only when inferences are implied which necessarily make no sense, rather than merely contingently making no sense.

```
void inference_problem(SIGIL_ARGUMENTS, world_object *wo, inference *inf,
        char *message, char *explanation) {
    ACT_ON_SIGIL
    quote_object(1, wo);
    quote_source(2, inf_inferred_from(inf));
    quote_text(3, message);
    quote_text(4, explanation);
    quote_property(5, inf_get_property_name(inf));
    issue_problem_begin(explanation);
    issue_problem_segment(
        "You wrote %2: but the property %5 for the %1 %3%|, %4");
    issue_problem_end();
    log_knowledge_about_object(wo);
}
```

The function inference_problem is called from 9/model.

§**30.** The cutest of all the object problems is the one which results when NI would need to construct a world model which was not well-founded, in the mathematical sense: one in which a given object must, directly or indirectly, contain itself.

```
void foundation_problem(SIGIL_ARGUMENTS, world_object *wo) {
    ACT_ON_SIGIL
    world_object *wo2, *wo3;
    quote_object(1, wo);
    issue_problem_begin("");
    issue_problem_segment("The %1 seems to be containing itself: ");
    wo2 = wo;
    while (TRUE) {
        parse_node *creator = new_nounphrase_raw(wo2->word_ref1, wo2->word_ref2);
        quote_object(2, wo2);
        quote_source(3, creator);
        issue_problem_segment("%2 (created by %3) ");
        wo3 = wo2->object_tree_parent;
        if (wo3) issue_problem_segment("in ");
        else { wo3 = wo2->part_of; issue_problem_segment("part of "); }
        wo2 = wo3;
        if (wo2 == wo) break;
    }
    issue_problem_segment("%1... and so on. This is forbidden.");
    issue_problem_end();
}
```

The function foundation_problem is called from 9/model.

§**31. Property problems.** Just occasionally there is a problem with the definition of a property in the abstract, rather than with the actual property of some specific object.

```
void property_problem(SIGIL_ARGUMENTS, property_name *prn, char *message, char *explanation) {
    ACT_ON_SIGIL
    quote_property(1, prn);
    quote_text(2, message);
    quote_text(3, explanation);
    issue_problem_begin(explanation);
    issue_problem_segment("The %1 %2%|, %3");
    issue_problem_end();
}
```

The function property_problem is called from 9/pp.

§**32. Extension problems.** These are generated when the user tries to employ a malformed extension.

```
void extension_problem(SIGIL_ARGUMENTS, extension_file *ef, char *message) {
    ACT_ON_SIGIL
    quote_extension(1, ef);
    quote_text(2, message);
    issue_problem_begin(message);
    issue_problem_segment(
        "The extension %1, which your source text makes use of, %2.");
    issue_problem_end();
}
```

The function extension_problem is called from 4/sent and 4/ext.

§**33. Releasing problems.** These occur when the release instructions do not make sense. Sometimes it's possible to pin down an exact place where the difficulty occurs, but sometimes not.

```
void release_problem(SIGIL_ARGUMENTS, char *message, char *filename) {
    ACT_ON_SIGIL
    quote_text(1, message);
    quote_text(2, filename);
    issue_problem_begin(message);
    issue_problem_segment("A problem occurred with the 'Release along with...': "
        "instructions: %1 (with the file '%2')");
    issue_problem_end();
}
void release_problem_at_sentence(SIGIL_ARGUMENTS, char *message, char *filename) {
    ACT_ON_SIGIL
    quote_text(1, message);
    quote_text(2, filename);
    quote_source(3, current_sentence);
    issue_problem_begin(message);
    issue_problem_segment("A problem occurred with the 'Release along with...': "
        "instructions (%3): %1 (with the file '%2')");
    issue_problem_end();
}
```

The function release_problem is called from 10/bib.
The function release_problem_at_sentence is called from 10/bib.

§**34. Cartographical problems.** The map-maker used for the World index and also the EPS-file output has its own quaint syntax, and where there is syntax, there are error messages:

```
void map_problem(SIGIL_ARGUMENTS, parse_node *q, char *message) {
    ACT_ON_SIGIL
    quote_source(1, q);
    quote_text(2, message);
    issue_problem_begin("");
    issue_problem_segment("You gave as a hint in map-making: %1. %2");
    issue_problem_end();
}
void map_problem_wanted_but(SIGIL_ARGUMENTS, parse_node *q, char *i_wanted_a, int vw1) {
    ACT_ON_SIGIL
    quote_source(1, q);
    quote_text(2, i_wanted_a);
    quote_words(3, vw1, vw1);
    issue_problem_begin("");
    issue_problem_segment(
        "You gave as a hint in map-making: %1. But the value '%3' did not "
        "fit - it should have been %2.");
    issue_problem_end();
}
```

The function map_problem is called from 9/map.
The function map_problem_wanted_but is called from 9/map.

§**35. Creating the Problems report.** We are at last able to print the text which appears at the top of the Problems report, sneakily treating this as if it were an error in its own right (except that it is sent only to the Problems file, not to `stderr` or the debugging log, and except that we temporarily disable the force-crash option); and this completes the code for errors. In my end is my beginning.

```
void start_problems_report(void) {
    int coae = crash_on_all_errors;
    char *fn = build_filename(PROBLEM_LOG_LEAFNAME);
    if (STREAM_OPEN_TO_FILE(problems_file, fn, UTF8_ENC) == FALSE)
        fatal_error2("Can't open problem log", fn);
    html_header(problems_file, "Translating the Source", NULL, NULL);
    crash_on_all_errors = FALSE;
    probl = problems_file;
    quote_text(1, NI_VERSION);
    quote_text(2, NI_BUILD);
    issue_problem_begin("**");
    issue_problem_segment(
        "This is the report produced by %1 (build %2) on its most recent "
        "run through:\n");
    issue_problem_end();
    probl = NULL;
    crash_on_all_errors = coae;
}
```

The function start_problems_report is called from 2/dl.

Debugging Log 2/dl

Purpose

To write to the debugging log, a plain text file which traces what NI is doing, in order to assist those lost souls debugging it.

2/dl.¶3-0 Debugging aspects; §2-3 Telemetry; §4 Subheadings; §5-10 Aspects; §11 The sentence trace; §12 Wrapping up; §13 Meaning list conversion backtrace; §14-19 Pretty printing

---

Definitions

¶1. Almost all of NI's algorithms are able to write a plain text trace of what they are doing in the debugging log file. To write to the log, we must in principle `fprintf` to `dl`, the file handle to which debugging log output is currently assigned. In practice we more often use a pair of pseudo-functions called `LOG` and `LOGIF`, which are macros defined below. For instance, the pseudo-function-call

```
LOGIF(HEADINGS, "Heading %d skipped\n", n);
```

prints the line in question to the debugging log only if the given aspect (see below) is currently switched on. Plain `LOG` does the same, but unconditionally.

```
#define LOG(args...) dlprintf(args)
#define LOGIF(aspect, args...) { \
    if (dl_this(aspect##_DA)) dlprintf(args); \
}
```

¶2. What makes this niftier than your common or garden `fprintf` is that in addition to `%d`, `%s`, and so forth, the log macros also support a number of NI-specific string escapes prefaced by a dollar sign. These pretty-print NI data structures as follows. Note that some dollar escapes soak up a pair of arguments, not just a single argument as `printf` escapes do.

`$A`	Action pattern (`action_pattern *`)
`$a`	Adjectival phrase (`adjectival_phrase *`)
`$B`	Table (`table *`)
`$b`	Booking for a rule (`booked_rule *`)
`$C`	Column (`table_column *`)
`$D`	Discourse representation (`pcalc_prop *`)
`$d`	Extension dictionary entry (`extension_dictionary_entry *`)
`$E`	Invocation list (`invocation_list *`)
`$e`	Invocation (`invocation *`)
`$F`	Word range showing formatting of white space (`int`, `int`)
`$G`	Grammar verb (`grammar_verb *`)
`$g`	Grammar line (`grammar_line *`)
`$H`	Heading (`heading *`)
`$h`	Phrase type (`ph_type_data *`)
`$I`	Inference (`inference *`)
`$i`	Inform 6 schema (`i6_schema *`)
`$j`	Inference subject (`inference_subject *`)
`$K`	Rulebook contents (`rulebook *`)
`$k`	Local variable (`local_variable *`)
`$L`	Action name list (`action_name_list *`)
`$l`	Action name (`action_name *`)
`$M`	Excerpt meaning (`excerpt_meaning *`)

`$m` Excerpt meaning list (`meaning_list *`)
`$N` Parse tree node type (`int`)
`$n` Inference type (`int`)
`$O` Object reference (`world_object *`)
`$o` Atom (`pcalc_prop *`)
`$P` Single parse tree node (`parse_node *`)
`$p` Production number (`int`)
`$Q` Quantity (`quantity *`)
`$q` Equation (`equation *`)
`$R` Rule (`phrase *`)
`$r` Type reference (`adjective_list_entry *`)
`$S` Type specification (`specification *`)
`$s` Type ID (`int`)
`$T` Subtree of parse tree (`parse_node *`)
`$t` Time period (`time_period *`)
`$U` Usage data for phrase (`ph_usage_data *`)
`$u` Kind of value (`kind_of_value *`)
`$V` Verb number (`int`)
`$v` Vocabulary entry (`vocabulary_entry *`)
`$W` Word range (`int`, `int`)
`$X` Exhaustive look at a SP (`specification *`)
`$x` Extension (`extension_file *`)
`$Y` Property (`property_name *`)
`$0` Predicate calculus term (`pcalc_term *`)
`$2` Binary predicate (`binary_predicate *`)

¶3. Debugging aspects. There are many different things which can go into the debugging file, or need not: since even a simple half-page source can result in a debugging log 5MB in size, we generally don't want everything included unless we ask for it with sentences in the source like:

Include object creations in the debugging log.

Inevitably, such sentences can only be parsed once NI has already done some work, so don't take effect until then.

A "debugging aspect" is a category of information that can be included, or not, as we please. Each has a unique number and a name of up to three words in length: "object creations" being a two-word example.

```
typedef struct debugging_aspect {
    char *da_word1;                                   first word of name
    char *da_word2;                                   second word, or "" if there is none
    char *da_word3;                                   third word, or "" if there is none
    int on_or_off;                                    whether or not active when writing to debugging log
    int alternate;                                    whether or not active when writing to trace log
} debugging_aspect;
```

The structure debugging_aspect is private to this section.

¶4. And now we must define all those constants and names. Note that the `TRUE` or `FALSE` settings below are the defaults, and apply unless the source says otherwise. The `alternate` settings are those used in trace-sentences mode, that is, between asterisk sentences.

```
define SET_DAS_DA 0
define TABLE_OF_DAS_DA 1
define VERIFICATIONS_DA 2
define RELATION_DEFINITIONS_DA 3
define OBJECT_CREATIONS_DA 4
define ACTION_CREATIONS_DA 5
define PROPERTY_CREATIONS_DA 6
define PLURALS_DA 7
define PAST_PARTICIPLES_DA 8
define RULE_ATTACHMENTS_DA 9
define LIBRARY_RULES_DA 10
define ASSERTIONS_DA 11
define INFERENCES_DA 12
define PROVISION_DA 13
define KIND_CHANGES_DA 14
define ASTERISKED_SYNTAX_DA 15
define TYPE_CREATIONS_DA 16
define BRIEF_PICTURE_DA 17
define PROPERTY_ASSIGNMENTS_DA 18
define TYPE_CASTS_DA 19
define PROPERTY_TRANSLATIONS_DA 20
define OBJECT_TREE_DA 21
define PHRASEBOOK_DA 22
define PHRASE_CREATIONS_DA 23
define PHRASE_ACTION_PATTERNS_DA 24
define PHRASE_COMPARISONS_DA 25
define QUANTITY_COMPILATION_DA 26
define OBJECT_COMPILATION_DA 27
define PHRASE_COMPILATION_DA 28
define TYPE_RESERVOIR_DA 29
define EXPRESSIONS_DA 30
define INVOCATIONS_DA 31
define COMPILE_RULEBOOKS_DA 32
define GRAMMAR_DA 33
define LEXER_DA 34
define HEADINGS_DA 35
define QUANTITY_CREATIONS_DA 36
define MEMORY_ALLOCATION_DA 37
define GRAMMAR_CONSTRUCTION_DA 38
define COMPILE_TEXT_ROUTINES_DA 39
define COMPILE_ACTION_PATTERNS_DA 40
define PRONOUNS_DA 41
define VOCABULARY_DA 42
define IMPLICATIONS_DA 43
define TABLE_CONSTRUCTION_DA 44
define TIME_PERIODS_DA 45
define DESCRIPTION_COMPILATION_DA 46
define LOCAL_VARIABLES_DA 47
define SPATIAL_MAP_DA 48
define EXCERPT_PARSING_DA 49
```

```
define EXCERPT_MEANINGS_DA 50
define S_GRAMMAR_DA 51
define NP_RESOLUTION_DA 52
define CALCULUS_DA 53
define WORKINGS_DA 54
define EXTCENSUS_DA 55
define AP_PARSING_DA 56
define TYPE_PERMISSIONS_DA 57
define TYPE_USAGE_DA 58
define ASSEMBLIES_DA 59
define INVWORKINGS_DA 60
define SPECIFICITIES_DA 61
define FIGURE_CREATIONS_DA 62
define SEGMENTS_DA 63
define CI_FOPEN_DA 64
define ML_ALLOC_DA 65

debugging_aspect the_debugging_aspects[] = {
    { "debugging", "log", "inclusions", TRUE, FALSE },
    { "debugging", "log", "contents", TRUE, FALSE },
    { "verifications", "", "", TRUE, FALSE },
    { "relation", "definitions", "", TRUE, FALSE },
    { "object", "creations", "", TRUE, FALSE },
    { "action", "creations", "", TRUE, FALSE },
    { "property", "creations", "", TRUE, FALSE },
    { "constructed", "plurals", "", TRUE, FALSE },
    { "constructed", "past", "participles", FALSE, FALSE },
    { "rule", "attachments", "", FALSE, FALSE },
    { "library", "rules", "", FALSE, FALSE },
    { "assertions", "", "", FALSE, TRUE },
    { "inferences", "", "", FALSE, TRUE },
    { "property", "provision", "", TRUE, FALSE },
    { "changes", "of", "kind", TRUE, TRUE },
    { "asterisked", "syntax", "", TRUE, FALSE },
    { "type", "creations", "", TRUE, FALSE },
    { "brief", "picture", "", TRUE, FALSE },
    { "property", "assignments", "", FALSE, TRUE },
    { "type", "casts", "", FALSE, FALSE },
    { "property", "translations", "", FALSE, FALSE },
    { "object", "tree", "", TRUE, FALSE },
    { "phrasebook", "", "", FALSE, FALSE },
    { "phrase", "creations", "", FALSE, FALSE },
    { "phrase", "action", "patterns", FALSE, FALSE },
    { "phrase", "comparisons", "", FALSE, FALSE },
    { "quantity", "compilation", "", TRUE, FALSE },
    { "object", "compilation", "", TRUE, FALSE },
    { "phrase", "compilation", "", FALSE, FALSE },
    { "type", "specification", "reservoir", FALSE, FALSE },
    { "expressions", "", "", FALSE, FALSE },
    { "invocations", "", "", FALSE, FALSE },
    { "rulebook", "compilation", "", FALSE, FALSE },
    { "grammar", "", "", FALSE, FALSE },
    { "lexical", "output", "", FALSE, FALSE },
    { "headings", "", "", FALSE, FALSE },
```

```
    { "quantity", "creations", "", TRUE, FALSE },
    { "memory", "allocation", "", FALSE, FALSE },
    { "grammar", "construction", "", FALSE, FALSE },
    { "text", "routine", "compilation", FALSE, FALSE },
    { "action", "pattern", "compilation", FALSE, FALSE },
    { "pronouns", "", "", TRUE, FALSE },
    { "vocabulary", "", "", FALSE, FALSE },
    { "implications", "", "", TRUE, TRUE },
    { "table", "construction", "", TRUE, FALSE },
    { "time", "periods", "", FALSE, FALSE },
    { "description", "compilation", "", FALSE, FALSE },
    { "local", "variables", "", FALSE, FALSE },
    { "spatial", "map", "", FALSE, FALSE },
    { "excerpt", "parsing", "", FALSE, FALSE },
    { "excerpt", "meanings", "", FALSE, FALSE },
    { "s", "grammar", "", FALSE, FALSE },
    { "noun", "phrase", "resolution", FALSE, FALSE },
    { "predicate", "calculus", "", TRUE, FALSE },
    { "predicate", "calculus", "workings", FALSE, FALSE },
    { "extensions", "census", "", FALSE, FALSE },
    { "action", "pattern", "parsing", FALSE, FALSE },
    { "type", "permissions", "", FALSE, FALSE },
    { "type", "usage", "", FALSE, FALSE },
    { "assemblies", "", "", FALSE, FALSE },
    { "invocation", "workings", "", FALSE, FALSE },
    { "specificities", "", "", FALSE, FALSE },
    { "figure", "creations", "", FALSE, FALSE },
    { "segments", "", "", FALSE, FALSE },
    { "case", "insensitive", "file-handling", FALSE, FALSE },
    { "meaning", "list", "allocation", FALSE, FALSE },
    { "", "", "", FALSE, FALSE }
};
```

---

§**1.** Two log files are generated by NI as it runs: the HTML report called "Problems", which accumulates the text of all error messages issued, and the debugging log, which provides an opportunity to see what has been happening behind the scenes. Such a log file is normally buffered by the filing system, so that a sudden crash of NI may result in the loss of recent data written to the log. Which is a pity, since this is exactly the most useful evidence as to the cause of the crash in the first place. Accordingly, we fairly often `fflush` the debug log file, forcing any buffered output to be written to disc, and in extreme cases NI can be run in a mode which directs all debugging information to `stdout`, which should ensure that no data is lost even in the event of a crash.

In this rest of this section, we always assume that `dl` is open. Note that it is possible this has been switched to be `stdout`, or even that it is temporarily the sentence tracing file: but we don't care.

```
void open_log_files(void) {
    char *fn = build_filename(DEBUG_LOG_LEAFNAME);
    if (STREAM_OPEN_TO_FILE(debug_log_file, fn, ISO_ENC) == FALSE)
        fatal_error2("Can't open debug log", fn);
    dl = debug_log_file;
    LOG("Debugging log of %s\n", NI_VERSION);
    start_problems_report();
```

```
}
void close_log_files(void) {
    show_debugging_contents();
    STREAM_CLOSE(debug_log_file); dl = NULL;
    if (trl) STREAM_CLOSE(trl); trl = NULL;
    if (telmy) {
        STREAM_WRITE(telmy, "End of telemetry for this run.\n");
        STREAM_CLOSE(telmy); telmy = NULL;
    }
    html_footer(problems_file); STREAM_CLOSE(problems_file);
}
```

The function open_log_files is called from 14/main.
The function close_log_files is called from 14/main.

§**2. Telemetry.** The telemetry file is optional, and transcribes the outcome of each run. This is mainly for testing Inform, but may also be useful for teachers who want to monitor how a whole class is using the system.

```
int attempts_to_open_telemetry = 0;
void ensure_telemetry_file(void) {
    if (telmy) return;
    if (attempts_to_open_telemetry++ > 0) return;
    if (STREAM_OPEN_TO_FILE_APPEND(telemetry_file, filename_of_telemetry, ISO_ENC) == FALSE)
        fatal_error2("Can't open telemetry file", filename_of_telemetry);
    telmy = telemetry_file;
    STREAM_WRITE(telmy, "\n-- -- -- -- -- -- -- --\n%s build %s: telemetry.\n",
        NI_VERSION, NI_BUILD);
    int this_month = the_present->tm_mon + 1;
    int this_day = the_present->tm_mday;
    int this_year = the_present->tm_year + 1900;
    STREAM_WRITE(telmy, "Running on %4d-%02d-%02d at %02d:%02d.\n\n",
        this_year, this_month, this_day, the_present->tm_hour, the_present->tm_min);
    LOG("Opening telemetry file.\n");
}
void write_telemetry_note(char *m) {
    ensure_telemetry_file();
    STREAM_WRITE(telmy, "The user says:\n\n%s\n\n", m);
}
```

The function ensure_telemetry_file is called from 2/prob2 and 2/prob3.
The function write_telemetry_note is called from 8/tass.

§**3.** The following reads in the text of the optional file of use options, when this has been created.

```
void read_further_mandatory_text(void) {
    FILE *FMAN = iso_fopen(filename_of_options, "r");
    if (FMAN == NULL) return;
    char line_buffer[300];
    while (TRUE) {
        int len = truncated_iso_fgets(FMAN, line_buffer, 255);
        if (len < 0) break;
        strcat(line_buffer, " | ");
        int w1 = lexer_wordcount, w2;
        feed_into_lexer(line_buffer, FALSE, FALSE);
        w2 = lexer_wordcount - 2;
        if [[w1, w2 == use ... FULLSTOP --> w1, w2]]
            set_immediate_option_flags(w1, w2, NULL);
    }
    fclose(FMAN);
}
```

The function read_further_mandatory_text is called from 3/read.

§**4. Subheadings.** To provide signposts in what is otherwise a huge amorphous pile of text, the debugging log is divided into "phases". And phases are themselves subdivided into "stages". This is how.

```
char debug_log_phase[32];
int debug_log_subheading = 1;
void log_new_phase_of_NIs_run(char *p, char *q) {
    strcpy(debug_log_phase, p);
    LOG("\n\n-------------------------------------------------------\n");
    LOG("Phase %s ... %s", p, q);
    LOG("\n-------------------------------------------------------\n\n");
    STREAM_FLUSH(dl);
    debug_log_subheading = 1;
}
void log_new_stage_of_NIs_run(char *p) {
    LOG("\n\n==== Phase %s.%d ... %s ====\n\n", debug_log_phase, debug_log_subheading, p);
    debug_log_subheading++;
    STREAM_FLUSH(dl);
}
```

The function log_new_phase_of_NIs_run is called from 14/i6t.
The function log_new_stage_of_NIs_run is called from 13/gv and 14/i6t.

§**5. Aspects.** As mentioned in Chapter 2 above, a wide range of activities can be logged to the debugging log: these are called "aspects" and we can switch logging of them off or on independently. The following routine tests whether a given aspect is currently being logged, and is used by our main macros. Aspect 0 mandates writing to the debug log, and is used when errors occur.

```
int dl_this(int aspect) {
    int decision = the_debugging_aspects[aspect].on_or_off;
    if (aspect == 0) decision = TRUE;
    if (decision) STREAM_FLUSH(dl);
    return decision;
}
```

The function dl_this is called from 3/read, 4/pn, 4/ext, 5/ml, 5/parse, 7/tids, 8/assem, 9/model, 12/toph and 12/br.

§**6.** We sometimes want to switch everything on, or switch everything off:

```
void set_all_aspects(int new_state) {
    int i;
    if (dl) LOGIF(SET_DAS, "Set debugging aspect: everything -> %s\n",
        new_state?"TRUE":"FALSE");
    for (i=0; ; i++) {
        debugging_aspect *da = &(the_debugging_aspects[i]);
        if (*(da->da_word1) == 0) break;
        da->on_or_off = new_state;
    }
}
```

§**7.** Here we parse a request to change the current logging state:

```
void set_debugging_aspect(int w1, int w2, int new_state) {
    LOGIF(SET_DAS, "Set contents of debugging log: $W -> %s\n",
        w1, w2, new_state?"TRUE":"FALSE");
    ⟨See if this is a compound request for debugging information 8⟩;
    ⟨See if this is a simple request for debugging information 9⟩;
    quote_source(1, current_sentence);
    quote_words(2, w1, w2);
    handmade_problem(_P_(C2UnknownDA));
    issue_problem_segment(
        "In the sentence %1, you asked to include '%2' in the "
        "debugging log, but there is no such debugging log topic.");
    issue_problem_end();
}
```

The function set_debugging_aspect is called from 4/verb.

§**8.** Requests can be divided as "R and S" (and can even use the serial comma), and we also understand "only R" and "everything" and "nothing".

⟨See if this is a compound request for debugging information 8⟩ ≡

```
if (is_list_divided(w1, w2, LOOK_FOR_AND)) {
    int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
    set_debugging_aspect(lw1, lw2, new_state);
    set_debugging_aspect(rw1, rw2, new_state);
    return;
}
if [[w1, w2 == only ...]] {
    set_all_aspects(1-new_state);
    set_debugging_aspect(w1+1, w2, new_state);
    return;
}
if [[w1, w2 == everything]] { set_all_aspects(new_state); return; }
if [[w1, w2 == nothing]] { set_all_aspects(1-new_state); return; }
```

This code is used in §7.

§**9.** Otherwise a request must be the name of a single debugging aspect.

⟨See if this is a simple request for debugging information 9⟩ ≡

```
int i;
debugging_aspect *da;
for (i=0; ; i++) {
    da = &(the_debugging_aspects[i]);
    if (*(da->da_word1) == 0) break;
    if (strcmp(lw_array[w1].lw_text, da->da_word1) != 0) continue;
    if (w2 > w1) {
        if (*(da->da_word2) == 0) continue;
        if (strcmp(lw_array[w1+1].lw_text, da->da_word2) != 0) continue;
    }
    if (w2 > w1+1) {
        if (*(da->da_word3) == 0) continue;
        if (strcmp(lw_array[w1+2].lw_text, da->da_word3) != 0) continue;
    }
    if (w2 > w1+2) continue;
    da->on_or_off = new_state;
    return;
}
```

This code is used in §7.

§**10.** We need the ability to change debugging log settings from the command line so that `intest` can collect the debugging information of its choice from a range of source texts. As can be seen, the command line form is derived from the textual form by replacing every space with a hyphen: for instance, the effect of

> Include property provision in the debugging log.

can be duplicated at the command line by running NI with the following switch:

```
--log=property-provision
```

We also recognise `no-property-provision` to switch this off again, `everything` and `nothing` with the obvious meanings, and `list` to print out a list of debugging aspects to `stdout`.

```
void set_dl_from_command_line(char *text) {
    int i, parity = TRUE, list_mode = FALSE;
    debugging_aspect *da;
    if (strcmp(text, "everything") == 0) { set_all_aspects(TRUE); return; }
    if (strcmp(text, "nothing") == 0) { set_all_aspects(FALSE); return; }
    if (strcmp(text, "list") == 0) list_mode = TRUE;
    parity = TRUE;
    if ((text[0] == 'n') && (text[1] == 'o') && (text[2] == '-')) {
        parity = FALSE;
        text += 3;
    }
    for (i=0; ; i++) {
        char tomatch[100];
        da = &(the_debugging_aspects[i]);
        if (*(da->da_word1) == 0) break;
        sprintf(tomatch, "%s", da->da_word1);
        if (*(da->da_word2)) sprintf(tomatch+strlen(tomatch), "-%s", da->da_word2);
        if (*(da->da_word3)) sprintf(tomatch+strlen(tomatch), "-%s", da->da_word3);
        if (list_mode) {
            STREAM_WRITE(STDOUT, "--log=%s  (%s)\n", tomatch, (da->on_or_off)?"on":"off");
        } else {
            if (strcmp(text, tomatch) == 0) {
                da->on_or_off = parity;
                return;
            }
        }
    }
    if (list_mode) return;
    STREAM_WRITE(STDOUT, "No such debugging log aspect as '%s'.\n"
        "(Try running --log=list for a list of the valid aspects.)\n", text);
    fatal_error("No such debugging log aspect.");
}
```

The function set_dl_from_command_line is called from 14/main.

§**11. The sentence trace.** This is a sort of abbreviated extract of the debug log which shows what was done with given assertions: how they were broken down into inferences, and so on. The sentence trace file has its own debugging aspect settings, independently of the main log, and only logs those sentences between asterisks.

```
void debug_log_tracing(int starred, char *heading) {
    int i, j;
    if (starred) {
        if (trl == NULL) {
            char *fn = filename_of_tracefile;
            STREAM_OPEN_TO_FILE(trace_log, fn, ISO_ENC);
            trl = trace_log;
        }
        if (trl) {
            LOG("\n*** Switching to trace log here for a while ***\n");
            dl = trace_log;
            LOG("*** %s ***\n\n", heading);
        }
    } else {
        if (trl != NULL) STREAM_WRITE(trl, "\n\n");
        dl = debug_log_file;
    }
    for (i=0; ; i++) {
        debugging_aspect *da = &(the_debugging_aspects[i]);
        if (*(da->da_word1) == 0) break;
        j = da->on_or_off;
        da->on_or_off = da->alternate;
        da->alternate = j;
    }
}
```

The function debug_log_tracing is called from 8/tass.

§**12. Wrapping up.** At the end of the debugging log we list what was in it, mostly to provide the reader with a list of other things which could have been put into it, but weren't.

```
void show_debugging_settings_with_state(int state) {
    int i, c;
    debugging_aspect *da;
    for (i=0, c=0; ; i++) {
        da = &(the_debugging_aspects[i]);
        if (*(da->da_word1) == 0) break;
        if (da->on_or_off == state) {
            c++;
            LOG("%s %s %s\n", da->da_word1, da->da_word2, da->da_word3);
        }
    }
    if (c == 0) { LOG("(Nothing.)\n"); }
}
void show_debugging_contents(void) {
    if (dl_this(TABLE_OF_DAS_DA) == FALSE) return;
    LOG("\n\nThat concludes the debugging log from this run of NI.\n"
        "Its contents were as follows, and can be changed by placing\n"
```

```
        "text like 'Include property creations in the debugging log.'\n"
        "or 'Omit everything from the debugging log.' in the source.\n\n");
    LOG("Included:\n"); show_debugging_settings_with_state(TRUE);
    LOG("Omitted:\n"); show_debugging_settings_with_state(FALSE);
}
```

§13. **Meaning list conversion backtrace.** It turns out to be convenient to produce a form of stack backtrace to the debugging log when strange problems turn up with the meaning-list conversion algorithm. This does nothing functional, i.e., does nothing other than write to the debugging log, unless a ludicrously massive backtrace occurs, which must be the result of a bug and so results in an internal error.

```
define MAXIMUM_MLC_BACKTRACE 1000
define CLEAR_MLC_BACKTRACE mlc_backtrace_sp = 0;
define RECORD_MLC_BACKTRACE
    if (mlc_backtrace_sp >= MAXIMUM_MLC_BACKTRACE)
        internal_error("MLC backtrace stack overflow");
    mlc_backtrace[mlc_backtrace_sp] = ml;
    mlc_backtrace_at[mlc_backtrace_sp++] = (char *) __func__;

int mlc_backtrace_sp = 0;
meaning_list *mlc_backtrace[MAXIMUM_MLC_BACKTRACE];
char *mlc_backtrace_at[MAXIMUM_MLC_BACKTRACE];

void log_mlc_backtrace(void) {
    int i;
    if (mlc_backtrace_sp == 0) return;
    LOG("MLC backtrace:\n");
    for (i=0; i<mlc_backtrace_sp; i++) {
        LOG("%d: %s\n$m", i, mlc_backtrace_at[i], mlc_backtrace[i]);
    }
    LOG("End of MLC backtrace\n");
}
```

The function log_mlc_backtrace is called from 2/prob3.

§14. **Pretty printing.** We can print formatted text to any stream with `STREAM_WRITE` or `WRITE`, and this offers `printf`-style escapes, but for the debugging log stream we want to supplement these with a much richer set of pretty-printing features. This means adding a new `$` escape, which is a signal to pretty-print internal Inform data structures.

```
void dlprintf(char *fmt, ...) {
    va_list ap;                                          the variable argument list signified by the dots
    char *p;
    if (dl == NULL) return;                              in case logging occurs for an error very early on
    STREAM_FLUSH(dl);
    va_start(ap, fmt);                                   macro to begin variable argument processing
    for (p = fmt; *p; p++)
        switch (*p) {
            case '%': ⟨Recognise traditional escape sequences 15⟩; break;
            case '$': ⟨Recognise new NI debugging log escape sequences 16⟩; break;
            case '\n': if (logging_to_I6_text) STREAM_PUT(dl, '^'); else STREAM_PUT(dl, '\n'); break;
            default: STREAM_PUT(dl, *p); break;
        }
    va_end(ap);                                          macro to end variable argument processing
}
```

§**15.** We don't trouble to check that correct `printf` escapes have been used: instead, we pass anything in the form of a percentage sign, followed by up to four nonalphabetic modifying characters, followed by an alphabetic category character, straight through to `STREAM_WRITE`, which in turn will forward the escape on to either `sprintf` or `fprintf` – so that we can use the implementation in the standard C library.

The only exception is `%%`, which makes a literal percentage sign. (It is an error to leave an escape half-finished at the end of the string.)

⟨Recognise traditional escape sequences 15⟩ ≡

```
int ival; double dval; char *sval;
char format_string[8], category = ' ';
int i = 1;
format_string[0] = '%';
p++;
while (*p) {
    format_string[i++] = *p;
    if ((islower(*p)) || (isupper(*p)) || (*p == '%')) category = *p;
    p++;
    if ((category != ' ') || (i==6)) break;
}
format_string[i] = 0; p--;
switch (category) {
    case 'c': case 'd': case 'x':                    char is promoted to int in variable arguments
        ival = va_arg(ap, int); STREAM_WRITE(dl, format_string, ival); break;
    case 'f':
        dval = va_arg(ap, double); STREAM_WRITE(dl, format_string, dval); break;
    case 's':
        for (sval = va_arg(ap, char *); *sval; sval++) STREAM_PUT(dl, *sval); break;
    case '%': STREAM_PUT(dl, '%'); break;
    default:
        STREAM_WRITE(dl, "*** Bad escape: <%s> ***\n", format_string);
        internal_error("Unknown % string escape in dlprintf");
}
```

This code is used in §14.

§**16.** Here the syntax is simpler: it's just a dollar sign followed by an identifying letter. `$$` makes a literal dollar; and it is, for some reason, legal for the text to end with a single `$`, which is treated similarly as a literal.

⟨Recognise new NI debugging log escape sequences 16⟩ ≡

```
p++;
switch (*p) {
    case 0: STREAM_PUT(dl, '$'); p--; break;          a literal dollar was last character
    case '$': STREAM_PUT(dl, '$'); break;             a double dollar means a literal dollar
    SINGLE_ESCAPE('0', pcalc_term *, log_pcalc_term)
    escape $1 is vacant
    SINGLE_ESCAPE('2', binary_predicate *, log_binary_predicate)
    SINGLE_ESCAPE('A', action_pattern *, log_action_pattern)
    SINGLE_ESCAPE('a', adjectival_phrase *, log_adjectival_phrase)
    SINGLE_ESCAPE('B', table *, log_table)
    SINGLE_ESCAPE('b', booked_rule *, log_booked_rule)
    SINGLE_ESCAPE('C', table_column *, log_table_column)
    escape $c is vacant
    SINGLE_ESCAPE('D', pcalc_prop *, log_pcalc_prop)
```

```
        SINGLE_ESCAPE('d', extension_dictionary_entry *, log_extension_dictionary_entry)
        SINGLE_ESCAPE('E', invocation_list *, log_invocation_list)
        SINGLE_ESCAPE('e', invocation *, log_invocation)
        DOUBLE_ESCAPE('F', int, int, log_word_range_with_whitespace)
        escape $f is vacant
        SINGLE_ESCAPE('G', grammar_verb *, log_grammar_verb)
        SINGLE_ESCAPE('g', grammar_line *, log_grammar_line)
        SINGLE_ESCAPE('H', heading *, log_heading)
        SINGLE_ESCAPE('h', ph_type_data *, log_ph_type_data)
        SINGLE_ESCAPE('I', inference *, log_inference)
        SINGLE_ESCAPE('i', i6_schema *, log_i6_schema)
        escape $J is vacant
        SINGLE_ESCAPE('j', inference_subject *, log_inference_subject)
        SINGLE_ESCAPE('K', rulebook *, log_rulebook)
        SINGLE_ESCAPE('k', local_variable *, log_local_variable)
        SINGLE_ESCAPE('L', action_name_list *, log_action_name_list)
        SINGLE_ESCAPE('l', action_name *, log_action_name)
        SINGLE_ESCAPE('M', excerpt_meaning *, log_excerpt_meaning)
        SINGLE_ESCAPE('m', meaning_list *, log_meaning_list)
        SINGLE_ESCAPE('N', int, log_node_type)
        SINGLE_ESCAPE('n', int, log_inference_type)
        SINGLE_ESCAPE('O', world_object *, log_world_object)
        SINGLE_ESCAPE('o', pcalc_prop *, log_pcalc_atom)
        SINGLE_ESCAPE('P', parse_node *, log_single_parse_node_at_current_margin)
        SINGLE_ESCAPE('p', int, log_production)
        SINGLE_ESCAPE('Q', quantity *, log_quantity)
        SINGLE_ESCAPE('q', equation *, log_equation)
        SINGLE_ESCAPE('R', phrase *, log_phrase)
        SINGLE_ESCAPE('r', adjective_list_entry *, log_adjective_list_entry)
        SINGLE_ESCAPE('S', specification *, log_specification_concisely)
        SINGLE_ESCAPE('s', int, log_data_type_ID)
        SINGLE_ESCAPE('T', parse_node *, log_parse_node_at_current_margin)
        SINGLE_ESCAPE('t', time_period *, log_time_period)
        SINGLE_ESCAPE('U', ph_usage_data *, log_ph_usage_data)
        SINGLE_ESCAPE('u', kind_of_value *, log_kind_of_value)
        SINGLE_ESCAPE('V', int, log_verb_number)
        SINGLE_ESCAPE('v', vocabulary_entry *, log_vocabulary_entry)
        DOUBLE_ESCAPE('W', int, int, log_word_range)
        escape $w is vacant
        SINGLE_ESCAPE('X', specification *,         log_specification_exhaustively)
        SINGLE_ESCAPE('x', extension_file *, log_extension_file)
        SINGLE_ESCAPE('Y', property_name *, log_property_name)
        escape $y is vacant
        escape $Z is vacant
        escape $z is vacant
        default:
            STREAM_WRITE(dl, "*** Bad escape: '$%c' ***\n", *p);
            internal_error("Unknown % string escape in dlprintf");
    }
```

This code is used in §14.

§**17.** All of those cases were built by the following macros, which extract data from the variable argument list, using the supplied type, and pass it to the supplied routine. One macro handles single-argument escapes, the other double-argument escapes.

```
define SINGLE_ESCAPE(ch, tp, rt)
    case ch: {
        tp data1;
        data1 = va_arg(ap, tp);                          extract next argument as tp
        rt(data1);
        break;
    }
define DOUBLE_ESCAPE(ch, tp1, tp2, rt)
    case ch: {
        tp1 data1; tp2 data2;
        data1 = va_arg(ap, tp1);                         extract next argument as tp1
        data2 = va_arg(ap, tp2);                         extract next argument as tp2
        rt(data1, data2);
        break;
    }
```

§**18.** That concludes the debugging log routines, and with them the service levels of NI. We can at last start the actual main stream of the program: reading the source text and breaking it into words.

3 Characters and Words

`3/lex`: *Lexer.w* To break down a stream of characters into a numbered sequence of words, literal strings and literal I6 inclusions, removing comments and unnecessary whitespace.

`3/read`: *Read Source Text.w* This is where source text is read in, whether from extension files or from the main source text file, and fed into the lexer.

`3/lwb`: *Lexical Writing Back.w* Feeding modified text back into the lexer, to ensure that inflected versions of phrases are also within the word stream.

`3/lexs`: *Lexical Services.w* A small suite of routines to examine the words stored by the lexer, which are provided as a utility to higher levels of NI. None are used by the lexer itself.

`3/vocab`: *Vocabulary.w* To classify the words in the lexical stream, where two different words are considered equivalent if they are unquoted and have the same text, taken case insensitively.

`3/words`: *Built-In Words.w* To stock up NI's vocabulary with words which are needed in order to compare the source text against various syntaxes known to NI.

Lexer 3/lex

Purpose

To break down a stream of characters into a numbered sequence of words, literal strings and literal I6 inclusions, removing comments and unnecessary whitespace.

3/lex.¶4-6 The lexical structure of source text; ¶7-0 What the lexer stores for each word; §5 External lexer states; §6 Definition of punctuation; §7 Definition of white space; §8-11 Internal lexer states; §12-15 Feeding the lexer; §16 Lexing one character at a time; §17-20 Dealing with whitespace; §21-23 Completing a word; §24-25 Entering and leaving literal mode; §26-28 Breaking strings up at text substitutions

---

Template interpreter commands

```
1   {-callv:start_lexer}
```

Definitions

¶1. Lexical analysis is the process of reading characters from the source text files and forming them into globs which we call "words": the part of Inform which does this is the "lexical analyser", or lexer for short. The algorithms in this chapter are entirely routine, but occasional eye-opening moments come because natural language does not have the rigorous division between lexical and semantic parsing which programming language theory expects. For instance, we want NI to be case insensitive for the most part, but we cannot discard upper case entirely at the lexical stage because we will need it later to decide whether punctuation at the end of a quotation is meant to end the sentence making the quote, or not. Humans certainly read these differently:

> Say "Hello!" with alarm, ... Say "Hello!" With alarm, ...

And paragraph breaks can also have semantic meanings. A gap between two words does not end a sentence, but a paragraph break between two words clearly does. So semantic considerations occasionally infiltrate themselves into even the earliest parts of this chapter.

¶2. We must never lose sight of the origin of text, because we may need to print problem messages back to the user which refer to that original material. We record the provenance of text using the following structure; the `lexer_position` is such a structure, and marks where the lexer is currently reading.

```
typedef struct source_location {
    struct source_file *file_of_origin;                    or NULL if internally written and not from a file
    int line_number;                                           counting upwards from 1 within file (if any)
} source_location;

source_location lexer_position;
```

The structure source_location is shared with 2/html, 2/index, 2/lexi, 2/prob1, 3/read, 3/lwb, 4/sent, 4/edoc, 4/head, 8/sob, 10/isin, 12/cinv and 12/rb.

¶3. A word can be an English word such as `bedspread`, or a piece of punctuation such as `!`, or a number such as `127`, or a piece of quoted text of arbitrary size such as `"I summon up remembrance of things past"`.

The words found are numbered 0, 1, 2, ... in order of being read by the lexer. The first eight or so words come from the mandatory insertion text (see Read Source Text.w), then come the words from the primary source text, then those from the extensions loaded.

References to text throughout NI's data structure are often in the form of a pair of word numbers, usually called `w1` and `w2` or some variation on that, indicating the text which starts at word `w1` and finishes at `w2` (including both ends). Thus if the text is

> When to the sessions of sweet silent thought

then the eight words are numbered 0 to 7 and a reference to `w1=2`, `w2=5` would mean the sub-text "the sessions of sweet". The special null value `wn=-1` is used when no word reference has been made: never 0, as that would mean the first word in the list. The maximum legal word number is always one less than the following variable's value.

```
int lexer_wordcount;                                    Number of words read in to arrays
```

¶4. The lexical structure of source text. The following definitions are fairly self-evident: they specify which characters cause word divisions, or signal literals.

```
define STRING_BEGIN '"'                                 Strings are always double-quoted
define STRING_END '"'
define TEXT_SUBSTITUTION_BEGIN '['                      Inside strings, this denotes a text substitution
define TEXT_SUBSTITUTION_END ']'
define TEXT_SUBSTITUTION_SEPARATOR ','
define COMMENT_BEGIN '['                                Text between these, outside strings, is comment
define COMMENT_END ']'
define INFORM6_ESCAPE_BEGIN_1 '('                       Text beginning with this pair is literal I6 code
define INFORM6_ESCAPE_BEGIN_2 '-'
define INFORM6_ESCAPE_END_1 '-'
define INFORM6_ESCAPE_END_2 ')'
define INFORM7_ESCAPE_BEGIN_1 '('                       Text beginning with this pair is I7 text within I6 literals
define INFORM7_ESCAPE_BEGIN_2 '+'
define INFORM7_ESCAPE_END_1 '+'
define INFORM7_ESCAPE_END_2 ')'
define PARAGRAPH_BREAK '|'                              Inserted as a special word to mark paragraph breaks
define NEWLINE_IN_STRING ((char) 0x7f)                  Within quoted text, all newlines are converted to this
define PUNCTUATION_MARKS ".,:;?!(){}[]"                 Do not add to this list lightly: it could break much existing source
```

¶5. This seems a good point to describe how best to syntax-colour source text, something which the user interfaces do on every platform. By convention we are sparing with the colours: ordinary word-processing is not a kaleidoscopic experience (even when Microsoft Word's impertinent grammar checker is accidentally left switched on), and we want the experience of writing Inform source text to be like writing, not like programming. So we use just a little colour, and that goes a long way.

Because the Inform applications generally syntax-colour source text in the Source panel of the user interface, it is probably worth writing down the lexical specification. There are eight basic categories of text, and they should be detected in the following order, with the first category that applies being the one to determine the colour and/or font weight:

(1) Titling text (primary source text only: not found in extensions). If the first non-whitespace in the file is a double-quoted text (see (4a)), this is the title of the work.

(2) Documentation text (extension text only: not found in primary source). If a paragraph consists of a single non-whitespace token only, and that token is `----` (four hyphens in a row), then this paragraph and all subsequent text down to the bottom of the file.

(3) Heading text. If a paragraph consists of a single line only and which begins with one of the five words Volume, Book, Part, Chapter or Section, capitalised as here, then that paragraph is a heading. (A paragraph division is found at the start and end of a file, and also at any run of white space containing two or more newline characters: a newline can be any of the Unicode characters `0x000A`, `0x2028` or `0x2029`.)

(4a) Quoted text. Outside of (4b) and (4c), a double-quotation mark (in principle any of Unicode `0x0022`, `0x201C`, `0x201D`) begins quoted text provided it follows either whitespace, or the start of the file, or one of the punctuation marks in the `PUNCTUATION_MARKS` string defined above. Quoted text continues until the next double-quotation mark (or the end of the file if there isn't one, though NI would issue Problems if asked to compile this).

(4a1) Text substitution text. Within (4a) only, an open square bracket introduced text substitution matter which continues until the next close square bracket or the end of the quoted text. (Again, NI would issue problem messages if given a string malformed in this way.)

(4b) Comment text. Outside of (4a) and (4c), an open square bracket begins comment. Comment continues until the next *matching* close square bracket. (This is the case even if that is in double quotes within the comment, i.e., quotation marks should be ignored when matching `[` and `]` inside a comment.) Thus, nested comments are allowed, and the following text contains a single comment running from just after "the" through to the full stop:

```
Snow White and the [Seven Dwarfs [but not Doc]].
```

(4c) Literal I6 code. Outside of (4a) and (4b), the combination `(-` begins literal I6 matter. This matter continues until the next `-)` is reached. Within literal I6 matter, one can escape back into I7 source text using a matched pair of `(+` and `+)` tokens, but it really doesn't seem worth syntax colouring this very much. And the authors of Inform will lose no sleep if we miscolour this, for instance, especially if it deters people from such horrible coding practices:

```
(- Constant BLOB = (+ the total weight of things in (- selfobj -) +); -)
```

(5) Normal text. Everything else.

NI regards all of the Unicode characters `0x0009`, `0x000A`, `0x000D`, `0x0020`, `0x0085`, `0x00A0`, `0x02000` to `0x200A`, `0x2028` and `0x2029` as instances of white space. Of course, it's entirely open to the Inform user interfaces to not allow the user to key some of these codes, but we should bear in mind that projects using them might be created on one platform and then reopened on another one, so it's probably best to be careful.

¶6. These categories of text are conventionally displayed as follows:

(1) Titling text: black boldface.

(2) Documentation text: grey type.

(3) Heading text: black boldface, perhaps of a slightly larger point size.

(4a) Quoted text: dark blue boldface.

(4a1) Text substitution text: lighter blue and not boldface.

(4b) Comment text: darkish green type, perhaps of a slightly smaller point size.

(4c) Literal I6 code: grey type. (Inform for OS X rather coolly goes into I6 syntax-colouring, which is considerably harder, for this material: see *The Inform 6 Technical Manual* for an algorithm.)

(5) Normal text: black type.

¶7. What the lexer stores for each word. The lexer builds a small data structure for each individual word it reads.

```
typedef struct lexer_details {
    char *lw_text;                                  text of word after treatment to normalise
    char *lw_rawtext;                               original untouched text of word
    struct source_location lw_source;               where it was read from
    char lw_break;                                  the divider (space, tab, etc.) preceding it
    struct vocabulary_entry *lw_identity;           which distinct word
} lexer_details;

lexer_details *lw_array = NULL;                     a dynamically allocated (and mobile) array
int lexer_details_memory_allocated = 0;             bytes allocated to this array
int lexer_workspace_allocated = 0;                  bytes allocated to text storage
```

The structure lexer_details is shared with 2/isn, 2/java, 2/hdoc, 2/index, 2/lexi, 2/prob1, 2/dl, 3/read, 3/lwb, 3/lexs, 3/vocab, 4/sent, 4/ext, 4/iext, 4/edoc, 4/vm, 4/head, 4/rtree, 5/rel, 5/conj, 5/aph, 5/litp, 5/em, 5/unitr, 5/arch, 5/parse, 5/lit, 5/candd, 5/tandv, 5/candp, 7/kov, 7/data, 7/dim, 9/rsdt, 7/kix, 7/tc, 8/tass, 8/sob, 8/refpt, 8/mass, 9/qty, 9/wo, 9/mapbp, 9/prop, 9/madj, 9/cmpbp, 9/vpbp, 9/inf, 9/model, 9/cot, 9/inpw, 9/map, 9/tmap, 10/str, 10/tab, 10/eqns, 10/bib, 10/fig, 10/sfx, 10/exf, 10/isin, 11/act, 11/anl, 11/nap, 11/av, 12/ph, 12/phud, 12/phtd, 12/phsf, 12/cinv, 12/cph, 12/toph, 12/phin, 12/def, 12/tiph, 12/br, 12/rb, 13/tfg, 13/gv, 13/gl, 13/gtok, 13/gprv, 13/gpr and 13/test.

¶8. The following bounds on how much we can read are immutable without editing and recompiling Inform.

```
define TEXT_STORAGE_CHUNK_SIZE 600000       Must exceed MAX_VERBATIM_LENGTH+MAX_WORD_LENGTH
define MAX_STRING_LENGTH 3000               This is supposed to be interactive fiction...
define MAX_VERBATIM_LENGTH 200000           Largest quantity of Inform 6 which can be quoted verbatim.
define MAX_WORD_LENGTH 92                   Maximum length of any unquoted word
```

§**1.** The main text area of memory has a simple structure: it is allocated in one contiguous block, and at any given time the memory is used from the lowest address up to (but not including) the "high water mark", a pointer in effect to the first free character.

```
char *lexer_workspace;                                  Large area of contiguous memory for text
char *lexer_word;                                       Start of current word in workspace
char *lexer_hwm;                                        High water mark of workspace
char *lexer_workspace_end;          Pointer to just past the end of the workspace: HWM must not exceed this
```

```
void start_lexer(void) {
    lexer_wordcount = 0;
    ensure_lexer_space_up_to(50000);                    the Standard Rules are about 44,000 words
    allocate_lexer_workspace_chunk();
    vocab_start_hash_table();
}
```

The function start_lexer is invoked by a command in a `.i6t` template file.

§**2.** These are quite hefty memory allocations, with the expensive one – `lw_source` – also being the least essential to NI's running. But at least we use memory in a way at least vaguely related to the size of the source text, never using more than twice what we need, and we impose no absolute upper limits.

```
int current_lw_array_size = 0, next_lw_array_size = 75000;

void ensure_lexer_space_up_to(int n) {
    if (n < current_lw_array_size) return;
    int new_size = current_lw_array_size;
    while (n >= new_size) {
        new_size = next_lw_array_size;
        next_lw_array_size = next_lw_array_size*2;
    }
    lexer_details_memory_allocated = new_size*sizeof(lexer_details);
    lexer_details *new_lw_array =
        ((lexer_details *) (I7_calloc(new_size, sizeof(lexer_details), LEXER_WORDS_MREASON)));
    if (new_lw_array == NULL) fatal_error("Out of memory: unable to create lexer workspace");
    int i;
    for (i=0; i<new_size; i++) {
        if (i < current_lw_array_size) new_lw_array[i] = lw_array[i];
        else {
            new_lw_array[i].lw_text = NULL;
            new_lw_array[i].lw_rawtext = NULL;
            new_lw_array[i].lw_break = ' ';
            new_lw_array[i].lw_source.file_of_origin = NULL;
            new_lw_array[i].lw_source.line_number = -1;
            new_lw_array[i].lw_identity = NULL;
        }
    }
    if (lw_array) I7_free(lw_array, LEXER_WORDS_MREASON);
    lw_array = new_lw_array;
    current_lw_array_size = new_size;
}
```

The function ensure_lexer_space_up_to is called from 3/lwb.

§**3.** Inform would almost certainly crash if we wrote past the end of the workspace, so we need to watch for the water running high. The following routine checks that there is room for another `n` characters, plus a termination character, plus breathing space for a single character's worth of lookahead:

```
void ensure_lexer_hwm_can_be_raised_by(int n, int transfer_partial_word) {
    if (lexer_hwm + n + 2 >= lexer_workspace_end) {
        char *old_hwm = lexer_hwm;
        allocate_lexer_workspace_chunk();
        if (transfer_partial_word) {
            *(lexer_hwm++) = ' ';
            char *new_lword = lexer_hwm;
            while (lexer_word < old_hwm) *(lexer_hwm++) = *(lexer_word++);
            lexer_word = new_lword;
        }
        if (lexer_hwm + n + 2 >= lexer_workspace_end)
            internal_error("further allocation failed to liberate enough space");
    }
}
void allocate_lexer_workspace_chunk(void) {
    lexer_workspace = ((char *) (I7_malloc(TEXT_STORAGE_CHUNK_SIZE, LEXER_TEXT_MREASON)));
    lexer_workspace_allocated += TEXT_STORAGE_CHUNK_SIZE;
    lexer_hwm = lexer_workspace;
    lexer_workspace_end = lexer_workspace + TEXT_STORAGE_CHUNK_SIZE;
}
```

§**4.** We occasionally want to reprocess the text of a word again in higher-level parsing, and it's convenient to use the lexer workspace to store the results of such a reprocessed text. The following routine makes a persistent copy of its argument, then: it should never be used while the lexer is actually running.

```
char *copy_to_lexer_memory(char *p) {
    ensure_lexer_hwm_can_be_raised_by(strlen(p), FALSE);
    char *q = lexer_hwm;
    lexer_hwm = q + strlen(p) + 1;
    strcpy(q, p);
    return q;
}
```

The function copy_to_lexer_memory is called from 3/lexs.

§5. External lexer states. The lexer is a finite state machine at heart. Its current state is the collective value of an extensive set of variables, almost all of them flags, but with three exceptions this state is used only within the lexer.

The three exceptional modes are by default both off and by default they stay off: the lexer never goes into either mode by itself.

`lexer_divide_strings_at_text_substitutions` is used by some of the lexical writing-back machinery, when it has been decided to compile something like

say "[The noun] falls onto [the second noun]."

In its ordinary mode, with this setting off, the lexer will render this as two words, the second being the entire quoted text. But if `lexer_divide_strings_at_text_substitutions` is set then the text is reinterpreted as

say The noun, " falls onto ", the second noun, "."

which runs to eleven words, three of them commas (punctuation always counts as a word).

`lexer_wait_for_dashes` is set by the extension-reading machinery, in cases where it wants to get at the documentation text of an extension but does not want to have to fill NI's memory with the source text of its code. In this mode, the lexer ignores the whole stream of words until it reaches `----`, the special marker used in extensions to divide source text from documentation: it then drops out of this mode and back into normal running, so that subsequent words are lexed as usual.

`lexer_treat_slash_as_punctuation` is a mode used when lexically writing back grammar tokens, such as the one here:

Understand "get away/off/out" as exiting.

NI would ordinarily lex the text away/off/out as one single word – so that something like "on/off switch" would be regarded as two words not four – but with slash treated as a punctuation mark, we instead read "away / off / out", a sequence of five lexical words.

```
int lexer_divide_strings_at_text_substitutions;          Break up text substitutions in quoted text
int lexer_wait_for_dashes;                               Ignore all text until first ---- found
int lexer_treat_slash_as_punctuation;                    Regard / as a word-dividing punctuation mark
```

§6. Definition of punctuation. As we have seen, the question of whether something is a punctuation mark or not depends slightly on the context:

```
int is_punctuation(char c) {
    int i;
    for (i=0; PUNCTUATION_MARKS[i]; i++)
        if (c == PUNCTUATION_MARKS[i])
            return TRUE;
    if ((c=='/') && (lexer_treat_slash_as_punctuation)) return TRUE;
    return FALSE;
}
```

The function is_punctuation is called from 2/java, 2/hdoc, 3/read and 9/map.

§7. Definition of white space. The following macro (to save time over a function call) is highly dangerous, and of the kind which all books on C counsel against. If it were called with any argument whose evaluation had side-effects, disaster would ensue. It is therefore used only twice, with care, and only in this section below.

```
define is_whitespace(c) ((c == ' ') || (c == '\n') || (c == '\t'))
```

§**8. Internal lexer states.** The current situation of the lexer is specified by the collective values of all of the following. First, the start of the current word being recorded, and the current high water mark – those are defined above. Second, we need the feeder machinery to maintain a variable telling us the previous character in the raw, un-respaced source. We need to be a little careful about the type of this: it needs to be an `int` so that it can on occasion hold the pseudo-character value `EOF`. (The problem is not that we need to recognise an `EOF` accurately: we don't. The problem is that `EOF` accidentally cast into a `char` collides with a French accented letter on some platforms, depending on whether `char` is signed or not.)

```
int lxs_previous_char_in_raw_feed;
```
Preceding character in raw file read

§**9.** There are four kinds of word: ordinary words, [comments in square brackets], "strings in double quotes," and `(- I6_inclusion_text -)`. The latter three are kinds are collectively called literals. As each word is read, the variable `lxs_kind_of_word` holds what it is currently believed to be.

```
define ORDINARY_KW 0
define COMMENT_KW 1
define STRING_KW 2
define I6_INCLUSION_KW 3
```

```
int lxs_kind_of_word;
```
One of the defined values above

§**10.** While there are a pile of state variables below, the basic situation is that the lexer has two main modes: ordinary mode and literal mode, determined by whether `lxs_literal_mode` is false or true. It might look as if this variable is redundant – can't we simply see whether `lxs_kind_of_word` is `ORDINARY_KW` or not? – but in fact we return to ordinary mode slightly before we finish recording a literal, as we shall see, so it is important to be able to switch in and out of literal mode without changing the kind of word.

```
int lxs_literal_mode;
```
Are we in literal or ordinary mode?

significant in ordinary mode:

```
char lxs_most_significant_space_char;
```
Most significant whitespace character preceding

```
int lxs_number_of_tab_stops;
```
Number of consecutive tabs

```
int lxs_this_line_is_empty_so_far;
```
Current line white space so far?

```
int lxs_this_word_is_empty_so_far;
```
Looking for a word to start?

```
int lxs_scanning_text_substitution;
```
Used to break up strings at [substitutions]

significant in literal mode:

```
int lxs_comment_nesting;
```
For square brackets within square brackets

```
int lxs_string_soak_up_spaces_mode;
```
Used to fold strings which break across lines

§**11.** The lexer needs to be reset each time it is used on a given feed of text, whether from a file or internally. Note that this resets both external and internal states to their defaults (the default for external states always being "off").

```
void reset_lexer(void) {
    lexer_word = lexer_hwm;
    lxs_previous_char_in_raw_feed = EOF;
    reset the external states
    lexer_wait_for_dashes = FALSE;
    lexer_treat_slash_as_punctuation = FALSE;
    lexer_divide_strings_at_text_substitutions = FALSE;
    reset the internal states
    lxs_most_significant_space_char = '\n';                 we imagine each lexer feed starting a new line
    lxs_number_of_tab_stops = 0;                            but not yet indented with tabs
    lxs_this_line_is_empty_so_far = TRUE;                   clearly
    lxs_this_word_is_empty_so_far = TRUE;                   likewise
    lxs_literal_mode = FALSE;                               begin in ordinary mode...
    lxs_kind_of_word = ORDINARY_KW;                         ...expecting an ordinary word
    lxs_string_soak_up_spaces_mode = FALSE;
    lxs_scanning_text_substitution = FALSE;
    lxs_comment_nesting = 0;
}
```

§**12. Feeding the lexer.** The lexer takes its input as a stream of characters, sent from a "feeder routine": there are two of these, one sending the stream from a file, the other from a C string. A feeder routine is required to:

(1) call `lexer_feed_begins` before sending the first character,

(2) send ISO Latin-1 characters which also exist in ZSCII, in sequence, via `lexer_feed_triplet`,

(3) conclude by calling `lexer_feed_ends`.

Only one feeder can be active at a time, as the following routines ensure.

```
int lexer_feed_started_at = -1;
void lexer_feed_begins(source_location sl) {
    if (lexer_feed_started_at >= 0) internal_error("one lexer feeder interrupted another");
    lexer_feed_started_at = lexer_wordcount;
    lexer_position = sl;
    reset_lexer();
    LOGIF(LEXER, "Lexer feed began at %d\n", lexer_feed_started_at);
}
void lexer_feed_ends(int *range_read_w1, int *range_read_w2, int extra_padding,
    char *problem_source_description) {
    if (lexer_feed_started_at == -1) internal_error("lexer feeder ended without starting");
    ⟨Feed whitespace as padding 13⟩;
    *range_read_w1 = lexer_feed_started_at; *range_read_w2 = lexer_wordcount-1;
    lexer_feed_started_at = -1;
    LOGIF(LEXER, "Lexer feed ended at %d\n", *range_read_w2);
    ⟨Issue Problem messages if feed ended in the middle of quoted text, comment or verbatim I6 14⟩;
}
```

The function lexer_feed_begins is called from 3/read and 3/lwb.

The function lexer_feed_ends is called from 3/read and 3/lwb.

§**13.** White space padding guarantees that a word running right up to the end of the feed will be processed, since (outside literal mode) that white space signals to the lexer that a word is complete. (If we are in literal mode at the end of the feed, problem messages are produced. We code NI to ensure that this never occurs when feeding our own C strings through.)

At the end of each complete file, we also want to ensure there is always a paragraph break, because this simplifies the parsing of headings (which in turn is because a file boundary counts as a super-heading-break, and headings are only detected as stand-alone paragraphs). We add a bit more white space than is strictly necessary, because it saves worrying about whether it is safe to look ahead to characters further on in the lexer's workspace when we are close to the high water mark, and because it means that a source file which is empty or contains only a byte-order marker comes out as at least one paragraph, even if a blank one.

⟨Feed whitespace as padding 13⟩ ≡

```
if (extra_padding == FALSE) {
    feed_char_into_lexer(' ');
} else {
    feed_char_into_lexer(' ');
    feed_char_into_lexer('\n');
    feed_char_into_lexer('\n');
    feed_char_into_lexer('\n');
    feed_char_into_lexer('\n');
    feed_char_into_lexer(' ');
}
```

This code is used in §12.

§**14.** These problem messages can, of course, never result from text which NI is feeding into the lexer itself, independently of source files. That would be a bug, and NI is bug-free, so it follows that it could never happen.

⟨Issue Problem messages if feed ended in the middle of quoted text, comment or verbatim I6 14⟩ ≡

```
if (lxs_kind_of_word != ORDINARY_KW) {
    if (lexer_wordcount >= 20) {
        LOG("Last words: $W\n", lexer_wordcount-20, lexer_wordcount-1);
    } else if (lexer_wordcount >= 1) {
        LOG("Last words: $W\n", 0, lexer_wordcount-1);
    } else {
        LOG("No words recorded\n");
    }
}
if (lxs_kind_of_word == STRING_KW) {
    lexical_problem(_P_(C3UnendingQuote),
        "Source file ended in the middle of quoted text", problem_source_description,
        "This probably means that a quotation mark is missing "
        "somewhere. If you are using Inform with syntax colouring, "
        "look for where the quoted-text colour starts. (Sometimes "
        "this problem turns up because a piece of quoted text contains "
        "a text substitution in square brackets which in turn contains "
        "another piece of quoted text - this is not allowed, and causes "
        "me to lose track.)");
}
if (lxs_kind_of_word == COMMENT_KW) {
    lexical_problem(_P_(C3UnendingComment),
        "Source file ended in the middle of a comment", problem_source_description,
        "This probably means that a ']' is missing somewhere. "
```

```
            "(If you are using Inform with syntax colouring, look for "
            "where the comment colour starts.) Inform's convention on "
            "'nested comments' is that each '[' in a comment must be "
            "matched by a corresponding ']': so for instance '[This "
            "[even nested like so] acts as a comment]' is a single "
            "comment - the first ']' character matches the second '[' "
            "and so doesn't end the comment: only the second ']' ends "
            "the comment.");
    }
    if (lxs_kind_of_word == I6_INCLUSION_KW) {
        lexical_problem(_P_(C3UnendingI6),
            "Source file ended in the middle of a verbatim passage "
            "of Inform 6 code", problem_source_description,
            "This probably means that a '-)' is missing.");
    }
    lxs_kind_of_word = ORDINARY_KW;
```

This code is used in §12.

§**15.** The feeder routine is required to send us a triple each time: `cr` must be a valid character (see above) and may not be `EOF`; `last_cr` must be the previous one or else perhaps `EOF` at the start of feed; while `next_cr` must be the next or else perhaps `EOF` at the end of feed.

Spaces, often redundant, are inserted around punctuation unless the lexer is in literal mode (inside strings, for instance), and except in the case where a single punctuation mark occurs in between two digits. This is done so that, for instance, "0.91" does not split into three words in the lexer. We do not count square brackets here, because if we did, that would cause trouble in parsing

```
say "[if M is less than 10]0[otherwise]1";
```

where the `0]0` would go unbroken in `lexer_divide_strings_at_text_substitutions` mode, and therefore the `]` would remain glued to the preceding text.

```
void lexer_feed_triplet(int last_cr, int cr, int next_cr) {
    lxs_previous_char_in_raw_feed = last_cr;
    if ((lxs_literal_mode == FALSE) && (is_punctuation(cr))) {
        if ((isdigit(last_cr))
            && ((next_cr == '-') || (isdigit(next_cr)))
            && (cr != '[') && (cr != ']')) {
            feed_char_into_lexer(cr);
        } else {
            feed_char_into_lexer(' ');
            feed_char_into_lexer(cr);              which might take us into literal mode, so to be careful...
            if (lxs_literal_mode == FALSE) feed_char_into_lexer(' ');
        }
    } else feed_char_into_lexer(cr);
    if ((cr == '\n') && (lexer_position.file_of_origin))
        lexer_position.line_number++;
}
```

The function lexer_feed_triplet is called from 3/read and 3/lwb.

§**16. Lexing one character at a time.** We can think of characters as a stream of differently-coloured marbles, flowing from various sources into a hopper above our marble-sorting machine. The hopper lets the marbles drop through one at a time into the mechanism below, but inserts transparent glass marbles of its own on either side of certain colours of marble, so that the sequence of marbles entering the mechanism is no longer the same as that which entered the hopper. Moreover, the mechanism can itself cause extra marbles of its choice to drop in from time to time, further interrupting the original flow.

The following routine is the mechanism which receives the marbles. We want the marbles to run swiftly through and either be pulverised to glass powder, or dropped into the output bucket, as the mechanism chooses. (Whatever marbles from the original source survive will always emerge in their original order, though.) Every so often the mechanism decides that it has completed one batch, and moves on to dropping marbles into the next bucket.

The marbles are characters; transparent glass ones are whitespace, which will always now be `' '`, `'\t'` or `'\n'`; the routine `lexer_feed_triplet` above was the hopper; the routine `feed_char_into_lexer`, which occupies the whole of the rest of this section, is the mechanism which takes each marble in turn. (On occasion it calls itself recursively to cause extra characters of its choice to drop in.) The batches are words, and the bucket receiving the surviving marbles is the sequence of characters starting at `lexer_word` and extending to `lexer_hwm-1`.

```
void feed_char_into_lexer(char c) {
    ensure_lexer_hwm_can_be_raised_by(MAX_WORD_LENGTH, TRUE);
    if (lxs_literal_mode) {
        ⟨Contemplate leaving literal mode 25⟩;
        if (lxs_kind_of_word == STRING_KW) {
            ⟨Force string division at the start of a text substitution, if necessary 26⟩;
            ⟨Soak up whitespace around line breaks inside a literal string 20⟩;
        }
    }
    whitespace outside literal mode ends any partly built word and need not be recorded
    if ((lxs_literal_mode == FALSE) && (is_whitespace(c))) {
        ⟨Admire the texture of the whitespace 17⟩;
        if (lexer_word != lexer_hwm) ⟨Complete the current word 21⟩;
        if (c == '\n') ⟨Line break outside a literal 19⟩;
        return;
    }
    otherwise record the current character as part of the word being built
    *(lexer_hwm++) = c;
    if (lxs_scanning_text_substitution) {
        ⟨Force string division at the end of a text substitution, if necessary 27⟩;
    }
    if (lxs_this_word_is_empty_so_far) {
        ⟨Look at recent whitespace to see what break it followed 18⟩;
        ⟨Contemplate entering literal mode 24⟩;
    }
    lxs_this_word_is_empty_so_far = FALSE;
    lxs_this_line_is_empty_so_far = FALSE;
}
```

§**17. Dealing with whitespace.** Let's deal with the different textures of whitespace first, as these are surprisingly rich all by themselves.

The following keeps track of the biggest white space character it has seen of late, ranking newlines bigger than tabs, which are in turn bigger than spaces; and it counts up the number of tabs it has seen (cancelling back to none if a newline is found).

⟨Admire the texture of the whitespace 17⟩ ≡

```
if (c == '\t') {
    lxs_number_of_tab_stops++;
    if (lxs_most_significant_space_char == ' ') lxs_most_significant_space_char = '\t';
}
if (c == '\n') {
    lxs_number_of_tab_stops = 0;
    lxs_most_significant_space_char = '\n';
}
```

This code is used in §16.

§**18.** To recall: we need to know what kind of whitespace prefaces each word the lexer records.

When we record the first character of a new word, it cannot be whitespace, but it probably follows a sequence of one or more whitespace characters, and the code in the previous paragraph has been watching them for us.

⟨Look at recent whitespace to see what break it followed 18⟩ ≡

```
if ((lxs_most_significant_space_char == '\n') && (lxs_number_of_tab_stops >= 1))
    lw_array[lexer_wordcount].lw_break = '0' + lxs_number_of_tab_stops;    newline followed by 1 or
more tabs
else
    lw_array[lexer_wordcount].lw_break = lxs_most_significant_space_char;
lxs_most_significant_space_char = ' ';          waiting for the next run of whitespace, after this word
lxs_number_of_tab_stops = 0;
```

This code is used in §16.

§**19.** Line breaks are usually like any other white space, if we are outside literal mode, but we want to keep an eye out for paragraph breaks, because these are sometimes semantically meaningful in NI and so cannot be discarded. A paragraph break is converted into a special "divider" word.

⟨Line break outside a literal 19⟩ ≡

```
if (lxs_this_line_is_empty_so_far) {
    feed_char_into_lexer(PARAGRAPH_BREAK);
    feed_char_into_lexer(' ');
}
lxs_this_line_is_empty_so_far = TRUE;
```

This code is used in §16.

§**20.** When working through a literal string, a new-line together with any preceding whitespace is converted into a single space character, and we enter "soak up spaces" mode: in which mode, any subsequent whitespace is ignored until something else is reached. If we reach another new-line while still soaking up, then the literal text contained a paragraph break. In this instance, the splurge of whitespace is converted not to a single space `" "` but to two forced newlines in quick succession. In other words, paragraph breaks in literal strings are converted to codes which will make Inform print a paragraph break at run-time.

⟨Soak up whitespace around line breaks inside a literal string 20⟩ ≡

```
if (lxs_string_soak_up_spaces_mode) {
    switch(c) {
        case ' ': case '\t': lexer_hwm--; break;
        case '\n':
            *(lexer_hwm-1) = NEWLINE_IN_STRING;
            c = NEWLINE_IN_STRING;
            break;
        default: lxs_string_soak_up_spaces_mode = FALSE; break;
    }
}
if (c == '\n') {
    while (is_whitespace(*(lexer_hwm-1))) lexer_hwm--;
    lxs_string_soak_up_spaces_mode = TRUE;
}
```

This code is used in §16.

§**21. Completing a word.** Outside of whitespace, then, our word (whatever it was – ordinary word, literal string, I6 insertion or comment) has been stored character by character at the steadily rising high water mark. We have now hit the end by reaching whitespace (in the case of a literal, this has happened because we found the end of the literal, escaped literal mode, and then hit whitespace). The start of the word is at `lexer_word`; the last character is stored just below `lexer_hwm`.

⟨Complete the current word 21⟩ ≡

```
*lexer_hwm++ = 0;                                   terminate the current word as a C string
if ((lexer_wait_for_dashes) && (strcmp(lexer_word, "----") == 0))
    lexer_wait_for_dashes = FALSE;                   our long wait for documentation is over
if ((lexer_wait_for_dashes == FALSE) && (lxs_kind_of_word != COMMENT_KW)) {
    ⟨Issue problem message and truncate if over maximum length for what it is 22⟩;
    ⟨Store everything about the word except its break, which we already know 23⟩;
}
now get ready for what we expect by default to be an ordinary word next
lexer_word = lexer_hwm;
lxs_this_word_is_empty_so_far = TRUE;
lxs_kind_of_word = ORDINARY_KW;
```

This code is used in §16.

§**22.** Note that here we are recording either an ordinary word, a literal string or a literal I6 insertion: comments are also literal, but are thrown away, and do not come here.

Some readers will be wondering about Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogochuchaf (the upper old part of the village of Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch, on the Welsh isle of Anglesey), but this has a mere 63 letters, and in any case the name was "improved" by the village cobbler in the mid-19th century to make it a tourist attraction for the new railway age.

⟨Issue problem message and truncate if over maximum length for what it is 22⟩ ≡

```
int len = strlen(lexer_word), max_len = MAX_WORD_LENGTH;
if (lxs_kind_of_word == STRING_KW) max_len = MAX_STRING_LENGTH;
if (lxs_kind_of_word == I6_INCLUSION_KW) max_len = MAX_VERBATIM_LENGTH;
if (len > max_len) {
    lexer_word[max_len] = 0;                                    truncate to its maximum length
    if (lxs_kind_of_word == STRING_KW)
        lexical_problem(_P_(C3TooMuchQuotedText),
            "Too much text in quotation marks", lexer_word,
            "...\" The maximum length is very high, so this is more "
            "likely to be because a close quotation mark was "
            "forgotten.");
    else {
        if (lxs_kind_of_word == I6_INCLUSION_KW) {
            lexer_word[100] = 0;                    to avoid an absurdly long problem message
            lexical_problem(_P_(Untestable),                    well, not at all conveniently
                "Verbatim Inform 6 extract too long", lexer_word,
                "... -). The maximum length is quite high, so this "
                "may be because a '-)' was forgotten. Still, if "
                "you do need to paste a huge I6 program in, try "
                "using several verbatim inclusions in a row.");
        } else
            lexical_problem(_P_(C3WordTooLong),
            "Word too long", lexer_word,
            "(Individual words of unquoted text can run up to "
            "92 letters long, in order to allow for the longest "
            "recognised place name in the English speaking world: "
            "a hill in New Zealand called Taumatawhakatang-"
            "ihangakoauauot-amateaturipukaka-pikimaunga-"
            "horonuku-pokaiwhenuak-itanatahu. You say tomato, "
            "I say taumatawhakatang-...)");
    }
}
```

This code is used in §21.

§**23.** We recorded the break for the word when it started (recall that, even if the current word is a literal, its first character was read outside literal mode, so it started out in life as an ordinary word and therefore had its break recorded). So now we need to set everything else about it, and to increment the word-count. We must not allow this to reach its maximum, since this would allow the next word's break setting to overwrite the array.

For ordinary words (but not literals), the copy of a word in the main array `lw_text` is lowered in case. The original is preserved in `lw_rawtext` and is used to print more attractive error messages, and also to enable a few semantic parts of NI to be case sensitive. This copying means that in the worst case – when we complete an ordinary word of maximal length – we need to consume an additional `MAX_WORD_LENGTH+2` bytes of the lexer's workspace, which is why that was the amount we checked to ensure existed when the lexer was called. The lowering loop can therefore never overspill the workspace.

⟨Store everything about the word except its break, which we already know 23⟩ ≡

```
lw_array[lexer_wordcount].lw_rawtext = lexer_word;
lw_array[lexer_wordcount].lw_source = lexer_position;
if (lxs_kind_of_word == ORDINARY_KW) {
    int i;
    lw_array[lexer_wordcount].lw_text = lexer_hwm;
    for (i=0; lexer_word[i]; i++) *(lexer_hwm++) = tolower(lexer_word[i]);
    *(lexer_hwm++) = 0;
} else {
    lw_array[lexer_wordcount].lw_text = lw_array[lexer_wordcount].lw_rawtext;
}
vocab_identify_word(lexer_wordcount);          which sets lw_array[lexer_wordcount].lw_identity
lexer_wordcount++;
ensure_lexer_space_up_to(lexer_wordcount);
```

This code is used in §21.

§**24. Entering and leaving literal mode.** After a character has been stored, in ordinary mode, we see if it provokes us into entering literal mode, by signifying the start of a comment, string or passage of verbatim Inform 6.

In the case of a string, we positively want to keep the opening character just recorded as part of the word: it's the opening double-quote mark. In the case of a comment, we don't care, as we're going to throw it away anyhow; as it happens, we keep it for now. But in the case of an I6 escape we are in danger, because of the auto-spacing around brackets, of recording two words

```
( -something
```

when in fact we want to record

```
(- something
```

We do this by adding a hyphen to the previous word (the `(` word), and by throwing away the hyphen from the material of the current word.

⟨Contemplate entering literal mode 24⟩ ≡

```
switch(c) {
    case COMMENT_BEGIN:
        lxs_literal_mode = TRUE; lxs_kind_of_word = COMMENT_KW;
        lxs_comment_nesting = 1;
        break;
    case STRING_BEGIN:
        lxs_literal_mode = TRUE; lxs_kind_of_word = STRING_KW;
        break;
```

```
        case INFORM6_ESCAPE_BEGIN_2:
            if (lxs_previous_char_in_raw_feed != INFORM6_ESCAPE_BEGIN_1) break;
            lxs_literal_mode = TRUE; lxs_kind_of_word = I6_INCLUSION_KW;
            because of spacing around punctuation outside literal mode, the ( became a word
            if (lexer_wordcount > 0) {                    this should always be true: just being cautious
                lw_array[lexer_wordcount-1].lw_text = "(-";     change the previous word's text from ( to
(-
                lw_array[lexer_wordcount-1].lw_rawtext = "(-";
                vocab_identify_word(lexer_wordcount-1);                                   and re-identify
            }
            lexer_hwm--;                          erase the just-recorded INFORM6_ESCAPE_BEGIN_2 character
            break;
    }
```

This code is used in §16.

§**25.** So literal mode is used for comments, strings and verbatim passages of Inform 6 code. We are in this mode when scanning only the *middle* of the literal: after all, we scanned (and recorded) the start of the literal in ordinary mode, before noticing that the character(s) marked the onset of a literal.

Note that, when we leave literal mode, we set the current character to a space. This means the character forcing our departure is lost and not recorded: but we only actually want it in the case of strings (because we prefer to record them in the form `"frogs and lilies"` rather than `"frogs and lilies`, for tidiness's sake). And so for strings we explicitly record a close quotation mark.

The new current character, being a space and thus whitespace outside of literal mode, triggers the completion of the word, recording whatever literal we have just made. (Or, if it was a comment, discarding it.) `lxs_kind_of_word` continues to hold the kind of literal we have just finished.

⟨Contemplate leaving literal mode 25⟩ ≡

```
    switch(lxs_kind_of_word) {
        case COMMENT_KW:
            if (c == COMMENT_BEGIN) lxs_comment_nesting++;
            if (c == COMMENT_END) {
                lxs_comment_nesting--;
                if (lxs_comment_nesting == 0) lxs_literal_mode = FALSE;
            }
            break;
        case STRING_KW:
            if (c == STRING_END) {
                lxs_string_soak_up_spaces_mode = FALSE;
                *(lexer_hwm++) = c;                    record the STRING_END character as part of the word
                lxs_literal_mode = FALSE;
            }
            break;
        case I6_INCLUSION_KW:
            if ((c == INFORM6_ESCAPE_END_2) &&
                (lxs_previous_char_in_raw_feed == INFORM6_ESCAPE_END_1)) {
                *lexer_hwm--;                  erase the INFORM6_ESCAPE_END_1 character recorded last time
                lxs_literal_mode = FALSE;
            }
            break;
        default: internal_error("in unknown literal mode");
    }
    if (lxs_literal_mode == FALSE) c = ' ';                                  trigger completion of this word
```

This code is used in §16.

§**26. Breaking strings up at text substitutions.** When text contains text substitutions, these are ordinarily ignored by the lexer, but in `lexer_divide_strings_at_text_substitutions` mode, we need to force strings to end and resume at the two ends of each substitution. For instance:

"Hello, [greeted person]. Do you make it [supper time]?"

must be split as

```
"Hello, " , greeted person , ". Do you make it " , supper time , "?"
```

where our original single text literal is now three text literals, plus eight ordinary words (four of them commas).

Note that each open square bracket, and each close square bracket, has been removed and become a comma word. We see to open squares before we come to recording the character, so to get rid of the `[` character, we change `c` to a space:

⟨Force string division at the start of a text substitution, if necessary 26⟩ ≡

```
if ((lexer_divide_strings_at_text_substitutions) && (c == TEXT_SUBSTITUTION_BEGIN)) {
    feed_char_into_lexer(STRING_END);                                   feed " to close the old string
    feed_char_into_lexer(' ');
    feed_char_into_lexer(TEXT_SUBSTITUTION_SEPARATOR);                  feed , to start new word
    c = ' ';                              the lexer now goes on to record a space, which will end the , word
    lxs_scanning_text_substitution = TRUE;                  but remember that we must get back again
}
```

This code is used in §16.

§**27.** Whereas we see to close squares after recording the character, so we have to erase it to get rid of the `]`. Note that since this was read in ordinary mode, it was automatically spaced (being punctuation), and that therefore the feeder above has just sent the second of a sequence of three characters: space, `]`, space. That means we have recorded, so far, a one-character word in ordinary mode, whose text consists only of `]`. By overwriting this with a comma, we instead get a one-character word in ordinary mode whose text consists only of a comma. We then feed a space to end that word; then feed a double-quote to start text again.

But, it might be objected: surely the feeder above is still poised with that third character in its sequence space, `]`, space, and that means it will now feed a spurious space into the start of our resumed text? Happily, the answer is no: this is why the feeder above checks that it is still in ordinary mode before sending that third character. Having open quotes again, we have put the lexer into literal mode: and so the spurious space is never fed, and there is no problem.

⟨Force string division at the end of a text substitution, if necessary 27⟩ ≡

```
if ((lexer_divide_strings_at_text_substitutions) && (c == TEXT_SUBSTITUTION_END)) {
    lxs_scanning_text_substitution = FALSE;
    *(lexer_hwm-1) = TEXT_SUBSTITUTION_SEPARATOR;              overwrite recorded copy of ] with ,
    feed_char_into_lexer(' ');                                 then feed a space to end the , word
    feed_char_into_lexer(STRING_BEGIN);                          then feed " to open a new string
}
```

This code is used in §16.

§**28.** Finally, note that the breaking-up process may result in empty strings where square brackets abut each other or the ends of the original string. Thus

"[The noun] is on the [colour][style] table."

is split as: `"" , The noun , " is on the " , colour , "" , style , " table."` This is not a bug: empty strings are legal. It's for higher-level code to remove them if they aren't wanted.

Read Source Text 3/read

Purpose

This is where source text is read in, whether from extension files or from the main source text file, and fed into the lexer.

3/read.§5-8 Feeding whole files into the lexer; §9-12 Reading UTF-8 files

---

Template interpreter commands

```
1   {-callv:read_primary_source_text}
```

Definitions

¶1. The source text is drawn almost entirely from the primary source file and the extensions, but NI does also inject small amounts of source text of its own (for instance, when a new kind of value is created, the data types interpreter does this).

In particular, every source text read into NI is automatically prefixed by the following eight words – if NI were a computer, this would be the BIOS which boots up its operating system. (In that the rest of the creation of the I7 world model is handled by source text in the Standard Rules.) The stroke word at the end of the sentence is the lexer's representation of a paragraph break.

```
define MANDATORY_INSERTED_TEXT "Include the Standard Rules by Graham Nelson. | "
```

¶2. Because of this mandatory insertion, one extension, the Standard Rules, is compulsorily included in every run. So there will certainly be at least two files of source text to be read, and quite possibly more.

Each separate file of source text actually read has its identity docketed in a `source_file` structure, as follows. The "primary" source file is the one with allocation ID 0: the first to be opened, which is always the one described to NI on the command line. (If we use NI as part of Inform 7, this will be the text in the Source panel.) The Standard Rules then come next, with allocation ID 1.

```
typedef struct source_file {
    char filename[MAX_FILENAME_LENGTH];
    char leafname[MAX_FILENAME_LENGTH];
    int words_of_source;                          word count, omitting comments and verbatim matter
    int words_of_quoted_text;                     word count for text in double-quotes
    FILE *handle;                                 file handle while open
    struct extension_file *extension_provided;    meets this extension request
    MEMORY_MANAGEMENT
} source_file;

source_file *primary_source_file = NULL;          first to be opened
```

The structure source_file is private to this section.

---

§**1.** There is no real difference between the loading of the primary source text and the loading of an extension's text, except for the descriptions we supply in case of any problem messages which might need to be issued, and for the fact that the mandatory insertion text is loaded before the primary source text.

```
int read_extension_source_text(extension_file *EF, char *filename,
    char *synopsis, int documentation_only) {
    int rv = read_file(pathname_of_extensions, filename, synopsis, EF, documentation_only);
    if (dl_this(LEXER_DA)) log_lexer_output();
    return rv;
}
void read_primary_source_text(void) {
    feed_into_lexer(MANDATORY_INSERTED_TEXT, FALSE, FALSE);
    read_further_mandatory_text();
    if (bundle_name != NULL)
        read_file(bundle_name, source_filename_relative_to_bundle("story.ni"),
            "your source text", NULL, FALSE);
    else read_file(NULL, source_text_file, "the source text", NULL, FALSE);
}
```

The function read_extension_source_text is called from 4/iext.
The function read_primary_source_text is invoked by a command in a `.i6t` template file.

§**2.** Either way, we use the following code. The `read_file` function returns one of the following values to indicate the source of the source: the value only really tells us something we didn't know in the case of extensions, but in that event the Extensions.w routines do indeed want to know this. (Area 51 is reserved for extensions of alien origin, but the relevant source code is classified.)

```
define ORIGIN_WAS_PRIMARY_SOURCE 0
define ORIGIN_WAS_USER_EXTENSIONS_AREA 1
define ORIGIN_WAS_BUILT_IN_EXTENSIONS_AREA 2
```

```
int read_file(char *pathname, char *leafname, char *synopsis, extension_file *EF,
    int documentation_only) {
    int origin_tried;
    source_file *sf = CREATE(source_file);
    if (EF == NULL) {
        primary_source_file = sf;
        origin_tried = ORIGIN_WAS_PRIMARY_SOURCE;
    } else {
        origin_tried = ORIGIN_WAS_USER_EXTENSIONS_AREA;
    }
    sf->words_of_source = 0;
    sf->words_of_quoted_text = 0;
    sf->extension_provided = EF;
    ⟨Set pathname and filename, and open file 3⟩;
    if (EF) ef_set_corresponding_source_file(EF, sf);
    feed_file_into_lexer(sf, leafname, documentation_only);
    fclose(sf->handle);
    ⟨Tell console output about the file 4⟩;
    return origin_tried;
}
```

§**3.** The primary source text must be found where we expect it, or a fatal error is issued. An extension, however, can be in one of two places: the user's own repository of installed extensions, or the built-in stock. We must try each possibility – in that order, so that the user can supplant the built-in extensions by installing hacked versions of her own – and in the event of failing, we issue only a standard Inform problem message and continue. While meaningful compilation is unlikely to succeed now, this is not a fatal error, because fatality would cause the user interface application to communicate the problem badly.

⟨Set pathname and filename, and open file 3⟩ ≡

```
Retry:
if (pathname == NULL) strcpy(sf->filename, leafname);
else sprintf(sf->filename, "%s%c%s", pathname, FOLDER_SEPARATOR, leafname);
strcpy(sf->leafname, leafname);
if (EF) sf->handle = iso_fopen_caseless(sf->filename, "r");
else sf->handle = iso_fopen(sf->filename, "r");
if ((sf->handle == NULL) && (EF)) {
    char extended_name[MAX_FILENAME_LENGTH];
    sprintf(extended_name, "%s.i7x", sf->filename);
    sf->handle = iso_fopen_caseless(extended_name, "r");
    if (sf->handle) strcpy(sf->filename, extended_name);
}
if (sf->handle == NULL) {
    if (EF) {
        if (origin_tried == ORIGIN_WAS_USER_EXTENSIONS_AREA) {
            origin_tried = ORIGIN_WAS_BUILT_IN_EXTENSIONS_AREA;
            pathname = pathname_of_built_in_extensions;
            goto Retry;
        }
        LOG("Extension failed: '%s'\n", sf->filename);
        sentence_problem(_P_(C3BogusExtension),
            "I can't find that extension",
            "which seems not to be installed. (You can get hold of extensions "
            "which people have made public at the Inform website.)");
        return -1;
    } else {
        fatal_error2("Error: can't open source text file", sf->filename);
    }
}
```

This code is used in §2.

§**4.** This is where messages like

```
I've also read Standard Rules by Graham Nelson, which is 27204 words long.
```

are printed to `stdout` (not `stderr`), in something of an affectionate nod to TEX's traditional console output, though occasionally I think silence is golden and that the messages could go. It's a moot point for almost all users, though, because the console output is concealed from them by the Inform application.

⟨Tell console output about the file 4⟩ ≡

```
int wc;
char *message;
if (EF == NULL) message = "I've now read %s, which is %d words long.\n";
else message = "I've also read %s, which is %d words long.\n";
wc = sf_total_word_count(sf);
STREAM_WRITE(STDOUT, message, synopsis, wc);
STREAM_FLUSH(STDOUT);
LOG(message, synopsis, wc);
```

This code is used in §2.

§**5. Feeding whole files into the lexer.** This is one of the two feeder routines for the lexer, the other being in Lexical Writing Back.w: see Lexer.w for its obligations.

We feed characters from an open file into the lexer, and continue until there is nothing left in it. Inform is used on operating systems which between them use all four of the sequences `0a`, `0d`, `0a0d` and `0d0a` to divide lines in text files, so each of these is converted to a single `'\n'`. Tabs are treated as if spaces in most contexts, but not when parsing formatted tables, for instance, so they are not similarly converted.

```
void feed_file_into_lexer(source_file *sf, char *leafname, int documentation_only) {
    source_location top_of_file;
    int words_fed_w1, words_fed_w2;
    int cr, last_cr, next_cr, read_cr, newline_char = 0;
    top_of_file.file_of_origin = sf;
    top_of_file.line_number = 1;
    lexer_feed_begins(top_of_file);
    if (documentation_only) lexer_wait_for_dashes = TRUE;
    last_cr = ' '; cr = ' '; next_cr = utf8_fgetc(sf->handle, TRUE);
    if (next_cr == 0xFEFF) next_cr = utf8_fgetc(sf->handle, TRUE);                Unicode BOM code
    if (next_cr != EOF)
        while (((read_cr = utf8_fgetc(sf->handle, TRUE)), next_cr) != EOF) {
            last_cr = cr; cr = next_cr; next_cr = read_cr;
            switch(cr) {
                case '\x0a':
                    if (newline_char == '\x0d') {
                        newline_char = 0; continue;          suppress 0x000A when it follows 0x000D
                    }
                    newline_char = cr; cr = '\n';                    and otherwise convert to '\n'
                    break;
                case '\x0d':
                    if (newline_char == '\x0a') {
                        newline_char = 0; continue;          suppress 0x000D when it follows 0x000A
                    }
                    newline_char = cr; cr = '\n';                    and otherwise convert to '\n'
                    break;
                default:
                    newline_char = 0;
```

```
                    break;
                }
                lexer_feed_triplet(last_cr, cr, next_cr);
            }
        if (primary_source_file == sf) leafname = "main source text";
        lexer_feed_ends(&words_fed_w1, &words_fed_w2, TRUE, leafname);
        ⟨Word count the new material 6⟩;
}
```

§**6.** We word count all source files, both as to their source text and their quoted text (i.e., their text within double-quotes).

⟨Word count the new material 6⟩ ≡

```
    int wc;
    for (wc=words_fed_w1; wc<=words_fed_w2; wc++) {
        char *p = lw_array[wc].lw_text;
        if (*p == '"') {
            inside quoted text, each run of non-whitespace counts as 1 word
            p++;                                                    skip opening quotation mark
            while (*p != 0) {
                while ((*p == ' ') || (*p == NEWLINE_IN_STRING)) p++;        move past white space
                if ((*p == '"') || (*p == 0)) break;                stop if this reaches the end
                sf->words_of_quoted_text++;                          otherwise we have a word
                while ((*p != ' ') && (*p != NEWLINE_IN_STRING)
                    && (*p != '"') && (*p != 0)) p++;                move to white space or end
            }
        } else {
            outside quoted text, each lexer word not wholly composed of punctuation scores 1
            for (; *p != 0; p++)
                if ((is_punctuation(*p) == FALSE) && (*p != '|')) {
                    sf->words_of_source++;
                    break;
                }
        }
    }
```

This code is used in §5.

§**7.** At present, though, the only use made of these two word counts is via the following routine, which combines them into one.

```
int sf_total_word_count(source_file *sf) {
    return sf->words_of_source + sf->words_of_quoted_text;
}
```

The function sf_total_word_count is called from 2/prob2.

§8. Finally, we translate between the tiresomely many representations of files we seem to be stuck with. The method used by `filename_to_source_file` looks vulnerable to case-insensitive filename issues, but isn't, because each filename is present in NI in only one form.

```
char *sf_get_filename(source_file *sf) {
    if (sf == NULL) internal_error("tried to read filename of null source file");
    return sf->filename;
}
extension_file *sf_get_extension_corresponding(source_file *sf) {
    if (sf == NULL) return NULL;
    return sf->extension_provided;
}
source_file *filename_to_source_file(char *filename) {
    source_file *sf;
    LOOP_OVER(sf, source_file) {
        if (strcmp(sf->filename, filename) == 0) return sf;
        if (strcmp(sf->leafname, filename) == 0) return sf;
    }
    return NULL;
}
```

The function sf_get_filename is called from 2/html, 2/prob1 and 4/head.
The function sf_get_extension_corresponding is called from 2/index, 2/prob1, 4/head, 8/sob, 10/isin and 12/rb.
The function filename_to_source_file is called from 2/html.

§9. **Reading UTF-8 files.** The following routine reads a sequence of Unicode characters from a UTF-8 encoded file, but returns them as a sequence of ISO Latin-1 characters, a trick it can only pull off by escaping non-ISO characters. This is done by taking character number N and feeding it out, one character at a time, as the text "[unicode N]", writing the number in decimal. Only one UTF-8 file like this will be being read at a time, and the routine will be repeatedly called until `EOF` or a line division.

Strictly speaking, we transmit not as ISO Latin-1 but as that subset of ISO which have corresponding (different) codes in the ZSCII character set. This excludes some typewriter symbols and a handful of letterforms, as we shall see.

There are two exceptions: `utf8_fgetc` can also return the usual C end-of-file pseudo-character `EOF`, and it can also return the Unicode BOM (byte-ordering marker) pseudo-character, which is legal at the start of a file and which is automatically prepended by some text editors and word-processors when they save a UTF-8 file (though in fact it is not required by the UTF-8 specification). Anyone calling `utf8_fgetc` must check the return value for `EOF` every time, and for `0xFEFF` every time we might be at the start of the file being read.

```
char unicode_feed_buffer[32];                    holds a single escape such as "[unicode 3106]"
int ufb_counter = -1;                                  position in the unicode feed buffer
int utf8_fgetc(FILE *from, int escape_oddities) {
    int c, conts;
    if (ufb_counter >= 0) {
        if (unicode_feed_buffer[ufb_counter] == 0) ufb_counter = -1;
        else return unicode_feed_buffer[ufb_counter++];
    }
    c = fgetc(from);
    if (c == EOF) return c;                      ruling out EOF leaves a genuine byte from the file
    if (c<0x80) return c;                    in all other cases, a UTF-8 continuation sequence begins
    ⟨Unpack one to five continuation bytes to obtain the Unicode character code 10⟩;
    ⟨Return non-ASCII codes in the intersection of ISO Latin-1 and ZSCII as literals 11⟩;
```

```
    ⟨Return Unicode fancy equivalents as simpler literals 12⟩;
    if (c == 0xFEFF) return c;                                    the Unicode BOM non-character
    if (escape_oddities == FALSE) return c;
    sprintf(unicode_feed_buffer, "[unicode %d]", c);
    ufb_counter = 1;
    return '[';
}
```

The function utf8_fgetc is called from 1/plat and 4/excen.

§**10.** Not every byte sequence is legal in a UTF-8 file: if we find a malformed continuation, we process it as a question mark rather than throwing a fatal error (which is pretty well the only alternative here). The user is likely to see problem messages later on which arise from the question marks, and that will have to do.

⟨Unpack one to five continuation bytes to obtain the Unicode character code 10⟩ ≡

```
    if (c<0xC0) return '?';                                                    malformed UTF-8
    if (c<0xE0) { c = c & 0x1f; conts = 1; }
    else if (c<0xF0) { c = c & 0xf; conts = 2; }
    else if (c<0xF8) { c = c & 0x7; conts = 3; }
    else if (c<0xFC) { c = c & 0x3; conts = 4; }
    else { c = c & 0x1; conts = 5; }
    while (conts > 0) {
        int d = fgetc(from);
        if (d == EOF) return '?';                                              malformed UTF-8
        c = c << 6;
        c = c + (d & 0x3F);
        conts--;
    }
```

This code is used in §9.

§**11.** For the ZSCII character set, see *The Inform 6 Designer's Manual*, or *The Z-Machine Standards Document*. It offers a range of west European accented letters which almost, but not quite, matches those on offer in ISO Latin-1. This omits certain obscure glyphs: among them, the Icelandic lower case eth, which is good because (as we saw in Chapter 2) this character value is used for hackery with error messages, so it is good to know that it cannot occur from the source text.

⟨Return non-ASCII codes in the intersection of ISO Latin-1 and ZSCII as literals 11⟩ ≡

```
    if ((c == 0xa1) || (c == 0xa3) || (c == 0xbf)) return c;                pound sign, inverted ! and ?
    if ((c >= 0xc0) && (c <= 0xff)) {                                  accented West European letters, but...
        if ((c != 0xd0) && (c != 0xf0) &&                                          not Icelandic eths
            (c != 0xde) && (c != 0xfe) &&                                          nor Icelandic thorns
            (c != 0xf7) && (c != 0xd7))                                nor multiplication or division signs
            return c;
    }
```

This code is used in §9.

§**12.** NI errs on the safe side, accepting em-rules and non-breaking spaces, etc., where it would normally expect hyphens and ordinary spaces: this is intended for the benefit of users with helpful word-processors which autocorrect hyphens into em-rules when they are flanked by spaces, and so on.

⟨Return Unicode fancy equivalents as simpler literals 12⟩ ≡

```
if (c == 0x85) return '\x0d';                                  NEL, or "next line"
if (c == 0xa0) return ' ';                                     non-breaking space
if ((c >= 0x2000) && (c <= 0x200a)) return ' ';                space variants
if ((c >= 0x2010) && (c <= 0x2014)) return '-';                rules and dashes
if ((c >= 0x2018) && (c <= 0x2019)) return '\'';               smart single quotes
if ((c >= 0x201c) && (c <= 0x201d)) return '"';                smart double quotes
if ((c >= 0x2028) && (c <= 0x2029)) return '\x0d';             fancy newlines
```

This code is used in §9.

Lexical Writing Back 3/lwb

Purpose

Feeding modified text back into the lexer, to ensure that inflected versions of phrases are also within the word stream.

3/lwb.§2-3 Splicing; §4-6 Stocking the plurals dictionary; §7-8 Searching the plural dictionary; §9 Constructing past participles; §10-11 Constructing comparatives and superlatives

---

Template interpreter commands

```
4   {-callv:traverse_for_plural_definitions}
```

Definitions

¶1. A modest dictionary of plurals is maintained, to allow the user to record better plurals than the ones we would make ourselves. This assumes that a plural can be constructed without knowledge of context, but that works in almost all cases. (Arguably "dwarf" should pluralise to "dwarfs" when discussing stars and to "dwarves" when reading Tolkien, but few works of IF will contain both at once.)

```
typedef struct plural_dictionary_entry {
    int singular_w1, singular_w2;                    words of singular form
    int plural_w1, plural_w2;                        words of plural form
    MEMORY_MANAGEMENT
} plural_dictionary_entry;
```

The structure plural_dictionary_entry is private to this section.

---

§**1.** I feel a certain embarrassment about the frequency with which the following routine is needed throughout NI, to create fresh source text from old. Is this a quick and dirty hack? Or a deep insight into the interdependency of lexical analysis and semantic understanding? The reader must decide.

Anyway, this is one of the two feeder routines for the lexer, the other being in Read Source Text.w: see Lexer.w for its obligations.

Our aim is to copy a string of text into the lexer, as if it had been read from a source file, to generate new words `(lexer_feed_w1, lexer_feed_w2)`.

When done, we call `vocab_identify_word_range`, because we are probably running long after the initial vocabulary identification phase of NI.

```
int lexer_feed_w1, lexer_feed_w2;                    word range made in last call to feed_into_lexer
void feed_into_lexer(char *text, int expand_strings, int grammar_mode) {
    int i;
    source_location as_if_from_nowhere;
    as_if_from_nowhere.file_of_origin = NULL;
    as_if_from_nowhere.line_number = 1;
    lexer_feed_begins(as_if_from_nowhere);
    lexer_divide_strings_at_text_substitutions = expand_strings;
    lexer_treat_slash_as_punctuation = grammar_mode;
    for (i=0; text[i] != 0; i++) {
        int last_cr, cr, next_cr;
```

```
        if (i > 0) last_cr = text[i-1]; else last_cr = EOF;
        cr = text[i];
        if (cr != 0) next_cr = text[i+1]; else next_cr = EOF;
        lexer_feed_triplet(last_cr, cr, next_cr);
    }
    lexer_feed_ends(&lexer_feed_w1, &lexer_feed_w2, FALSE, "<internal>");
    vocab_identify_word_range(lexer_feed_w1, lexer_feed_w2);
}
```

The function feed_into_lexer is called from 2/dl, 3/read, 4/excen, 4/edoc, 4/rtree, 5/litp, 5/candp, 7/dti, 8/creat, 9/scene, 9/mapbp, 9/prop, 9/madj, 9/vpbp, 10/str, 10/eqns, 10/bib, 11/act, 11/av, 12/phtd, 12/phsf, 12/cinv, 13/tfg, 13/gtok and 14/i6t.

§**2. Splicing.** A different form of lexer abuse is called for to overcome the fact that, once in a while, we need to have a run of words in the lexer which all do occur in the source text, but not contiguously, so that they cannot be represented by a pair `(w1, w2)`. In that event we use the following routine to splice duplicate references at the end of the word list (this does not duplicate the text itself, only references to it): for instance, if we start with 10 words (0 to 9) and then splice `(2,3)` and then `(6,8)`, we end up with 15 words, and the text of `(10,14)` contains the same material as words 2, 3, 6, 7, 8.

```
int splice_words(int w1, int w2) {
    int i;
    ensure_lexer_space_up_to(lexer_wordcount + w2 - w1 + 1);
    for (i=0; i<=w2-w1; i++) {
        lw_array[lexer_wordcount+i].lw_text = lw_array[w1+i].lw_text;
        lw_array[lexer_wordcount+i].lw_rawtext = lw_array[w1+i].lw_rawtext;
        lw_array[lexer_wordcount+i].lw_source = lw_array[w1+i].lw_source;
        lw_array[lexer_wordcount+i].lw_identity = lw_array[w1+i].lw_identity;
    }
    i = lexer_wordcount;
    lexer_wordcount += w2-w1+1;
    return i;
}
```

The function splice_words is called from 4/rtree, 5/litp, 9/scene, 9/prop, 10/str, 11/act and 11/av.

§**3.** The following is almost identical, but effectively forces words into lower case by splicing only the case-normalised originals (thus not preserving the original untouched style).

```
int lower_case_splice_words(int w1, int w2) {
    int i;
    ensure_lexer_space_up_to(lexer_wordcount + w2 - w1 + 1);
    for (i=0; i<=w2-w1; i++) {
        lw_array[lexer_wordcount+i].lw_text = lw_array[w1+i].lw_text;
        lw_array[lexer_wordcount+i].lw_rawtext = lw_array[w1+i].lw_text;
        lw_array[lexer_wordcount+i].lw_source = lw_array[w1+i].lw_source;
    }
    i = lexer_wordcount;
    lexer_wordcount += w2-w1+1;
    return i;
}
```

§**4. Stocking the plurals dictionary.** The user gives us plurals with special sentences, but we don't handle these during the two main source text traverses because that would be too late for some of the plurals we need. (Inform used to work this way, but with the consequent restriction that plurals had to be defined earlier in the source than anything needing them: this was much misunderstood by users, and generally tiresome, especially for extensions where the ordering was not obvious.)

```
sentence_handler PLURAL_SH_handler =
    { SENTENCE_NT, PLURAL_VB, 1, handle_plural_definition };
void handle_plural_definition(parse_node *p) {
    do nothing: these sentences have already been caught by the traverse below
}
void traverse_for_plural_definitions(void) {
    parse_node *p, *prevp;
    for (prevp=NULL, TREE_START(p); p; prevp=p, TREE_NEXT(p)) {
        if ((pn_get_node_type(p) == SENTENCE_NT)
            && (p->down)
            && (pn_int_annotation(p->down, verb_id_ANNOT) == PLURAL_VB)
            && (p->down->next) && (p->down->next->next))
            register_plural_form(p->down->next->word_ref1,
                p->down->next->word_ref2,
                p->down->next->next->word_ref1,
                p->down->next->next->word_ref2);
    }
}
```

The function traverse_for_plural_definitions is invoked by a command in a `.i6t` template file.

§**5.** Note that we are entirely allowed to register a new plural for a phrase which already has a plural in the dictionary (see below), which is why we do not trouble to search the existing dictionary here.

```
void register_plural_form(int sing_w1, int sing_w2, int pl_w1, int pl_w2) {
    plural_dictionary_entry *pde = CREATE(plural_dictionary_entry);
    [[sing_w1, sing_w2 == the plural of ... --> sing_w1, sing_w2]];
    pde->singular_w1 = sing_w1;
    pde->singular_w2 = sing_w2;
    pde->plural_w1 = pl_w1;
    pde->plural_w2 = pl_w2;
    ⟨Forbid plural declarations containing quoted text 6⟩;
    LOGIF(PLURALS, "[Registering plural of $W as $W]\n", sing_w1, sing_w2, pl_w1, pl_w2);
}
```

§**6.** In general names of things which we need plurals for cannot contain quoted text anyway, so the following problem messages are not too gratuitous.

⟨Forbid plural declarations containing quoted text 6⟩ ≡

```
int i;
for (i=sing_w1; i<=sing_w2; i++)
    if (vocab_test_flags(i, TEXT_MC)) {
        sentence_problem(_P_(C3PluralOfQuoted),
            "declares a plural for a phrase containing quoted text",
            "which is forbidden. Sentences like this are supposed to "
            "declare plurals without quotation marks: for instance, "
            "'The plural of attorney general is attorneys general.'");
        return;
    }
for (i=pl_w1; i<=pl_w2; i++)
    if (vocab_test_flags(i, TEXT_MC)) {
        sentence_problem(_P_(C3PluralIsQuoted),
            "declares a plural for a phrase using quoted text",
            "which is forbidden. Sentences like this are supposed to "
            "declare plurals without quotation marks: for instance, "
            "'The plural of procurator fiscal is procurators fiscal.'");
        return;
    }
```

This code is used in §5.

§**7. Searching the plural dictionary.** The following routine can either be called once only – in which case it yields up the best known plural for the phrase – or iteratively, in which case it serves up all known plurals of the given phrase, starting with the best (the earliest defined in the text, if any plural for this phrase has been so defined) and finishing up with the worst (a mechanically-made one not found in the dictionary).

```
int plw1, plw2;
plural_dictionary_entry *make_plural_of(int w1, int w2,
    plural_dictionary_entry *search_from) {
    int i;
    char pluralised[MAX_WORD_LENGTH+3];
    plural_dictionary_entry *pde;
    if ((w1 == -1) || (w2 < w1)) { plw1 = -1; plw2 = -1; return NULL; }
    if (search_from == NULL) search_from = FIRST_OBJECT(plural_dictionary_entry);
    else search_from = NEXT_OBJECT(search_from, plural_dictionary_entry);
    for (pde = search_from; pde; pde = NEXT_OBJECT(pde, plural_dictionary_entry)) {
        if (compare_word_range(w1, w2, pde->singular_w1, pde->singular_w2)) {
            plw1 = pde->plural_w1;
            plw2 = pde->plural_w2;
            return pde;
        }
    }
    ⟨Make a new plural by lexical writing back 8⟩;
    return NULL;
}
```

The function make_plural_of is called from 7/tids, 7/kov and 9/wo.

§**8.** When the dictionary fails us, we use lexical rewriting to construct plurals of phrases found only in the singular in the source. For instance, if the designer says that "A wicker basket is a kind of container" then NI will need to recognise not only "wicker basket" but also "wicker baskets", a pair of words not found in the source text anywhere. So the following routine takes the text `(w1, w2)` and feeds a suitable plural into the lexer, emerging with the text `(plw1, plw2)`.

The pluralising algorithm looks only at the ending of the final word, turning "-s" into "-ses" ("wine glass" to "wine glasses"), "-sh" into "-shes" ("wish" to "wishes"), "-y" into "-ies" ("nappy" to "nappies"; less happily, "time of day" to "time of daies") and otherwise tacking an "-s" on the end ("sheep" to "sheeps". Sigh...).

We do not write the new plural into the dictionary: there is no need, as it can be rebuilt quickly whenever needed again.

⟨Make a new plural by lexical writing back 8⟩ ≡

```
plw1 = lexer_wordcount;
if (w1<w2) splice_words(w1, w2-1);
if (*(lw_array[w2].lw_text) == '\"') strcpy(pluralised, "some-long-text");
else strcpy(pluralised, lw_array[w2].lw_text);
i = strlen(pluralised)-1;
switch (pluralised[i]) {
    case 's': case 'x': strcat(pluralised, "es"); break;
    case 'h':
        if ((i>0) && (pluralised[i-1]=='s')) strcat(pluralised, "es");
        else strcat(pluralised, "s"); break;
    case 'y':
        if ((i>0) && (pluralised[i-1]=='a')) { strcat(pluralised, "s"); break; }
        if ((i>0) && (pluralised[i-1]=='e')) { strcat(pluralised, "s"); break; }
        if ((i>0) && (pluralised[i-1]=='i')) { strcat(pluralised, "s"); break; }
        if ((i>0) && (pluralised[i-1]=='o')) { strcat(pluralised, "s"); break; }
        if ((i>0) && (pluralised[i-1]=='u')) { strcat(pluralised, "s"); break; }
        pluralised[i] = 0; strcat(pluralised, "ies"); break;
    default: strcat(pluralised, "s"); break;
}
feed_into_lexer(pluralised, FALSE, FALSE);
plw2 = lexer_wordcount-1;
LOGIF(PLURALS, "[Constructing plural of $W as $W]\n", w1, w2, plw1, plw2);
```

This code is used in §7.

§**9. Constructing past participles.** A similar construction works out the past participle of actions, such as "turning" to "turned", though here we don't need any dictionary because action definitions already have the option to specify their own irregularities.

English is replete with exceptions ("catching" must become "caught", for instance), and there are no sensible rules, so exceptions are mostly left to the Standard Rules or the user. ("Doing" is turned automatically into "done" because it makes life easier when we create the patterns for "doing something", and "going" to "gone" by analogy.)

```
void make_past_of_participle(int w1, int w2) {
    char pasturised[MAX_WORD_LENGTH+3];
    int i;
    plw1 = lexer_wordcount;
    i = w1;
    while (i <= w2) {
        if (*(lw_array[i].lw_text) == '\"') strcpy(pasturised, "some-long-text");
        else strcpy(pasturised, lw_array[i].lw_text);
```

```
        if ((strlen(pasturised) >= 5) &&
            (pasturised[strlen(pasturised)-3] == 'i') &&
            (pasturised[strlen(pasturised)-2] == 'n') &&
            (pasturised[strlen(pasturised)-1] == 'g')) {
            if (i > w1) splice_words(w1, i-1);
            pasturised[strlen(pasturised)-3] = 0;
            if (strcmp(pasturised, "do") == 0) strcpy(pasturised, "done");
            else if (strcmp(pasturised, "go") == 0) strcpy(pasturised, "gone");
            else strcat(pasturised, "ed");
            feed_into_lexer(pasturised, FALSE, FALSE);
            if (i < w2) splice_words(i+1, w2);
            break;
        }
        i++;
    }
    plw2 = lexer_wordcount-1;
    LOGIF(PAST_PARTICIPLES,
        "[Past participle of $W is $W]\n", w1, w2, plw1, plw2);
}
void set_past_participle(int *past_w1, int *past_w2, int irregular_pp) {
    plw1 = lexer_wordcount;
    splice_words(irregular_pp, irregular_pp);
    if (*past_w1 < *past_w2) splice_words(*past_w1 + 1, *past_w2);
    *past_w1 = plw1;
    *past_w2 = lexer_wordcount-1;
}
```

The function make_past_of_participle is called from 11/act.
The function set_past_participle is called from 11/act.

§**10. Constructing comparatives and superlatives.** And these are constructed very similarly. Thus "happy" to "happier" and "happiest"; "rich" to "richer" and "richest".

```
int make_comparative(int w1) {
    char comprised[MAX_WORD_LENGTH+3];
    plw1 = lexer_wordcount;
    if (*(lw_array[w1].lw_text) == '\"') strcpy(comprised, "some-long-text");
    else strcpy(comprised, lw_array[w1].lw_text);
    if (comprised[strlen(comprised)-1] == 'y')
        comprised[strlen(comprised)-1] = 'i';
    if (comprised[strlen(comprised)-1] != 'e')
        strcat(comprised, "e");
    strcat(comprised, "r");
    feed_into_lexer(comprised, FALSE, FALSE);
    strcpy(comprised, " than ");
    feed_into_lexer(comprised, FALSE, FALSE);
    plw2 = lexer_wordcount-1;
    LOG("[Comparative of $W is $W]\n", w1, w1, plw1, plw2);
    return plw1;
}
int make_superlative(int w1) {
    char comprised[MAX_WORD_LENGTH+3];
    if (*(lw_array[w1].lw_text) == '\"') strcpy(comprised, "some-long-text");
```

```
    else strcpy(comprised, lw_array[w1].lw_text);
    if (comprised[strlen(comprised)-1] == 'y')
        comprised[strlen(comprised)-1] = 'i';
    if (comprised[strlen(comprised)-1] != 'e')
        strcat(comprised, "e");
    strcat(comprised, "st");
    feed_into_lexer(comprised, FALSE, FALSE);
    LOG("[Superlative of $W is $W]\n", w1, w1, lexer_wordcount-1, lexer_wordcount-1);
    return lexer_wordcount-1;
}
```

The function make_comparative is called from 9/cmpbp.
The function make_superlative is called from 9/madj.

§**11.** There has to be a better term than "quiddity" for this grammatical construct, but what I mean is the property for which the given adjective makes a comparison: for instance, "tallness" for "tall", or "steeliness" for "steely".

```
int make_quiddity(int w1) {
    char comprised[MAX_WORD_LENGTH+3];
    if (*(lw_array[w1].lw_text) == '\"') strcpy(comprised, "some-long-text");
    else strcpy(comprised, lw_array[w1].lw_text);
    if (comprised[strlen(comprised)-1] == 'y')
        comprised[strlen(comprised)-1] = 'i';
    strcat(comprised, "ness");
    feed_into_lexer(comprised, FALSE, FALSE);
    LOG("[Quiddity of $W is $W]\n", w1, w1, lexer_wordcount-1, lexer_wordcount-1);
    return lexer_wordcount-1;
}
```

The function make_quiddity is called from 9/cmpbp.

Lexical Services 3/lexs

Purpose

A small suite of routines to examine the words stored by the lexer, which are provided as a utility to higher levels of NI. None are used by the lexer itself.

3/lexs.§3-4 Casing and sentence division; §5 Dequoting literal text; §6 Correct use of text substitutions; §7 Ordinals; §8 Digital times; §9 Title casing for extension names; §10-12 Comparing word ranges; §13-17 Searching word ranges; §18 Searching for unusual spacing in ranges; §19 Dividing lists; §20 Parenthesis matching; §21-22 A detour for encoding text; §23-24 Printing word ranges; §25-26 Printing raw word ranges; §27-29 Printing word ranges to the debugging log

---

Definitions

¶1. The following definitions are for use with a utility routine for looking through lists to pick off the first entry:

```
define LOOK_FOR_AND 1
define LOOK_FOR_OR 2

int left_w1, left_w2;                    words making up first entry in list
int right_w1, right_w2;                  words making up tail of list
```

---

§**1.** Comparison of the word at a given numbered position against some known word, say "rulebook", must be done very quickly. The whole point of the vocabulary bank identifying each distinct word was to enable this to be done by a single comparison of pointers: and to avoid the overhead of a function call, we perform this with macros.

Recall that words are numbered from 0 to `lexer_wordcount - 1` inclusive, and that the word range `(w1, w2)` means all words from `w1` to `w2` inclusive: it would be an error to use the routines below in any case where `w1` or `w2` are out of range, or where it is not true that `w1 <= w2`. For efficiency's sake these error conditions are not checked.

```
define compare_word(w, voc) (lw_array[w].lw_identity == (voc))
define compare_words(w1, w2) (lw_array[w1].lw_identity == lw_array[w2].lw_identity)
```

§**2.** We can also, more slowly, perform a direct string comparison. If carried out on the original, raw, text, this will be case sensitive – which is usually wrong for Inform purposes. On the `lw_text`, however, we are comparing a case-normalised version of the original word, which is likely to be safely case insensitive comparison, provided that the content of `t` is also normalised.

```
int compare_word_by_strcmp(int w, char *t) {
    return (strcmp(lw_array[w].lw_text, t) == 0);
}
int compare_raw_word_by_strcmp(int w, char *t) {
    return (strcmp(lw_array[w].lw_rawtext, t) == 0);
}
```

The function compare_word_by_strcmp is called from 2/index, 4/vm and 5/candp.

The function compare_raw_word_by_strcmp is called from 2/index and 10/isin.

§**3. Casing and sentence division.** Casing is only sometimes informative in English: for the first word in a sentence, we expect to find an upper-case letter, so that there is no easy way to tell the name of a person or institution from a common noun. But in other cases an upper case initial letter is unexpected, and can tell us something.

```
int unexpectedly_upper_case(int wn) {
    if (wn<1) return FALSE;
    if (compare_word(wn-1, FULLSTOP_V)) return FALSE;
    if (compare_word(wn-1, STROKE_V)) return FALSE;
    if (isupper(*(lw_array[wn].lw_rawtext))) {
        if (text_ending_sentence(wn-1)) return FALSE;
        return TRUE;
    }
    return FALSE;
}
```

The function unexpectedly_upper_case is called from 2/prob3, 4/verb, 4/noun, 4/ofs, 5/det, 5/conj, 8/refpt, 11/ap and 13/test.

§**4.** Does the word at `wn` appear to be a piece of quoted text which, because it ends with punctuation, may also end the sentence which quotes it?

```
int text_ending_sentence(int wn) {
    char *p = lw_array[wn].lw_rawtext;
    if (p[0] != '"') return FALSE;
    p += strlen(p) - 2;
    if ((p[0] == '.') && (p[1] == '"')) return TRUE;
    if ((p[0] == '?') && (p[1] == '"')) return TRUE;
    if ((p[0] == '!') && (p[1] == '"')) return TRUE;
    p--;
    if ((p[0] == '.') && (p[1] == ')') && (p[2] == '"')) return TRUE;
    if ((p[0] == '?') && (p[1] == ')') && (p[2] == '"')) return TRUE;
    if ((p[0] == '!') && (p[1] == ')') && (p[2] == '"')) return TRUE;
    if ((p[0] == '.') && (p[1] == '\'') && (p[2] == '"')) return TRUE;
    if ((p[0] == '?') && (p[1] == '\'') && (p[2] == '"')) return TRUE;
    if ((p[0] == '!') && (p[1] == '\'') && (p[2] == '"')) return TRUE;
    return FALSE;
}
```

The function text_ending_sentence is called from 4/sent and 12/cinv.

§**5. Dequoting literal text.** A utility for stripping double-quotes from literal text, along with initial or trailing spaces inside those quotes.

```
void dequote_word(int wn) {
    char *previous_text = lw_array[wn].lw_text;
    char *dequoted_text;
    if (previous_text[0] != '"') return;
    lw_array[wn].lw_rawtext = copy_to_lexer_memory(lw_array[wn].lw_rawtext);
    dequoted_text = previous_text + 1;
    while (*(dequoted_text) == ' ') dequoted_text++;
    if ((strlen(dequoted_text) > 0) &&
        (*(dequoted_text+strlen(dequoted_text)-1) == '"'))
        *(dequoted_text+strlen(dequoted_text)-1) = 0;
    while ((strlen(dequoted_text) > 0) &&
        (*(dequoted_text+strlen(dequoted_text)-1) == ' '))
        *(dequoted_text+strlen(dequoted_text)-1) = 0;
    lw_array[wn].lw_text = dequoted_text;
    LOGIF(VOCABULARY, "Dequoting word %d <%s> to <%s>\n",
        wn, previous_text, dequoted_text);
    vocab_identify_word(wn);
    vocab_set_raw_exemplar_to_text(wn);
}
```

The function dequote_word is called from 2/isn, 2/index, 8/sob, 8/mass, 9/map, 10/bib, 10/fig, 10/sfx, 10/isin, 11/act, 12/def, 13/tfg, 13/gtok and 13/test.

§**6. Correct use of text substitutions.** If a "word" is going to be quoted literal text, then it has to use the characters `[` and `]` in a matched way, and without nesting them. The following verifies that.

These rules are quite strict. It could be argued that nested brackets should be allowed, allowing comments in text substitutions, but the result would be hard to read and tricky for the user interface applications to syntax-colour.

```
int well_formed_text_routine(char *fw) {
    int i, escaped = NOT_APPLICABLE;
    for (i=0; fw[i] != 0; i++) {
        if (fw[i] == TEXT_SUBSTITUTION_BEGIN) {
            if (escaped == TRUE) return FALSE;
            escaped = TRUE;
        }
        if (fw[i] == TEXT_SUBSTITUTION_END) {
            if (escaped != TRUE) return FALSE;
            escaped = FALSE;
        }
    }
    if (escaped == NOT_APPLICABLE) return escaped;
    if (escaped) return FALSE;
    return TRUE;
}
```

The function well_formed_text_routine is called from 3/vocab and 13/tfg.

§**7. Ordinals.** The following parses the string to see if it is a non-negative integer, written as an English ordinal: 0th, 1st, 2nd, 3rd, 4th, 5th, ... Note that we don't bother to police the finicky rules on which suffix should accompany which value (22nd not 22th, and so on).

```
int an_ordinal_number(char *fw) {
    int i;
    for (i=0; fw[i] != 0; i++)
        if (!(isdigit(fw[i]))) {
            if ((i>0) &&
                (((fw[i] == 's') && (fw[i+1] == 't') && (fw[i+2] == 0)) ||
                ((fw[i] == 'n') && (fw[i+1] == 'd') && (fw[i+2] == 0)) ||
                ((fw[i] == 'r') && (fw[i+1] == 'd') && (fw[i+2] == 0)) ||
                ((fw[i] == 't') && (fw[i+1] == 'h') && (fw[i+2] == 0))))
                return atoi(fw);
            break;
        }
    return -1;
}
```

The function an_ordinal_number is called from 3/vocab.

§**8. Digital times.** Times, in other words, written in the style of a digital clock: "00:00", "5:21", "17:21". The syntax must be one or two digits, followed by a colon, followed by exactly two digits; it is permissible for the first of two digits to be zero; when that is discarded, the hours part must be in the range 0 to 23, and the minutes part in the range 0 to 59.

```
int is_a_digital_clock_time(int w1, int *time_minutes, int *time_hours) {
    int ratchet = 0, t, colons = 0, digits = 0;
    char *wd = lw_array[w1].lw_text;
    *time_hours = 0; *time_minutes = 0;
    for (t=0; wd[t]; t++) {
        if (((t==1) || (t==2)) && (wd[t] == ':') && (wd[t+1])) {
            if (ratchet >= 24) return FALSE;
            *time_hours = ratchet;
            ratchet = 0; digits = 0;
            colons++;
        } else if (isdigit(wd[t])) {
            ratchet = 10*ratchet + (wd[t]-'0'); digits++;
            if ((ratchet >= 60) || (digits > 2)) return FALSE;
        } else return FALSE;
    }
    if (colons != 1) return FALSE;
    *time_minutes = ratchet;
    return TRUE;
}
```

The function is_a_digital_clock_time is called from 5/lit.

§**9. Title casing for extension names.** Until the spring of 2008, extension titles and author names were case normalised, and the following routine was used to perform the necessary surgery on the word ranges used for these. That changed in the 5Sxx series of builds, and the routine is now no longer used. It is something of a trick – for once we break our promises and touch the raw lexical source text: but not the touched text, because that would wreck the vocabulary hash table.

```
void rewrite_raw_text_in_title_casing(int w1, int w2) {
    int w;
    for (w=w1; w<=w2; w++) {
        char *p = lw_array[w].lw_rawtext;
        if (*p == 0) continue;
        *p = toupper(*p);
        p++;
        while (*p != 0) {
            *p = tolower(*p);
            p++;
        }
    }
}
```

§**10. Comparing word ranges.** The calculation overhead makes it marginally not worth using a hash function to speed up the following comparison, which requires two excerpts to be absolutely equal.

```
int compare_word_range(int w1, int w2, int w3, int w4) {
    int j;
    if (w4-w3 != w2-w1) return FALSE;
    if ((w1<0) || (w3<0)) return FALSE;
    for (j=0; j<=w2-w1; j++)
        if (compare_words(w1+j, w3+j) == FALSE) return FALSE;
    return TRUE;
}
```

The function compare_word_range is called from 3/lwb, 4/iext, 4/head, 4/ofs, 5/conj, 5/litp, 8/creat, 9/qty, 9/scene, 9/vpbp, 10/tab, 11/act, 12/phud, 12/phod, 12/phsf, 12/stv, 12/phin, 12/rb and 13/gv.

§**11.** This alternative form does not use the identifications of vocabulary and resorts to `strcmp`, which has to look at every character of the raw text: so although it looks similar, it's much, much slower. Happily, it is now no longer used, and retained only in case of future need.

```
int compare_raw_word_range(int w1, int w2, int w3, int w4) {
    int j;
    if (w4-w3 != w2-w1) return FALSE;
    if ((w1<0) || (w3<0)) return FALSE;
    for (j=0; j<=w2-w1; j++)
        if (strcmp(lw_array[w1+j].lw_rawtext,
            lw_array[w3+j].lw_rawtext) != 0) return FALSE;
    return TRUE;
}
```

§**12.** And relatedly, used for sorting into alphabetical order, a direct analogue of `strcmp` but for word ranges:

```
int rangecmp(int x1, int x2, int y1, int y2) {
    if (x1 < 0) { if (y1 < 0) return 0; return -1; }
    if (y1 < 0) return 1;
    int n;
    int l1 = x2 - x1 + 1;
    int l2 = y2 - y1 + 1;
    for (n=0; (n<l1) && (n<l2); n++) {
        int delta = strcmp(lw_array[x1 + n].lw_text, lw_array[y1 + n].lw_text);
        if (delta != 0) return delta;
    }
    return l1 - l2;
}
```

The function rangecmp is called from 9/tmap and 11/ina.

§**13. Searching word ranges.** Look to see if a given text occurs as a word with number $w_1 < n < w_2$. In the second test, the word is not allowed to be inside brackets, nor to be unexpectedly capitalised. In the third, the ends are included. On success we return the word number n, which could be any non-negative number; on failure, we return -1.

```
int is_word_intermediate(vocabulary_entry *word, int w1, int w2) {
    int i;
    for (i=w1+1; i<=w2-1; i++) if (lw_array[i].lw_identity == word) return i;
    return -1;
}
int is_word_intermediate_unbracketed(vocabulary_entry *word, int w1, int w2) {
    int i, bl = 0;
    for (i=w1; i<=w2-1; i++) {
        if ((i>w1) && (bl==0) && (lw_array[i].lw_identity == word) && (unexpectedly_upper_case(i) ==
FALSE))
            return i;
        if ((lw_array[i].lw_identity == OPENBRACKET_V) || (lw_array[i].lw_identity == OPENBRACE_V))
bl++;
        if ((lw_array[i].lw_identity == CLOSEBRACKET_V) || (lw_array[i].lw_identity == CLOSEBRACE_V))
bl--;
    }
    return -1;
}
int is_word_intermediate_inc(vocabulary_entry *word, int w1, int w2) {
    int i;
    for (i=w1; i<=w2; i++) if (lw_array[i].lw_identity == word) return i;
    return -1;
}
```

The function is_word_intermediate is called from 4/sent, 4/noun, 7/data, 8/refpt, 9/qty, 9/map, 10/tab, 10/eqns, 11/act, 11/av, 12/phud, 12/phtd, 12/phsf, 12/def and 13/gtok.

The function is_word_intermediate_unbracketed is called from 5/candp.

The function is_word_intermediate_inc is called from 8/creat, 8/mass, 8/knowc, 11/anl, 11/ap, 12/phud and 12/phsf.

§**14.** We are going to need to look for paired brackets at the outside of an excerpt reasonably quickly, and the following routine performs that test.

```
int paired_brackets(int w1, int w2) {
    if [[w1, w2 == OPENBRACKET ... CLOSEBRACKET --> w1, w2]] {
        int i, bl = 1;
        for (i=w1+1; i<=w2-1; i++) {
            if [[word i == OPENBRACKET]] bl++;
            if [[word i == CLOSEBRACKET]] bl--;
            if (bl == 0) return FALSE;
        }
        if (bl == 1) return TRUE;
    }
    return FALSE;
}
```

The function paired_brackets is called from 4/head, 5/tandv and 5/candp.

§**15.** The following looks for "callings", that is, names occurring in noun phrases with the shape "blah blah (called the rhubarb rhubarb)".

```
int word_range_ends_with_calling(int w1, int w2, int *m1, int *m2, int *cw1, int *cw2) {
    int i, n1, n2;
    if (![[w1, w2 == ... CLOSEBRACKET]]) return FALSE;
    for (i=w2-3; i>w1; i--)
        if [[i, w2 == OPENBRACKET called ... CLOSEBRACKET --> n1, n2]] {
            [[n1, n2 == the ... --> n1, n2]];
            if (m1) { *m1 = w1; *m2 = i-1; }          the text before the bracketed clause, "blah blah"
            if (cw1) { *cw1 = n1; *cw2 = n2; }                the calling name, "rhubarb rhubarb"
            return TRUE;
        }
    return FALSE;
}
int word_range_includes_a_calling(int w1, int w2) {
    int j;
    for (j = w2; j > w1+2; j--)
        if (word_range_ends_with_calling(w1, j, NULL, NULL, NULL, NULL)) return TRUE;
    return FALSE;
}
```

The function word_range_ends_with_calling is called from 5/rel, 5/candd and 12/def.
The function word_range_includes_a_calling is called from 9/scene.

§**16.** A somewhat more specialised test, which happens to be useful in spotting multi-word verbs in sentences: we look for a run of one to three known words, starting and ending wholly within the range, and not occurring immediately after a word to avoid.

```
int is_sequence_intermediate(vocabulary_entry *word1, vocabulary_entry *word2,
    vocabulary_entry *word3, vocabulary_entry *avoid_word, int w1, int w2) {
    int i, c = 1;
    if (word2 != 0) c++;
    if (word3 != 0) c++;
    for (i=w1+1; i<=w2-c; i++)
        if ((lw_array[i].lw_identity == word1) &&
            ((c < 2) || (lw_array[i+1].lw_identity == word2)) &&
            ((c < 3) || (lw_array[i+2].lw_identity == word3)) &&
            (lw_array[i-1].lw_identity != avoid_word))
            return i;
    return -1;
}
```

The function is_sequence_intermediate is called from 4/verb.

§**17.** The following finds the first "in" between `w1` and `w2` inclusive. It is not really a loop, and iterates only to retry if it finds an "in" that we consider bogus for our own syntactic purposes: for instance, we don't want to count the "in" from "fixed in place", so if we should find it we retry the search starting from "place". The left wall thus moves rightwards until it reaches the right, in $\leq (w_2 - w_1 + 1)/2$ steps: the worst case is searching "is in is in is in is in ... is in".

```
int is_the_word_in_intermediate(int w1, int w2) {
    int left = w1, right = w2;
    while (left <= right) {
        int j = is_word_intermediate_inc(in_V, left, right);
        if (j >= 1) {
            int prev = j-1, next = j+1;
            if (compare_word(prev, is_V)) { left = next; continue; }
            if (compare_word(prev, are_V)) { left = next; continue; }
            if (compare_word(prev, was_V)) { left = next; continue; }
            if (compare_word(prev, were_V)) { left = next; continue; }
            if (compare_word(prev, been_V)) { left = next; continue; }
            if (compare_word(prev, listed_V)) { left = next; continue; }
            if ((compare_word(prev, fixed_V)) &&
                (next <= right) && (compare_word(next, place_V))) { left = next; continue; }
        }
        return j;
    }
    return -1;
}
```

The function is_the_word_in_intermediate is called from 11/anl and 11/ap.

§18. Searching for unusual spacing in ranges. Looking forward to see how far the current column or row extends, for formatted tables. The idea is that we have a range `w1` to `w2`, and that the current column or row extends from `w1` but may run only part-way through: we look for the first point at which there is a tab break (for column scanning) or a newline break (for row scanning), and return the word position just before that break. If we do not find one, it follows that the entire range holds the current column or row, and we return `w2`.

```
int last_word_of_formatted_text(int w1, int w2, int tab_flag) {
    int i;
    if (w2 == w1) return w1;
    for (i=w1+1; i<=w2; i++)
        if (((tab_flag) && (lw_array[i].lw_break == '\t')) ||
            (isdigit(lw_array[i].lw_break)) ||
            (lw_array[i].lw_break == '\n'))
                return i-1;
    return w2;
}
```

The function last_word_of_formatted_text is called from 10/tab and 10/eqns.

§19. Dividing lists. A little routine used to detect list divisions, paying great attention to the optional use of the serial comma, since this gives Emily such delight. It tries to divide `(w1, w2)` at the leftmost possible position, regarding either "and" or "or" as a divider, or both: the parameter `and_or` is a bitmap consisting of some sum of the constants `LOOK_FOR_AND` and `LOOK_FOR_OR`. If it succeeds, `(left_w1, left_w2)` and `(right_w1, right_w2)` hold the left and right material respectively: thus it is guaranteed that `(left_w1, left_w2)` is indivisible further. Note that, because these four variables are global, it is important to copy them into local variables in any routine making recursive use of `is_list_divided`.

```
int is_list_divided(int w1, int w2, int and_or) {
    int a, o, c, b;
    if ((w1<0) || (w2<0) || (w1>=w2)) return FALSE;
    a = is_word_intermediate_unbracketed(and_V, w1, w2);
    o = is_word_intermediate_unbracketed(or_V, w1, w2);
    c = is_word_intermediate_unbracketed(COMMA_V, w1, w2);
    if ((and_or & LOOK_FOR_AND) == 0) a = -1;
    if ((and_or & LOOK_FOR_OR) == 0) o = -1;
    b = -1;
    if ((a>=0) && ((b<0) || (a<=b))) b = a;
    if ((o>=0) && ((b<0) || (o<=b))) b = o;
    if ((c>=0) && ((b<0) || (c<=b))) b = c;
    if (b<0) return FALSE;
    left_w1 = w1; left_w2 = b-1; right_w1 = b+1; right_w2 = w2;
    if (compare_word(left_w2, COMMA_V)) left_w2--;
    if (compare_word(right_w1, COMMA_V)) right_w1++;
    if (compare_word(right_w1, and_V)) right_w1++;
    if (compare_word(right_w1, or_V)) right_w1++;
    if ((left_w1 <= left_w2) && (right_w1 <= right_w2))
        return TRUE;
    return FALSE;
}
```

The function is_list_divided is called from 2/dl, 4/iext, 4/vm, 4/noun, 5/litp, 7/tc, 9/map, 10/eqns, 10/bib, 10/isin, 11/anl, 11/av, 12/phud, 12/phod, 12/rb and 13/tfg.

§**20. Parenthesis matching.** Note that the empty word range is deemed to have balancing brackets, as is a range with no brackets at all.

```
int brackets_unbalanced(int w1, int w2) {
    int i, lev = 0;
    if ((w1<0) || (w2<w1)) return FALSE;
    for (i=w1; i<=w2; i++) {
        if (compare_word(i, OPENBRACKET_V)) lev++;
        if (compare_word(i, CLOSEBRACKET_V)) lev--;
        if (lev < 0) return TRUE;
    }
    if (lev != 0) return TRUE;
    return FALSE;
}
```

The function brackets_unbalanced is called from 5/rel and 8/tass.

§**21. A detour for encoding text.** XML escapes like `>` instead of `>` are needed when copying literal text into a file which uses HTML or XML.

```
void print_literal_string_to_file(OUTPUT_STREAM, char *p) {
    if (STREAM_ENCODING(OUT) == UTF8_ENC) STREAM_USE_XML_ESCAPES(OUT, TRUE);
    WRITE("%s", p);
    STREAM_USE_XML_ESCAPES(OUT, FALSE);
}
```

The function print_literal_string_to_file is called from 2/lexi, 4/ext, 4/eid, 4/edict, 5/aph and 5/litp.

§**22.** And the following is needed to deal with extension filenames on platforms whose locale is encoded as UTF-8.

```
void transcode_ISO_string_to_UTF8(char *p, char *dest) {
    int i, j;
    for (i=0, j=0; p[i]; i++) {
        int charcode = (int) (((unsigned char *)p)[i]);
        if (charcode >= 128) {
            dest[j++] = 0xC0 + (charcode >> 6);
            dest[j++] = 0x80 + (charcode & 0x3f);
        } else {
            dest[j++] = p[i];
        }
    }
    dest[j] = 0;
}
```

The function transcode_ISO_string_to_UTF8 is called from 1/plat.

§**23. Printing word ranges.** There are two complications here: first, whether to print the original raw text or whether to print the case-normalised text (which will be lower case except for literals); and second, if the destination is a file, what text encoding to use. When writing to a string, we always preserve the ISO Latin-1 encoding of the lexed text.

```
void print_text_to_file(int w1, int w2, OUTPUT_STREAM) {
    int j;
    for (j=w1; j<=w2; j++) {
        print_literal_string_to_file(OUT, lw_array[j].lw_text);
        if (j<w2) WRITE(" ");
    }
}
void print_text_to_string(int w1, int w2, char *str) {
    int j;
    str[0] = 0;
    for (j=w1; j<=w2; j++) {
        sprintf(str+strlen(str), "%s", lw_array[j].lw_text);
        if (j<w2) sprintf(str+strlen(str), " ");
    }
}
void print_text_to_string_truncated(int w1, int w2, char *str, int max) {
    int j;
    str[0] = 0;
    for (j=w1; j<=w2; j++) {
        char *p = lw_array[j].lw_text;
        char *p2 = "";
        int sp = FALSE;
        if (j<w2) { sp = TRUE; p2 = lw_array[j+1].lw_rawtext; }
        if (strlen(str) + strlen(p) + 2 > max) break;
        sprintf(str+strlen(str), "%s", p);
        if (strcmp(p2, ":")==0) sp = FALSE;
        if (strcmp(p2, ",")==0) sp = FALSE;
        if (strcmp(p2, ")")==0) sp = FALSE;
        if (strcmp(p, "(")==0) sp = FALSE;
        if (sp) sprintf(str+strlen(str), " ");
    }
}
```

The function print_text_to_file is called from 2/index, 4/vm, 5/bp, 5/litp, 9/rsdt, 7/kix, 9/pp, 11/act, 11/ina, 11/av and 12/rb.
The function print_text_to_string is called from 4/iext, 7/data, 9/mapbp, 9/madj, 9/map and 12/cinv.
The function print_text_to_string_truncated is called from 12/br.

§**24.** A variation on this tries to contain unreasonably long pastes of quoted literals, and is used in printing out problem messages, where quoting back the offending source text might make unhelpfully vast paragraphs in which the actual information is more or less hidden.

```
define STRING_TOLERANCE_LIMIT 70

void print_modest_sized_text_to_string(int w1, int w2, char *str) {
    int i, j, l; char *p;
    if (w1 < 0) {
        sprintf(str+strlen(str), "<no text>"); return;
    }
    str[0] = 0;
    for (i=w1; i<=w2; i++) {
        p = lw_array[i].lw_rawtext;
        if ((i>w1) && (strcmp(p, ",") != 0) && (strcmp(p, ":") != 0))
            sprintf(str+strlen(str), " ");
        l = strlen(p);
        if (l > STRING_TOLERANCE_LIMIT+5) {
            for (j=0; j<STRING_TOLERANCE_LIMIT/2; j++)
                sprintf(str+strlen(str), "%c", p[j]);
            sprintf(str+strlen(str), " [...] ");
            for (j=STRING_TOLERANCE_LIMIT/2; j>0; j--)
                sprintf(str+strlen(str), "%c", p[l-j]);
        } else {
            sprintf(str+strlen(str), "%s", p);
        }
    }
}
```

The function print_modest_sized_text_to_string is called from 2/prob1, 7/vasp, 9/vpbp and 12/phtd.

§**25. Printing raw word ranges.** Raw, in this context, means that it retains its original case, but not that it retains its original spacing. This is sometimes problematic: we need to go to some trouble to make punctuation look reasonably nice again, to obtain, say,

> The auctioneer said: "I'm not through yet -".

in preference to:

> The auctioneer said : "I'm not through yet -" .

Note that we are not actually preserving the spacing in the original source – that might have line breaks or other curiosities which we don't want: we are instead imposing what we think are normal English conventions. While this will sometimes be wrong, this is only likely to affect the index and problem messages, so the user is not likely to be bothered.

```
void print_raw_text_to_file(int w1, int w2, OUTPUT_STREAM) {
    int j;
    if ((w1 < 0) || (w1>w2)) { WRITE("?%d,%d?", w1, w2); return; }
    for (j=w1; j<=w2; j++) {
        char *p = lw_array[j].lw_rawtext;
        char *p2 = "";
        int sp = FALSE;
        if (j<w2) { sp = TRUE; p2 = lw_array[j+1].lw_rawtext; }
        print_literal_string_to_file(OUT, p);
        if (strcmp(p2, ":")==0) sp = FALSE;
        if (strcmp(p2, ",")==0) sp = FALSE;
```

```
        if (strcmp(p2, ")")==0) sp = FALSE;
        if (strcmp(p, "(")==0) sp = FALSE;
        if (sp) WRITE(" ");
    }
}
void print_raw_text_to_string(int w1, int w2, char *str) {
    int j;
    str[0] = 0;
    for (j=w1; j<=w2; j++) {
        char *p = lw_array[j].lw_rawtext;
        char *p2 = "";
        int sp = FALSE;
        if (j<w2) { sp = TRUE; p2 = lw_array[j+1].lw_rawtext; }
        sprintf(str+strlen(str), "%s", p);
        if (strcmp(p2, ":")==0) sp = FALSE;
        if (strcmp(p2, ",")==0) sp = FALSE;
        if (strcmp(p2, ")")==0) sp = FALSE;
        if (strcmp(p, "(")==0) sp = FALSE;
        if (sp) sprintf(str+strlen(str), " ");
    }
}
void print_raw_text_to_string_truncated(int w1, int w2, char *str, int max) {
    int j;
    str[0] = 0;
    for (j=w1; j<=w2; j++) {
        char *p = lw_array[j].lw_rawtext;
        char *p2 = "";
        int sp = FALSE;
        if (j<w2) { sp = TRUE; p2 = lw_array[j+1].lw_rawtext; }
        if (strlen(str) + strlen(p) + 2 > max) break;
        sprintf(str+strlen(str), "%s", p);
        if (strcmp(p2, ":")==0) sp = FALSE;
        if (strcmp(p2, ",")==0) sp = FALSE;
        if (strcmp(p2, ")")==0) sp = FALSE;
        if (strcmp(p, "(")==0) sp = FALSE;
        if (sp) sprintf(str+strlen(str), " ");
    }
}
```

The function print_raw_text_to_file is called from 2/index, 2/lexi, 4/ext, 4/edoc, 4/head, 5/rel, 5/aph, 5/litp, 7/kix, 9/qty, 9/scene, 9/prop, 9/pp, 9/cot, 9/inpw, 9/map, 9/tmap, 10/tab, 10/eqns, 10/fig, 10/sfx, 10/exf, 11/act, 11/nap, 12/phud, 12/phod, 12/phsf, 12/phin, 12/br, 12/rb, 13/gl and 13/test.
The function print_raw_text_to_string is called from 2/lexi, 4/ext, 4/iext, 4/edoc, 7/data, 8/creat, 9/map and 10/bib.
The function print_raw_text_to_string_truncated is called from 4/ext, 4/edict, 4/head, 12/br and 12/rb.

§**26.** And something similar, but formatted for use in I6 double-quoted text. In effect this is a further encoding again, but we only need it for this one routine. Here we don't care about punctuation spacing, because we are only writing in comments and I6 debugging routines, but we do need to use the I6 escape ~ for a double-quotation mark.

```
void print_raw_text_within_i6_literal(OUTPUT_STREAM, int w1, int w2) {
    int j, k;
    for (j=w1; j<=w2; j++) {
        char *str = lw_array[j].lw_rawtext;
        if (j>w1) WRITE(" ");
        for (k=0; str[k] != 0; k++) {
            char c = str[k];
            if (c == '"') c = '~';
            WRITE("%c", c);
        }
    }
}
```

The function print_raw_text_within_i6_literal is called from 4/ext, 5/aph, 12/phud, 12/br and 12/rb.

§**27. Printing word ranges to the debugging log.** The plain vanilla version of this routine is defended against malformed word ranges, since bugs can cause these to appear, and that is when we need the debugging log most.

```
void log_word_range(int w1, int w2) {
    if (w1 >= 0) {
        if (w2 < w1) w2 = w1;
        print_text_to_file(w1, w2, dl);
    }
}
```

The function log_word_range is called from 2/dl, 9/prop and 10/isin.

§**28.** Well, so this is useful because the form of breaks used between words is significant when parsing tables, which use tabs to divide columns.

```
void log_word_range_with_whitespace(int w1, int w2) {
    int i;
    if (w1<0) return;
    for (i=w1; i<=w2; i++) {
        char *del = "[sp]";
        if (lw_array[i].lw_break == '\n') del = "[cr]";
        if (lw_array[i].lw_break == '\t') del = "[tab]";
        if (isdigit(lw_array[i].lw_break)) del = "[cr+tab(s)]";
        LOG("%s%s", lw_array[i].lw_text, del);
        if (i<w2) LOG(" ");
    }
    LOG("\n");
}
```

The function log_word_range_with_whitespace is called from 2/dl.

§**29.** Not to be done lightly: the output can be enormous.

```
void log_lexer_output(void) {
    int i;
    LOG("Entire lexer output to date:\n");
    for (i=0; i<lexer_wordcount; i++) {
        LOG("%d: <%s> <%s> <%c>\n", i, lw_array[i].lw_text, lw_array[i].lw_rawtext, lw_array[i].lw_break);
    }
    LOG("------\n");
}
```

The function log_lexer_output is called from 3/read.

Vocabulary 3/vocab

Purpose

To classify the words in the lexical stream, where two different words are considered equivalent if they are unquoted and have the same text, taken case insensitively.

3/vocab.§10 Hash coding of words; §11-14 The hash table of vocabulary

---

Definitions

¶1. The following structure is created for each different word found in the source. (Recall that these are not necessarily words in the usual English sense: for instance, `17` is a word here.)

The vocabulary entry structure exists to make textual comparisons faster, which is essential to make NI run tolerably quickly: NI's speed on typical source texts increased by a factor of 5-10 when this structure was introduced. Firstly, the vocabulary is hashed so that it is not too painful to compare a newly-read word against the known vocabulary; secondly, each word stores linked lists of meanings which it begins, occurs in the middle of, ends, or is optionally part of (in the sense that "brown" is optionally part of the name "small brown shoe", which could also be written "small shoe"); and thirdly, each word also carries a bitmap of flags indicating the possible contexts in which it might be used. Finally, to avoid parsing the same text over and over for its possible meaning as a literal integer, we cache the result: for instance, 17 for the text `17`.

The meaning codes alluded to below are also used for excerpts of text (i.e., are not just for single words) and are defined in Excerpt Meanings.

```
typedef struct vocabulary_entry {
    int flags;                                        bitmap of "meaning codes" indicating possible usages
    int literal_number_value;                         evaluation as a literal number, if any
    char *exemplar;                                   text of one instance of this word
    char *raw_exemplar;                               text of one instance in its raw untreated form
    struct meaning_list *start_list;                  meanings starting with this
    struct meaning_list *end_list;                    meanings ending with this
    struct meaning_list *middle_list;                 meanings with this inside but at neither end
    struct meaning_list *subset_list;                 meanings allowing subsets which include this
    int subset_list_length;                           number of meanings in the subset list
    int hash;                                         hash code derived from text of word
    struct vocabulary_entry *next_in_vocab_hash;      next in list with this hash
    struct vocabulary_entry *lower_case_form;         or null if none exists
    struct vocabulary_entry *upper_case_form;         or null if none exists
} vocabulary_entry;
```

The structure vocabulary_entry is shared with 3/words, 5/em, 5/arch and 5/parse.

---

§**1.** Each distinct word is to have a unique `vocabulary_entry` structure, and the "identity" at word number `wn` is to point to the structure for the text at that word. Two words are distinct if their lower-case forms are different, except that two quoted literal texts are always distinct, even if they have the same content. So for instance,

Daleks conquer and destroy! "Ba-dum." Exterminate, exterminate! "Ba-dum."

would be identified as

```
ve0 ve1 ve2 ve3 ve4 ve5 ve6 ve6 ve4 ve7
```

where `ve4` is the common identity of both exclamation marks, and `ve6` that of the two "exterminate"s, even though they have different casings; while the quoted text `"Ba-dum."` came out with two different identities `ve5` and `ve7`.

When we want to set the identity for a given word, we call these front-door routines, either on a single word or on a range.

```
void vocab_identify_word(int wn) {
    lw_array[wn].lw_identity = vocab_entry_for_text(lw_array[wn].lw_text);
    lw_array[wn].lw_identity->raw_exemplar = lw_array[wn].lw_rawtext;
}
void vocab_identify_word_range(int w1, int w2) {
    int i;
    for (i=w1; i<=w2; i++) vocab_identify_word(i);
}
```

The function vocab_identify_word is called from 3/lex and 3/lexs.

The function vocab_identify_word_range is called from 3/lwb, 4/sent and 12/phsf.

§**2.** Should we ever change the text of a word, it's essential to re-identify it, as otherwise its `lw_identity` points to the wrong vocabulary entry.

```
void change_text_of_word(int wn, char *new) {
    lw_array[wn].lw_text = new;
    lw_array[wn].lw_rawtext = new;
    vocab_identify_word(wn);
}
```

The function change_text_of_word is called from 4/iext and 4/rtree.

§**3.** We now need some utilities for dealing with vocabulary entries. Here is a creator, and a debugging logger:

```
vocabulary_entry *vocab_entry_new(char *text, int hash_code, int flags, int val) {
    vocabulary_entry *ve;
    int l = strlen(text);
    ve = CREATE(vocabulary_entry);
    ve->exemplar = text; ve->raw_exemplar = text;
    ve->start_list = NULL; ve->end_list = NULL; ve->middle_list = NULL;
    ve->subset_list = NULL; ve->subset_list_length = 0;
    ve->next_in_vocab_hash = NULL;
    ve->lower_case_form = NULL; ve->upper_case_form = NULL;
    ve->hash = hash_code;
    ve->flags = flags;
    if ((l>3) && (text[l-3] == 'i') && (text[l-2] == 'n') && (text[l-1] == 'g'))
        ve->flags |= ING_MC;
    ve->literal_number_value = val;
    return ve;
}
void log_vocabulary_entry(vocabulary_entry *ve) {
    if (ve == NULL) { LOG("NULL"); return; }
    if (ve->exemplar == NULL) { LOG("NULL-EXEMPLAR"); return; }
    LOG("%08x-%s", ve->hash, ve->raw_exemplar);
}
```

The function log_vocabulary_entry is called from 2/dl.

§**4.** It's perhaps unexpected that a vocabulary entry not only stores a (pointer to) a copy of the text, the "exemplar" (since it is text which is an example of this vocabulary being used), but also a separate raw copy of the text: raw in the sense of retaining the original form in the source files which the word came from. This looks strange because we normally identify words on their case-lowered text, not on their raw text. In the source material:

> Former Marillion vocalist Fish derived his nickname not from a fish, but from habitual bathing.

words 4, "Fish", and 11, "fish", each have the same vocabulary entry as identity, even though their raw texts differ. Clearly the ordinary exemplar of this entry must be "fish". But what should the raw exemplar be, "Fish" or "fish"? The answer is the latter, or in general, the raw exemplar will always be the same as the exemplar; unless we have amended it by hand, using the following routine.

```
void vocab_set_raw_exemplar_to_text(int wn) {
    lw_array[wn].lw_identity->raw_exemplar = lw_array[wn].lw_text;
}
```

The function vocab_set_raw_exemplar_to_text is called from 3/lexs.

§**5.** Here are some access routines for the data stored in this structure:

```
char *vocab_get_exemplar(vocabulary_entry *ve, int raw) {
    if (raw) return ve->raw_exemplar;
    else return ve->exemplar;
}
```

The function vocab_get_exemplar is called from 2/lexi, 5/bp, 5/rel, 5/conj, 5/em, 7/tids, 7/data, 7/kix, 9/cmpbp, 9/vpbp, 9/model and 11/act.

§**6.** An integer is stored at each vocabulary entry, recording its value if it every turns out to parse as a literal number:

```
int vocab_get_literal_number_value(vocabulary_entry *ve) {
    return ve->literal_number_value;
}
void vocab_set_literal_number_value(vocabulary_entry *ve, int val) {
    ve->literal_number_value = val;
}
```

The function vocab_get_literal_number_value is called from 4/vm, 5/litp, 5/unitr, 5/lit, 5/candd, 8/refpt, 9/map and 10/isin.
The function vocab_set_literal_number_value is called from 5/lit and 7/tids.

§**7.** Almost all text is used case insensitively in Inform source, but we do occasionally need to distinguish "The" from "the" and the like, when parsing the names of text substitutions. When a new text substitution is declared whose first word, in the definition, begins with a capital letter, `vocab_make_case_sensitive` is called on the first word, and its identity is changed to the upper case variant form.

```
int vocab_used_case_sensitively(vocabulary_entry *ve) {
    if ((ve->upper_case_form) || (ve->lower_case_form)) return TRUE;
    return FALSE;
}
vocabulary_entry *vocab_get_lower_case_form(vocabulary_entry *ve) {
    return ve->lower_case_form;
}
vocabulary_entry *vocab_make_case_sensitive(vocabulary_entry *ve) {
    if (ve->upper_case_form) return ve->upper_case_form;
    ve->upper_case_form =
        vocab_entry_new(ve->exemplar, ve->hash, ve->flags, ve->literal_number_value);
    ve->upper_case_form->lower_case_form = ve;
    return ve->upper_case_form;
}
```

The function vocab_used_case_sensitively is called from 5/parse.
The function vocab_get_lower_case_form is called from 5/em.
The function vocab_make_case_sensitive is called from 5/em.

§**8.** Finally, each vocabulary entry comes with a bitmap of flags, and here we get to set and test them:

```
void vocab_set_flags(vocabulary_entry *ve, int t) {
    ve->flags |= t;
}
int vocab_test_flags(int wn, int t) {
    return (lw_array[wn].lw_identity->flags) & t;
}
```

The function vocab_set_flags is called from 5/conj, 7/tids and 7/dti.
The function vocab_test_flags is called from 2/isn, 2/java, 2/hdoc, 2/index, 3/lwb, 4/sent, 4/vm, 4/verb, 4/noun, 4/ofs, 5/conj, 5/aph, 5/litp, 5/unitr, 5/parse, 5/lit, 5/candd, 5/varc, 5/candp, 8/tass, 8/refpt, 8/creat, 8/mass, 9/wo, 9/model, 9/map, 10/tab, 10/eqns, 10/exf, 10/isin, 11/anl, 11/ap and 12/def.

§**9.** It can be useful to find the disjunction of the flags for all the words in a range, as that gives us a single bitmap which tells us quickly whether any of the words in that range is a number, or is a word ending in "-ing", and so on:

```
int vocab_disjunction_of_flags(int w1, int w2) {
    int i, d = 0;
    for (i=w1; i<=w2; i++) d |= (lw_array[i].lw_identity->flags);
    return d;
}
```

The function vocab_disjunction_of_flags is called from 11/ap.

§**10. Hash coding of words.** To find all the different words used in the source text, we need in principle to make an enormous number of comparisons of their texts. It is slow to make a correct identification of two texts as being equal: we have to compare their every characters against each other. Fortunately, it can be much faster to tell if they are different. We do this by rapidly deriving a number from their texts, and then comparing the numbers: if different, the texts were different.

The most obvious number would be the length of the text, but this produces too little variation, and too many false positives: "blue" and "cyan", for instance, would each produce the number 4.

Instead we use a standard method to derive a number traditionally called a "hash code". This is the algorithm called "X 30011" in Aho, Sethi and Ullman's standard reference *Compilers: Principles, Techniques and Tools* (1986). Because it is derived from constantly overflowing integer arithmetic, it will produce different codes on different architectures (say, where `int` is 64 bits long rather than 32, or where `char` is unsigned). All that matters is that it provides a good spread of hash codes for typical texts fed into it on any given occasion.

Good results depend on the number of possible codes being not too tiny compared to the number of different texts fed in, and also on the key value 30011 being coprime to this number (but 30011 is prime, so that's easily arranged). A typical source text of 50,000 words has an unquoted vocabulary of only about 2000 different words. The variation in vocabulary size between the smallest text source and the largest is only about a factor of three or four, so there is no need to make a dynamic estimate of the size of the source. We will always choose 997 as the number of possible hash codes produced by X 30011: we reserve a further three special codes to be the hashes of literals rather than ordinary words, and this brings us up to a round 1000.

Inside the lexer, decimal integers such as `-506` were treated as ordinary words, as there were no lexical difficulties in parsing them. Here they begin to semantically diverge from the way other ordinary words are handled: they're treated more like literal texts and I6 inclusions.

```
define HASH_TAB_SIZE 1000                    the possible hash codes are 0 up to this minus 1
define NUMBER_HASH 0                         literal decimal integers, and no other words, have this hash code
define TEXT_HASH 1                           double quoted texts, and no other words, have this hash code
define I6_HASH 2                             the (- word introducing an I6 inclusion uniquely has this hash code

int hash_code_from_word(char *text) {
    unsigned int hash_code = 0;
    char *p = text;
    switch(*p) {
        case '-': if (p[1] == 0) break;                    an isolated minus sign is an ordinary word
            and otherwise fall into...
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            the first character may prove to be the start of a number: is this true?
            for (p++; *p; p++) if (isdigit(*p) == FALSE) goto Try_Text;
            return NUMBER_HASH;
        case ' ': return I6_HASH;
        case '(': if (p[1] == '-') return I6_HASH;
```

```
            break;
        case '"': return TEXT_HASH;
    }
    Try_Text:
    for (p=text; *p; p++) hash_code = hash_code*30011 + (*p);
    return (int) (3+(hash_code % (HASH_TAB_SIZE-3)));                    result of X 30011, plus 3
}
```

§**11. The hash table of vocabulary.** Armed with these hash codes, we now store the pointers to the vocabulary entry structures in linked lists, one for each possible hash code. These begin empty.

```
vocabulary_entry *list_of_vocab_with_hash[HASH_TAB_SIZE];
void vocab_start_hash_table(void) {
    int i;
    for (i=0; i<HASH_TAB_SIZE; i++) list_of_vocab_with_hash[i] = NULL;
}
```

The function vocab_start_hash_table is called from 3/lex.

§**12.** And that leaves only one routine: for finding the unique vocabulary entry pointer associated with the material in `text`. We search the hash table to see if we have the word already, and if not, we add it.

It is in order to set the initial values of the flags for the new word (if it does turn out to be new) that we mandated special hash codes for any number, any text, or any I6 inclusion.

```
int no_vocabulary_entries = 0;
vocabulary_entry *vocab_entry_for_text(char *text) {
    vocabulary_entry *new_entry;
    int hash_code = hash_code_from_word(text), f = 0, val = 0;
    switch(hash_code) {
        case NUMBER_HASH: f = NUMBER_MC; val = atoi(text); break;
        case TEXT_HASH:
            switch (well_formed_text_routine(text)) {
                case TRUE: f = TEXTWITHSUBS_MC; break;
                case FALSE: f = TEXT_MC; break;
                case NOT_APPLICABLE: f = TEXT_MC; break;
            }
            break;
        case I6_HASH: f = I6_MC; break;
        default:
            val = an_ordinal_number(text);
            if (val >= 0) f = NUMBER_MC;          so that "4th", say, picks up the NUMBER_MC flag
            break;
    }
    if (list_of_vocab_with_hash[hash_code] == NULL) {
        ⟨Pi-ty? That word is not in my vocabulary banks 13⟩;
    } else {
        vocabulary_entry *old_entry = NULL;
        int n;
        search the non-empty list of words with this hash code
        for (n=0, new_entry = list_of_vocab_with_hash[hash_code];
            new_entry != NULL;
            n++, old_entry = new_entry, new_entry = new_entry->next_in_vocab_hash)
            if (strcmp(new_entry->exemplar, text) == 0)
```

```
            return new_entry;
        and if we do not find text in there, then...
        ⟨My vision is impaired! I cannot see! 14⟩;
    }
}
```

The function vocab_entry_for_text is called from 3/words and 7/dti.

§**13.** (Cf. episode 6 of the Doctor Who serial *Genesis of the Daleks*, 1975.) Here the list for this word's hash code was empty, either meaning that this is a hash code never seen for any word before (in which case we start the list for that hash code with the new word), or that the word is a text literal – because, for efficiency's sake, we deliberately keep the hash list for all text literals empty.

⟨Pi-ty? That word is not in my vocabulary banks 13⟩ ≡

```
new_entry = vocab_entry_new(text, hash_code, f, val);
if (hash_code != TEXT_HASH) list_of_vocab_with_hash[hash_code] = new_entry;
LOGIF(VOCABULARY, "Word %d <%s> is first vocabulary with hash %d\n",
    no_vocabulary_entries++, text, hash_code);
return new_entry;
```

This code is used in §12.

§**14.** And here, we exhausted the list at entry `n-1`, with the last entry being pointed to by `old_entry`. We add the new word at the end.

⟨My vision is impaired! I cannot see! 14⟩ ≡

```
new_entry = vocab_entry_new(text, hash_code, f, val);
old_entry->next_in_vocab_hash = new_entry;
LOGIF(VOCABULARY, "Word %d <%s> is vocabulary entry no. %d with hash %d\n",
    no_vocabulary_entries++, text, n, hash_code);
return new_entry;
```

This code is used in §12.

Built-In Words 3/words

Purpose

To stock up NI's vocabulary with words which are needed in order to compare the source text against various syntaxes known to NI.

Template interpreter commands

```
1   {-callv:make_reserved_words}
```

Definitions

¶1. If we want to see if a given word in the source is "relation", we want to do so the quick way: by comparing the identities in the vocabulary bank, which involves only a single pointer comparison. All the words in the source have identities: but we have no idea what the pointer for "relation" would be. So when NI begins, it stocks its vocabulary banks by identifying texts of about 550 words it will at one time or another need to compare against. (Usually this is done in the double-square-bracket parsing extension syntaxes, where the `_V` suffixes are dropped).

```
vocabulary_entry *ASTERISK_V;
vocabulary_entry *BRACEDSTAR_V;
vocabulary_entry *CAPD_A_V;
vocabulary_entry *CAPD_An_V;
vocabulary_entry *CAPD_The_V;
vocabulary_entry *CLOSEBRACE_V;
vocabulary_entry *CLOSEBRACKET_V;
vocabulary_entry *COLON_V;
vocabulary_entry *COMMA_V;
vocabulary_entry *COMPILER_CRASH10_V;
vocabulary_entry *COMPILER_CRASH11_V;
vocabulary_entry *COMPILER_CRASH1_V;
vocabulary_entry *DASHDASH_V;
vocabulary_entry *DASH_V;
vocabulary_entry *DOCUMENTATION_V;
vocabulary_entry *DOTDOTDOT_V;
vocabulary_entry *DOUBLEDOUBLEQUOTE_V;
vocabulary_entry *EQUALS_V;
vocabulary_entry *FORWARDSLASH_V;
vocabulary_entry *FOURASTERISKS_V;
vocabulary_entry *FOURDASHES_V;
vocabulary_entry *FULLSTOP_V;
vocabulary_entry *GE_V;
vocabulary_entry *GT_V;
vocabulary_entry *HYPHEN_V;
vocabulary_entry *LE_V;
vocabulary_entry *LT_V;
vocabulary_entry *OPENBRACKET_V;
vocabulary_entry *OPENBRACE_V;
vocabulary_entry *OPENI6_V;
vocabulary_entry *PLUS_V;
vocabulary_entry *SEMICOLON_V;
vocabulary_entry *STROKE_V;
```

```
vocabulary_entry *THREEASTERISKS_V;
vocabulary_entry *TWOASTERISKS_V;
vocabulary_entry *ZERO_V;
vocabulary_entry *a_V;
vocabulary_entry *abide_V;
vocabulary_entry *able_V;
```

...and so on...

```
vocabulary_entry *z_V;
vocabulary_entry *zero_V;
vocabulary_entry *zeros_V;
vocabulary_entry *zmachine_V;
```

---

§**1.** This catalogue of assignments looks as if it should have been mechanically generated, and indeed the first draft of it was, but it is now maintained by hand (in that very occasional additions are made by hand).

```
void make_reserved_words(void) {
    ASTERISK_V = vocab_entry_for_text("*");
    BRACEDSTAR_V = vocab_entry_for_text("*}");
    CAPD_A_V = vocab_entry_for_text("A");
    CAPD_An_V = vocab_entry_for_text("An");
    CAPD_The_V = vocab_entry_for_text("The");
    CLOSEBRACE_V = vocab_entry_for_text("}");
    CLOSEBRACKET_V = vocab_entry_for_text(")");
    COLON_V = vocab_entry_for_text(":");
    COMMA_V = vocab_entry_for_text(",");
    COMPILER_CRASH10_V = vocab_entry_for_text("ni__crash__10");
    COMPILER_CRASH11_V = vocab_entry_for_text("ni__crash__11");
    COMPILER_CRASH1_V = vocab_entry_for_text("ni__crash__1");
    DASHDASH_V = vocab_entry_for_text("--");
    DASH_V = vocab_entry_for_text("-");
    DOCUMENTATION_V = vocab_entry_for_text("documentation");
    DOTDOTDOT_V = vocab_entry_for_text("dotdotdot");
    DOUBLEDOUBLEQUOTE_V = vocab_entry_for_text("\"\"");
    EQUALS_V = vocab_entry_for_text("=");
    FORWARDSLASH_V = vocab_entry_for_text("/");
    FOURASTERISKS_V = vocab_entry_for_text("****");
    FOURDASHES_V = vocab_entry_for_text("----");
    FULLSTOP_V = vocab_entry_for_text(".");
    GE_V = vocab_entry_for_text(">=");
    GT_V = vocab_entry_for_text(">");
    HYPHEN_V = vocab_entry_for_text("-");
    LE_V = vocab_entry_for_text("<=");
    LT_V = vocab_entry_for_text("<");
    OPENBRACE_V = vocab_entry_for_text("{");
    OPENBRACKET_V = vocab_entry_for_text("(");
    OPENI6_V = vocab_entry_for_text("(-");
    PLUS_V = vocab_entry_for_text("+");
    SEMICOLON_V = vocab_entry_for_text(";");
    STROKE_V = vocab_entry_for_text("|");
    THREEASTERISKS_V = vocab_entry_for_text("***");
    TWOASTERISKS_V = vocab_entry_for_text("**");
```

```
    ZERO_V = vocab_entry_for_text("0");
    a_V = vocab_entry_for_text("a");
    abide_V = vocab_entry_for_text("abide");
    able_V = vocab_entry_for_text("able");
```

...and so on...

```
    z_V = vocab_entry_for_text("z");
    zero_V = vocab_entry_for_text("zero");
    zeros_V = vocab_entry_for_text("zeros");
    zmachine_V = vocab_entry_for_text("z-machine");

    first_V->flags = ORDINAL_MC; first_V->literal_number_value = 1;
    second_V->flags = ORDINAL_MC; second_V->literal_number_value = 2;
    third_V->flags = ORDINAL_MC; third_V->literal_number_value = 3;
    fourth_V->flags = ORDINAL_MC; fourth_V->literal_number_value = 4;
    fifth_V->flags = ORDINAL_MC; fifth_V->literal_number_value = 5;
    sixth_V->flags = ORDINAL_MC; sixth_V->literal_number_value = 6;
    seventh_V->flags = ORDINAL_MC; seventh_V->literal_number_value = 7;
    eighth_V->flags = ORDINAL_MC; eighth_V->literal_number_value = 8;
    ninth_V->flags = ORDINAL_MC; ninth_V->literal_number_value = 9;
    tenth_V->flags = ORDINAL_MC; tenth_V->literal_number_value = 10;
    eleventh_V->flags = ORDINAL_MC; eleventh_V->literal_number_value = 11;
    twelfth_V->flags = ORDINAL_MC; twelfth_V->literal_number_value = 12;

    zero_V->flags = NUMBER_MC; zero_V->literal_number_value = 0;
    one_V->flags = NUMBER_MC; one_V->literal_number_value = 1;
    two_V->flags = NUMBER_MC; two_V->literal_number_value = 2;
    three_V->flags = NUMBER_MC; three_V->literal_number_value = 3;
    four_V->flags = NUMBER_MC; four_V->literal_number_value = 4;
    five_V->flags = NUMBER_MC; five_V->literal_number_value = 5;
    six_V->flags = NUMBER_MC; six_V->literal_number_value = 6;
    seven_V->flags = NUMBER_MC; seven_V->literal_number_value = 7;
    eight_V->flags = NUMBER_MC; eight_V->literal_number_value = 8;
    nine_V->flags = NUMBER_MC; nine_V->literal_number_value = 9;
    ten_V->flags = NUMBER_MC; ten_V->literal_number_value = 10;
    eleven_V->flags = NUMBER_MC; eleven_V->literal_number_value = 11;
    twelve_V->flags = NUMBER_MC; twelve_V->literal_number_value = 12;

    not_V->flags = NOT_MC;

    in_V->flags = INORON_MC + PREPOSITION_MC;
    on_V->flags = INORON_MC + PREPOSITION_MC;

    which_V->flags = PREPOSITION_MC;
    who_V->flags = PREPOSITION_MC;
    that_V->flags = PREPOSITION_MC;

    a_V->flags = ARTICLE_MC;
    an_V->flags = ARTICLE_MC;
    the_V->flags = ARTICLE_MC;
    some_V->flags = ARTICLE_MC;
}
```

The function make_reserved_words is invoked by a command in a `.i6t` template file.

§**2.** That completes the assembly of the basic parsing tools we need for dealing with individual source words: we are finally ready to begin attacking the source text on the level of sentences and clauses.

4 Sentences and Extensions

`4/pn`: *Parse Tree.w* To construct a single parse tree with excerpts of text, and in some cases also annotations, at each node.

`4/sent`: *Sentences.w* To break up the stream of words produced by the lexer into English sentences, and join each to the parse tree.

`4/ext`: *Extension Files.w* To keep details of the extensions currently loaded, their authors, titles, versions and rubrics, and to index and credit them suitably.

`4/iext`: *Including Extensions.w* To fulfill requests to include extensions, adding their material to the parse tree as needed, and removing INCLUDE nodes.

`4/eid`: *Extension Identifiers.w* To store, hash code and compare title/author pairs used to identify extensions which, though installed, are not necessarily used in the present source text.

`4/excen`: *Extension Census.w* To conduct a census of all the extensions installed (whether used on this run or not), and keep the documentation index for them up to date.

`4/edict`: *Extension Dictionary.w* To maintain a database of names and constructions in all extensions so far used by this installation of Inform, and spot potential namespace clashes.

`4/edoc`: *Extension Documentation.w* To generate HTML documentation for extensions.

`4/vm`: *Virtual Machines.w* I7 supports a variety of virtual machines as targets. Most source text should be independent of the target VM, but sometimes numbering is needed, and this is where any VM dependencies are decided.

`4/head`: *Headings.w* To keep track of the hierarchy of headings and subheadings found in the source text.

`4/verb`: *Verb Phrases.w* To construct verb-phrase nodes in the parse tree.

`4/noun`: *Noun Phrases.w* To construct noun-phrase subtrees for assertion sentences found in the parse tree.

`4/ofs`: *Of and From.w* To verify that "of" and "from" subtrees in assertion sentence noun phrases are validly used, and reconstruct their sentences without them if they are not.

`4/rtree`: *Rule Subtrees.w* To tidy up `COMMAND_NT` nodes into a list of children under the relevant `ROUTINE_NT` node, and so turn each rule definition into a single subtree.

`4/verif`: *Verify Parse Tree.w* To check that the parse tree has been correctly constructed, in hopes of detecting bugs in the program: this check should never fail.

Parse Tree 4/pn

Purpose

To construct a single parse tree with excerpts of text, and in some cases also annotations, at each node.

Template interpreter commands

```
14  {-callv:plant_parse_tree}
```

Definitions

¶1. Most algorithms for parsing natural language involve the construction of trees, in which the original words appear as leaves at the top of the tree, while the grammatical functions they serve appear as the branches and trunk: thus the word "orange", as an adjective, might be growing from a branch which represents a noun clause ("the orange envelope"), growing in turn from a trunk which in turn might represent a assertion sentence:

> The card is in the orange envelope.

NI is no exception, though nodes seldom go down to the level of individual words. As we shall see, the entire source text is converted to a single but fairly "lightweight" parse tree, containing just enough structure to enable us to see which more sophisticated parsing techniques should be used. The parse tree is more of a navigational aid on the journey to meaning than the destination itself.

Though the parse tree is never going to tell the whole story, its nodes are nonetheless gradually annotated with richer and richer semantic information as NI proceeds. In the above example, for instance, the two noun phrases are both marked as having the definite article, and this has subsumed the usage of the word "the" twice in the original text: neither "the" remains in the parse tree. (Nor does the punctuation, nor the word "is".)

¶2. Node types. Each node in the parse tree has a type, which must be one of the following. The numbers have no significance except that they are all different, and that they must correspond exactly with the sequence of entries in the constant array defined when we verify the parse tree.

```
define ROOT_NT 1                          Only one such node exists: the tree root
define BIBLIOGRAPHIC_NT 2                 For the initial title sentence
define HEADING_NT 3                       "Chapter VIII: Never Turn Your Back On A Shreve"
define INCLUDE_NT 4                       "Include School Rules by Argus Filch"
define BEGINHERE_NT 5                     "The Standard Rules begin here"
define ENDHERE_NT 6                       "The Standard Rules end here"
define SENTENCE_NT 7                      "The Garden is a room"
define ROUTINE_NT 8                       "Instead of taking something, ..."
define INFORM6CODE_NT 9                   "Include (- ... -)
define TABLE_NT 10                        "Table 1 - Counties of England"
define EQUATION_NT 11                     "Equation 2 - Newton's Second Law"
define TRACE_NT 12                        A sentence consisting of a single asterisk, perhaps followed by quoted text
define RELATIONSHIP_NT 13                 "on"
define CALLED_NT 14                       "On the table is a container called the box"
define WITH_NT 15                         "The footstool is a supporter with capacity 2"
```

```
define AND_NT 16                    "whisky and soda"
define KIND_NT 17                   "A woman is a kind of person"
define X_OF_Y_NT 18                 "description of the painting"
define VERB_NT 19                   "is"
define CREATED_NT 20                "a vehicle called Sarah Jane's car"
define NOUNPHRASE_NT 21             "the red handkerchief"
define PROPERTYLIST_NT 22           "capacity 2"
define FROM_NT 23                   "East from the Garden is the Grove"
define COMMAND_NT 24                "change the Marble Door to open"
define ALLOWED_NT 25                "An animal is allowed to have a description"
define EVERY_NT 26                  "every container"
define INSTANCE_NT 27               "a container"
define VALUE_NT 28                  "102"
define ACTION_NT 29                 "taking something closed"
define ADJECTIVE_NT 30              "open"
define PROPERTYNOUN_NT 31           "score for finding"
define PROPERTYCALLED_NT 32         "A man has a number called age"
define TOKEN_NT 33                  Used for tokens in grammar
define HIGHEST_NODE_TYPE 34         Well, actually the highest plus 1
```

¶3. The parse node structure. This is implemented as a basic framework structure to which any number of annotations can be attached, but such that an unannotated node occupies a reasonably small amount of memory. The number of PNs created is typically of the same order as the number of words in the original source text: so perhaps 100,000 for a large work. Since PNs have a `sizeof` of about 36 on a 32-bit CPU, this keeps the storage for the main structural tree down to about 3MB for a large work, which doesn't seem worth reducing.

```
typedef struct parse_node {
    int node_type;                                  One of the list above
    int word_ref1, word_ref2;                       The text
    struct parse_node_annotation *annotations;      Linked list: see below
    struct parse_node *down;                        Pointers to navigate the tree structure
    struct parse_node *next;
    MEMORY_MANAGEMENT
} parse_node;
```

The structure parse_node is private to this section.

¶4. The parse tree annotations are miscellaneous, and many are needed only at a few unusual nodes. Rather than have the structure grow large, we store annotations in the following:

```
typedef struct parse_node_annotation {
    int kind_of_annotation;
    int annotation_integer;
    general_pointer annotation_pointer;
    struct parse_node_annotation *next_annotation;
} parse_node_annotation;
```

The structure parse_node_annotation is private to this section.

¶5. The annotations used are identified by ID numbers, as follows:

define implicit_in_creation_of_ANNOT 1 — world_object: *for assemblies*
define implicitness_count_ANNOT 2 — *int: keeping track of recursive assemblies*
define relationship_ANNOT 3 — binary_predicate: *for RELATIONSHIP nodes*
define property_ANNOT 4 — property_name: *for names of properties*
define resolved_ANNOT 5 — *int: temp storage when resolving NPs*
define full_phrase_evaluation_ANNOT 6 — specification: *temp storage when resolving NPs*
define negated_boolean_ANNOT 7 — *int: set if adjective meant negatively*
define verbal_certainty_ANNOT 8 — *int: a certainty level attached to a sentence*
define slash_class_ANNOT 9 — *int: used when partitioning grammar tokens*
define table_cell_unspecified_ANNOT 10 — *int: used to mark table entries as unset*
define grammar_token_code_ANNOT 11 — *int: used to identify grammar tokens*
define plural_reference_ANNOT 12 — *int: used by NOUNPHRASE nodes for evident plurals*
define heading_level_ANNOT 13 — *int: for HEADING nodes, a hierarchical level, 0 (highest) to 9 (lowest)*
define grammar_token_literal_ANNOT 14 — *int: for grammar tokens which are literal words*
define nounphrase_article_ANNOT 15 — *int: definite or indefinite article: see below*
define sentence_unparsed_ANNOT 16 — *int: set if verbs haven't been sought yet here*
define verb_id_ANNOT 17 — *int: identifying what kind of VERB node*
define relationship_node_type_ANNOT 18 — *int: what kind of inference this assertion makes*
define I6_routine_wn_ANNOT 19 — *int: word number for literal I6 routine name*
define evaluation_ANNOT 20 — specification: *result of evaluating the text*
define refers_to_ANNOT 21 — world_object: *specific object or kind referred to*
define creation_modifier_ANNOT 22 — *int: bitmap of modifiers for kind of value*
define grammar_token_relation_ANNOT 23 — binary_predicate: *for relation tokens*
define table_cell_allocated_ANNOT 24 — *int: table entry already correctly allocated*
define nounphrase_multiplicity_ANNOT 25 — *int: multiplicity of bare number determiner*
define interpretation_of_subject_ANNOT 26 — world_object: *subject, during passes*
define category_of_I6_translation_ANNOT 27 — *int: what sort of "translates into I6" sentence this is*
define suppress_heading_dependencies_ANNOT 28 — *int: ignore extension dependencies on this heading node*
define row_amendable_ANNOT 29 — *int: a candidate row for a table amendment*
define colon_block_command_ANNOT 30 — *int: this COMMAND uses the ":" not begin/end syntax*
define suppress_debug_text_ANNOT 31 — *int: don't print the RULES ALL text for this phrase*
define aph_ANNOT 32 — adjectival_phrase: *which adjective is asserted*
define creation_proposition_ANNOT 33 — pcalc_prop: *proposition which newly created value satisfies*

¶6. A few of the more popular annotations are worth documenting here. First, NOUNPHRASE nodes use the article annotation to record the article used:

define DEF_ART 1 — *the definite article*
define INDEF_ART 2 — *the indefinite article*
define NO_ART 3 — *no article supplied*
define IT_ART 4 — *a special case to handle "it"*

¶7. Second, RELATIONSHIP nodes, which indicate a (usually spatial) relationship, are annotated with a choice of inference type to indicate the particular relationship which seems to be intended: PARTOF_INF, for instance. Such constants are declared in Inferences.

In similar vein, VERB nodes are annotated with a verb type constant, and these are defined in Verb Phrases.

¶8. Where we currently are in the text. NI makes many traverses through the parse tree, often modifying as it goes, and keeps track of its position so that it can make any problem messages correctly refer to the location of the faulty text in the original source files.

During such traverses, `current_sentence` is always the subtree being looked at: it is always a child of the tree root, and is usually a `SENTENCE_NT` node, hence the name.

During the main assertion traverses, we also keep track of whether we are near the start of an extension file or not. (If we are, then a lone string of text is interpreted as the rubric of the extension – an exception to Inform's normal rules.)

```
parse_node *tree_root = NULL;
parse_node *current_sentence = NULL;
int near_start_of_extension = 0;
```

¶9. Two macros are frequently used in NI in order to abstract and abbreviate the traversal process. For instance, we might write:

```
for (TREE_START(p); p; TREE_NEXT(p))
```

and this is expanded using the following macro definitions:

```
define TREE_START(p) p=tree_root->down,current_sentence=p
define TREE_NEXT(p) p=p->next,current_sentence=p
```

---

§**1.** The following routine creates a new parse node. (Here, as for many other NI structures, there is no garbage collection: if we allocate one, and then forget the pointer, it is lost forever and the memory is wasted. This only seldom happens in NI and we are not in pressing need of memory. But we should bear it in mind before coding something which only speculatively grabs new structures.)

```
parse_node *new_node(int type) {
    parse_node *PN = CREATE(parse_node);
    PN->node_type = type;
    PN->word_ref1 = -1; PN->word_ref2 = -1;
    PN->down = NULL; PN->next = NULL;
    PN->annotations = NULL;
    return PN;
}
```

The function new_node is called from 4/sent, 4/verb, 4/noun, 4/rtree, 8/refpt, 8/creat, 8/mass, 8/assem and 9/wo.

§**2.** It's easily overlooked that the single most useful piece of information at each node is its node type, accessed as follows:

```
int pn_get_node_type(parse_node *pn) { return pn->node_type; }
void pn_set_node_type(parse_node *pn, int nt) { pn->node_type = nt; }
```

The function pn_get_node_type is called from 2/prob2, 2/prob3, 3/lwb, 4/iext, 4/head, 4/verb, 4/ofs, 4/rtree, 4/verif, 6/treec, 8/tass, 8/refpt, 8/creat, 8/mass, 8/assem, 8/knowc, 8/knowp, 8/relv, 8/imp, 9/pp, 9/model, 9/map, 10/tab, 10/eqns, 11/act, 11/av, 12/cs, 12/ph, 12/cph, 12/def, 12/rb, 13/tfg and 13/gl.
The function pn_set_node_type is called from 4/sent, 4/noun, 4/ofs, 4/verif, 8/refpt, 8/creat, 8/mass, 8/assem, 8/relv, 9/pp, 10/tab and 13/gtok.

§**3. Annotations.** A blank annotation is like a blank luggage ticket, waiting to be filled out and attached to some suitcase:

```
parse_node_annotation *pna_new(int koa) {
    parse_node_annotation *pna = CREATE(parse_node_annotation);
    pna->kind_of_annotation = koa;
    pna->annotation_integer = 0;
    pna->annotation_pointer = NULL_GENERAL_POINTER;
    pna->next_annotation = NULL;
    return pna;
}
```

§**4.** Annotations are identified by an enumerated range of constants (KOA here stands for "kind of annotation"). Each node is permitted an arbitrary selection of these, storing them as a linked list: it will always be short (worst case about 5), so there is no need for a more efficient algorithm to search this list.

```
int pn_has_annotation(parse_node *PN, int koa) {
    parse_node_annotation *pna;
    for (pna=PN->annotations; pna; pna=pna->next_annotation)
        if (pna->kind_of_annotation == koa)
            return TRUE;
    return FALSE;
}
```

The function pn_has_annotation is called from 8/sob and 8/creat.

§**5.** Reading annotations is similar. We need two variant forms: one for reading integer-valued annotations (which is most of them, as it happens) and the other for reading pointers to structures.

```
int pn_int_annotation(parse_node *PN, int koa) {
    parse_node_annotation *pna;
    for (pna=PN->annotations; pna; pna=pna->next_annotation)
        if (pna->kind_of_annotation == koa)
            return pna->annotation_integer;
    return 0;
}
general_pointer pn_pointer_annotation(parse_node *PN, int koa) {
    parse_node_annotation *pna;
    for (pna=PN->annotations; pna; pna=pna->next_annotation)
        if (pna->kind_of_annotation == koa)
            return pna->annotation_pointer;
    return NULL_GENERAL_POINTER;
}
```

The function pn_int_annotation is called from 2/isn, 2/prob2, 3/lwb, 4/head, 4/verb, 4/ofs, 4/rtree, 6/treec, 8/tass, 8/refpt, 8/creat, 8/mass, 8/assem, 8/relv, 8/imp, 9/scene, 9/model, 9/map, 10/tab, 12/cs, 12/cph, 13/tfg, 13/gl and 13/gtok.

§**6.** Integer-valued annotations are set with the following routine. Note that any second or subsequent annotation with the same KOA as an existing one overwrites it.

```
void pn_annotate_int(parse_node *PN, int koa, int v) {
    parse_node_annotation *newpna, *pna, *final = NULL;
    for (pna=PN->annotations; pna; pna=pna->next_annotation) {
        if (pna->kind_of_annotation == koa) {
            an annotation with this KOA exists already: overwrite it
            pna->annotation_integer = v;
            return;
        }
        if (pna->next_annotation == NULL) final = pna;
    }
    no annotation with this KOA exists: create a new one and add to end of node's list
    newpna = pna_new(koa); newpna->annotation_integer = v;
    if (final) final->next_annotation = newpna; else PN->annotations = newpna;
}
```

The function pn_annotate_int is called from 2/isn, 4/sent, 4/head, 4/verb, 4/noun, 4/ofs, 4/rtree, 8/refpt, 8/mass, 8/assem, 10/tab, 12/cs, 13/gl and 13/gtok.

§**7.** Again, almost identical code handles the case of pointer-valued annotations:

```
void pn_annotate_pointer(parse_node *PN, int koa, general_pointer data) {
    parse_node_annotation *newpna, *pna, *final = NULL;
    for (pna=PN->annotations; pna; pna=pna->next_annotation) {
        if (pna->kind_of_annotation == koa) {
            an annotation with this KOA exists already: overwrite it
            pna->annotation_pointer = data;
            return;
        }
        if (pna->next_annotation == NULL) final = pna;
    }
    no annotation with this KOA exists: create a new one and add to end of node's list
    newpna = pna_new(koa); newpna->annotation_pointer = data;
    if (final) final->next_annotation = newpna; else PN->annotations = newpna;
}
```

§**8.** It turns out to be convenient to access annotations with standard-form get and set functions, for pointers, to avoid difficulties with null pointers (which would throw run-time errors as being invalid if the store and retrieve routines were allowed to work on them). It's also less verbose.

```
define MAKE_ANNOTATION_FUNCTIONS(annotation_name, pointer_type)
void pn_set_##annotation_name(parse_node *pn, pointer_type *bp) {
    pn_annotate_pointer(pn, annotation_name##_ANNOT,
        STORE_POINTER_##pointer_type(bp));
}
pointer_type *pn_get_##annotation_name(parse_node *pn) {
    pointer_type *pt = NULL;
    if (pn_has_annotation(pn, annotation_name##_ANNOT))
        pt = RETRIEVE_POINTER_##pointer_type(
            pn_pointer_annotation(pn, annotation_name##_ANNOT));
    return pt;
}
```

§**9.** So we itemise the pointer-valued annotations below, and the macro expands to provide their get and set functions:

```
MAKE_ANNOTATION_FUNCTIONS(evaluation, specification)
MAKE_ANNOTATION_FUNCTIONS(full_phrase_evaluation, specification)
MAKE_ANNOTATION_FUNCTIONS(property, property_name)
MAKE_ANNOTATION_FUNCTIONS(relationship, binary_predicate)
MAKE_ANNOTATION_FUNCTIONS(refers_to, world_object)
MAKE_ANNOTATION_FUNCTIONS(grammar_token_relation, binary_predicate)
MAKE_ANNOTATION_FUNCTIONS(interpretation_of_subject, world_object)
MAKE_ANNOTATION_FUNCTIONS(implicit_in_creation_of, world_object)
MAKE_ANNOTATION_FUNCTIONS(aph, adjectival_phrase)
MAKE_ANNOTATION_FUNCTIONS(creation_proposition, pcalc_prop)
```

§**10. Copying parse nodes.** If we want to duplicate a parse node, we cannot do so with a shallow bit copy: the node points to a list of its annotations, and the duplicated node would therefore point to the same list. If, subsequently, one of the two nodes were annotated further, then the other would change in synchrony, which would be the source of mysterious bugs. We therefore need to perform a deep copy which duplicates not only the node, but also its annotation list.

```
void pn_copy(parse_node *to, parse_node *from) {
    parse_node_annotation *pna, *latest = NULL;
    *to = *from;
    to->annotations = NULL;
    for (pna=from->annotations; pna; pna=pna->next_annotation) {
        parse_node_annotation *pna_copy = CREATE(parse_node_annotation);
        *pna_copy = *pna; pna_copy->next_annotation = NULL;
        if (to->annotations == NULL) to->annotations = pna_copy;
        else latest->next_annotation = pna_copy;
        latest = pna_copy;
    }
}
```

The function pn_copy is called from 8/refpt, 8/creat and 8/assem.

§**11. Detection of subnodes.** This is needed when producing problem messages: we may need to work up from an arbitrary leaf to the main sentence branch containing it. At any rate, given a node `PN`, we want to know if another node `to_find` lies beneath it. (This will never be called when `PN` is the root, and from all other nodes it will certainly run quickly, since the tree is otherwise neither wide nor deep.)

```
int pn_contains(parse_node *PN, parse_node *to_find) {
    parse_node *to_try;
    if (PN == to_find) return TRUE;
    for (to_try = PN->down; to_try; to_try = to_try->next)
        if (pn_contains(to_try, to_find))
            return TRUE;
    return FALSE;
}
```

The function pn_contains is called from 2/prob2.

§**12. The word range beneath a given node.** Any given node may be the root of a subtree concerning the structure of a given contiguous range of words in the original source text. The "left edge" of a node `PN` is the least-numbered word considered by any node at or below `PN` in the tree; the "right edge" is the highest-numbered word similarly considered.

The left edge is calculated by taking the minimum value of the `word_ref1` for `PN` and the left edges of its children, except that -1 is not counted. (A left edge of -1 means no source text is here.)

```
int left_edge_of(parse_node *PN) {
    parse_node *child;
    int l = PN->word_ref1, lc;
    for (child = PN->down; child; child = child->next) {
        lc = left_edge_of(child);
        if ((lc >= 0) && ((l == -1) || (lc < l))) l = lc;
    }
    return l;
}
```

The function left_edge_of is called from 8/mass.

§**13.** Symmetrically, the right edge is found by taking the maximum of `word_ref2` for `PN` and the right edges of its children.

```
int right_edge_of(parse_node *PN) {
    parse_node *child;
    int r = PN->word_ref2, rc;
    if (PN->word_ref1 < 0) r = -1;
    for (child = PN->down; child; child = child->next) {
        rc = right_edge_of(child);
        if ((rc >= 0) && ((r == -1) || (rc > r))) r = rc;
    }
    return r;
}
```

The function right_edge_of is called from 8/mass.

§**14. Planting and grafting the tree.** Further horticultural metaphors follow: this is where we plant the tree, and graft new branches onto it.

```
void plant_parse_tree(void) {
    tree_root = new_node(ROOT_NT);
}
```

The function plant_parse_tree is invoked by a command in a `.i6t` template file.

§**15.** Every node in the tree is indirectly a child of the root node, which is stored in `tree_root`. The tree tends to be very wide: since each sentence in the original source text is a different child of the root, the root may have 5000 or so children, though the maximum depth of the tree might be only 10.

That means that perpetually scanning through them in order to add another one on the end is inefficient: so we cache the "last sentence" in the tree, meaning, the youngest child of root. (But we must only do this when we are not also performing surgery on the tree at the same time, which is why it is not always allowed.)

```
parse_node *youngest_child_of_root = NULL;                    youngest child of tree root
int allow_last_sentence_cacheing = FALSE;

void enable_last_sentence_cacheing(void) {
    youngest_child_of_root = NULL;                    because this may have changed since last enabled
    allow_last_sentence_cacheing = TRUE;
}

void disable_last_sentence_cacheing(void) {
    allow_last_sentence_cacheing = FALSE;
}
```

The function enable_last_sentence_cacheing is called from 4/sent.
The function disable_last_sentence_cacheing is called from 4/sent.

§**16.** This is where metaphors become mixed. The routine below is called `graft` by analogy with gardening, where the rootstock of one plant is joined to a scion (or cutting) of another, so that a root chosen for strength can be combined with the fruits or blossom of the scion. This is fairly apt for the process of joining one subtree onto a node of another. But since gardening lacks words to describe branches as being eldest or youngest, and so on, for the actual body of the routine we talk about family trees instead.

`graft` returns the node for which `newborn` is the immediate sibling, that is, it returns the previously youngest child of the `parent` (or `NULL` if it previously had no children).

```
parse_node *graft(parse_node *newborn, parse_node *parent) {
    parse_node *elder = NULL;
    if (newborn == NULL) internal_error("newborn is null in tree graft");
    if (parent == NULL) internal_error("parent is null in tree graft");
    is the new node to be the only child of the old?
    if (parent->down == NULL) { parent->down = newborn; return NULL; }
    can last sentence cacheing save us a long search through many children of root?
    if ((parent == tree_root) && (allow_last_sentence_cacheing)) {
        if (youngest_child_of_root) {
            elder = youngest_child_of_root;
            elder->next = newborn;
            youngest_child_of_root = newborn;
            return elder;
        }
        we don't know who's the youngest child now, but we know who soon will be:
        youngest_child_of_root = newborn;
    }
    find youngest child of attach node...
    for (elder = parent->down; elder->next; elder = elder->next) ;
    ...and make the new node its younger sibling
    elder->next = newborn; return elder;
}
```

The function graft is called from 4/sent, 4/iext, 4/verb, 9/pp, 9/model, 10/tab and 13/gtok.

§**17. Logging the parse tree.** For most trees, logging is a fearsome prospect, but here we only mean printing out a textual representation to the debugging log.

We usually want to log recursive pictures of the tree, but not always.

```
void log_single_parse_node_at_current_margin(parse_node *p) {
    log_single_node(p, 1);
}
void log_parse_node_at_current_margin(parse_node *p) {
    log_subtree(p, 1);
}
```

The function log_single_parse_node_at_current_margin is called from 2/dl.
The function log_parse_node_at_current_margin is called from 2/dl.

§**18.** The recursive logger covers the subtree from the current node down, but does not include the siblings of the current node. (If it did, logging any sentence node would produce humungous output, as the sentences are typically among thousands of siblings as children of the root.)

```
void log_subtree(parse_node *p, int indentation) {
    if (p == NULL) { LOG("<null-parse-node>"); return; }
    log_single_node(p, indentation);
    if (p->down) { log_subtree_recursively(p->down, indentation+1); }
}
void log_subtree_recursively(parse_node *p, int indentation) {
    if (p == NULL) { LOG("<null-parse-node>"); return; }
    log_single_node(p, indentation);
    if (p->down) { log_subtree_recursively(p->down, indentation+1); }
    if (p->next) { log_subtree_recursively(p->next, indentation); }
}
```

The function log_subtree is called from 2/prob0, 2/prob3, 4/verif, 8/tass, 8/refpt, 8/mass, 8/assem, 8/knowp and 13/gtok.

§**19.** All of those routines make use of the following, which actually performs the log of a parse node. Note that this always produces exactly one line of text in the debugging log, indented as requested.

```
void log_single_node(parse_node *p, int indentation) {
    int t;
    while (indentation-- > 0) LOG("    ");
    if (p == NULL) { LOG("<null-parse-node>"); return; }
    t = p->node_type;
    if ((t<1) || (t>HIGHEST_NODE_TYPE)) { LOG("<node with bad type %d>\n", t); return; }
    LOG("node:$N ", t);
    if (p->word_ref1 >= 0) {
        int w1 = p->word_ref1, w2 = p->word_ref2;
        if (w2 < 0) w2 = w1;
        LOG(" <$W>", w1, w2);
    }
    ⟨Log annotations of parse tree nodes, depending on the node type 20⟩;
    LOG("\n");
}
```

§**20.** We do not log every annotation: only the few which are most illuminating.

⟨Log annotations of parse tree nodes, depending on the node type 20⟩ ≡

```
int show_eval = FALSE, show_prop = FALSE, show_refers = FALSE;
switch(t) {
    case ADJECTIVE_NT: show_prop = TRUE; show_eval = TRUE; break;
    case HEADING_NT: LOG(" (level %d)", pn_int_annotation(p, heading_level_ANNOT)); break;
    case INSTANCE_NT: show_refers = TRUE; show_eval = TRUE; break;
    case KIND_NT: show_refers = TRUE; break;
    case RELATIONSHIP_NT: LOG(" (type:");
        switch (pn_int_annotation(p, relationship_node_type_ANNOT)) {
            case STANDARD_RELN: LOG("standard"); break;
            case PARENTAGE_HERE_RELN: LOG("here"); break;
            case DIRECTION_RELN: LOG("direction"); break;
        }
        LOG(")"); break;
    case NOUNPHRASE_NT:
        switch (pn_int_annotation(p, nounphrase_article_ANNOT)) {
            case NO_ART: LOG(" (no article)"); break;
            case IT_ART: LOG(" (pronoun)"); break;
            case DEF_ART: LOG(" (definite)"); break;
            case INDEF_ART: LOG(" (indefinite)"); break;
        }
        if (pn_int_annotation(p, plural_reference_ANNOT)) LOG(" (plural)");
        if (pn_int_annotation(p, nounphrase_multiplicity_ANNOT))
            LOG(" (multiplicity %d)", pn_int_annotation(p, nounphrase_multiplicity_ANNOT));
        show_refers = TRUE;
        break;
    case PROPERTYNOUN_NT: show_prop = TRUE; break;
    case VALUE_NT: show_eval = TRUE; break;
    case VERB_NT: LOG(" ($V)", pn_int_annotation(p, verb_id_ANNOT)); break;
    case TOKEN_NT: LOG(" [%d]", pn_int_annotation(p, slash_class_ANNOT)); break;
}
if ((show_refers) && (pn_get_refers_to(p))) LOG(" refers:$O", pn_get_refers_to(p));
if ((show_eval) && (pn_get_evaluation(p))) {
    LOG(" evaluation:$S", pn_get_evaluation(p));
    if (pn_int_annotation(p, creation_modifier_ANNOT) != 0)
        LOG(" (modifier %d)", pn_int_annotation(p, creation_modifier_ANNOT));
    if (pn_get_creation_proposition(p))
        LOG(" (subject to $D)", pn_get_creation_proposition(p));
}
```

This code is used in §19.

§**21. The asterisked tree.** The entire parse tree is a bewilderingly large object to take in at a glance, so NI contains a secret debugging feature: a sentence consisting of a single asterisk only switches tracing on and off. Thus, any text in between two asterisk sentences will be watched in the debugging log: the "asterisked tree" is the subset of the tree which fell between asterisks in this way. For instance:

The Court is a room. *. The Judge is a man in the Court. *.

Here a single sentence is watched: the one about the Judge.

```
void log_asterisked_parts_of_parse_tree(char *desc) {
    parse_node *p; int asterisked = FALSE;
    if (dl_this(ASTERISKED_SYNTAX_DA) == FALSE) return;
    LOG("\n%s (asterisked areas only):\n", desc);
    for (TREE_START(p); p; TREE_NEXT(p))
        if (p->node_type == TRACE_NT) asterisked = 1-asterisked;
        else if (asterisked) log_subtree(p, 1);
}
```

The function log_asterisked_parts_of_parse_tree is called from 8/tass.

Sentences 4/sent

Purpose

To break up the stream of words produced by the lexer into English sentences, and join each to the parse tree.

4/sent.§2-10 Sentence breaking; §11-17 Making sentence nodes; §18-19 Begins and ends here; §20-32 Unskipped material which is not a heading

---

Template interpreter commands

```
{-callv:declare_source_loaded}
{-callv:break_source_into_sentences}
```

---

§**1.** Sentence division can happen either early in NI's run, when the vast bulk of the source text is read, or at intermittent periods later when fresh text is generated internally. New sentences need to be treated slightly differently in these cases, so this seems as good a point as any to define the routine which the `.i6t` interpreter calls when it wants to signal that the source text has now officially been read.

```
void declare_source_loaded(void) {
    text_loaded_from_source = TRUE;
}
```

The function declare_source_loaded is invoked by a command in a `.i6t` template file.

§**2. Sentence breaking.** The `break_into_sentences` routine is used for long stretches of text, normally entire files. The following provides a way for the `.i6t` interpreter to apply it to the whole text as lexed, which provides the original basis for parsing. (This won't be the entire source text, though: extensions, including the Standard Rules, have yet to be read.)

```
void break_source_into_sentences(void) {
    break_into_sentences(0, lexer_wordcount-1, NULL);
}
```

The function break_source_into_sentences is invoked by a command in a `.i6t` template file.

§**3.** What breaks a sentence? In ordinary English, question marks, exclamation marks, in some cases ellipses, but mainly full stops. In Inform source text, only full stops are used outside quoted text; but we do have to recognise the other cases when they occur at the end of quoted matter. Moreover, we actually subdivide a little further, because we also want to break up rule "sentences" into their subordinate clauses. Thus, going on punctuation, we recognise rules as having the following model:

Preamble: phrase 1; phrase 2; ...; phrase N.

It is even, in certain limited circumstances, possible that a comma can divide a sentence:

Instead of eating, say "You really aren't hungry just now."

This means that context is important even here, where it might have been expected that all we needed to do was to spot the punctuation marks. So we carry out the sentence breaking with a simple finite state machine – the last sentence having been a rule preamble tells us that the current one is probably a phrase, and so on – and the following is its state.

```
source_file *sfsm_source_file = NULL;
int sfsm_inside_rule_mode = FALSE;
int sfsm_skipping_material_at_level = -1;
int sfsm_in_table_mode = FALSE;
extension_file *sfsm_extension = NULL;
int sfsm_extension_position = 0;    0: not an extension; 1: before "begins here"; 2: before "ends here"; 3: after
```

§**4.** Now for the routine itself. `w1` is the first word the lexer read, `w2` is the last. We break into bite-sized chunks, each of which is despatched to the `make_sentence_node` routine with a note of the punctuation which was used to end it. Each call to this routine represents one cycle of our finite state machine.

```
void break_into_sentences(int w1, int w2, extension_file *from_extension) {
    int position, sentence_start = w1;
    enable_last_sentence_cacheing();
    ⟨Reset the sentence finite state machine 5⟩;
    ⟨Go into table sentence mode if necessary 6⟩;
    for (position=w1; position<=w2; position++)
        if (sentence_start < position) {
            int no_stop_words, back_up_one_word;
            char stop_character;
            ⟨Look for a sentence break, finding the number of stop words and the stop character 8⟩;
            if (no_stop_words > 0) {
                make_sentence_node(sentence_start, position-1, stop_character);
                position = position + no_stop_words - 1;
                if (back_up_one_word) sentence_start = position;
                else sentence_start = position + 1;
                ⟨Go into table sentence mode if necessary 6⟩;
            }
        }
    if ((sentence_start < w2) ||
        ((sentence_start == w2) && (![[word w2 == STROKE]])))
        make_sentence_node(sentence_start, w2, '.');
    disable_last_sentence_cacheing();
    if (from_extension)
        ⟨Issue a problem message if we are missing the begin and end here sentences 19⟩;
}
```

The function break_into_sentences is called from 4/iext.

§**5.** Each call to `break_into_sentences` starts afresh, with no residual state left over from previous calls. (The same cannot be said for `make_sentence_node`, which constructs individual sentences and is repeatedly called by us, and that is why these are global variables rather than locals in `break_into_sentences`.)

⟨Reset the sentence finite state machine 5⟩ ≡

```
sfsm_source_file = NULL;
sfsm_inside_rule_mode = FALSE;
sfsm_skipping_material_at_level = -1;
sfsm_extension = from_extension;
if (from_extension) sfsm_extension_position = 1;
else sfsm_extension_position = 0;
```

This code is used in §4.

§**6.** A table is any sentence beginning with the word "Table". (Bad news for anyone writing "Table Mountain is a room.", of course, but there are other ways to do that, and it seems wise to keep the syntax for tables clear, since their entries are governed by different lexical and semantic rules even at the lowest levels of NI.)

⟨Go into table sentence mode if necessary 6⟩ ≡

```
if [[sentence_start, w2 == table ...]] sfsm_in_table_mode = TRUE;
else sfsm_in_table_mode = FALSE;
```

This code is used in §4.

§**7.** We now come to the definition of a sentence break, which is more complicated than might have been expected.

For one thing, a run of sentence divisions is treated as a single division, only the last of which is the one which counts. This looks odd at first sight, because it means that Inform considers

> The cat is on the table;.

to be a valid sentence, equivalent to

> The cat is on the table.

But it has the advantage that it enables us to avoid being pointlessly strict over the punctuation which precedes a paragraph break. Some people like to write paragraphs like this:

```
Before going north:
    say "Northward ho!";
    now the compass points north;
```

And properly speaking that ends with a semicolon then a paragraph break, which is a doubled sentence division. But we forgive it as harmless, and that forgiveness is provided by the loop arrangement below.

We also avoid the need for empty sentences, because it is not possible for the code below to detect them: thus

> say "Look behind you!";;;;; now the Wug is in the Cave

is broken as two sentences, not six sentences of which four are empty. Perhaps we ought to be stricter, and reject more of these dubious forms, but at this point we have too little understanding of the semantics of the text to risk annoying the user with problem messages.

§**8.** Full stops, semicolons and paragraph breaks (all rendered by the lexer as individual words: the stroke word in the case of the latter) are always sentence divisions. The other cases are more complicated: see below.

⟨Look for a sentence break, finding the number of stop words and the stop character 8⟩ ≡

```
int at = position;
no_stop_words = 0; stop_character = '?'; back_up_one_word = FALSE;
while (at < w2) {
    int stopped = FALSE;
    if [[word at == STROKE]] { stop_character = '|'; stopped = TRUE; }
    if [[word at == FULLSTOP]] { stop_character = '.'; stopped = TRUE; }
    if [[word at == SEMICOLON]] { stop_character = ';'; stopped = TRUE; }
    ⟨Consider if a colon divides a sentence 9⟩;
    ⟨Consider if punctuation within a preceding quoted text divides a sentence, making an X break 10⟩;
    if (stopped == FALSE) break;
    no_stop_words++; at++;
}
if (stop_character == 'X') {            X breaks are like full stops, but there is no stop word to skip over
    stop_character = '.'; back_up_one_word = TRUE;
}
if (no_stop_words > 0)
    LOGIF(LEXER, "Stop character '%c', no_stop_words %d, sentence_break %d, position %d\n",
        stop_character, no_stop_words, sentence_start, position);
```

This code is used in §4.

§**9.** Colons are normally dividers, too, but an exception is made if they come between two apparently numerical constructions, because this suggests that the colon is being used not as punctuation but within a literal pattern. (For instance, "He went out at 1:34 PM." is a sentence with just one clause, not two clauses divided by the colon; but "He went out at 1 PM: the snow was still falling." is indeed divided. Our rule here correctly distinguishes these cases, and although it can be fooled by really contrived sentences – "He went out at 1: 22 Company, the Parachute Regiment, was marching." – it's robust enough in practice.)

Note that here we are at a word position which is strictly within the word range being sentence-broken, so that it is safe to examine both the word before and the word after the current position.

⟨Consider if a colon divides a sentence 9⟩ ≡

```
if ([[word at == COLON]] &&
    ((isdigit(*(lw_array[at-1].lw_rawtext)) == FALSE) ||
        (isdigit(*(lw_array[at+1].lw_rawtext))) == FALSE)) {
    stop_character = ':'; stopped = TRUE;
}
```

This code is used in §8.

§**10.** Inform authors habitually use the punctuation in quoted text to end sentences, just as other writers of English do. The text

"Look out!" The explosion shattered the calm of the hillside.

is certainly intended as two sentences, not one.

An exception is made for table declarations, because a table needs to be formed as one long sentence, and it clearly does not abide by the ordinary punctuation rules of English. The point is that in the random line of table entries...

"Of cabbages and kings." Walrus "Carroll"

...the full stop after "kings" has no significance: the semantics of the table would be no different if it were not there.

⟨Consider if punctuation within a preceding quoted text divides a sentence, making an X break 10⟩ ≡

```
if ((stopped == FALSE) &&                                   only look if we are not already at a division
    (no_stop_words == 0) &&                                 be sure not to elide two such texts in a row
    (sfsm_in_table_mode == FALSE) &&                        check that we are not scanning the body of a table
    (isupper(*(lw_array[at].lw_rawtext))) &&                and the current word begins with a capital letter
    (text_ending_sentence(at-1))) {                         and the preceding one was quoted text ending in punctuation
    stop_character = 'X'; stopped = TRUE;
}
```

This code is used in §8.

§**11. Making sentence nodes.** At this point we have established that `make_sentence_node` is called sequentially for every divided-off sentence in the original source text. But we need a little machinery to skip past sentences which are being excluded for one reason or another.

The design of Inform deliberately excludes conditional compilation in the traditional C sense of `#ifdef` and `#endif`. This takes us too far from what natural language would do, faced with the same basic issue. A book, or a government form, would more naturally have a heading making clear that the section beneath it is not universal in application. This is what Inform does, too: it parses a heading to decide whether to skip the material, and if so, the state `sfsm_skipping_material_at_level` is set to the level of the heading in question. We then skip all subsequent sentences until reaching the next heading of the same or higher status, or until reaching the "... ends here." sentence (if we are reading an extension), or until reaching the end of the text: whichever comes first.

```
int no_sentences_read = 0;

void make_sentence_node(int w1, int w2, char stop_character) {
    int heading_level = 0, start_of_paragraph = FALSE, begins_or_ends = 0;
    parse_node *new;

    no_sentences_read++;
    LOGIF(LEXER, "MSN(%d): %d,%d='$W' stop_character <%c>\n",
            no_sentences_read, w1, w2, w1, w2, stop_character);

    if (w2<w1) internal_error("empty sentence generated");
    vocab_identify_word_range(w1, w2);                      a precaution to catch any late unidentified text

    if ((w1 == 0) || (compare_word(w1-1, STROKE_V))) start_of_paragraph = TRUE;

    ⟨Detect a change of source file, and declare it as an implicit heading 12⟩;
    if (start_of_paragraph) ⟨Detect any explicit heading and set heading level 13⟩;
    ⟨Detect a begins here or ends here sentence 18⟩;

    if ((begins_or_ends == -1) ||
        ((heading_level > 0) && (heading_level <= sfsm_skipping_material_at_level)))
        sfsm_skipping_material_at_level = -1;
```

```
    if (sfsm_skipping_material_at_level >= 0) return;
    if (heading_level > 0) {
        ⟨Issue a problem message if the heading incorporates a line break 14⟩;
        ⟨Issue a problem message if the heading does not end with a line break 15⟩;
        ⟨Make a new HEADING node, possibly beginning to skip material 16⟩;
        return;
    }
    ⟨If this is the top of the source text and starts with quoted matter, make a BIBLIOGRAPHIC node and return 17⟩;
    ⟨Accept the new sentence as one or more nodes in the parse tree 21⟩;
}
```

The function make_sentence_node is called from 5/litp, 7/dti, 9/scene and 9/madj.

§**12.** For reasons gone into in the section on Headings below, a change of source file (e.g., when one extension has been read in and another begins) is declared as if it were a super-heading in the text.

⟨Detect a change of source file, and declare it as an implicit heading 12⟩ ≡

```
    if (lw_array[w1].lw_source.file_of_origin != sfsm_source_file) {
        parse_node *implicit_heading = new_node(HEADING_NT);
        implicit_heading->word_ref1 = w1;
        implicit_heading->word_ref2 = w2;
        graft(implicit_heading, tree_root);
        pn_annotate_int(implicit_heading, sentence_unparsed_ANNOT, FALSE);
        pn_annotate_int(implicit_heading, heading_level_ANNOT, 0);
        declare_heading(implicit_heading);
        sfsm_skipping_material_at_level = -1;
    }
    sfsm_source_file = lw_array[w1].lw_source.file_of_origin;
```

This code is used in §11.

§**13.** Sentences beginning with “book”, “volume”, “part”, “chapter” or “section”, which occur at the start of the source text or the start of a paragraph, and which do not contain line breaks, are construed as headings.

⟨Detect any explicit heading and set heading level 13⟩ ≡

```
    heading_level = 0;
    if [[w1, w2 == volume ...]]  heading_level = 1;
    if [[w1, w2 == book ...]]    heading_level = 2;
    if [[w1, w2 == part ...]]    heading_level = 3;
    if [[w1, w2 == chapter ...]] heading_level = 4;
    if [[w1, w2 == section ...]] heading_level = 5;
```

This code is used in §11.

§**14.** We have already looked to see if the sentence could be a heading, and set the variable `heading_level` to be its ranking in the hierarchy (with 1, for "volume", the highest). But we also want to check that the heading does not have a line break in, because this is almost certainly a mistake by the designer, and likely to be a difficult one to understand: so we should help out if we can. Such a problem is best recovered from by continuing regardless.

⟨Issue a problem message if the heading incorporates a line break 14⟩ ≡

```
int k;
for (k=w1+1; k<=w2; k++)
    if ((lw_array[k].lw_break == '\n') || (isdigit(lw_array[k].lw_break))){
        quote_source(1, new_nounphrase_raw(w1, w2));
        quote_source(2, new_nounphrase_raw(w1, k-1));
        quote_source(3, new_nounphrase_raw(k, w2));
        handmade_problem(_P_(C4HeadingOverLine));
        issue_problem_segment(
            "The text %1 seems to be a heading, but contains a "
            "line break, which is not allowed: so I am reading it "
            "as just %2 and ignoring the continuation %3. The rule "
            "is that a heading must be a single line which is the "
            "only sentence in its paragraph, so there must be a "
            "skipped line above and below.");
        issue_problem_end();
        break;
    }
```

This code is used in §11.

§**15.** And similarly... Here we take the liberty of looking a little ahead of the current word range in order to make the problem message more helpful: we check that we are still looking at valid words in the lexer, just to be on the safe side, but in fact we cannot run on past the end of the lexer feed which fed the malformed heading, because of all of the run-off newlines automatically added at the end of the feed of any source file.

⟨Issue a problem message if the heading does not end with a line break 15⟩ ≡

```
if (lw_array[w2+1].lw_break != '\n') {
    int k;
    for (k = w2+1;
        (k<=w2+8) && (k<lexer_wordcount) && (lw_array[k].lw_break != '\n');
        k++) ;
    quote_source(1, new_nounphrase_raw(w1, w2));
    quote_source(2, new_nounphrase_raw(w2+1, k-1));
    handmade_problem(_P_(C4HeadingStopsBeforeEndOfLine));
    issue_problem_segment(
        "The text %1 seems to be a heading, but does not occupy "
        "the whole of its line of source text, which continues %2. "
        "The rule is that a heading must occupy a whole single line "
        "which is the only sentence in its paragraph, so there "
        "must be a skipped line above and below, and it must not "
        "contain any full stop characters '.', even if they occur "
        "in an ellipsis '...' or a number '2.3.13'. (I mention this "
        "because sometimes this problem arises when a decimal point is "
        "misread as a full stop.)");
    issue_problem_end();
}
```

This code is used in §11.

§**16.** We now have a genuine heading, and can declare it, calling a routine in Headings to determine whether we should include the material.

⟨Make a new HEADING node, possibly beginning to skip material 16⟩ ≡

```
heading *h;
new = new_node(HEADING_NT);
new->word_ref1 = w1;
new->word_ref2 = w2;
graft(new, tree_root);
pn_annotate_int(new, sentence_unparsed_ANNOT, FALSE);
pn_annotate_int(new, heading_level_ANNOT, heading_level);
h = declare_heading(new);
if (hd_include_material(h) == FALSE)
    sfsm_skipping_material_at_level = heading_level;
return;
```

This code is used in §11.

§**17.** The second sentence in the source text is construed as containing bibliographic data if it begins with a quoted piece of text, perhaps with substitutions. For instance,

"A Dream of Fair to Middling Women" by Samuel Beckett

might be a `BIBLIOGRAPHIC_NT` sentence. (The reason this is the second sentence read, not the first, is that all source texts read by NI implicitly begin with an inclusion of the Standard Rules.)

⟨If this is the top of the source text and starts with quoted matter, make a BIBLIOGRAPHIC node and return 17⟩ ≡

```
if ((no_sentences_read == 2) && (vocab_test_flags(w1, TEXT_MC+TEXTWITHSUBS_MC))) {
    new = new_node(BIBLIOGRAPHIC_NT);
    new->word_ref1 = w1; new->word_ref2 = w2;
    pn_annotate_int(new, sentence_unparsed_ANNOT, FALSE);
    graft(new, tree_root);
    return;
}
```

This code is used in §11.

§**18. Begins and ends here.** The final kind of pseudo-heading we shall consider are the sentences ending "begins here" or "ends here" in an extension, and which mark its start and end.

⟨Detect a begins here or ends here sentence 18⟩ ≡

```
begins_or_ends = 0;
if (sfsm_extension_position > 0) {                    i.e., if this is an extension at all
    if [[w1, w2 == ... begin/begins here]] {
        switch (sfsm_extension_position) {
            case 1: sfsm_extension_position++; break;
            case 2: extension_problem(_P_(C4ExtMultipleBeginsHere),
                sfsm_extension, "has more than one 'begins here' sentence"); break;
            case 3: extension_problem(_P_(C4ExtBeginsAfterEndsHere),
                sfsm_extension, "has a further 'begins here' after an 'ends here'"); break;
        }
        begins_or_ends = 1;
    }
    if [[w1, w2 == ... end/ends here]] {
        switch (sfsm_extension_position) {
            case 1: extension_problem(_P_(C4ExtEndsWithoutBegins),
                sfsm_extension, "has an 'ends here' with nothing having begun"); break;
            case 2: sfsm_extension_position++; break;
            case 3: extension_problem(_P_(C4ExtMultipleEndsHere),
                sfsm_extension, "has more than one 'ends here' sentence"); break;
        }
        begins_or_ends = -1;
    }
}
```

This code is used in §11.

§**19.** When we finish scanning all the sentences in a given batch, and if they came from an extension, we need to make sure we saw both beginning and end:

⟨Issue a problem message if we are missing the begin and end here sentences 19⟩ ≡

```
switch (sfsm_extension_position) {
    case 1: extension_problem(_P_(C4ExtNoBeginsHere),
        sfsm_extension, "has no 'begins here' sentence"); break;
    case 2: extension_problem(_P_(C4ExtNoEndsHere),
        sfsm_extension, "has no 'ends here' sentence"); break;
    case 3: break;
}
```

This code is used in §4.

§**20. Unskipped material which is not a heading.** Each of the sentences which are to be included is given its own node on the parse tree, which for the time being is a direct child of the root. Sentences are classified by their node types, the main identification attached to each unit in the tree, and one purpose of this code is to sort them into three broad categories:

(a) **Sentences with primary verbs**, which will be subject to further parsing work later: these have node type `SENTENCE_NT`. Anything we cannot place into categories (b) or (c) below will go here.

(b) **Sentences making up rules**. These are sequences of sentences in which a preamble (ending with a colon, or in certain cases a comma) of node type `ROUTINE_NT` is followed by a sequence of phrases (ending with semicolons until the last, which ends with a full stop or paragraph break), each of node type `COMMAND_NT`. For instance, the following produces three nodes:

To look upwards: say "Look out!"; something else.

(c) **Structural sentences**. These demarcate the text, call for other text or unusual matter to be included, etc.: the types in question are `TRACE_NT`, `HEADING_NT`, `INCLUDE_NT`, `INFORM6CODE_NT`, `BEGINHERE_NT`, `ENDHERE_NT`, `TABLE_NT`, `EQUATION_NT` and `BIBLIOGRAPHIC_NT`.

§**21.** And here is how we handle each new sentence to be added. If we do end up with a SENTENCE, which is how a typical assertion ends up, then it will initially not have its primary verb identified. If we are still in the early stages of NI's run, we simply mark it as unparsed for later. But if we have already passed the stage at which the identification of primary verbs normally happens, then we cannot defer the point: we must look for the verb straight away.

```
⟨Accept the new sentence as one or more nodes in the parse tree 21⟩ ≡
    ⟨Convert comma-divided rule into two sentences, if this is allowed 22⟩;
    ⟨Otherwise, make a SENTENCE node 24⟩;
    ⟨Convert a rule preamble to a ROUTINE node and enter rule mode 25⟩;
    if (sfsm_inside_rule_mode)
        ⟨Convert to a COMMAND node and exit rule mode unless a semicolon implies further commands 26⟩;
    at this point we are certainly in assertion mode, not rule mode
    ⟨Convert an asterisk sentence to a TRACE node and return 27⟩;
    if (start_of_paragraph) {
        ⟨Convert a Table paragraph to a TABLE node and return 28⟩;
        ⟨Convert an Equation paragraph to an EQUATION node and return 29⟩;
    }
    ⟨Convert an extension Include sentence to an INCLUDE node and return 30⟩;
    ⟨Convert a verbatim I6 Include sentence to an INFORM6CODE node and return 31⟩;
    ⟨Convert a begins here or ends here sentence to a BEGINHERE or ENDHERE node and return 32⟩;
    none of that happened, so we have a SENTENCE node for certain
    if (text_loaded_from_source) look_for_verb_phrase(new);
    else pn_annotate_int(new, sentence_unparsed_ANNOT, TRUE);
```

This code is used in §11.

§**22.** Rules are ordinarily noticed by the colon, which divides the header from the rest: colons are not otherwise legal in NI. However, we allow five exceptions, provided that we are outside a rule at the moment. (If we were inside a rule, the stop character would be a semicolon or colon.)

⟨Convert comma-divided rule into two sentences, if this is allowed 22⟩ ≡

```
if ((sfsm_inside_rule_mode == FALSE)
    && ((stop_character == '.') || (stop_character == '|'))
    && ([[w1, w2 == instead of ...]] ||
        [[w1, w2 == every turn ...]] ||
        [[w1, w2 == before ...]] ||
        [[w1, w2 == after ...]] ||
        [[w1, w2 == when ...]]))
    ⟨Look for a comma and split the sentence at it, unless in serial list 23⟩;
```

This code is used in §21.

§**23.** In such sentences a comma is read as if it were a colon. (The text up to the comma will then be given a `ROUTINE_NT` node and the text beyond the comma will make a `COMMAND_NT` node.)

Note that we make an exception for the serial comma used in a list of alternatives: thus the comma in "Aeschylus, Sophocles, or Euripides" does not trigger this rule. We need this exception because such lists of alternatives often occur in rule preambles, where it's the third comma which divides rule from preamble:

Instead of pushing, dropping, or taking the talisman, say "It is cursed."

⟨Look for a comma and split the sentence at it, unless in serial list 23⟩ ≡

```
int comma_pos, earliest_comma_position = is_word_intermediate(or_V, w1, w2);
if (earliest_comma_position < 0) earliest_comma_position = w1;
comma_pos = is_word_intermediate(COMMA_V, earliest_comma_position, w2);
if (comma_pos >= 0) {
    make_sentence_node(w1, comma_pos-1, ':');                    rule preamble stopped with a colon
    make_sentence_node(comma_pos+1, w2, '.');          rule body with one sentence, stopped with a stop
    return;
}
```

This code is used in §22.

§**24.** At this point we know that the text `w1` to `w2` will make one and only one sentence node in the parse tree, so we may as well create and graft it now. There are a number of special cases with variant node types, but the commonest outcome is a SENTENCE node, so that's what we shall assume for now.

⟨Otherwise, make a SENTENCE node 24⟩ ≡

```
new = new_node(SENTENCE_NT);
new->word_ref1 = w1;
new->word_ref2 = w2;
pn_annotate_int(new, sentence_unparsed_ANNOT, FALSE);
graft(new, tree_root);
```

This code is used in §21.

§**25.** Rules are sequences of phrases with a preamble in front, which we detect by its terminating colon. For instance:

To look upwards: say "Look out!"; something else.

(which arrives at this routine as three separate "sentences") will produce nodes with type `ROUTINE_NT`, `COMMAND_NT` and `COMMAND_NT` respectively.

This paragraph of code might look as if it should only be used in assertion mode, not in rule mode, because how can a rule preamble legally occur in the middle of another rule? But in fact it can, in two ways. One is the officially sanctioned way to make a definition with a complex phrase:

Definition: a supporter is wobbly: if the player is on it, decide yes; decide no.

This produces four nodes: `ROUTINE_NT`, `ROUTINE_NT`, `COMMAND_NT` and `COMMAND_NT` respectively.

The other arises somewhat less officially when people treat phrases as if they were C (or Inform 6) statements, always to be terminated with semicolons, and also run two rules together with no skipped paragraph between:

```
To do one thing: something here;
To do another thing: something else here;
```

A strict reading of our rules would oblige us to consider "To do another thing:" as a phrase within the definition of "To do one thing", and we would then have to issue a problem message. But this would be pettifogging. (People who habitually shuffle phrases about in their editors tend not to want to fuss about changing the punctuation of the last to a full stop instead of a semicolon. We may lament this, but it is so.)

⟨Convert a rule preamble to a ROUTINE node and enter rule mode 25⟩ ≡

```
if (stop_character == ':') {
    if ((sfsm_inside_rule_mode) && (detect_control_structure(w1, w2))) {
        pn_set_node_type(new, COMMAND_NT);
        pn_annotate_int(new, colon_block_command_ANNOT, TRUE);
        sfsm_inside_rule_mode = TRUE;
        return;
    } else {
        pn_set_node_type(new, ROUTINE_NT);
        sfsm_inside_rule_mode = TRUE;
        return;
    }
}
```

This code is used in §21.

§**26.** Subsequent commands are divided by semicolons, and any failure of a semicolon to appear indicates an end of the rule.

⟨Convert to a COMMAND node and exit rule mode unless a semicolon implies further commands 26⟩ ≡

```
pn_set_node_type(new, COMMAND_NT);
if (stop_character != ';') sfsm_inside_rule_mode = FALSE;
return;
```

This code is used in §21.

§**27.** A sentence consisting only of a single asterisk toggles whether sentences should be traced to the special sentence tracing file: such a node has type `TRACE_NT` and is otherwise ignored. A sentence of an asterisk followed by a double-quoted text is a note for the benefit of the telemetry file (again, a convenience for testing Inform).

⟨Convert an asterisk sentence to a TRACE node and return 27⟩ ≡
```
if ([[w1, w2 == ASTERISK]] ||
    ([[w1, w2 == ASTERISK ###]] && (vocab_test_flags(w2, TEXT_MC)))) {
    pn_set_node_type(new, TRACE_NT); return;
}
```

This code is used in §21.

§**28.** Sentences beginning with "table" which occur at the start of the source text or the start of a paragraph are construed as table declarations, and given type `TABLE_NT`.

⟨Convert a Table paragraph to a TABLE node and return 28⟩ ≡
```
if [[w1, w2 == table ...]] {
    pn_set_node_type(new, TABLE_NT); return;
}
```

This code is used in §21.

§**29.** Sentences beginning with "equation" which occur at the start of the source text or the start of a paragraph are similarly given type `EQUATION_NT`.

⟨Convert an Equation paragraph to an EQUATION node and return 29⟩ ≡
```
if [[w1, w2 == equation ...]] {
    pn_set_node_type(new, EQUATION_NT); return;
}
```

This code is used in §21.

§**30.** Sentences in the form "include ... by ..." are construed as requiring extensions to be loaded, and typed `INCLUDE_NT`.

⟨Convert an extension Include sentence to an INCLUDE node and return 30⟩ ≡
```
if [[w1, w2 == include ... by ...]] {
    pn_set_node_type(new, INCLUDE_NT); return;
}
```

This code is used in §21.

§**31.** Sentences in the form "Include (- ..." are construed as verbatim I6 code inclusions, and typed `INFORM6CODE_NT`.

⟨Convert a verbatim I6 Include sentence to an INFORM6CODE node and return 31⟩ ≡
```
if [[w1, w2 == include OPENI6 ...]] {
    pn_set_node_type(new, INFORM6CODE_NT); return;
}
```

This code is used in §21.

§**32.** Finally, we must tidy away the previously detected "begins here" and "ends here" sentences into nodes on the tree.

⟨Convert a begins here or ends here sentence to a BEGINHERE or ENDHERE node and return 32⟩ ≡

```
if (begins_or_ends == 1) {
    pn_set_node_type(new, BEGINHERE_NT);
    new->word_ref1 = w1;
    new->word_ref2 = w2 - 2;
    check_extension_begins_here(new, sfsm_extension);
    return;
}
if (begins_or_ends == -1) {
    new->word_ref1 = w1;
    new->word_ref2 = w2 - 2;
    pn_set_node_type(new, ENDHERE_NT);
    check_extension_ends_here(new, sfsm_extension);
    return;
}
```

This code is used in §21.

Extension Files 4/ext

Purpose

To keep details of the extensions currently loaded, their authors, titles, versions and rubrics, and to index and credit them suitably.

Template interpreter commands

```
{-callv:check_extension_versions}
{-routine:ShowExtensionVersions}
{-callv:index_extensions}
{-callv:handle_census_mode}
{-callv:update_census_of_extensions}
```

Definitions

¶1. Extensions are files of source text, normally combined with an appendix of documentation, provided by third parties for users to include with the source text of their own projects. Extensions are intended to provide general solutions to typical needs, and an archive of extensions for download is provided by the Inform website: those are licenced by their authors with a Creative Commons Attribution licence, and because of that, Inform behaves as if all extensions it sees require attribution (unless the author requests anonymity). A user of Inform 7 will certainly have nine or ten extensions available to him, because that many are built into the standard distribution: as of 2007 it is typical for users to have installed 50 to 90 others, and the total is steadily rising. Our routines here therefore need to be scalable.

Extensions are stored in two places: a built-in area, inside the Inform 7 application, and an external area. NI knows the location of the former because the application passes this as `-rules` on the command line, and it knows the location of the latter because this is standard and fixed for each platform: see Platform-Specific Definitions.

¶2. An extension has a title and an author name, each of which is limited in length to one character less than the following constants:

```
define MAX_EXTENSION_TITLE_LENGTH 51
define MAX_EXTENSION_AUTHOR_LENGTH 51
```

¶3. How the application should install extensions. When the Inform 7 application looks at a file chosen by the user to be installed, it should look at the first line. (Note that this might have any of `0a`, `0d`, `0a0d`, `0d0a`, or Unicode line division as its line ending: and that the file might, or might not, begin with a Unicode BOM, "byte order marker", code. Characters within the line will be encoded as UTF-8, though – except possibly for some exotic forms of space – they will all be found in the ISO Latin-1 set.) The first line is required to have one of the following forms, possibly with white space before or after, but definitely without line breaks before:

Locksmith Extra by Emily Short begins here.

Version 2 of Locksmith Extra by Emily Short begins here.

Version 060430 of Locksmith Extra by Emily Short begins here.

Version 2/060430 of Locksmith Extra by Emily Short begins here.

If the name of the extension finishes with a bracketed clause, that should be disregarded. Such clauses are used to specify virtual machine requirements, at present, and could conceivably be used for other purposes later, so let's reserve them now.

Version 2 of Glulx Text Effects (for Glulx only) by Emily Short begins here.

The application should reject (that is, politely refuse to install) any purported extension file whose first line does not conform to the above.

Ignoring any version number given, the Inform application should then store the file in the external extensions area. For instance,

`~/Library/Inform/Extensions/Emily Short/Glulx Text Effects` (OS X)
`My Documents\Inform\Extensions\Emily Short\Glulx Text Effects` (Windows)

Note that the file will probably not have the right name initially, and will need to be renamed as well as moved. (Note the lack of a file extension.) The subfolders `Inform`, `Extensions` and `Emily Short` must be created if not already present.

If to install such an extension would result in over-writing an extension already present at that filename, the user should be given a warning and asked if he wants to proceed.

However, note that it is *not* an error to install an extension with the same name and author as one in the built-in extensions folder. This does not result in overwriting, since the newly installed version will live in the external area, not the built-in area.

An extension may be uninstalled simply by deleting the file: but the application must not allow the user to uninstall any extension from the built-in area. We must assume that the latter could be on a read-only disc, or could be part of a cryptographically signed application bundle.

¶4. The extension census. The Inform application should run NI in "census mode" in order to keep extension documentation up to date. NI should be run in census mode on three occasions:

(a) when the Inform application starts up;
(b) when the Inform application installs a new extension;
(c) when the Inform application uninstalls an extension.

When NI is run in "census mode", it should be run with the command

```
ni -rules (...) -census
```

where the argument for `-rules` is the same as for any other run. All output from NI should be ignored, including its return code: ideally, not even a fatal error should provoke a reaction from the application. If the census doesn't work for some file-system reason, never mind – it's not mission-critical.

¶5. What happens in census mode. The census has two purposes: first, to create provisional documentation where needed for new and unused extensions; and second, to create the following index files in the external documentation area (*not* in the external extension area):

`.../Extensions.html` (basically a contents page)
`.../ExtIndex.html` (basically an index)

Documentation for any individual extension is stored at, e.g.,

`.../Extensions/Victoria Saxe-Coburg-Gotha/Werewolves.html`

NI can generate such a file, for an individual extension, in two ways: (a) provisionally, with much less detail, and (b) fully. Whenever it successfully compiles a work using extension X, it rewrites the documentation for X fully, and updates both the two indexing pages.

When NI runs in `-census` mode, what it does is to scan for all extensions. If NI finds a valid extension with no documentation page, it writes a provisional one; and again, it updates both the two indexing pages.

(NI in fact runs a census on every compilation, as well, so `-census` runs do nothing "extra" that a normal run of NI does not also do. On every census, NI automatically checks for misfiled or broken extensions, and places a descriptive report of what's wrong on the `Extensions.html` index page – if people move around or edit extensions by hand, they may run into these errors.)

¶6. With that general discussion out of the way, we can get on with implementation. A modest structure is used to store details of extension files loaded into NI: or rather, to store requests to include them, and then to keep track of the results.

The rubric of an extension is text found near its opening, describing its purpose. This is truncated to the following length, which must remain strictly less than `MAX_STRING_LENGTH`:

```
define MAX_RUBRIC_LENGTH 500

typedef struct extension_file {
    struct extension_identifier ef_id;                    Texts of title and author with hash code
    int author_w1, author_w2;                                          Author's name
    int name_w1, name_w2;                                              Extension name
    int body_w1, body_w2;                          Body of source text supplied in extension, if any
    int body_text_unbroken;                   Does this contain text waiting to be sentence-broken?
    int doc_w1, doc_w2;                            Documentation supplied in extension, if any
    int VM_restriction_w1, VM_restriction_w2;                  Restricting use to certain VMs
    int min_version_needed;                                    As stipulated by source
    int version_loaded;                                        As actually loaded
    int footnote_number;                          In the index, which footnote refers to this
    int loaded_from_built_in_area;                             Located within Inform application
    int authorial_modesty;                                     Do not credit in the compiled game
    struct parse_node *inclusion_sentence;                     Where the source called for this
    struct source_file *read_into_file;                        Which source file loaded this
    char rubric_as_lexed[MAX_RUBRIC_LENGTH+1];
    char extra_credit_as_lexed[MAX_RUBRIC_LENGTH+1];
    MEMORY_MANAGEMENT
} extension_file;

extension_file *standard_rules_extension;              the Standard Rules by Graham Nelson
```

The structure extension_file is shared with 4/iext, 4/edict and 4/edoc.

§**1.** We begin with some housekeeping, really: the code required to create new extension file structures, and to manage existing ones.

```
extension_file *new_extension_file(int author_w1, int author_w2,
    int nw1, int nw2, int vm1, int vm2, int version_word) {
    char violation[160];                                          enough for two lines of type
    violation[0] = 0;
    extension_file *ef = CREATE(extension_file);
    ef->author_w1 = author_w1; ef->author_w2 = author_w2;
    ef->name_w1 = nw1; ef->name_w2 = nw2;
    ⟨Create EID for new extension file 2⟩;
    ef->min_version_needed = version_word;
    ef->inclusion_sentence = current_sentence;
    ef->VM_restriction_w1 = vm1; ef->VM_restriction_w2 = vm2;
    ef->body_w1 = -1; ef->body_w2 = -1;
    ef->body_text_unbroken = FALSE;
    ef->doc_w1 = -1; ef->doc_w2 = -1;
    ef->version_loaded = -1;
    ef->loaded_from_built_in_area = FALSE;
    ef->authorial_modesty = FALSE;
    ef->rubric_as_lexed[0] = 0;
    ef->extra_credit_as_lexed[0] = 0;
    if (violation[0]) extension_problem(_P_(Untestable), ef, violation);          see below
    return ef;
}
```

The function new_extension_file is called from 4/iext.

§**2.** We protect ourselves a little against absurdly long requested author or title names, and then produce problem messages in the event of only longish ones, unless the census is going on: in which case it's better to leave the matter to the census errors system elsewhere.

⟨Create EID for new extension file 2⟩ ≡

```
    char exft[MAX_FILENAME_LENGTH], exfa[MAX_FILENAME_LENGTH];
    print_raw_text_to_string_truncated(ef->author_w1, ef->author_w2, exfa, MAX_FILENAME_LENGTH);
    print_raw_text_to_string_truncated(ef->name_w1, ef->name_w2, exft, MAX_FILENAME_LENGTH);
    if (currently_recording_census_errors() == FALSE) {
        if (strlen(exfa) >= MAX_EXTENSION_AUTHOR_LENGTH)                  intest cannot easily test this
            sprintf(violation,
                "has an author's name which is too long, exceeding the maximum "
                "allowed (%d characters) by %d",
                MAX_EXTENSION_AUTHOR_LENGTH-1,
                (int) (1+strlen(exfa)-MAX_EXTENSION_AUTHOR_LENGTH));
        if (strlen(exft) >= MAX_EXTENSION_AUTHOR_LENGTH)                  intest cannot easily test this
            sprintf(violation,
                "has a title which is too long, exceeding the maximum allowed "
                "(%d characters) by %d",
                MAX_EXTENSION_TITLE_LENGTH-1,
                (int) (1+strlen(exft)-MAX_EXTENSION_TITLE_LENGTH));
    }
    eid_new(&(ef->ef_id), exfa, exft, LOADED_EIDBC);
    if (eid_is_standard_rules(&(ef->ef_id))) standard_rules_extension = ef;
```

This code is used in §1.

§**3.** Three pieces of information (not available when the EF is created) will be set later on, by other parts of NI calling the routines below.

The rubric text for an extension, which is double-quoted matter just below its "begins here" line, is parsed as a sentence and will be read as an assertion in the usual way when the material from this extension is being worked through (quite a long time after the EF structure was created). When that happens, the following routine will be called to set the rubric; and the one after for the optional extra credit line, used to acknowledge I6 sources, collaborators, translators and so on.

```
void ef_set_rubric(extension_file *ef, char *text) {
    truncated_strcpy(ef->rubric_as_lexed, text, MAX_RUBRIC_LENGTH);
    LOGIF(EXTCENSUS, "Extension rubric: %s\n", ef->rubric_as_lexed);
}
void ef_set_extra_credit(extension_file *ef, char *text) {
    truncated_strcpy(ef->extra_credit_as_lexed, text, MAX_RUBRIC_LENGTH);
    LOGIF(EXTCENSUS, "Extension extra credit: %s\n", ef->extra_credit_as_lexed);
}
```

The function ef_set_rubric is called from 8/sob.
The function ef_set_extra_credit is called from 8/sob.

§**4.** Once we start reading text from the file (if it can successfully be found: when the EF structure is created, we don't know that yet), we need to tally it up with the corresponding source file structure.

```
void ef_set_corresponding_source_file(extension_file *ef, source_file *sf) {
    ef->read_into_file = sf;
}
```

The function ef_set_corresponding_source_file is called from 3/read.

§**5.** When headings cross-refer to extensions, they need to read extension IDs, so:

```
extension_identifier *ef_get_eid(extension_file *ef) {
    return &(ef->ef_id);
}
```

The function ef_get_eid is called from 2/index, 2/prob1, 4/head and 12/rb.

§**6.** The use option "authorial modesty" is unusual in applying to the extension it is found in, not the whole source text. When we read it, we call one of the following routines, depending on whether it was in an extension or in the main source text:

```
int general_authorial_modesty = FALSE;
void ef_set_authorial_modesty(extension_file *ef) { ef->authorial_modesty = TRUE; }
void set_general_authorial_modesty(void) { general_authorial_modesty = TRUE; }
```

The function ef_set_authorial_modesty is called from 10/isin.
The function set_general_authorial_modesty is called from 10/isin.

§**7. Index footnotes.** These are used only in indexing, to show that a given meaning is defined by an extension rather than by the source text.

Before compiling each index, we clear the current footnote numbers:

```
int no_extensions_footnoted_in_current_index = 0;
int ef_no_of_footnoted_extensions(void) {
    return no_extensions_footnoted_in_current_index;
}
void ef_clear_extension_footnotes(void) {
    extension_file *ef;
    no_extensions_footnoted_in_current_index = 0;
    LOOP_OVER(ef, extension_file) ef->footnote_number = 0;
}
```

The function ef_no_of_footnoted_extensions is called from 2/index.

The function ef_clear_extension_footnotes is called from 2/index.

§**8.** When and as needed, each EF is allocated the next available positive integer as its footnote number. Thus the first to be footnoted on a given index page becomes footnote 1, and so on.

```
int ef_get_footnote(extension_file *ef) {
    if (ef->footnote_number == 0)
        ef->footnote_number = ++no_extensions_footnoted_in_current_index;
    return ef->footnote_number;
}
```

The function ef_get_footnote is called from 2/index.

§**9.** Lastly, we recover the EF corresponding to a given footnote number thus. The routine returns `NULL` if the EF is not footnoted on the current index page so far.

```
extension_file *ef_by_footnote(int n) {
    extension_file *ef;
    LOOP_OVER(ef, extension_file)
        if (ef->footnote_number == n) return ef;
    return NULL;
}
```

The function ef_by_footnote is called from 2/index.

§**10. Printing names of extensions.** We can print the name of the extension in a variety of ways and to a variety of destinations, but it all comes down to the same thing in the end. First, printing the name to an arbitrary UTF-8 file:

```
void ef_write_name_to_file(extension_file *ef, OUTPUT_STREAM) {
    print_raw_text_to_file(ef->name_w1, ef->name_w2, OUT);
}
void ef_write_author_to_file(extension_file *ef, OUTPUT_STREAM) {
    print_raw_text_to_file(ef->author_w1, ef->author_w2, OUT);
}
void ef_write_full_title_to_file(extension_file *ef, OUTPUT_STREAM) {
    ef_write_name_to_file(ef, OUT);
    INDEX(" by ");
    ef_write_author_to_file(ef, OUT);
}
```

The function ef_write_name_to_file is called from 11/act and 12/phin.
The function ef_write_author_to_file is called from 11/act and 12/phin.
The function ef_write_full_title_to_file is called from 2/index.

§**11.** Next, the debugging log:

```
void log_extension_file(extension_file *ef) {
    if (ef == NULL) { LOG("<null-extension-file>"); return; }
    LOG("$W by $W", ef->name_w1, ef->name_w2, ef->author_w1, ef->author_w2);
}
```

The function log_extension_file is called from 2/dl.

§**12.** Next, printing the name in the form of a comment in an (ISO Latin-1) Inform 6 source file:

```
void ef_write_I6_comment_describing(extension_file *ef, OUTPUT_STREAM) {
    if (ef == standard_rules_extension) {
        WRITE("! From the Standard Rules\n");
    } else {
        WRITE("! From \"");
        print_raw_text_within_i6_literal(OUT, ef->name_w1, ef->name_w2);
        WRITE("\" by ");
        print_raw_text_within_i6_literal(OUT, ef->author_w1, ef->author_w2);
        WRITE("\n");
    }
}
```

The function ef_write_I6_comment_describing is called from 12/cph.

§**13.** And finally printing the name to a C string:

```
void ef_write_full_title_to_string(extension_file *ef, char *p) {
    print_raw_text_to_string(ef->name_w1, ef->name_w2, p);
    sprintf(p + strlen(p), " by ");
    print_raw_text_to_string(ef->author_w1, ef->author_w2, p + strlen(p));
}
```

The function ef_write_full_title_to_string is called from 2/prob2.

§14. Checking version numbers. It's only at the end of semantic analysis, when all extensions have been loaded, that we check that all the version numbers are sufficient to meet the requests made. The reason we don't do this one at a time, as we load them in, is that we might load E at a time when version V is required, and find that it matches; but then an extension loaded later might turn out to require E version $V + 1$. So it is only when all extensions have been loaded that we know the full set of requirements, and only then do we check that they have been met.

```
void check_extension_versions(void) {
    extension_file *ef;
    LOOP_OVER(ef, extension_file) {
        int have = parse_version_number(ef->version_loaded),
            need = parse_version_number(ef->min_version_needed);
        if (need > have) {
            LOG("Need %d, have %d\n", need, have);
            current_sentence = ef->inclusion_sentence;
            quote_source(1, current_sentence);
            quote_extension(2, ef);
            if (ef->version_loaded >= 0) {
                quote_words(3, ef->version_loaded, ef->version_loaded);
                handmade_problem(_P_(C4ExtVersionTooLow));
                issue_problem_segment(
                    "You wrote %1: but my copy of %2 is only version %3.");
                issue_problem_end();
            } else {
                handmade_problem(_P_(C4ExtNoVersion));
                issue_problem_segment(
                    "You wrote %1: but my copy of %2 contains no version "
                    "number, and is therefore considered to be earlier than "
                    "all numbered versions.");
                issue_problem_end();
            }
        }
    }
}
```

The function check_extension_versions is invoked by a command in a `.i6t` template file.

§15. Credit for extensions. Here we compile an I6 routine to print out credits for all the extensions present in the compiled work. This is important because the extensions published at the Inform website are available under a Creative Commons license which requires users to give credit to the authors: Inform ensures that this happens automatically.

Use of authorial modesty (see above) will suppress a credit in the `ShowExtensionVersions` routine, but the system is set up so that one can only be modest about one's own extensions: this would otherwise violate a CC license of somebody else. General authorial modesty thus suppresses credits for all extensions used which are by the user himself. On the other hand, if an extension contains an authorial modesty disclaimer in its own text, then that must have been the wish of its author, so we can suppress the credit whoever that author was.

In `I7FullExtensionVersions` all extensions are credited whatever anyone's feelings of modesty.

```
void compile_ShowExtensionVersions_routine(OUTPUT_STREAM) {
    extension_file *ef;
    WRITE("[ ShowExtensionVersions;\n"); INDENT;
    LOOP_OVER(ef, extension_file) {
```

```
        char the_author_name[512];
        print_raw_text_to_string(ef->author_w1, ef->author_w2, the_author_name);
        if ((ef->authorial_modesty == FALSE) &&                    if (1) extension doesn't ask to be modest
            ((general_authorial_modesty == FALSE) ||          and (2) author doesn't ask to be modest, or...
            (story_author_is(the_author_name) == FALSE)))                    ...didn't write this extension
                credit_ef(OUT, ef);                                                then we award a credit
    }
    OUTDENT; WRITE("];\n");
    WRITE("[ ShowFullExtensionVersions;\n"); INDENT;
    LOOP_OVER(ef, extension_file) credit_ef(OUT, ef);
    OUTDENT; WRITE("];\n");
}
```

The function compile_ShowExtensionVersions_routine is invoked by a command in a `.i6t` template file.

§**16.** The actual credit consists of a single line, with name, version number and author. These are printed as I6 strings, hence the ISO encoding.

```
void credit_ef(OUTPUT_STREAM, extension_file *ef) {
    WRITE("print \"%s", ef->ef_id.raw_title);
    if (ef->version_loaded >= 0) {
        WRITE(" version ");
        print_raw_text_to_file(ef->version_loaded, ef->version_loaded, OUT);
    }
    WRITE(" by %s", ef->ef_id.raw_author_name);
    if (ef->extra_credit_as_lexed[0]) WRITE(" (%s)", ef->extra_credit_as_lexed);
    WRITE("^\";\n");
}
```

§**17. Indexing extensions in the Contents index.** The routine below places a list of extensions used in the Contents index, giving only minimal entries about them.

```
void index_extensions(void) {
    extension_file *ef;
    index_anchor("EXTLIST");
    INDEX("<p><b>List of extensions included</b> <i>(About these ");
    index_doc_link("EXTENSIONS");
    INDEX(")</i><p>\n");
    LOOP_OVER(ef, extension_file) {
        print_raw_text_to_file(ef->name_w1, ef->name_w2, ifl);
        if (ef != standard_rules_extension) {              give author and inclusion links, but not for SR
            INDEX(" by ");
            print_raw_text_to_file(ef->author_w1, ef->author_w2, ifl);
            index_link(lw_array[ef->inclusion_sentence->word_ref1].lw_source);
        } else index_doc_link("SRULES");
        if (ef->version_loaded >= 0) {
            INDEX(" version ");
            print_raw_text_to_file(ef->version_loaded, ef->version_loaded, ifl);
        }
        if (ef->extra_credit_as_lexed[0]) {
            INDEX(" (");
            print_literal_string_to_file(ifl, ef->extra_credit_as_lexed);
            INDEX(")");
```

```
        }
        INDEX("<br>\n");
    }
}
```

The function index_extensions is invoked by a command in a `.i6t` template file.

§**18. Updating the documentation.** This is done in the course of taking an extension census, which is called for in one of two circumstances: when NI is being run in "census mode" to notify it that extensions have been installed or uninstalled; or when NI has completed the successful compilation of a source text. In the latter case, it knows quite a lot about the extensions actually used in that compilation, and so can write detailed versions of their documentation: since it is updating extension documentation anyway, it conducts a census as well. (In both cases the extension dictionary is also worked upon.) The two alternatives are expressed here:

```
void handle_census_mode(void) {
    if (census_mode) {
        load_extension_dictionary();
        extension_census();
        write_top_level_of_extensions_documentation();
        write_sketchy_documentation_for_extensions_found();
        abort_I6T_interpreter = TRUE;                our work is done: exit at the next graceful opportunity
    }
}

void update_census_of_extensions(void) {
    extension_file *ef;
    load_extension_dictionary();
    extension_census();
    write_top_level_of_extensions_documentation();
    LOOP_OVER(ef, extension_file) write_detailed_extension_documentation(ef);
    write_sketchy_documentation_for_extensions_found();
    write_dictionary_back();
    if (dl_this(EXTCENSUS_DA)) log_EID_hash_table();
}
```

The function handle_census_mode is invoked by a command in a `.i6t` template file.
The function update_census_of_extensions is invoked by a command in a `.i6t` template file.

§**19.** Documenting extensions seen but not used: we run through the census results in no particular order and create a sketchy page of documentation, if there's no better one already.

```
void write_sketchy_documentation_for_extensions_found(void) {
    extension_census_datum *ecd;
    LOOP_OVER(ecd, extension_census_datum)
        write_sketchy_extension_documentation(ecd);
}
```

§**20. Writing the extensions home pages.** Extensions documentation forms a mini-website within the Inform documentation. There is a top level consisting of two home pages: a directory of all installed extensions, and an index to the terms defined in those extensions. A cross-link switches between them. Each of these links down to the bottom level, where there is a page for every installed extension (wherever it is installed). The picture is therefore something like this:

```
(Main documentation contents page)
        |
Extensions.html--ExtIndex.html
        |      \/      |
        |      /\      |
Nigel Toad/Eggs  Barnabas Dundritch/Neopolitan Iced Cream   ...
```

§**21.** These pages are stored at the relative pathnames

```
Extensions/Documentation/Extensions.html
Extensions/Documentation/ExtIndex.html
```

They are made by inserting content in place of the material between the HTML anchors `on` and `off` in a template version of the page kept in the `Reserved` subfolder of the built-in extensions area (and therefore within the application), with a leafname which varies from platform to platform, for reasons as always to do with the vagaries of Internet Explorer 7 for Windows.

```
void write_top_level_of_extensions_documentation(void) {
    write_top_level_extensions_page("Extensions",
        "As extensions are installed, documentation on them is added to this "
        "page. Click on the name of an extension for more. See also the "
        "<a href=ExtIndex.html STYLE=\"text-decoration: none\"><font color=\"#4040FF\">"
        "index of extension definitions</font></a>.",
        1);
    write_top_level_extensions_page("ExtIndex",
        "Whenever an extension is used, its definitions are entered into the "
        "following index. (Thus, a newly installed but never-used extension "
        "is not indexed yet.) See also the "
        "<a href=Extensions.html STYLE=\"text-decoration: none\"><font color=\"#4040FF\">"
        "directory of installed extensions</font></a>.",
        2);
}
```

§**22.**

```
void write_top_level_extensions_page(char *leafname, char *explanatory_heading,
    int content) {
    STREAM HOMEPAGE_struct;
    STREAM *HOMEPAGE = &HOMEPAGE_struct;
    FILE *TEMPLATE;
    char path_to_census[MAX_FILENAME_LENGTH];
    char path_to_model[MAX_FILENAME_LENGTH];
    sprintf(path_to_census, "%s%c%s.html", pathname_of_extension_docs, FOLDER_SEPARATOR,
        leafname);
    if (STREAM_OPEN_TO_FILE(HOMEPAGE, path_to_census, UTF8_ENC) == FALSE)
        fatal_error2("Unable to open extensions documentation index for writing",
            path_to_census);
    sprintf(path_to_model, "%s%cReserved%c%s",
        pathname_of_built_in_extensions, FOLDER_SEPARATOR, FOLDER_SEPARATOR,
        EXTENSIONS_MODEL_HTML);
    TEMPLATE = iso_fopen(path_to_model, "r");
    if (TEMPLATE == NULL)
        fatal_error("Unable to open model extensions documentation for reading");
    ⟨Copy the template, inserting the census results at the appropriate place 23⟩;
    STREAM_CLOSE(HOMEPAGE);
    fclose(TEMPLATE);
}
```

§**23.** The template must be a valid HTML page encoded as UTF-8 whose lines are divided with Unix line endings and in which every line is shorter than the following number of characters in length:

```
define LONGEST_LINE_IN_EX_TEMPLATE 1024
define LINE_MARKING_START_OF_CENSUS "<a name=on>"
define LINE_MARKING_END_OF_CENSUS "<a name=off>"
```

⟨Copy the template, inserting the census results at the appropriate place 23⟩ ≡

```
    char line_of_template[LONGEST_LINE_IN_EX_TEMPLATE];
    int skipping_between_markers = FALSE;
    while (!(feof(TEMPLATE))) {
        int n = truncated_iso_fgets(TEMPLATE, line_of_template, LONGEST_LINE_IN_EX_TEMPLATE);
        if (n == -1) break;                                    file I/O error, or end of file
        if (strcmp(line_of_template, LINE_MARKING_START_OF_CENSUS) == 0) {
            STREAM_WRITE(HOMEPAGE, "<p>%s<p>", explanatory_heading);
            switch (content) {
                case 1: write_extensions_census_to_HTML_file(HOMEPAGE); break;
                case 2: write_extension_dictionary_to_HTML_file(HOMEPAGE); break;
            }
            skipping_between_markers = TRUE;
        }
        if (skipping_between_markers == FALSE) STREAM_WRITE(HOMEPAGE, "%s", line_of_template);
        if (strcmp(line_of_template, LINE_MARKING_END_OF_CENSUS) == 0)
            skipping_between_markers = FALSE;
    }
```

This code is used in §22.

§**24.** Note the cross-link to the extension dictionary, which is reciprocated from there.

⟨Print the headnote explaining what the census is 24⟩ ≡

```
STREAM_WRITE(HOMEPAGE,
    "<p>As extensions are installed, documentation on them is added to this "
    "page. Click on the name of an extension for more. See also the "
    "<a href=ExtIndex.html STYLE=\"text-decoration: none\"><font color=\"#4040FF\">"
    "index of extension definitions</font></a>.<p>");
```

This code is used in §.

Including Extensions 4/iext

Purpose

To fulfill requests to include extensions, adding their material to the parse tree as needed, and removing INCLUDE nodes.

4/iext.§6-10 Extension loading; §11-12 Parsing extension version numbers; §13-20 Checking the begins here and ends here sentences; §21 Sentence handlers for begins here and ends here

---

Template interpreter commands

```
1   {-callv:traverse_for_extensions}
```

---

§**1.** At this point in the narrative of a typical run of NI, we have read in the source text supplied by the user. The lexer automatically prefaced this with "Include Standard Rules by Graham Nelson", and the sentence-breaker converted all such sentences to nodes of type `INCLUDE_NT` which are children of the parse tree root. (The eldest child, therefore, is the Standard Rules inclusion.)

We now look through the parse tree in sentence order – something we shall do many times, and which we call a "traverse" – and look for INCLUDE nodes. Each is replaced with a mass of further nodes for the material in whatever new extensions were required. This process is repeated until there are no "Include" sentences left. In principle this could go on forever if A includes B which includes A, or some such, but we log each extension read in to ensure that nothing is read twice.

At the end of this routine, provided no Problems have been issued, there are guaranteed to be no INCLUDE nodes remaining in the parse tree.

```
void traverse_for_extensions(void) {
    int includes_cleared;
    do {
        parse_node *pn, *elder;
        includes_cleared = TRUE;
        if (problem_count > 0) return;
        for (elder=NULL, TREE_START(pn); pn; TREE_NEXT(pn)) {
            current_sentence = pn;                          to report problems at the right place
            if (pn_get_node_type(pn) == INCLUDE_NT) {
                ⟨Replace INCLUDE node with sentence nodes for any extensions required 2⟩;
                includes_cleared = FALSE;
                continue;
            }
            elder = pn;
        }
    } while (includes_cleared == FALSE);
}
```

The function traverse_for_extensions is invoked by a command in a `.i6t` template file.

§**2.** We slice out the INCLUDE node and everything after it, which we call the tail: that gives us a clean branch from which new included matter can grow. We then graft the tail back again. The net effect is that the INCLUDE node disappears, since it is never regrafted, while any new matter now appears where the INCLUDE node used to be. (It's possible for that new matter to be empty, if the inclusion request was for an extension which has already been included. If so then the process simply excises the INCLUDE and leaves the tree otherwise unchanged.)

⟨Replace INCLUDE node with sentence nodes for any extensions required 2⟩ ≡

```
    if (elder == NULL) tree_root->down = NULL; else elder->next = NULL;          trim off pn and the tail
    fulfill_request_to_include_extension(pn, pn->word_ref1+1, pn->word_ref2);       which may add new
nodes
    if (pn->next) elder = graft(pn->next, tree_root);          put the tail (not pn) back after the new nodes
```

This code is used in §1.

§**3.** Here we parse requests to include one or more extensions. People mostly don't avail themselves of the opportunity, but it is legal to include several at once, with a line like:

> Include Carrots by Peter Rabbit and Green Lettuce by Flopsy Bunny.

The problem message here can only be seen if the author tries to include multiple extensions in the same sentence, one of which has an author but another of which does not, since the presence of the word "by" is an essential marker in detecting an inclusion sentence. So the test case to provoke this problem is a little bit of a stretch.

A consequence of this convention is that "and" is not permitted in the name of an extension. We might change this some day.

```
void fulfill_request_to_include_extension(parse_node *incl, int w1, int w2) {
    int i, aw1, aw2;
    if ([[w1, w2 == ... by ... : i]] &&
        (is_list_divided(i, w2, LOOK_FOR_AND))) {
        int lw1 = w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        fulfill_request_to_include_extension(incl, lw1, lw2);
        fulfill_request_to_include_extension(incl, rw1, rw2);
        return;
    }
    if ([[w1, w2 == ... by ... : i --> w1, w2 ... aw1, aw2]] == FALSE) {
        current_sentence = incl;
        sentence_problem(_P_(C4IncludeExtWithoutBy),
            "the name of an included extension should have the form 'General "
            "Relativity by Albert Einstein'",
            "though real examples are usually more weighty. (The problem here "
            "is that you do not say who the extension is 'by'.)");
        return;
    }
    ⟨Fulfill request to include a single extension 4⟩;
}
```

§**4.** A request consists of author, name and version, the latter being optional. We obtain the extension file structure corresponding to this: it may have no text at all (for instance if NI could not open the file), or it may be one we have seen before, thanks to an earlier inclusion. Only when it provided genuinely new text will its `body_text_unbroken` flag be set, and then we call the sentence-breaker to graft the new material on to the parse tree.

⟨Fulfill request to include a single extension 4⟩ ≡

```
int version_word = -1;
extension_file *requested_extension;
if [[w1, w2 == version ### of ...]] {
    version_word = w1+1;
    parse_version_number(version_word);                this checks the formatting of the version number
    w1 += 3;
}
[[w1, w2 == the ... --> w1, w2]];
⟨Issue problem message if the extension title is written in double quotes 5⟩;
requested_extension = load_extension(aw1, aw2, w1, w2, version_word);
if (requested_extension->body_text_unbroken) {
    break_into_sentences(requested_extension->body_w1, requested_extension->body_w2,
        requested_extension);
    requested_extension->body_text_unbroken = FALSE;
}
```

This code is used in §3.

§**5.** Quite a popular mistake, this:

⟨Issue problem message if the extension title is written in double quotes 5⟩ ≡

```
if (is_kova(is_a_literal(w1, w2), TEXT_TY)) {
    sentence_problem(_P_(C4IncludeExtQuoted),
        "the name of an included extension should be given without double "
        "quotes in an Include sentence",
        "so for instance 'Include Oh My God by Janice Bing.' rather than "
        "'Include \"Oh My God\" by Janice Bing.')");
    return;
}
```

This code is used in §4.

§**6. Extension loading.** Extensions are loaded here.

```
extension_file *load_extension(int author_w1, int author_w2,
    int name_w1, int name_w2, int version_word) {
    int i, vm1 = -1, vm2 = -1;
    extension_file *ef;
    [[name_w1, name_w2 == ... OPENBRACKET ... CLOSEBRACKET : i --> name_w1, name_w2 ... vm1, vm2]];
    LOOP_OVER(ef, extension_file)
        if ((compare_word_range(ef->author_w1, ef->author_w2, author_w1, author_w2))
            && (compare_word_range(ef->name_w1, ef->name_w2, name_w1, name_w2)))
            ⟨This is an extension already loaded, so note any version number hike and return 7⟩;
    ef = new_extension_file(author_w1, author_w2, name_w1, name_w2, vm1, vm2, version_word);
    ⟨Read the extension file into the lexer, and break it into body and documentation 8⟩;
    return ef;
}
```

The function load_extension is called from 4/edoc.

§**7.** Note that we ignore a request for an extension which has already been loaded, except if the new request ups the ante in terms of the minimum version permitted: in which case we need to record that the requirement has been tightened. That is, if we previously wanted version 2 of Pantomime Sausages by Mr Punch, and loaded it, but then read the sentence

> Include version 3 of Pantomime Sausages by Mr Punch.

then we need to note that the version requirement on PS has been raised to 3. (This is why version numbers are not checked at load time: in general, we can't know at load time what we will ultimately require.)

⟨This is an extension already loaded, so note any version number hike and return 7⟩ ≡

```
    if (parse_version_number(ef->min_version_needed) < parse_version_number(version_word)) {
        ef->min_version_needed = version_word;
        ef->inclusion_sentence = current_sentence;
    }
    return ef;
```

This code is used in §6.

§**8.** We finally make our call out of the Extensions section, down through the trap-door into Read Source Text, to seek and open the file.

⟨Read the extension file into the lexer, and break it into body and documentation 8⟩ ≡

```
    char partial_pathname[MAX_FILENAME_LENGTH];
    char synopsis[MAX_FILENAME_LENGTH];
    int start_wn, end_wn;
    ⟨Concoct a partial pathname and synopsis for the extension to be read 9⟩;
    start_wn = lexer_wordcount;
    switch (read_extension_source_text(ef, partial_pathname, synopsis, census_mode)) {
        case ORIGIN_WAS_USER_EXTENSIONS_AREA:
            ef->loaded_from_built_in_area = FALSE; break;
        case ORIGIN_WAS_BUILT_IN_EXTENSIONS_AREA:
            ef->loaded_from_built_in_area = TRUE; break;
        default:                                    which can happen if the extension file cannot be found
            ef->loaded_from_built_in_area = FALSE; break;
    }
    end_wn = lexer_wordcount-1;
    if (end_wn >= start_wn) ⟨Break the extension's text into body and documentation 10⟩;
```

This code is used in §6.

§**9.** We need to concoct the specific part of the pathname for the file to read:

```
Mr Punch/Pantomime Sausages
```

but we do so without specifying where the `Mr Punch` directory is, because it could be in either the built-in or the external area: a detail which we delegate to `read_extension_source_text` to sort out for us. We also concoct a textual synopsis in the form

```
"Pantomime Sausages by Mr Punch"
```

to be used by `read_extension_source_text` for printing to `stdout`. Since we dare not assume `stdout` can manage characters outside the basic ASCII range, we flatten them from general ISO to plain ASCII.

⟨Concoct a partial pathname and synopsis for the extension to be read 9⟩ ≡

```
int i;
print_text_to_string(author_w1, author_w2, partial_pathname);
sprintf(partial_pathname+strlen(partial_pathname), "%c", FOLDER_SEPARATOR);
print_text_to_string(name_w1, name_w2, partial_pathname+strlen(partial_pathname));
print_raw_text_to_string(name_w1, name_w2, synopsis);
sprintf(synopsis+strlen(synopsis), " by ");
print_raw_text_to_string(author_w1, author_w2, synopsis+strlen(synopsis));
for (i=0; synopsis[i]; i++) synopsis[i] = iso_remove_accents(synopsis[i]);
```

This code is used in §8.

§**10.** If an extension file contains the special text (outside literal mode) of

```
---- Documentation ----
```

then this is taken as the end of the NI source, and the beginning of a snippet of documentation about the extension; text from that point on is saved until later, but not broken into sentences for the parse tree, and it is therefore invisible to the rest of NI. If this division line is not present then the extension contains only body source and no documentation.

⟨Break the extension's text into body and documentation 10⟩ ≡

```
int i;
ef->body_w1 = start_wn; ef->body_w2 = end_wn;
for (i=start_wn; i<=end_wn; i++)
    if [[i, end_wn == FOURDASHES DOCUMENTATION FOURDASHES ...]] {
        ef->body_w1 = start_wn; ef->body_w2 = i-1;
        ef->doc_w1 = i+3; ef->doc_w2 = end_wn;
        break;
    }
ef->body_text_unbroken = TRUE;                    mark this to be sentence-broken
```

This code is used in §8.

§11. Parsing extension version numbers. Extensions can have versions in the form N/DDDDDD, a format which was chosen for sentimental reasons: IF enthusiasts know it well from the banner text of the Infocom titles of the 1980s. This story file, for instance, was compiled at the time of the Reykjavik summit between Presidents Gorbachev and Reagan:

```
Moonmist
Infocom interactive fiction - a mystery story
Copyright (c) 1986 by Infocom, Inc. All rights reserved.
Moonmist is a trademark of Infocom, Inc.
Release number 9 / Serial number 861022
```

Story file collectors customarily abbreviate this in catalogues to `9/861022`.

In our scheme, DDDDDD can be omitted (in which case so must the slash be). Spacing is not allowed around the slash (if present), so the version number always occupies a single lexical word.

The following routine parses the version number at word `vwn` to give an non-negative integer – in fact it really just construes the whole thing, with the slash removed, as a 7-digit number – in such a way that an earlier version always has a lower integer than a later one. It is legal for `vwn` to be −1, which means "no version number quoted", and evaluates as 0 – corresponding to `0/000000`, lower than the lowest version number it is legal to quote explicitly, which is `1`. (It follows that requiring no version in particular is equivalent to requiring `0/000000` or better, since every extension passes that test.)

In order that the numerical form of a version number should be a signed 32-bit integer which does not overflow, we require that the release number `N` be at most 999. It could in fact rise to 2146 without incident, but it seems cleaner to constrain the number of digits than the value.

define `MAX_VERSION_NUMBER_LENGTH 10` *for* `999/991231`

```
int parse_version_number(int vwn) {
    int i, rv, slashes = 0, digits = 0, slash_at = 0;
    char *p, *q;
    if (vwn == -1) return 0;                                    an unspecified version equates to 0/000000
    p = lw_array[vwn].lw_text; q = p;
    for (i=0; p[i] != 0; i++)
        if (p[i] == '/') {
            slashes++; if ((i == 0) || (slashes > 1)) goto Malformed;
            slash_at = i; q = p+i+1;
        } else {
            if (!(isdigit(p[i]))) goto Malformed;
            digits++;
        }
    if ((p[0] == '0') || (digits == 0)) goto Malformed;
    if ((slashes == 0) && (digits <= 3))                    so that p points to 1 to 3 digits, not starting with 0
        return atoi(p)*1000000;
    p[slash_at] = 0;                temporarily replace the slash with a null, making p and q distinct C strings
    if (strlen(p) > 3) goto Malformed;                        now p points to 1 to 3 digits, not starting with 0
    if (strlen(q) != 6) goto Malformed;
    while (*q == '0') q++;                                    now q points to 0 to 6 digits, not starting with 0
    if (q[0] == 0) q--;                                     if it was 0 digits, backspace to make it a single digit 0
    rv = (atoi(p)*1000000) + atoi(q);
    p[slash_at] = '/';                        put the slash back over the null byte temporarily dividing the string
    return rv;
    Malformed: ⟨Issue a problem message for a malformed version number 12⟩;
}
```

The function parse_version_number is called from 4/ext.

§**12.** Because we tend to call `parse_version_number` repeatedly on the same word, we want to recover tidily from this problem, and not report it over and over. We do this by altering the text to `1`, the lowest well-formed version number text.

⟨Issue a problem message for a malformed version number 12⟩ ≡

```
LOG("Offending word number %d <%s>\n", vwn, lw_array[vwn].lw_text);
sentence_problem(_P_(C4ExtVersionMalformed),
    "a version number must have the form N/DDDDDD",
    "as in the example '2/040426' for release 2 made on 26 April 2004. "
    "(The DDDDDD part is optional, so '3' is a legal version number too. "
    "N must be between 1 and 999: in particular, there is no version 0.)");
change_text_of_word(vwn, "1");
return 1000000;                                    which equates to 1/000000
```

This code is used in §11.

§13. Checking the begins here and ends here sentences. When a newly loaded extension is being sentence-broken, problem messages will be turned up unless it contains the matching pair of "begins here" and "ends here" sentences. Assuming it does, the sentence breaker has no objection, but it also calls the two routines below to verify that these sentences have the correct format. (The point of this is to catch a malformed extension at the earliest possible moment after loading it: it's easy to mis-install extensions, especially if doing so by hand, and the resulting problem messages could be quite inscrutable if one extension was wrongly identified as another.)

First, we check the "begins here" sentence. We also identify where the version number is given (if it is), and check that we are not trying to use an extension which is marked as not working on the current VM.

```
void check_extension_begins_here(parse_node *PN, extension_file *ef) {
    int i, w1, w2, aw1, aw2, vm1, vm2;
    current_sentence = PN;                                  in case problem messages need to be issued
    [[w1, w2 <-- PN]];
    ef->version_loaded = -1;
    if [[w1, w2 == version ### of ...]] {
        ef->version_loaded = w1+1; w1 += 3;
        ⟨Check that the extension's version number is well-formed 14⟩;
    }
    [[w1, w2 == the ... --> w1, w2]];
    if ([[w1, w2 == ... by ... : i --> w1, w2 ... aw1, aw2]] == FALSE)
        ⟨Issue a problem message pointing out that the author is unspecified 16⟩;
    ef->VM_restriction_w1 = -1; ef->VM_restriction_w2 = -1;
    if [[w1, w2 == ... OPENBRACKET ... CLOSEBRACKET : i --> w1, w2 ... vm1, vm2]] {
        ef->VM_restriction_w1 = vm1; ef->VM_restriction_w2 = vm2;
        ⟨Check that the extension's stipulation about the virtual machine can be met 15⟩;
    }
    if ((compare_word_range(ef->name_w1, ef->name_w2, w1, w2) == FALSE) ||
        (compare_word_range(ef->author_w1, ef->author_w2, aw1, aw2) == FALSE))
        ⟨Issue a problem message pointing out that name and author do not agree with filename 17⟩;
}
```

The function check_extension_begins_here is called from 4/sent.

§14. It is sufficient to try parsing the version number in order to check it: we throw away the answer, as we can't use it yet, but this will provoke problem messages if it is malformed.

⟨Check that the extension's version number is well-formed 14⟩ ≡

```
    parse_version_number(ef->version_loaded);
```

This code is used in §13.

§**15.** On the other hand, we do already know what virtual machine we are compiling for, so we can immediately object if the loaded extension cannot be used with our VM de jour.

⟨Check that the extension's stipulation about the virtual machine can be met 15⟩ ≡

```
switch (current_VM_matches_text(ef->VM_restriction_w1, ef->VM_restriction_w2)) {
    case TRUE: break;
    case FALSE:
        ⟨Issue a problem message saying that the VM does not meet requirements 19⟩;
        break;
    case NOT_APPLICABLE:
        ⟨Issue a problem message saying that the VM requirements are malformed 18⟩;
        break;
    default: internal_error("improper response from current_VM_matches_text");
}
```

This code is used in §13.

§**16.** The extension has to identify its author, or we come here:

⟨Issue a problem message pointing out that the author is unspecified 16⟩ ≡

```
quote_extension(1, ef);
quote_words(2, PN->word_ref1, PN->word_ref2);
handmade_problem(_P_(C4ExtMiswordedBeginsHere));
issue_problem_segment(
    "has a misworded 'begins here' sentence ('%2'), which contains "
    "no 'by'. Recall that every extension should begin with a "
    "sentence such as 'Quantum Mechanics by Max Planck begins "
    "here.', and end with a matching 'Quantum Mechanics ends "
    "here.', perhaps with documentation to follow.");
issue_problem_end();
return;
```

This code is used in §13.

§**17.** Suppose we wanted Onion Cookery by Delia Smith. We loaded the extension file called Onion Cookery in the Delia Smith folder of the (probably external) extensions area: but suppose that file turns out instead to be French Cuisine by Elizabeth David, according to its "begins here" sentence? Then the following problem message is produced:

⟨Issue a problem message pointing out that name and author do not agree with filename 17⟩ ≡

```
quote_extension(1, ef);
quote_words(2, PN->word_ref1, PN->word_ref2);
handmade_problem(_P_(C4ExtMisidentified));
issue_problem_segment(
    "The extension %1, which your source text makes use of, seems to be "
    "misidentified: its 'begins here' sentence declares it as '%2'. "
    "(Perhaps it was wrongly installed?)");
issue_problem_end();
return;
```

This code is used in §13.

§**18.** See Virtual Machines for the grammar of what can be given as a VM requirement.

⟨Issue a problem message saying that the VM requirements are malformed 18⟩ ≡

```
quote_extension(1, ef);
quote_words(2, ef->VM_restriction_w1, ef->VM_restriction_w2);
handmade_problem(_P_(C4ExtMalformedVM));
issue_problem_segment(
    "Your source text makes use of the extension %1: but my copy "
    "stipulates that it is '%2', which is a description of the required "
    "story file format which I can't understand, and should be "
    "something like '(for Z-machine version 5 or 8 only)'.");
issue_problem_end();
```

This code is used in §15.

§**19.** Here the problem is not that the extension is broken in some way: it's just not what we can currently use. Therefore the correction should be a matter of removing the inclusion, not of altering the extension, so we report this problem at the inclusion line.

⟨Issue a problem message saying that the VM does not meet requirements 19⟩ ≡

```
current_sentence = ef->inclusion_sentence;
quote_source(1, current_sentence);
quote_words(2, ef->name_w1, ef->name_w2);
quote_words(3, ef->author_w1, ef->author_w2);
quote_words(4, ef->VM_restriction_w1, ef->VM_restriction_w2);
handmade_problem(_P_(C4ExtInadequateVM));
issue_problem_segment(
    "You wrote %1: but my copy of %2 by %3 stipulates that it "
    "is '%4'. That means it can only be used with certain of "
    "the possible compiled story file formats, and at the "
    "moment, we don't fit the requirements. (You can change "
    "the format used for this project on the Settings panel.)");
issue_problem_end();
```

This code is used in §15.

§**20.** Similarly, we check the "ends here" sentence. Here there are no side-effects: we merely verify that the name matches the one quoted in the "begins here". We only check this if the problem count is still 0, since we don't want to keep on nagging somebody who has already been told that the extension isn't the one he thinks it is.

```
void check_extension_ends_here(parse_node *PN, extension_file *ef) {
    int w1, w2;
    [[w1, w2 <-- PN]];
    [[w1, w2 == the ... --> w1, w2]];
    if ((problem_count == 0) &&
        (compare_word_range(ef->name_w1, ef->name_w2, w1, w2) == FALSE)) {
        current_sentence = PN;
        quote_extension(1, ef);
        quote_words(2, PN->word_ref1, PN->word_ref2);
        handmade_problem(_P_(C4ExtMisidentifiedEnds));
        issue_problem_segment(
            "The extension %1, which your source text makes use of, seems to be "
            "malformed: its 'begins here' sentence correctly identifies it, but "
            "then the 'ends here' sentence calls it '%2' instead. (They need "
```

```
            "to be a matching pair except that the end does not name the "
            "author: for instance, 'Hocus Pocus by Jan Ackerman begins here.' "
            "would match with 'Hocus Pocus ends here.')");
        issue_problem_end();
        return;
    }
}
```

The function check_extension_ends_here is called from 4/sent.

§**21. Sentence handlers for begins here and ends here.** The main traverses of the assertions are handled by code which calls "sentence handler" routines on each node in turn, depending on type. Here are the handlers for BEGINHERE and ENDHERE. As can be seen, all we really do is start again from a clean piece of paper.

Note that, because one extension can include another, these nodes may well be interleaved: we might find the sequence A begins, B begins, B ends, A ends. The careful checking done so far ensures that these will always properly nest. We don't at present make use of this, but we might in future.

```
sentence_handler BEGINHERE_SH_handler =
    { BEGINHERE_NT, -1, 0, handle_extension_begins };
sentence_handler ENDHERE_SH_handler =
    { ENDHERE_NT, -1, 0, handle_extension_ends };

void handle_extension_begins(parse_node *PN) {
    empty_headings(0); new_discussion(); near_start_of_extension = 1;
}

void handle_extension_ends(parse_node *PN) {
    empty_headings(0); near_start_of_extension = 0;
}
```

Extension Identifiers 4/eid

Purpose

To store, hash code and compare title/author pairs used to identify extensions which, though installed, are not necessarily used in the present source text.

4/eid.§5-8 The database of known EIDs; §9 How casing is normalised; §10 Documentation links

---

Definitions

¶1. Extensions are identified by the pair of title and author name, each of which is an ISO Latin-1 string limited in length, with certain bad-news characters excluded (such as `/` and `:`) so that they can be used directly in filenames. However, we will not want to compare these by string comparison: so we hash-code the combination for speed. The following structure holds a combination of the textual names and the hash code:

```
typedef struct extension_identifier {
    char author_name[MAX_EXTENSION_AUTHOR_LENGTH];
    char raw_author_name[MAX_EXTENSION_AUTHOR_LENGTH];
    char title[MAX_EXTENSION_TITLE_LENGTH];
    char raw_title[MAX_EXTENSION_TITLE_LENGTH];
    int extension_id_hash_code;                          hash code derived from the above
} extension_identifier;
```

The structure extension_identifier is shared with 4/ext, 4/excen, 4/edict and 4/edoc.

¶2. Each EID is given a hash code – an integer between 0 and the following constant minus 1, derived from its title and author name.

```
define EI_HASH_CODING_BASE 499
```

¶3. EIDs are created with one of the following contexts:

```
define NO_EIDB_CONTEXTS 5
define LOADED_EIDBC 0
define INSTALLED_EIDBC 1
define DICTIONARY_REFERRED_EIDBC 2
define HYPOTHETICAL_EIDBC 3
define USEWITH_EIDBC 4
```

```
typedef struct extension_identifier_database_entry {
    struct extension_identifier *eide_id;
    struct extension_identifier_database_entry *hash_next;   next one in hashed EID database
    int incidence_count[NO_EIDB_CONTEXTS];
} extension_identifier_database_entry;
```

The structure extension_identifier_database_entry is private to this section.

---

§**1.** Each EID structure is written only once, and its title and author name are not subsequently altered. We therefore hash-code on arrival. As when hashing vocabulary, we apply the X 30011 algorithm, this time with 499 (coprime to 30011) as base, to the text of the Unix-style pathname `Author/Title`.

It is important that no EID structure ever be modified or destroyed once created, so it must not be stored inside a transient data structure like a `specification`.

Though it is probably the case that the author name and title supplied are already of normalised casing, we do not want to rely on that. EIDs of the same extension but named with different casing conventions would fail to match: and this could happen if a new build of NI were published which made a subtle change to the casing conventions, but which continued to use an extension dictionary file first written by previous builds under the previous conventions.

```
void eid_new(extension_identifier *eid, char *an, char *ti, int context) {
    unsigned int hc = 0, i;
    truncated_strcpy(eid->raw_author_name, an, MAX_EXTENSION_AUTHOR_LENGTH);
    truncated_strcpy(eid->author_name, an, MAX_EXTENSION_AUTHOR_LENGTH);
    normalise_extension_casing(eid->author_name);
    truncated_strcpy(eid->raw_title, ti, MAX_EXTENSION_TITLE_LENGTH);
    truncated_strcpy(eid->title, ti, MAX_EXTENSION_TITLE_LENGTH);
    normalise_extension_casing(eid->title);
    char *norm_an = eid->author_name, *norm_ti = eid->title;
    for (i=0; norm_an[i]; i++) hc = hc*30011 + (norm_an[i]);
    hc = hc*30011 + '/';
    for (i=0; norm_ti[i]; i++) hc = hc*30011 + (norm_ti[i]);
    hc = hc % EI_HASH_CODING_BASE;
    eid->extension_id_hash_code = hc;
    add_EID_to_database(eid, context);
}

void eid_set_raw(extension_identifier *eid, char *raw_an, char *raw_ti) {
    strcpy(eid->raw_author_name, raw_an);
    strcpy(eid->raw_title, raw_ti);
}

void eid_write_to_HTML_file(OUTPUT_STREAM, extension_identifier *eid, int fancy) {
    print_literal_string_to_file(OUT, eid->raw_title);
    if (fancy) WRITE("<font color=\"#404040\">");
    WRITE(" by ");
    if (fancy) WRITE("</font>");
    print_literal_string_to_file(OUT, eid->raw_author_name);
}

void eid_write_link_to_HTML_file(OUTPUT_STREAM, extension_identifier *eid) {
    WRITE("<a href='Extensions/");
    print_literal_string_to_file(OUT, eid->author_name);
    WRITE("/");
    print_literal_string_to_file(OUT, eid->title);
    WRITE(".html' STYLE=\"text-decoration: none\"><font color=\"#404040\">");
    if (eid_is_standard_rules(eid)) print_literal_string_to_file(OUT, eid->title);
    else eid_write_to_HTML_file(OUT, eid, FALSE);
    WRITE("</font></a>");
}
```

The function eid_new is called from 4/ext, 4/excen, 4/edict and 4/head.
The function eid_set_raw is called from 4/excen.
The function eid_write_to_HTML_file is called from 4/edict, 4/edoc and 12/rb.
The function eid_write_link_to_HTML_file is called from 4/edict.

§**2.** Two EIDs with different hash codes definitely identify different extensions; if the code is the same, we must use `strcmp` on the actual title and author name. This is in effect case insensitive, since we normalised casing when the EIDs were created.

(Note that this is not a lexicographic function suitable for sorting EIDs into alphabetical order: it cannot be, since the hash code is not order-preserving. To emphasise this we return true or false rather than a `strcmp`-style delta value. For `eidcmp`, see below...)

```
int eid_match(extension_identifier *eid1, extension_identifier *eid2) {
    if ((eid1 == NULL) || (eid2 == NULL)) internal_error("bad eid match");
    if (eid1->extension_id_hash_code != eid2->extension_id_hash_code) return FALSE;
    if (strcmp(eid1->author_name, eid2->author_name) != 0) return FALSE;
    if (strcmp(eid1->title, eid2->title) != 0) return FALSE;
    return TRUE;
}
```

The function eid_match is called from 4/edict and 4/head.

§**3.** This is quite a deal slower, but trichotomous.

```
int eidcmp(extension_identifier *eid1, extension_identifier *eid2) {
    int d;
    if ((eid1 == NULL) || (eid2 == NULL)) internal_error("bad eid match");
    d = strcmp(eid1->author_name, eid2->author_name);
    if (d != 0) return d;
    return strcmp(eid1->title, eid2->title);
}
```

The function eidcmp is called from 4/edict.

§**4.** Because the Standard Rules are treated slightly differently by the documentation, and so forth, it's convenient to provide a single function which asks if a given EID is talking about them.

```
int an_eid_for_standard_rules_created = FALSE;
extension_identifier an_eid_for_standard_rules;
int eid_is_standard_rules(extension_identifier *eid) {
    if (an_eid_for_standard_rules_created == FALSE) {
        an_eid_for_standard_rules_created = TRUE;
        eid_new(&an_eid_for_standard_rules,
            "Graham Nelson", "Standard Rules", HYPOTHETICAL_EIDBC);
    }
    return eid_match(eid, &an_eid_for_standard_rules);
}
```

The function eid_is_standard_rules is called from 2/prob1, 4/ext and 4/edoc.

§**5. The database of known EIDs.** We will need to be able to give rapid answers to questions like "is there an installed extension with this EID?" and "does any entry in the dictionary relate to this EID?": there may be many extensions and very many dictionary entries, so we keep an incidence count of each EID and in what context it has been used, and store that in a hash table. Note that each distinct EID is recorded only once in the table: this is important, as although an individual extension can only be loaded or installed once, it could be referred to in the dictionary dozens or even hundreds of times.

The table is unsorted and is intended for rapid searching. Typically there will be only a handful of EIDs in the list of those with a given hash code: indeed, the typical number will be 0 or 1.

```
int EID_database_created = FALSE;
extension_identifier_database_entry *hash_of_EIDEs[EI_HASH_CODING_BASE];

void add_EID_to_database(extension_identifier *eid, int context) {
    extension_identifier_database_entry *eide;
    int i, hc;
    if (EID_database_created == FALSE) {
        EID_database_created = TRUE;
        for (hc=0; hc<EI_HASH_CODING_BASE; hc++) hash_of_EIDEs[hc] = NULL;
    }
    hc = eid->extension_id_hash_code;
    for (eide = hash_of_EIDEs[hc]; eide; eide = eide->hash_next)
        if (eid_match(eid, eide->eide_id)) {
            eide->incidence_count[context]++;
            return;
        }
    eide = CREATE(extension_identifier_database_entry);
    eide->hash_next = hash_of_EIDEs[hc]; hash_of_EIDEs[hc] = eide;
    eide->eide_id = eid;
    for (i=0; i<NO_EIDB_CONTEXTS; i++) eide->incidence_count[i] = 0;
    eide->incidence_count[context] = 1;
}
```

§**6.** The purpose of the hash table is to enable us to reply quickly when asked for one of the following usage counts:

```
int no_times_EID_used_in_context(extension_identifier *eid, int context) {
    extension_identifier_database_entry *eide;
    for (eide = hash_of_EIDEs[eid->extension_id_hash_code]; eide; eide = eide->hash_next)
        if (eid_match(eid, eide->eide_id)) return eide->incidence_count[context];
    return 0;
}
```

The function no_times_EID_used_in_context is called from 4/excen, 4/edict and 4/head.

§**7.** The EID hash table makes quite interesting reading, so:

```
void log_EID_hash_table(void) {
    int hc, total = 0;
    LOG("Extension identifier hash table:\n");
    for (hc=0; hc<EI_HASH_CODING_BASE; hc++) {
        extension_identifier_database_entry *eide;
        for (eide = hash_of_EIDEs[hc]; eide; eide = eide->hash_next) {
            total++;
            LOG("%03d %3d %3d %3d %3d  %50s  %50s\n",
                hc, eide->incidence_count[0], eide->incidence_count[1],
                eide->incidence_count[2], eide->incidence_count[3],
                eide->eide_id->author_name, eide->eide_id->title);
        }
    }
    LOG("%d entries in all\n", total);
}
```

The function log_EID_hash_table is called from 4/ext.

§**8.** I dislike to use `strncpy` because, and for some reason this surprises me every time, it truncates but fails to write a null termination character if the string to be copied is larger than the buffer to write to: the result is therefore not a well-formed string and we have to fix matters by hand. This I think makes for opaque code. So:

```
void truncated_strcpy(char *to, char *from, int max) {
    int i;
    for (i=0; ((from[i]) && (i<max-1)); i++) to[i] = from[i];
    to[i] = 0;
}
```

The function truncated_strcpy is called from 2/prob3, 4/ext, 4/edict, 5/bp, 7/kov, 12/phsf, 12/br and 14/i6t.

§**9. How casing is normalised.** Every word is capitalised, where a word begins at the start of the text, after a hyphen, or after a bracket. Thus "Every Word Counts", "Even Double-Barrelled Ones (And Even Parenthetically)".

```
void normalise_extension_casing(char *p) {
    int i, uc;
    for (i=0, uc = TRUE; p[i]; i++) {
        if (uc) p[i] = toupper(p[i]);
        else p[i] = tolower(p[i]);
        uc = FALSE;
        if (p[i] == ' ') uc = TRUE;
        if (p[i] == '-') uc = TRUE;
        if (p[i] == '(') uc = TRUE;
    }
}
```

The function normalise_extension_casing is called from 4/excen.

§**10. Documentation links.** This is where HTML links to extension documentation are created; the URL for each extension's page is generated from its ID.

```
void begin_extension_link(OUTPUT_STREAM, extension_identifier *eid, char *rubric) {
    WRITE("<a href='inform://Extensions/Extensions/%s/%s.html' ", eid->author_name, eid->title);
    if (rubric) WRITE("title=\"%s\" ", rubric);
    else WRITE("title=\"%s by %s\" ", eid->title, eid->author_name);
    WRITE("STYLE=\"text-decoration: none\">");
}

void end_extension_link(OUTPUT_STREAM, extension_identifier *eid) {
    WRITE("</a>\n");
}
```

The function begin_extension_link is called from 2/index and 4/excen.
The function end_extension_link is called from 2/index and 4/excen.

Extension Census 4/excen

Purpose

To conduct a census of all the extensions installed (whether used on this run or not), and keep the documentation index for them up to date.

Definitions

¶1. In addition to the extensions read in, there are the roads not taken: the ones which NI has at its disposal, but which the source text never asks to include. NI performs a "census" of installed extensions on every run, essentially by scanning the directories which hold them to see what the user has installed there. Each extension discovered will produce a single "extension census datum", or ECD.

```
typedef struct extension_census_datum {
    struct extension_identifier ecd_id;                    title, author, hash code
    char version_text[MAX_VERSION_NUMBER_LENGTH+1];        such as 23 or 14/060527
    char VM_requirement[512];                              such as "(for Z-machine only)"
    int built_in;                                          found in the Inform 7 application's private stock
    int overriding_a_built_in_extension;                   not built in, but overriding one which is
    char *domain;                                          pathname of the stock in which this was found
    char rubric[MAX_RUBRIC_LENGTH+1];                      brief description found in opening lines
    struct extension_census_datum *next;                   next one in lexicographic order
    MEMORY_MANAGEMENT
} extension_census_datum;
```

The structure extension_census_datum is shared with 4/edoc.

¶2. The extension census results are then tabulated in lexicographic order in an array of linked lists, one for each possible first-letter-of-author-name:

```
extension_census_datum *extension_census_results[256];     list starting with each letter
```

---

§**1.** This is a narrative section and describes the story of the census. Just as Caesar Augustus decreed that all the world should be taxed, and that each should return to his place of birth, so we will open and inspect every extension we can find, checking that each is in the right place.

Note that if the same extension is found in more than one domain, the first to be found is considered the definitive version: this is why the external area is searched first, so that the user can override built-in extensions by placing his own versions in the external area. (Should this convention ever be reversed, a matching change would need to be made in the code which opens extension files in Read Source Text.)

```
void extension_census(void) {
    ⟨Make sure the external extensions area exists, so that it can be searched 2⟩;
    begin_recording_census_errors();
    count user-installed extensions first
    take_census_of_domain(pathname_of_extensions, FALSE);
    and then count the built-in ones
    take_census_of_domain(pathname_of_built_in_extensions, TRUE);
    end_recording_census_errors();
}
```

The function extension_census is called from 4/ext.

§**2.** We will need to search both the built-in extension area, and also the external one. The latter cannot be guaranteed to exist – and indeed will not, if NI is being run for the first time after Inform is installed. Should anything go wrong we simply abandon the census: it wasn't all that crucial, and we don't want to deprive the user of the enjoyment of his successfully compiled story file just because we're (say) running with inadequate permissions to read the external area.

⟨Make sure the external extensions area exists, so that it can be searched 2⟩ ≡

```
    if (verify_installed_extensions_tree() == FALSE) return;
```

This code is used in §1.

§**3. Conducting the census.** An "extension domain", for these purposes, is a directory in the filing system which can contain extensions (providing these are themselves placed in subdirectories identifying their authorship). The built-in domain is special (it is read-only, for one thing), but in principle we could have any number of other domains. The following code scans one.

```
char *current_extension_domain;
void take_census_of_domain(char *pathname, int built_in) {
    current_extension_domain = pathname;
    census_from(pathname, TRUE, built_in, "");
}
```

§**4.** The following routine is not as recursive as it looks, since it runs at only two levels: a top level, when it is scanning the domain, and an inner level, when it is scanning the subfolder of the domain for a given author.

The reader may wince at the way we scan a directory – we essentially dump a catalogue listing to a temporary text file and then read it back in, one line at a time – but it works properly on all platforms, which is important given how poorly directory handling is standardised in C.

Because of the two-level recursion, there are two such temporary files, `Temporary1.txt` at the upper level and `Temporary0.txt` below. These are stored in the external Extensions area. In principle that could be problematic since two processes running Inform on the same machine may be simultaneously taking an extension census, so could lock each other out: probably we ought to include a process ID in the filenames. But then we would have further platform hassles, and we would need to delete the files when done, and worry about what happens if Inform is aborted half-way. (When `intest` does large multiprocessing test runs of Inform, indexing is turned off, so extension censuses are not being taken.)

```
void census_from(char *pathname, int top_level, int built_in, char *parent) {
    char path_to_folder[2*MAX_FILENAME_LENGTH];
    char path_to_temp[MAX_FILENAME_LENGTH];
    char item_name[MAX_FILENAME_LENGTH+1];
    FILE *TEMPF;
    int linecount = 0;
    transcode_ISO_string_to_locale(pathname, path_to_folder);
    sprintf(path_to_temp, "%s%cTemporary%d.txt", pathname_of_extension_docs,
        FOLDER_SEPARATOR, top_level);
    if (write_folder_contents_to_file(path_to_folder, path_to_temp) == FALSE) {
        LOGIF(EXTCENSUS, "Unable to obtain contents of <%s>\n", pathname);
        return;
    }
    TEMPF = iso_fopen(path_to_temp, "r");
    if (TEMPF == NULL) fatal_error("Unable to open temporary file for reading");
    while (truncated_locale_fgets(TEMPF, item_name, MAX_FILENAME_LENGTH+1) >= 1) {
```

```
        int n = strlen(item_name) - 1;
        if (n<0) continue;
        LOGIF(EXTCENSUS, "%d: %s\n", linecount++, item_name);
        if (top_level) ⟨Take census from a possible author folder 5⟩
        else ⟨Take census from a possible extension 6⟩;
    }
    fclose(TEMPF);
}
```

§**5.** At the upper level, we expect every item to be a subfolder whose name is that of a given author. We can identify folder names because of the terminating slash `/`: note that this character is used regardless of whether or not it is the platform's file separator, and that it is illegal in extension titles or author names. (See Filenames for more on how `write_folder_contents_to_file` describes a directory.)

It is a "census error" – meaning, we have detected a misinstallation of extensions – if any file is present, or any folder too long for its name to be an author name, except that items with names beginning with a `.` are ignored. (In particular, the folders `./` and `../` are ignored, but so too are OS X Finder files like `.DS_Store`, and other such junk. No author name or title is allowed to contain a `.` character, so this cannot throw away valid census entries.)

The author name `Reserved` is, of course, reserved: the author of an extension is not allowed to be "Reserved", and this subfolder is used by Inform for its own purposes. Otherwise any reasonably short name of filename-safe characters is valid as an author name.

Once we have a valid, non-reserved author subfolder, we recurse down to lower level to scan it.

⟨Take census from a possible author folder 5⟩ ≡

```
    if (item_name[0] == '.') continue;
    if (item_name[n] == '/') {
        char path_to_sub[MAX_FILENAME_LENGTH];
        item_name[n] = 0;                                   remove the terminal slash: it has served its purpose
        if (strcmp(item_name, "Reserved") == 0) continue;
        if (strlen(item_name) > MAX_EXTENSION_TITLE_LENGTH-1) {
            census_error("author name exceeds the maximum permitted length",
                item_name, NULL, NULL, NULL); continue;
        }
        sprintf(path_to_sub, "%s%c%s", pathname, FOLDER_SEPARATOR, item_name);
        census_from(path_to_sub, FALSE, built_in, item_name);
        continue;
    }
    census_error("non-folder found where author folders should be",
        item_name, NULL, NULL, NULL); continue;
```

This code is used in §4.

§**6.** At the lower level, . files or folders are again skipped; any other folders are a census error, since there should only be extension files present; and once again, we enforce the title length restriction.

As will be seen from the logic below, an extension is only given a census entry (and therefore included in the HTML documentation of installed extensions) if no census errors arise from it. An Inform user can therefore know that if an extension shows up on his documentation page, it is properly installed and identifies itself correctly, and moreover that it can be installed correctly on other Informs elsewhere (perhaps on other platforms).

define `MAX_TITLING_LINE_LENGTH 501` *lots, allowing for an improbably large number of virtual machine restrictions*

⟨Take census from a possible extension 6⟩ ≡

```
int overridden_by_an_extension_already_found = FALSE;
char candidate_title[MAX_EXTENSION_TITLE_LENGTH + 1],
    raw_title[MAX_EXTENSION_TITLE_LENGTH + 1];
char candidate_author_name[MAX_EXTENSION_TITLE_LENGTH + 1],
    raw_author_name[MAX_EXTENSION_TITLE_LENGTH + 1];
char titling_line[MAX_TITLING_LINE_LENGTH], rubric_text[MAX_RUBRIC_LENGTH + 1];
char version_text[MAX_TITLING_LINE_LENGTH], requirement_text[MAX_TITLING_LINE_LENGTH];
char claimed_author_name[MAX_TITLING_LINE_LENGTH], claimed_title[MAX_TITLING_LINE_LENGTH];
if (item_name[0] == '.') continue;
if (item_name[n] == '/') {
    census_error("folder or application in author folder",
        parent, item_name, NULL, NULL); continue;
}
if (strlen(item_name) > MAX_EXTENSION_TITLE_LENGTH-1) {
    census_error("title exceeds the maximum permitted length",
        parent, item_name, NULL, NULL); continue;
}
⟨Make candidate title and author name from normalised casing versions of filename and parent folder name 7⟩;
⟨Extract the titling line and rubric, if any, from the extension file 9⟩;
⟨Parse the version, title, author and VM requirements from the titling line 13⟩;
⟨Check that the candidate name and title match those claimed in the titling line 17⟩;
⟨See if we duplicate the title and author name of an extension already found in another domain 18⟩;
if (overridden_by_an_extension_already_found == FALSE) {
    extension_census_datum *ecd;
    ⟨Create a new census datum for this extension, which has passed all tests 19⟩;
    ⟨Add the new census datum into the census list, keeping this in lexicographic order 20⟩;
}
```

This code is used in §4.

§**7.** If we find an extension at the relative pathname `Emily Short/Locksmith` then we expect it to be Locksmith by Emily Short: these are the candidate author name and title respectively. (What the file actually contains is another matter, as we shall see.)

All titles and author names have to be stored carefully in case-normalised form at all times, and this extends to the filename and folder name. Enforcing this rule may seem needlessly bureaucratic, since for the majority of users (using typical Mac OS X and Windows installations) the filing system preserves the case of filenames but is not sensitive to case when searching for files: thus the folders `jimmy stewart` and `Jimmy Stewart` behave identically in practice. But we want to use `strcmp` for case-sensitive comparisons, for instance.

There are active Inform users who do have a case-sensitive filing system (on some of the Linux and Solaris ports), and we want extension files to continue to work perfectly if taken from one Inform installation and added to another. So the previously strict rules on casing of the filenames as stored have been waived.

⟨Make candidate title and author name from normalised casing versions of filename and parent folder name 7⟩ ≡

```
strcpy(candidate_author_name, parent);
strcpy(candidate_title, item_name);
⟨Remove filename extension for extensions, if any 8⟩;
strcpy(raw_title, candidate_title);
strcpy(raw_author_name, candidate_author_name);
normalise_extension_casing(candidate_author_name);
normalise_extension_casing(candidate_title);
if (FALSE) {
    if (strcmp(candidate_author_name, parent) != 0) {
        census_error("file is in a folder whose name does not use upper and "
            "lower case correctly (should be, e.g., 'Cary Grant' not 'cary "
            "grant' or 'CARY GRANT')",
            parent, item_name, NULL, NULL); continue;
    }
    if (strcmp(candidate_title, item_name) != 0) {
        census_error("file name does not use upper and "
            "lower case correctly (should be, e.g., 'Going To The Zoo' not 'going "
            "to the zoo' or 'Going to the Zoo')",
            parent, item_name, NULL, NULL); continue;
    }
}
```

This code is used in §6.

§**8.** We permit (encourage, actually) the filename extension ".i7x" for extensions, in any of its possible casings, and that of course isn't part of the title:

⟨Remove filename extension for extensions, if any 8⟩ ≡

```
if ((n>=5) &&
    (candidate_title[n-3] == '.') &&
    ((candidate_title[n-2] == 'i') || (candidate_title[n-2] == 'I')) &&
    (candidate_title[n-1] == '7') &&
    ((candidate_title[n] == 'x') || (candidate_title[n-2] == 'X')))
        candidate_title[n-3] = 0;
```

This code is used in §7.

§**9. Handling the extension file.** This and the next three paragraphs do the file-handling necessary to open the extension, extract its titling line and rubric, then close it again.

⟨Extract the titling line and rubric, if any, from the extension file 9⟩ ≡

```
FILE *EXTF;
⟨Open the extension file for reading 10⟩;
⟨Read the titling line of the extension and normalise its casing 11⟩;
⟨Read the rubric text, if any is present 12⟩;
fclose(EXTF);
```

This code is used in §6.

§**10.** The following should only fail in the event of some peculiar filing system failure (or of some other process having moved or deleted the file since we scanned the folder only moments ago).

⟨Open the extension file for reading 10⟩ ≡

```
char path_to_ext[MAX_FILENAME_LENGTH];
sprintf(path_to_ext, "%s%c%s%c%s",
    current_extension_domain, FOLDER_SEPARATOR, parent, FOLDER_SEPARATOR, item_name);
EXTF = iso_fopen_caseless(path_to_ext, "r");
if (EXTF == NULL) {
    census_error("file cannot be read",
        parent, item_name, NULL, NULL); continue;
}
```

This code is used in §9.

§**11.** The actual maximum number of characters in the titling line is one less than `MAX_TITLING_LINE_LENGTH`, to allow for the null terminator. The titling line is terminated by any of `0A`, `0D`, `0A 0D` or `0D 0A`, or by the local `\n` for good measure.

⟨Read the titling line of the extension and normalise its casing 11⟩ ≡

```
int titling_chars_read = 0, c;
titling_line[0] = 0;
while ((c = utf8_fgetc(EXTF, FALSE)) != EOF) {
    if (c == 0xFEFF) continue;                    skip the optional Unicode BOM pseudo-character
    if ((c == '\x0a') || (c == '\x0d') || (c == '\n')) break;
    if (titling_chars_read < MAX_TITLING_LINE_LENGTH - 1)
        titling_line[titling_chars_read++] = c;
}
titling_line[titling_chars_read--] = 0;
normalise_extension_casing(titling_line);
```

This code is used in §9.

§**12.** In the following, all possible newlines are converted to white space, and all white space before a quoted rubric text is ignored. We need to do this partly because users have probably keyed a double line break before the rubric, but also because we might have stopped reading the titling line halfway through a line division combination like `0A 0D`, so that the first thing we read here is a meaningless `0D`.

⟨Read the rubric text, if any is present 12⟩ ≡

```
int rubric_chars_read = 0, c, found_start = FALSE;
while ((c = utf8_fgetc(EXTF, FALSE)) != EOF) {
    if (rubric_chars_read >= MAX_RUBRIC_LENGTH) break;
    if ((c == '\x0a') || (c == '\x0d') || (c == '\n') || (c == '\t')) c = ' ';
    if ((c != ' ') && (found_start == FALSE)) {
        if (c == '"') found_start = TRUE;
        else break;
    } else {
        if (c == '"') break;
        if (found_start) rubric_text[rubric_chars_read++] = c;
    }
}
rubric_text[rubric_chars_read] = 0;
```

This code is used in §9.

§13. Parsing the titling line. In general, once case-normalised, a titling line looks like this:

Version 2/070423 Of Going To The Zoo (For Glulx Only) By Cary Grant Begins Here.

and the version information, the VM restriction and the full stop are all optional, but the division word "of" and the concluding "begin[s] here" are not. We break it up into pieces, so that

```
version_text = "2/070423"
claimed_title = "Going To The Zoo"
claimed_author_name = "Cary Grant"
requirement_text = "(For Glulx Only)"
```

It's tempting to do this by feeding it into the lexer and then reusing some of the code which parses these lines during sentence-breaking, but in fact we want to use the information rather differently, and besides: it seems useful to record some C code here which correctly parses a titling line, since this can easily be extracted and used in other utilities handling Inform extensions.

⟨Parse the version, title, author and VM requirements from the titling line 13⟩ ≡

```
int start = 0, end = strlen(titling_line) - 1;
while ((end>=0) && ((titling_line[end] == ' ') ||
    (titling_line[end] == '\t') || (titling_line[end] == '.'))) {
    titling_line[end] = 0; end--;          trim white space or full stops from the end of the titling line
}
if ((end < strlen(" Begin Here")) || (strcmp(titling_line+end-4, " Here") != 0)) {
    census_error("appears not to be an extension (its first line does "
        "not end 'begin(s) here', as extension titling lines must)",
        parent, item_name, NULL, NULL); continue;
}
end -= 5; if (titling_line[end] == 's') end--;
titling_line[end+1] = 0;                                        trim "s Here" or " Here"
if (strcmp(titling_line+end-5, " Begin") != 0) {
    census_error("appears not to be an extension (its first line does "
        "not end 'begin(s) here', as extension titling lines must)",
        parent, item_name, NULL, NULL); continue;
}
end -= 6; titling_line[end+1] = 0;                                        trim " Begin"
⟨Scan the version text, if any, and advance to the position past Version... Of 14⟩;
if (strncmp(titling_line+start, "The ", 4) == 0) start += 4;          advance past any "The "
⟨Divide the remaining text into a claimed author name and title, divided by By 15⟩;
⟨Extract the VM requirements text, if any, from the claimed title 16⟩;
```

This code is used in §6.

§**14.** Ah, the pleasures of C string hackery. Here, `begin` and `end` are the positions of the first and last character (not the null terminator) in the unparsed part of the titling line. To advance means to increase `begin`.

We make no attempt to check the version number for validity: the purpose of the census is to identify extensions and reject accidentally included other files, not to syntax-check all extensions to see if they would work if used.

⟨Scan the version text, if any, and advance to the position past Version... Of 14⟩ ≡

```
int l;
version_text[0] = 0;
if (strncmp(titling_line, "Version ", 8) == 0)
    for (l=0, start=0; titling_line[l]; l++)
        if (strncmp(titling_line + l, " Of ", 4) == 0) {
            for (start=8; start<l; start++) version_text[start-8] = titling_line[start];
            version_text[l-8] = 0;
            start = l+4;
            break;
        }
```

This code is used in §13.

§**15.** The earliest "by" is the divider: note that extension titles are not allowed to contain this word, so "North By Northwest By Cary Grant" is not a situation we need to contend with.

⟨Divide the remaining text into a claimed author name and title, divided by By 15⟩ ≡

```
int l, j;
claimed_author_name[0] = 0;
claimed_title[0] = 0;
for (l=start; titling_line[l]; l++)
    if (strncmp(titling_line+l, " By ", 4) == 0) {
        for (j=start; j<l; j++) claimed_title[j-start] = titling_line[j];
        claimed_title[l-start] = 0;
        for (j=l+4; titling_line[j]; j++) claimed_author_name[j-l-4] = titling_line[j];
        claimed_author_name[j-l-4] = 0;
        break;
    }
if ((claimed_author_name[0] == 0) || (claimed_title[0] == 0)) {
    census_error("appears not to be an extension (the titling line does "
        "not give both author and title)",
        parent, item_name, NULL, NULL); continue;
}
```

This code is used in §13.

§**16.** Similarly, extension titles are not allowed to contain parentheses, so this is unambiguous.

⟨Extract the VM requirements text, if any, from the claimed title 16⟩ ≡

```
int l;
requirement_text[0] = 0;
for (l=0; claimed_title[l]; l++)
    if (claimed_title[l] == '(') {
        strcpy(requirement_text, claimed_title + l);
        claimed_title[l] = 0;                                   truncate to just before open bracket
        while ((l>0) && (claimed_title[l-1] == ' ')) {          trim spaces from the end
            l--; claimed_title[l] = 0;
        }
        break;
    }
```

This code is used in §13.

§**17. Making sure this is the extension we expected to find.** It's easier for these confusions to arise than might be thought. For instance, if two different authors independently write extensions called "Followers", it would be easy to put a file of this name in the wrong author folder by mistake. Since extension installation is usually handled mechanically (and, we hope, correctly) by the Inform application, such problems are only likely to arise when users install extensions by hand. Still, it's prudent to check.

⟨Check that the candidate name and title match those claimed in the titling line 17⟩ ≡

```
int right_leafname = FALSE, right_folder = FALSE;
if (strcmp(claimed_title, candidate_title) == 0) right_leafname = TRUE;
if (strcmp(claimed_author_name, candidate_author_name) == 0) right_folder = TRUE;
if ((right_leafname == TRUE) && (right_folder == FALSE)) {
    census_error("an extension with the right filename but in the wrong "
        "author's folder",
        parent, item_name, claimed_author_name, claimed_title); continue;
}
if ((right_leafname == FALSE) && (right_folder == TRUE)) {
    census_error("an extension stored in the correct author's folder, but "
        "with the wrong filename",
        parent, item_name, claimed_author_name, claimed_title); continue;
}
if ((right_leafname == FALSE) && (right_folder == FALSE)) {
    census_error("an extension but with the wrong filename and put in the "
        "wrong author's folder",
        parent, item_name, claimed_author_name, claimed_title); continue;
}
```

This code is used in §6.

§**18. Adding the extension to the census, or not.** Recall that the higher-priority external domain is scanned first; the built-in domain is scanned second. So if we find that our new extension has the same title and author as one already known, it must be the case that we are now scanning the built-in area and that the previous one was an extension which the user had installed to override this built-in extension.

⟨See if we duplicate the title and author name of an extension already found in another domain 18⟩ ≡

```
extension_census_datum *other;
LOOP_OVER(other, extension_census_datum)
    if ((strcmp(candidate_author_name, other->ecd_id.author_name) == 0)
        && (strcmp(candidate_title, other->ecd_id.title) == 0)) {
        other->overriding_a_built_in_extension = TRUE;
        overridden_by_an_extension_already_found = TRUE;
    }
```

This code is used in §6.

§**19.** Assuming the new extension was not overridden in this way, we come here. Because we didn't check the version number text for validity, it might through being invalid be longer than we expect: in case this is so, we truncate it.

⟨Create a new census datum for this extension, which has passed all tests 19⟩ ≡

```
ecd = CREATE(extension_census_datum);
eid_new(&(ecd->ecd_id), candidate_author_name, candidate_title, INSTALLED_EIDBC);
eid_set_raw(&(ecd->ecd_id), raw_author_name, raw_title);
strcpy(ecd->VM_requirement, requirement_text);
version_text[MAX_VERSION_NUMBER_LENGTH] = 0;                    truncate to maximum legal length
strcpy(ecd->version_text, version_text);
ecd->domain = current_extension_domain;
ecd->built_in = built_in;
ecd->overriding_a_built_in_extension = FALSE;
ecd->next = NULL;
rubric_text[MAX_RUBRIC_LENGTH-1] = 0;
strcpy(ecd->rubric, rubric_text);
```

This code is used in §6.

§**20.** We now insertion-sort the new ECD into the census: that is, run through it from A to Z, stopping where the new one should be entered. It may seem that sorting is unnecessary, since the filename listings made to carry out the census will be in alphabetical order anyway: and on some platforms, some of the time, that might even be true.

We insertion-sort these ECDs, which would have rather poor running time if there are thousands rather than dozens of extensions. So we ameliorate this by using an ordered hash table – though that is a very grand way of putting it: the hash code used here is just the first letter of the author's name and the hash table is therefore only an array of alphabetised lists of all extensions by authors beginning with A, with B, and so on. The bunching of names means that this probably only speeds us up by a factor of 10 or so (E is a particular spike, thanks to the prolific work of Emily Short, but also Eric Eve): using the first letter of the title would be more efficient, but that wouldn't respect lexicographic order, since we sort by author then title not vice versa. The empty author name is not allowed, so there always is first letter (in the case of "Anonymous", it will be A).

Because the extension identifiers for ECDs always have title and name extension-normalised in casing, we can use `strcmp` to compare them and not worry about its being case-sensitive.

⟨Add the new census datum into the census list, keeping this in lexicographic order 20⟩ ≡

```
int initial_letter = candidate_author_name[0];
```

```
    extension_census_datum *prev = NULL;
    extension_census_datum *link = extension_census_results[initial_letter];
    while ((link) && (strcmp(link->ecd_id.author_name, ecd->ecd_id.author_name) < 0)) {
        prev = link; link = link->next;                    skip extensions by alphabetically earlier authors
    }
    while ((link) && (strcmp(link->ecd_id.author_name, ecd->ecd_id.author_name) == 0) &&
        (strcmp(link->ecd_id.title, ecd->ecd_id.title) < 0)) {
        prev = link; link = link->next;                  skip extensions by same author with earlier titles
    }
    if (prev == NULL) {                                          make new ECD the new first item in the list
        ecd->next = extension_census_results[initial_letter];
        extension_census_results[initial_letter] = ecd;
    } else {                                                  insert new ECD after prev in the list
        ecd->next = prev->next;
        prev->next = ecd;
    }
```

This code is used in §6.

§**21. Census errors.** To recap, the extension census process involves looking for all the extensions NI can find, but also checking them: are the files genuine, that is, do they appear to be suitable text files with the correct identifying first lines? Or have binary files crept in, or genuine extension but which have been given the wrong location or filename? When any of these checks fails, a census error is generated: there are about a dozen different kinds.

These are dumped to a temporary text file in the external extensions area, to be appended to the extensions directory page later on. Once again, this is not thread-safe: two simultaneous attempts by different Inform processes to take a census might collide.

```
int no_census_errors = 0;
STREAM CENERR_struct;
STREAM *CENERR = NULL;

void begin_recording_census_errors(void) {
    no_census_errors = 0;
    CENERR = NULL;
}

int currently_recording_census_errors(void) {
    if (CENERR) return TRUE;
    return FALSE;
}

void end_recording_census_errors(void) {
    if (CENERR) STREAM_CLOSE(CENERR);
    CENERR = NULL;
}
```

The function currently_recording_census_errors is called from 4/ext.

§**22.** When a census error arises, then, we write it as a line to the text file. The line ends with an HTML break tag `<br>` since this copy is destined to be used in an HTML catalogue.

```
void census_error(char *message, char *auth, char *title,
    char *claimed_author, char *claimed_title) {
    no_census_errors++;
    if (CENERR == NULL) {
        char path_to_errors[MAX_FILENAME_LENGTH];
        sprintf(path_to_errors, "%s%cReserved%cCensusErrors.txt",
            pathname_of_extensions, FOLDER_SEPARATOR, FOLDER_SEPARATOR);
        CENERR = &CENERR_struct;
        if (STREAM_OPEN_TO_FILE(CENERR, path_to_errors, UTF8_ENC) == FALSE)
            fatal_error2("Unable to write census error log", path_to_errors);
    }
    if (claimed_author)
        STREAM_WRITE(CENERR, "%s by %s - %s (the extension says it is '%s by %s')<br>\n",
            title, auth, message, claimed_title, claimed_author);
    else if ((auth) && (title)) STREAM_WRITE(CENERR, "%s by %s - %s<br>\n", title, auth, message);
    else STREAM_WRITE(CENERR, "%s - %s<br>\n", auth, message);
}
```

§**23.** And this is where the inclusion of that material into the catalogue is taken care of. First, we sometimes position a warning prominently at the top of the listing, because otherwise its position at the bottom will be invisible unless the user scrolls a long way:

```
void warn_about_census_errors(OUTPUT_STREAM) {
    if (no_census_errors == 0) return;                                  no need for a warning
    if (NUMBER_CREATED(extension_census_datum) < 20) return;            it's a short page anyway
    WRITE("<p><img border=0 src=inform:/doc_images/census_problem.png> "
        "<b>Warning</b>. One or more extensions have been installed incorrectly: "
        "see the details at the foot of the page for more.<p>");
}
```

§**24.** A minor subtlety here is that the census errors file is ISO Latin-1 encoded, with HTML tags in: we need to write it to a UTF-8 file, and therefore have to convert each line we copy.

define `LONGEST_POSSIBLE_CENSUS_ERROR MAX_TITLING_LINE_LENGTH*2`

```
void transcribe_census_errors(OUTPUT_STREAM) {
    FILE *RESULTS;
    char path_to_errors[MAX_FILENAME_LENGTH];
    char error_line[LONGEST_POSSIBLE_CENSUS_ERROR];
    if (no_census_errors == 0) return;                                  nothing to include, then
    sprintf(path_to_errors, "%s%cReserved%cCensusErrors.txt",
        pathname_of_extensions, FOLDER_SEPARATOR, FOLDER_SEPARATOR);
    RESULTS = iso_fopen(path_to_errors, "r");
    if (RESULTS == NULL) fatal_error2("Unable to read census error log", path_to_errors);
    ⟨Include the headnote explaining what census errors are 25⟩;
    while (truncated_iso_fgets(RESULTS, error_line, LONGEST_POSSIBLE_CENSUS_ERROR) >= 1) {
        WRITE("%s", error_line);
        WRITE("\n");
    }
    fclose(RESULTS);
}
```

§**25.** We only want to warn people here: not to stop them from using Inform until they put matters right. (Suppose, for instance, they are using an account not giving them sufficient privileges to modify files in the external extensions area: they'd then be locked out if anything was amiss there.)

⟨Include the headnote explaining what census errors are 25⟩ ≡

```
WRITE("<p><img border=0 src=inform:/doc_images/census_problem.png> "
    "<b>Warning</b>. Inform checks the folder of user-installed extensions "
    "each time it translates the source text, in order to keep this directory "
    "page up to date. Each file must be a properly labelled extension (with "
    "its titling line correctly identifying itself), and must be in the right "
    "place - e.g. 'Marbles by Daphne Quilt' must have the filename 'Marbles' "
    "(with no file extension) and be stored in the folder 'Daphne Quilt'. "
    "The title should be at most %d characters long; the author name, %d. "
    "At the last check, these rules were not being followed:<p>",
        MAX_EXTENSION_TITLE_LENGTH, MAX_EXTENSION_AUTHOR_LENGTH);
```

This code is used in §24.

§**26.** Here we write the copy for the directory page of the extensions documentation: the one which the user currently sees by clicking on the "Installed Extensions" link from the contents page of the documentation. It contains an alphabetised catalogue of extensions by author and then title, along with some useful information about them, and then a list of any oddities found in the external extensions area.

```
void write_extensions_census_to_HTML_file(OUTPUT_STREAM) {
    int key_vms = FALSE, key_override = FALSE, key_builtin = FALSE, key_bullet = FALSE;
    int initial_letter;
    warn_about_census_errors(OUT);
    for (initial_letter = 0x20; initial_letter < 0x100; initial_letter++) {
        extension_census_datum *ecd;
        char *current_author_name = "";
        for (ecd = extension_census_results[initial_letter]; ecd; ecd = ecd->next) {
            if (strcmp(current_author_name, ecd->ecd_id.author_name) != 0) {
                current_author_name = ecd->ecd_id.author_name;
                WRITE("<p><b>%s</b><br>", ecd->ecd_id.raw_author_name);
            }
            ⟨Print the census line for this extension 27⟩;
        }
    }
    ⟨Print the key to any symbols used in the census lines 29⟩;
    transcribe_census_errors(OUT);
}
```

The function write_extensions_census_to_HTML_file is called from 4/ext.

§**27.** The appearance of the line is

(bullet) The Title (icon) (VM requirement icons) Version (number)

where all is optional except the title part. The bullet appears if the extension has been loaded on this run of NI, or is in the dictionary as a result of having been loaded on some previous run of NI: it therefore indicates that this an extension the user sometimes uses.

```
define UNINDEXED_SYMBOL "<img border=\"0\" src=\"inform:/doc_images/unindexed_bullet.png\">"
define INDEXED_SYMBOL "<img border=\"0\" src=\"inform:/doc_images/indexed_bullet.png\">"
define BUILT_IN_SYMBOL "<img border=\"0\" src=\"inform:/doc_images/builtin_ext.png\">"
define OVERRIDING_SYMBOL "<img border=\"0\" src=\"inform:/doc_images/override_ext.png\">"
```

⟨Print the census line for this extension 27⟩ ≡

```
    char *bulletornot = UNINDEXED_SYMBOL;
    if ((no_times_EID_used_in_context(&(ecd->ecd_id), LOADED_EIDBC) > 0) ||
        (no_times_EID_used_in_context(&(ecd->ecd_id), DICTIONARY_REFERRED_EIDBC) > 0)) {
        bulletornot = INDEXED_SYMBOL; key_bullet = TRUE;
    }
    WRITE("    %s", bulletornot);
    begin_extension_link(OUT, &(ecd->ecd_id), ecd->rubric);
    WRITE("<font color=\"#404040\">%s</font>", ecd->ecd_id.title);
    end_extension_link(OUT, &(ecd->ecd_id));
    if (ecd->built_in) {
        WRITE(" %s", BUILT_IN_SYMBOL); key_builtin = TRUE; }
    if (ecd->overriding_a_built_in_extension) {
        WRITE(" %s", OVERRIDING_SYMBOL); key_override = TRUE; }
    if (ecd->VM_requirement[0]) {
        ⟨Append icons which signify the VM requirements of the extension 28⟩;
        key_vms = TRUE; }
    if (ecd->version_text[0])
        WRITE("   <small>Version %s</small> ", ecd->version_text);
    WRITE("<br>");
```

This code is used in §26.

§**28.** VM requirements are parsed by feeding them into the lexer and calling the same routines as would be used when parsing headings about VM requirements in a normal run of NI. Note that because the requirements are in round brackets, which the lexer will split off as distinct words, we can ignore the first and last word and just look at what is in between:

⟨Append icons which signify the VM requirements of the extension 28⟩ ≡

```
    int rw1 = lexer_wordcount, rw2;
    feed_into_lexer(ecd->VM_requirement, FALSE, FALSE);
    rw2 = lexer_wordcount - 1;
    feed_into_lexer("Another firebreak. ", FALSE, FALSE);
    WRITE(" ");
    write_VM_icons(OUT, rw1+1, rw2-1);
```

This code is used in §27.

§**29.** The key at the foot only explicates those symbols actually used, and doesn't explicate the "unindexed" symbol at all, since that's actually just a blank image used for horizontal spacing to keep margins straight.

⟨Print the key to any symbols used in the census lines 29⟩ ≡

```
if ((key_builtin) || (key_override) || (key_bullet) || (key_vms)) {
    WRITE("<p>Key: ");
    if (key_bullet) WRITE("%s Used and indexed ", INDEXED_SYMBOL);
    if (key_builtin) WRITE("%s Built in to Inform ", BUILT_IN_SYMBOL);
    if (key_override) WRITE("%s User-installed version overrides "
        "the one built in to Inform ", OVERRIDING_SYMBOL);
    if (key_vms) {
        WRITE("<br>");
        write_VM_key(OUT);
    }
}
```

This code is used in §26.

Extension Dictionary 4/edict

Purpose

To maintain a database of names and constructions in all extensions so far used by this installation of Inform, and spot potential namespace clashes.

Definitions

¶1. Not during the census, but rather when extensions are successfully used, a dictionary is kept of the terms defined in them: this is used to generate the dynamic documentation on installed extensions, and is stored between runs in a cache file inside the I7 application. This means dictionary entries are first read in from the cache; then the entries for any extension used by NI in its current run are revised, which may mean deleting some entries or adding new ones; and at last NI writes the final state of the dictionary back to the cache. In this way, changes in what an extension defines are reflected in the dictionary after each successful use of that extension.

```
define MAX_ED_CATEGORY_LENGTH 32
define MAX_ED_HEADWORD_LENGTH 128

typedef struct extension_dictionary_entry {
    struct extension_identifier ede_id;                          author name and title, with hash code
    char entry_text[MAX_ED_HEADWORD_LENGTH];                     text of the dictionary entry
    char sorting[MAX_ED_HEADWORD_LENGTH + 10];                   text reprocessed for sorting purposes
    char type[MAX_ED_CATEGORY_LENGTH];                           grammatical category, such as "kind"
    int erased;                                                  marked to be erased
    struct extension_dictionary_entry *next_in_sorted_dictionary;  link in linked list
    MEMORY_MANAGEMENT
} extension_dictionary_entry;

extension_dictionary_entry *first_in_sorted_dictionary = NULL;
```

The structure extension_dictionary_entry is private to this section.

¶2. Clashes occur if, say, two extensions define "chopper" as a kind of vehicle (for instance, meaning a helicopter in one and a motorcycle in the other). This results in two dictionary entries under "chopper" and is recorded as a clash between them. Often, more will turn up: perhaps "chopper" might elsewhere mean a butchery tool. In the event of 3 or more clashing entries, $A, B, C, ...$, a linked list of ordered pairs $(A, B), (A, C), ...$ is maintained where in each pair the first term (the left one) is from an extension lexicographically earlier than the second (the right one): see below.

```
typedef struct known_extension_clash {
    int first_known;                                  heads a linked list of clashes with a given ede1
    struct known_extension_clash *next;               next in linked list of clashes
    struct extension_dictionary_entry *leftx;         clash is between this entry...
    struct extension_dictionary_entry *rightx;        ...and this one
    int number_clashes;                               number of entries clashing between ede1 and ede2
    MEMORY_MANAGEMENT
} known_extension_clash;
```

The structure known_extension_clash is private to this section.

§**1.** The extension dictionary has no natural order as such. In order to generate the dictionary page of the documentation, we will sort it alphabetically, but it is not alphabetically stored either in memory or in its serialised form on disc. (It might seem advantageous, since we're going to sort it anyway, to use the sorted ordering when saving it back to disc, as at least the structure will then be nearly sorted most of the time: but in fact the reverse is true, because we will sort using the C library's implementation of quicksort, an algorithm whose worst-case performance is on nearly sorted lists.)

§**2.** The following sample is an extract of the dictionary in its serialised form on disc. The four columns are author, title, headword and category. The special entries with headword `---` and category "indexing" are simply markers that the extension in question is indexed in the dictionary.

Note that the stroke character is illegal in unquoted Inform source text, and therefore also in excerpts with meanings, in extension titles and in author names. It can therefore safely be used as a record divider.

In December 2007, the dictionary file of a user who had employed 155 different extensions (by 33 different authors) contained 2223 entries, the longest of which formed a line 95 characters long: the most prolific extension made 380 definitions. The total file size was about 130K. Some typical entries:

```
...
|Emily Short|Plurality|---|indexing|
|Emily Short|Plurality|prior named noun|value|
|Emily Short|Plurality|ambiguously plural|property|
|Emily Short|Plurality|ordinarily enumerated|property|
|Emily Short|Locksmith|---|indexing|
|Emily Short|Locksmith|passkey|kind|
|Emily Short|Locksmith|keychain|kind|
...
```

§**3.** The file is encoded as ISO Latin-1 and can in principle have any of `0A`, `0D`, `0A 0D` or `0D 0A` as line divider. Each line must be no longer than the following number of characters minus 1:

```
define MAX_ED_LINE_LENGTH 512
```

§**4.** The following logs the dictionary as it stands in memory, in a similar format but also recorded the erasure flag.

```
void log_extension_dictionary_entry(extension_dictionary_entry *ede) {
    LOG("ede: %4d %d |%s|%s|%s|%s|\n", ede->allocation_id,
        ede->erased, ede->ede_id.author_name, ede->ede_id.title,
        ede->entry_text, ede->type);
}

void log_extension_dictionary(void) {
    extension_dictionary_entry *ede;
    int n=0;
    LOGIF(EXTCENSUS, "Extension dictionary:\n");
    LOOP_OVER(ede, extension_dictionary_entry) {
        n++; LOGIF(EXTCENSUS, "$d", ede);
    }
    if (n==0) LOGIF(EXTCENSUS, "no entries\n");
}
```

The function log_extension_dictionary_entry is called from 2/dl.

§**5. Erasing entries.** The erasure flag is used to mark entries in the dictionary which are to be erased, in that they will not survive when we save it back from memory to disc. (Entries are never physically deleted from the memory structures.)

There are two reasons to erase entries. First, the following routine sets the erased flag for dictionary entries corresponding to an extension which, according to the census returns, is no longer installed. (This can happen if the user has uninstalled an extension since the last time NI successfully ran.)

```
void erase_entries_of_uninstalled_extensions(void) {
    extension_dictionary_entry *ede;
    LOGIF(EXTCENSUS, "Erasure of dictionary entries for uninstalled extensions\n");
    LOOP_OVER(ede, extension_dictionary_entry)
        if ((ede->erased == FALSE) &&
            (no_times_EID_used_in_context(&(ede->ede_id), INSTALLED_EIDBC) == 0)) {
            ede->erased = TRUE;
            LOGIF(EXTCENSUS, "Erased $d", ede);
        }
}
```

§**6.** The second reason arises when we are making the dictionary entries for an extension which was used on the current run. (For instance, if it created a kind of vehicle called "dragster", then we will make a dictionary entry for that.) Before making its dictionary entries, we first erase all entries for the same extension which are left in the dictionary from some previous run of NI, as those are now out of date.

```
void erase_entries_of_this_loaded_extension(extension_file *ef) {
    extension_dictionary_entry *ede;
    LOGIF(EXTCENSUS, "Erasure of dictionary entries for $x\n", ef);
    LOOP_OVER(ede, extension_dictionary_entry)
        if ((ede->erased == FALSE) &&
            (eid_match(&(ede->ede_id), &(ef->ef_id)))) {
            ede->erased = TRUE;
            LOGIF(EXTCENSUS, "Erased $d", ede);
        }
}
```

The function erase_entries_of_this_loaded_extension is called from 4/edoc.

§**7. Making new entries.** We provide two ways to add a new entry: from a C string or from a word range.

```
void new_dictionary_entry(char *category, extension_file *ef, int w1, int w2) {
    char headword[MAX_ED_HEADWORD_LENGTH];
    if ((w1<0) || (w2<w1)) return;                    a safety precaution: never index the empty text
    print_raw_text_to_string_truncated(w1, w2, headword, MAX_ED_HEADWORD_LENGTH);
    new_dictionary_entry_raw(category, ef->ef_id.author_name, ef->ef_id.title, headword);
}

void new_dictionary_entry_str(char *category, extension_file *ef, char *headword) {
    new_dictionary_entry_raw(category, ef->ef_id.author_name, ef->ef_id.title, headword);
}

void new_dictionary_entry_raw(char *category, char *author, char *title, char *headword) {
    extension_dictionary_entry *ede;
    ede = CREATE(extension_dictionary_entry);
    eid_new(&(ede->ede_id), author, title, DICTIONARY_REFERRED_EIDBC);
```

```
    truncated_strcpy(ede->entry_text, headword, MAX_ED_HEADWORD_LENGTH);
    truncated_strcpy(ede->type, category, MAX_ED_CATEGORY_LENGTH);
    ede->erased = FALSE;
    ede->next_in_sorted_dictionary = NULL;
    LOGIF(EXTCENSUS, "Created $d", ede);
}
```

The function new_dictionary_entry is called from 4/edoc.
The function new_dictionary_entry_str is called from 2/lexi and 4/edoc.

§**8. Loading from disc.** Not a surprising routine: open, convert one line at a time to dictionary entries, close.

```
void load_extension_dictionary(void) {
    FILE *DICTF; FILE *EMPTY_DICTF;
    char line_entry[MAX_ED_LINE_LENGTH];
    ⟨Open the serialised extensions dictionary file for reading 9⟩;
    LOGIF(EXTCENSUS, "Reading dictionary file\n");
    while (truncated_iso_fgets(DICTF, line_entry, MAX_ED_LINE_LENGTH-1) >= 1)
        ⟨Make dictionary entry from single line from the dictionary file 10⟩;
    LOGIF(EXTCENSUS, "Finished reading dictionary file\n");
    fclose(DICTF);
}
```

The function load_extension_dictionary is called from 4/ext.

§**9.** The extension dictionary file is stored as `Dictionary.txt` in the folder for the `Reserved` "author" in the external extensions area. If it doesn't exist, we try to make an empty one; as it may happen that the `Reserved` doesn't exist either, we try to make that too. Should these attempts to write to the external area fail, we simply return: there might be permissions reasons, and it doesn't matter too much if the dictionary isn't read. A fatal error results only if, having written the empty file, we are then unable to open it again: that must mean a file I/O error of some kind, which is bad enough news to bother the user with.

⟨Open the serialised extensions dictionary file for reading 9⟩ ≡

```
    char path_to_dictionary[MAX_FILENAME_LENGTH];
    sprintf(path_to_dictionary, "%s%cReserved%cDictionary.txt",
        pathname_of_extensions, FOLDER_SEPARATOR, FOLDER_SEPARATOR);
    DICTF = iso_fopen(path_to_dictionary, "r");
    if (DICTF == NULL) {
        if (verify_library_folder("Inform", "Extensions", NULL, "Reserved") == 0)
            return;
        LOGIF(EXTCENSUS, "Creating new empty dictionary file\n");
        EMPTY_DICTF = iso_fopen(path_to_dictionary, "w");
        if (EMPTY_DICTF == NULL) return;
        fclose(EMPTY_DICTF);
        DICTF = iso_fopen(path_to_dictionary, "r");
        if (DICTF == NULL)
            fatal_error2("Unable to open dictionary file", path_to_dictionary);
    }
```

This code is used in §8.

§**10.** We parse lines in a fairly forgiving way. Material before the initial stroke is ignored (this helps us cope with any spare newline characters if there are blank lines, or if the line division is multi-byte); material after the final stroke is also ignored, and any line not containing five vertical strokes (i.e., four stroke-divided columns) is ignored altogether. This means that any truncated, overlong lines are ineffectual but safe.

⟨Make dictionary entry from single line from the dictionary file 10⟩ ≡

```
int strokes = 0, pos = 0, inpos = 0;
char author[MAX_ED_LINE_LENGTH], title[MAX_ED_LINE_LENGTH];
char headword[MAX_ED_LINE_LENGTH], category[MAX_ED_LINE_LENGTH];
author[0] = 0; title[0] = 0; headword[0] = 0; category[0] = 0;
while (strokes <= 4) {
    if ((line_entry[pos] == 0) || (inpos >= MAX_ED_LINE_LENGTH-1)) break;
    if (line_entry[pos] == '|') {
        pos++; strokes++; inpos = 0; continue;
    }
    switch(strokes) {
        case 1: author[inpos] = line_entry[pos]; author[++inpos] = 0; break;
        case 2: title[inpos] = line_entry[pos]; title[++inpos] = 0; break;
        case 3: headword[inpos] = line_entry[pos]; headword[++inpos] = 0; break;
        case 4: category[inpos] = line_entry[pos]; category[++inpos] = 0; break;
    }
    pos++;
}
if (strokes >= 5) new_dictionary_entry_raw(category, author, title, headword);
```

This code is used in §8.

§**11. Saving to disc.** And inversely...

```
void write_dictionary_back(void) {
    extension_dictionary_entry *ede;
    STREAM DICTF_struct;
    STREAM *DICTF = &DICTF_struct;
    char path_to_dictionary[MAX_FILENAME_LENGTH];
    if (verify_installed_extensions_tree()) return;
    sprintf(path_to_dictionary, "%s%cReserved%cDictionary.txt",
        pathname_of_extensions, FOLDER_SEPARATOR, FOLDER_SEPARATOR);
    if (STREAM_OPEN_TO_FILE(DICTF, path_to_dictionary, UTF8_ENC) == FALSE) return;
    LOGIF(EXTCENSUS, "Writing dictionary file\n");
    LOOP_OVER(ede, extension_dictionary_entry)
        if (ede->erased == FALSE) {
            LOGIF(EXTCENSUS, "Writing $d", ede);
            ⟨Write line to the dictionary file from single entry 12⟩;
        } else LOGIF(EXTCENSUS, "Suppressing $d\n", ede);
    LOGIF(EXTCENSUS, "Finished writing dictionary file\n");
    STREAM_CLOSE(DICTF);
}
```

The function write_dictionary_back is called from 4/ext.

§**12.** We needn't worry overmuch about exceeding the maximum length, since any such lines are handled safely by the loading code above. In any case, they could only occur if it were true that

```
4 + MAX_ED_CATEGORY_LENGTH + MAX_ED_HEADWORD_LENGTH +
    MAX_EXTENSION_TITLE_LENGTH + MAX_EXTENSION_AUTHOR_LENGTH >= MAX_ED_LINE_LENGTH
```

and this is not nearly the case. (`MAX_ED_LINE_LENGTH` is larger than strictly necessary since it costs us only temporary stack space and allows for any future increase of the above maxima without fuss.)

⟨Write line to the dictionary file from single entry 12⟩ ≡

```
STREAM_WRITE(DICTF, "|%s|%s|%s|%s|\n",
    ede->ede_id.author_name, ede->ede_id.title,
    ede->entry_text, ede->type);
```

This code is used in §11.

§**13. Sorting the extension dictionary.** We pass this job on to the standard C library `qsort`, in hopes that it is reasonably efficiently implemented. We need to bear in mind that the extensions database can be expected to have some thousands of entries, and that the $O(n^2)$ insertion sorts used so casually elsewhere in NI – where lists are certainly much smaller – could cause misery here.

This routine returns the number of (unerased) entries in the dictionary, and on its exit the (unerased) entries each occur once in alphabetical order in the linked list beginning at `first_in_sorted_dictionary`. If two entries have identical headwords, the earliest created is the one which appears earlier in the sorted dictionary.

```
int sort_extension_dictionary(void) {
    extension_dictionary_entry **sorted_extension_dictionary = NULL;
    int no_entries = 0;
    LOGIF(EXTCENSUS, "Beginning dictionary sort\n");
    sorted_extension_dictionary = NULL;
    ⟨Count headwords and reprocess their texts for dictionary sorting 14⟩;
    if (no_entries == 0) {
        first_in_sorted_dictionary = NULL;
        return 0;
    }
    ⟨Allocate memory for, and fill, an array of pointers to the EDEs 15⟩;
    qsort(sorted_extension_dictionary, no_entries, sizeof(extension_dictionary_entry *),
        compare_ed_entries);
    ⟨String the sorted array together into a sorted linked list of EDEs 16⟩;
    ⟨Deallocate memory for the array again 17⟩;
    LOGIF(EXTCENSUS, "Sorted dictionary: %d entries\n", no_entries);
    return no_entries;
}
```

§**14.** Dictionary entries must be in mixed case: we might have both "green" the colour and "Green" the kind of person (an environmental activist), say. But we want to compare them with `strcmp`, which is much faster than its case-insensitive analogue. So we trade memory for speed and store a modified form of the headword in which spaces are removed and letters are reduced to lower case; note that this is no larger than the original, so there is no risk of the `sorting` string (which is 10 characters longer than the unprocessed version) overflowing. Note: later we shall rely on the first character of the sorting text being the lower-case form of the first character of the original word.

We then append the allocation ID number, padded with initial zeros. We do this so that (i) all sorting texts will be distinct, and (ii) alphabetical order for sorting texts derived from two identical headword texts will correspond to creation order. This means that `qsort`'s output will be predictable (whereas different implementations of Quicksort might use the freedom to sort unstably in different ways), and this seems a good idea for better testing.

⟨Count headwords and reprocess their texts for dictionary sorting 14⟩ ≡

```
extension_dictionary_entry *ede;
int i, j;
LOOP_OVER(ede, extension_dictionary_entry)
    if (ede->erased == FALSE) {
        no_entries++;
        strcpy(ede->sorting, ede->entry_text);
        for (i=0, j=0; ede->entry_text[j]; j++)
            if (ede->entry_text[j] != ' ')
                ede->sorting[i++] = tolower(ede->entry_text[j]);
        ede->sorting[i] = 0;
        sprintf(ede->sorting + i, "-%09d", ede->allocation_id);
        LOGIF(EXTCENSUS, "Sorted under '%s': $d", ede->sorting, ede);
    }
```

This code is used in §13.

§**15.** We unbundle the linked list of EDEs in creation order into an array temporarily allocated in memory:

⟨Allocate memory for, and fill, an array of pointers to the EDEs 15⟩ ≡

```
extension_dictionary_entry *ede;
int i = 0;
sorted_extension_dictionary = I7_calloc(no_entries,
    sizeof(extension_dictionary_entry *), EXTENSION_DICTIONARY_MREASON);
LOOP_OVER(ede, extension_dictionary_entry) {
    if (ede->erased == FALSE)
        sorted_extension_dictionary[i++] = ede;
}
```

This code is used in §13.

§**16.** We then use the sorted version of the same array to reorder the EDEs:

⟨String the sorted array together into a sorted linked list of EDEs 16⟩ ≡

```
int i;
first_in_sorted_dictionary = sorted_extension_dictionary[0];
for (i=0; i<no_entries-1; i++)
    sorted_extension_dictionary[i]->next_in_sorted_dictionary =
        sorted_extension_dictionary[i+1];
if (no_entries > 0)
    sorted_extension_dictionary[no_entries-1]->next_in_sorted_dictionary = NULL;
```

This code is used in §13.

§**17.** And for the sake of tidiness:

⟨Deallocate memory for the array again 17⟩ ≡
```
I7_free(sorted_extension_dictionary, EXTENSION_DICTIONARY_MREASON);
```

This code is used in §13.

§**18.** As always with `qsort`, there's a palaver about the types used for the comparison function so that the result will compile without errors. The comparison of two EDEs is in fact delegated to a `strcmp` comparison of their sorting texts:

```
int compare_ed_entries(const void *elem1, const void *elem2) {
    const extension_dictionary_entry **e1 = (const extension_dictionary_entry **) elem1;
    const extension_dictionary_entry **e2 = (const extension_dictionary_entry **) elem2;
    if ((*e1 == NULL) || (*e2 == NULL))
        internal_error("Disaster while sorting extension dictionary");
    return strcmp((*e1)->sorting, (*e2)->sorting);
}
```

§**19. Extension clashes.** All Inform extensions included share the main name-space of the source text, and this causes potential problems with name clashes between two different extensions if ever an author wants to include both at once. To try to detect these clashes, we automatically scan the dictionary for them, and provide warnings on the dynamic extension index.

```
known_extension_clash *kec_new(extension_dictionary_entry *L, extension_dictionary_entry *R,
    int first_known_flag) {
    known_extension_clash *kec = CREATE(known_extension_clash);
    kec->leftx = L;
    kec->rightx = R;
    kec->number_clashes = 1;
    kec->first_known = first_known_flag;
    kec->next = NULL;
    return kec;
}
```

§**20.** Every clash of names arises from definitions made in a pair of EDEs, which we shall call left and right. Each distinct KEC ("known extension clash") represents a different pair of extensions which clash, one example of a name clashing between them, and a count of the number of such names.

(a) Given a pair of extensions, the left one is the one whose author name followed by title is lexicographically earlier. Since we are only concerned with clashes between different extensions, this unambiguously decides which is leftmost, as title and author suffice to identify extensions.

(b) Similarly, given a pair of EDEs, the left one is the one whose definition arises from the lefthand extension. (So, for instance, any definition made in one of Eric Eve's extensions is always to the left of any definition in one of John Clemens's.) Different EDEs deriving from the same extension do not exemplify a clash.

(c) For each extension L, there is at most one KEC whose left EDE derives from L and which has the "first known" flag set.

(c.1) If such a KEC does not exist, then L does not clash with any other extension.

(c.2) If such a KEC does exist, then it is the head of a linked list of KECs all of which have lefthand EDE deriving from L, and in which no two entries have righthand EDEs deriving from the same extension as each other.

It follows that we can determine if extensions X and Y clash by arranging them as L and R (rule (a)), looking for L among the left EDEs of all KECs with the "first known" flag set (rule (c)), and then looking for Y among the right EDEs of all KECs in the list which hangs from that (rule (c.2)). Should either of these searches fail, there is no clash between X and Y. Should both succeed, then the KEC found provides a single example of the clash (in its left and right EDEs), together with the number of clashes.

If there are n extensions then there could in theory be $n(n-1)/2$ KECs, which might amount to a lot of storage. In practice, though, Inform source text tends to be dispersed around the cloud of English nouns and adjectives fairly well, and since extension authors use each other's extensions, there is also some social pressure to reduce the number of clashes. The user mentioned above who had installed 155 different extensions – for a possible 11,935 distinct clashing pairs – in fact observed 15 such pairs, mostly arising from part-finished drafts which had borrowed source text from pieces of other extensions. Of the few remaining, several were cases where the same name occurred in rival extensions aspiring to do much the same thing as each other: for instance, "current quip" was defined by two different conversation extensions. The only clashes of different meanings which might both be needed, and which seem to have arisen spontaneously, were from definitions of the words "seen" and "implicit", both treacherously ambiguous. Clashes did not seem to have arisen from homonyms like "lead" (the substance) versus "lead" (the attachment to a collar).

```
void extension_clash(extension_dictionary_entry *ede1, extension_dictionary_entry *ede2) {
    extension_dictionary_entry *left = NULL, *right = NULL;
    extension_identifier *leftx, *rightx;
    known_extension_clash *kec;
    if ((ede1 == NULL) || (ede2 == NULL)) internal_error("bad extension clash");
    int d = eidcmp(&(ede1->ede_id), &(ede2->ede_id));                    compare source extensions
    ⟨Ignore apparent clashes which are in fact not troublesome 21⟩;
    if (d < 0) { left = ede1; right = ede2; }
    if (d > 0) { left = ede2; right = ede1; }
    leftx = &(left->ede_id); rightx = &(right->ede_id);
    LOOP_OVER(kec, known_extension_clash)
        if ((kec->first_known) && (eid_match(leftx, &(kec->leftx->ede_id)))) {
            ⟨Search list of KECs deriving from the same left extension as this clash 22⟩;
            return;
        }
    kec = kec_new(left, right, TRUE);
}
```

§**21.** If two name clashes occur in the same extension then, since we can presume that this extension does actually work, the clash cannot cause problems. We also ignore a clash of a property name against some other form of name, because these occur quite often and cause little difficulty in practice: so they would only clutter up the dictionary with spurious warnings.

⟨Ignore apparent clashes which are in fact not troublesome 21⟩ ≡

```
    if (d == 0) return;                                  both definitions come from the same extension
    if ((strcmp(ede1->type, "property") == 0) && (strcmp(ede2->type, "property") != 0)) return;
    if ((strcmp(ede1->type, "property") != 0) && (strcmp(ede2->type, "property") == 0)) return;
```

This code is used in §20.

§**22.** If we can find the righthand extension on the righthand side of any KEC in the list, then the clash is not a new one: we simply increment the number of definition pairs clashing between the left and right extensions, and return. (Thus forgetting what the actual definitions causing the present clash were: we don't need them, as we already have an example of the definitions clashing between the two.) But if we can't find righthand extension anywhere in the list, we must add the new pair of definitions:

⟨Search list of KECs deriving from the same left extension as this clash 22⟩ ≡

```
while (kec) {
    if (eid_match(rightx, &(kec->rightx->ede_id))) {
        kec->number_clashes++; return;
    }
    if (kec->next == NULL) {
        kec->next = kec_new(left, right, FALSE); return;
    }
    kec = kec->next;
}
```

This code is used in §20.

§**23.** The above arrangement was designed to make it easy to print out the clashes in a concise, human-readable way, which is what we now do.

```
void list_known_extension_clashes(OUTPUT_STREAM) {
    known_extension_clash *kec;
    if (NUMBER_CREATED(known_extension_clash) == 0) return;
    ⟨Write the headnote about what extension clashes mean 24⟩;
    LOOP_OVER(kec, known_extension_clash)
        if (kec->first_known)
            ⟨Write a paragraph about extensions clashing with the lefthand one here 25⟩;
}
```

§**24.** Not the end of the world! Extension clashes are not an error condition: they are, if anything, a sign of life and activity.

⟨Write the headnote about what extension clashes mean 24⟩ ≡

```
WRITE("<p>\n<b>Clashes found.</b> The dictionary above shows that some "
    "extensions make incompatible definitions of the same words or phrases. "
    "When two extensions disagree like this, it is not necessarily a bad "
    "sign (they might simply be two ways to approach the same problem), "
    "but in general it means that it may not be safe to use both "
    "extensions at the same time. The following list shows some potential "
    "clashes.<p>\n");
```

This code is used in §23.

§**25.** As always, we need to be careful about writing the ISO text of clashing matter to the UTF-8 HTML file:

⟨Write a paragraph about extensions clashing with the lefthand one here 25⟩ ≡

```
known_extension_clash *example;
WRITE("<b>");
eid_write_to_HTML_file(OUT, &(kec->leftx->ede_id), FALSE);
WRITE("</b>: ");
for (example = kec; example; example = example->next) {
    WRITE("clash with <b>");
    eid_write_to_HTML_file(OUT, &(example->rightx->ede_id), FALSE);
    WRITE("</b>");
    if (example->number_clashes > 1)
        WRITE(" (on %d names, for instance ", example->number_clashes);
    else WRITE(" (on ");
    print_literal_string_to_file(OUT, example->leftx->entry_text);
    WRITE(")");
    if (example->next) WRITE("; ");
}
WRITE("<p>\n");
```

This code is used in §23.

§**26. Writing the HTML extension index.** This is the index of terms, not the directory of extensions: it is, in fact, the HTML rendering of the dictionary constructed above.

```
void write_extension_dictionary_to_HTML_file(OUTPUT_STREAM) {
    int n;
    char first_letter = 'a';
    extension_dictionary_entry *ede, *previous_ede, *next_ede;
    erase_entries_of_uninstalled_extensions();
    n = sort_extension_dictionary();
    if (n <= 0) return;
    for (previous_ede = NULL, ede = first_in_sorted_dictionary; ede;
        previous_ede = ede, ede = ede->next_in_sorted_dictionary) {
        if (strcmp(ede->type, "indexing") == 0) continue;
        next_ede = ede->next_in_sorted_dictionary;
        char this_first = tolower(ede->entry_text[0]);
        if (first_letter != this_first) {
            WRITE("<br>"); first_letter = this_first;
        }
        ⟨Write extension dictionary entry for this headword 27⟩;
    }
    list_known_extension_clashes(OUT);
}
```

The function write_extension_dictionary_to_HTML_file is called from 4/ext.

§**27.** A run of N words which are all the same should appear in tinted type throughout, while $N(N-1)/2$ clashes should be reported to the machinery above: if we find definitions A, B, C, for instance, the clashes are reported as A vs B, A vs C, then B vs C. This has $O(N^2)$ running time, so if there are 1000 extensions, each of which gives 1000 different meanings to the word "frog", we would be in some trouble here. Let's take the risk.

```
define TINT_FOR_SAME_WORD "#FF8080"
define EDES_DEFINE_SAME_WORD(X, Y) ((X) && (Y) && (strcmp(X->sorting, Y->sorting) == 0))
```

⟨Write extension dictionary entry for this headword 27⟩ ≡

```
int tint = FALSE;
if (EDES_DEFINE_SAME_WORD(ede, previous_ede)) tint = TRUE;
while (EDES_DEFINE_SAME_WORD(ede, next_ede)) {
    tint = TRUE;
    extension_clash(ede, next_ede);
    next_ede = next_ede->next_in_sorted_dictionary;
}
if (tint) WRITE("<font color=\"%s\">", TINT_FOR_SAME_WORD);
print_literal_string_to_file(OUT, ede->entry_text);
if (tint) WRITE("</font>");
WRITE(" - <i>%s</i>   <small>", ede->type);
eid_write_link_to_HTML_file(OUT, &(ede->ede_id));
WRITE("</small><br>");
```

This code is used in §26.

Extension Documentation 4/edoc

Purpose

To generate HTML documentation for extensions.

---

§**1.** Each extension gets its own page in the external documentation area, but this page can have two forms: the deluxe version, only produced if an extension is successfully used, and a cut-down placeholder version, used if NI has detected the extension but never used it (and so does not really understand what it entails). The following routine writes both kinds of page.

```
void write_detailed_extension_documentation(extension_file *ef) {
    write_extension_documentation(NULL, ef);
}
void write_sketchy_extension_documentation(extension_census_datum *ecd) {
    write_extension_documentation(ecd, NULL);
}
```

The function write_detailed_extension_documentation is called from 4/ext.

The function write_sketchy_extension_documentation is called from 4/ext.

§**2.** Thus we pass two arguments, `ecd` and `ef`, to `write_extension_documentation`: one is a valid pointer, the other null. If `ef` is valid, we can write a full page: if `ecd` is valid, only a sketchy one.

The outer shell routine calls the inner one first to generate the main page of the documentation (where `eg_number` is `-1`), then uses its return value (the number of examples provided, which may be 0) to generate associated files for each example. For instance, we might end up making, in sequence,

```
Documentation/Extensions/Emily Short/Locksmith.html
Documentation/Extensions/Emily Short/Locksmith-eg1.html
Documentation/Extensions/Emily Short/Locksmith-eg2.html
Documentation/Extensions/Emily Short/Locksmith-eg3.html
Documentation/Extensions/Emily Short/Locksmith-eg4.html
```

where these are pathnames relative to the external resources area.

```
void write_extension_documentation(extension_census_datum *ecd, extension_file *ef) {
    int c, eg_count;
    eg_count = write_extension_documentation_page(ecd, ef, -1);
    for (c=1; c<=eg_count; c++)
        write_extension_documentation_page(ecd, ef, c);
}
```

§**3.** Here then is the nub of it. An ECD is not really enough information to go on. We are not always obliged to make a sketchy page from an ECD: we decide against in a normal run where a page exists for it already, as otherwise a user with many extensions installed would detect an annoying slight delay on every run of NI – whereas a slight delay on each census-mode run is acceptable, since census-mode runs are made only when extensions are installed or uninstalled. If we do decide to make a page from an ECD, we in fact read the extension into the lexer so as to make an EF of it. Of course, it won't be a very interesting EF – since it wasn't used in compilation there will be no definitions arising from it, so the top half of its documentation page will be vacant – but it will at least provide the extension author's supplied documentation, if there is any, as well as the correct identifying headings and requirements.

```
int write_extension_documentation_page(extension_census_datum *ecd, extension_file *ef,
    int eg_number) {
    extension_identifier *eid = NULL;
    STREAM DOCF_struct;
    STREAM *DOCF = &DOCF_struct;
    FILE *TEST_DOCF;
    int page_exists_already, no_egs = 0;
    char leafname[MAX_FILENAME_LENGTH], pathname[MAX_FILENAME_LENGTH];
    if (ecd) eid = &(ecd->ecd_id); else if (ef) eid = &(ef->ef_id);
    else internal_error("WEDP incorrectly called");
    LOGIF(EXTCENSUS, "WEDP %s (%s by %s)/%d\n",
        (ecd)?"ecd":" ef", eid->title, eid->author_name, eg_number);
    ⟨Work out the leafname and pathname of the documentation file 4⟩;
    page_exists_already = FALSE;
    TEST_DOCF = iso_fopen(pathname, "r");
    if (TEST_DOCF) { page_exists_already = TRUE; fclose(TEST_DOCF); }
    LOGIF(EXTCENSUS, "WEDP %s: %s\n", (page_exists_already)?"exists":"does not exist",
        pathname);
    if (ecd) {
        if ((page_exists_already == FALSE) || (census_mode))
            ⟨Convert ECD to a text-only EF 5⟩;
        return 0;                          ensure no requests sent for further pages about the ECD: see below
    }
    if (ef == NULL) internal_error("null EF in extension documentation writer");
    if (verify_library_folder("Inform", "Documentation", "Extensions", eid->author_name) == 0)
        return 0;
    if (STREAM_OPEN_TO_FILE(DOCF, pathname, UTF8_ENC) == FALSE)
        return 0;                                  if we lack permissions, e.g., then write no documentation
    ⟨Write the actual extension documentation page to DOCF 6⟩;
    STREAM_CLOSE(DOCF);
    return no_egs;
}
```

§**4.** Straightforwardly:

⟨Work out the leafname and pathname of the documentation file 4⟩ ≡

```
    strcpy(leafname, eid->title);
    if (eg_number > 0) sprintf(leafname+strlen(leafname), "-eg%d", eg_number);
    sprintf(pathname, "%s%cExtensions%c%s%c%s.html", pathname_of_extension_docs,
        FOLDER_SEPARATOR, FOLDER_SEPARATOR, eid->author_name, FOLDER_SEPARATOR, leafname);
```

This code is used in §3.

§**5.** The reader may wonder why we perform the conversion in this slightly recursive way, by calling our parent routine again. Wouldn't it be simpler just to set `ecd` to null and let events take their course? The answer is that this would fail if there were examples, because we would return (say) 3 for the number of examples, and then the routine would be called 3 more times – but with the original ECD as argument each time: that would mean reading the file thrice more, reconverting to EF each time. So we restart the process from our EF, and return 0 in response to the ECD call to prevent any further ECD calls.

⟨Convert ECD to a text-only EF 5⟩ ≡

```
int aw1 = lexer_wordcount, aw2, tw1, tw2;
feed_into_lexer(eid->raw_author_name, FALSE, FALSE);
feed_into_lexer(" ", FALSE, FALSE);
aw2 = lexer_wordcount - 1;
tw1 = lexer_wordcount;
feed_into_lexer(eid->raw_title, FALSE, FALSE);
feed_into_lexer(" ", FALSE, FALSE);
tw2 = lexer_wordcount - 1;
feed_into_lexer("This sentence provides a firebreak, no more. ", FALSE, FALSE);
ef = load_extension(aw1, aw2, tw1, tw2, -1);
if (ef == NULL) return 0;                                   shouldn't happen: it was there only moments ago
write_extension_documentation(NULL, ef);
```

This code is used in §3.

§**6.** We now make much the same "paste into the gap in the template" copying exercise as when generating the home pages for extensions, though with a different template:

⟨Write the actual extension documentation page to DOCF 6⟩ ≡

```
char path_to_model[MAX_FILENAME_LENGTH];
char line_of_template[LONGEST_LINE_IN_EX_TEMPLATE];
FILE *TEMPLATE;
int skipping_between_markers = FALSE;
sprintf(path_to_model, "%s%cReserved%c%s",
    pathname_of_built_in_extensions, FOLDER_SEPARATOR, FOLDER_SEPARATOR,
    EXTENSION_FILE_MODEL_HTML);
TEMPLATE = iso_fopen(path_to_model, "r");
if (TEMPLATE == NULL)
    fatal_error2("Unable to open model extension documentation file for reading",
        path_to_model);
while (!(feof(TEMPLATE))) {
    int n = truncated_iso_fgets(TEMPLATE, line_of_template, LONGEST_LINE_IN_EX_TEMPLATE);
    if (n < 0) break;                                       file I/O error or end of file
    if (strcmp(line_of_template, "<a name=on>") == 0) {
        ⟨Write documentation for a specific extension into the page 7⟩;
        skipping_between_markers = TRUE;
    }
    if (skipping_between_markers == FALSE) STREAM_WRITE(DOCF, "%s", line_of_template);
    if (strcmp(line_of_template, "<a name=off>") == 0)
        skipping_between_markers = FALSE;
}
fclose(TEMPLATE);
```

This code is used in §3.

§**7.** And this is what we paste into the gap:

⟨Write documentation for a specific extension into the page 7⟩ ≡
```
STREAM_WRITE(DOCF, "<p>");
if (eid_is_standard_rules(eid) == FALSE)
    ⟨Write Javascript paste icon for source text to include this extension 8⟩;
STREAM_WRITE(DOCF, "<b>");
eid_write_to_HTML_file(DOCF, eid, TRUE);
STREAM_WRITE(DOCF, "</b><p><small>");
⟨Write up any restrictions on VM usage 9⟩;
⟨Write up the version number, if any, and location 10⟩;
STREAM_WRITE(DOCF, "</small><p>");
⟨Write up the rubric, if any 11⟩;
⟨Write up the table of contents for the extension author's supplied documentation, if any 12⟩;
⟨Document and dictionary the definitions made in extension file ef 14⟩;
STREAM_WRITE(DOCF, "<p><hr><p>");
⟨Write up the extension author's supplied documentation, if any 13⟩;
```

This code is used in §6.

§**8.** UTF-8 transcoding in the following is delegated to `write_javascript_paste`:

⟨Write Javascript paste icon for source text to include this extension 8⟩ ≡
```
char inclusion_text[MAX_EXTENSION_TITLE_LENGTH + MAX_EXTENSION_AUTHOR_LENGTH + 50];
sprintf(inclusion_text, "Include %s by %s.\n\n\n", eid->title, eid->author_name);
write_javascript_paste(DOCF, -1, -1, inclusion_text);
STREAM_WRITE(DOCF, " ");
```

This code is used in §7.

§**9.**

⟨Write up any restrictions on VM usage 9⟩ ≡
```
int rw1 = ef->VM_restriction_w1, rw2 = ef->VM_restriction_w2;
if (rw1 >= 0) {
    print_raw_text_to_file(rw1, rw2, DOCF);
    STREAM_WRITE(DOCF, " ");
    write_VM_icons(DOCF, rw1, rw2);
}
```

This code is used in §7.

§**10.**

⟨Write up the version number, if any, and location 10⟩ ≡
```
char the_version[MAX_VERSION_NUMBER_LENGTH+1];
if (ef->version_loaded >= 0)
    print_raw_text_to_string(ef->version_loaded, ef->version_loaded, the_version);
else the_version[0] = 0;
if (the_version[0]) STREAM_WRITE(DOCF, "<p>Version %s", the_version);
if (ef->loaded_from_built_in_area) {
    if (the_version[0] == 0) STREAM_WRITE(DOCF, "<p>Extension");
    STREAM_WRITE(DOCF, " built in to Inform");
}
```

This code is used in §7.

§**11.**

⟨Write up the rubric, if any 11⟩ ≡
```
if (ef->rubric_as_lexed[0])
    STREAM_WRITE(DOCF, "%s<p>", ef->rubric_as_lexed);
if (ef->extra_credit_as_lexed[0])
    STREAM_WRITE(DOCF, "<i>%s</i><p>", ef->extra_credit_as_lexed);
```
This code is used in §7.

§**12.** This appears above the definition paragraphs because it tends to be only large extensions which provide TOCs: and they, ipso facto, make many definitions. If the TOC were directly at the top of the supplied documentation, it might easily be scrolled down off screen when the user first visits the page.

⟨Write up the table of contents for the extension author's supplied documentation, if any 12⟩ ≡
```
if (ef->doc_w1 >= 0)
    edoc_set_table_of_contents(ef->doc_w1, ef->doc_w2, DOCF, leafname);
```
This code is used in §7.

§**13.**

⟨Write up the extension author's supplied documentation, if any 13⟩ ≡
```
if (ef->doc_w1 >= 0)
    no_egs = edoc_set_body_text(ef->doc_w1, ef->doc_w2, DOCF, eg_number, leafname);
else
    STREAM_WRITE(DOCF, "The extension provides no documentation.");
```
This code is used in §7.

§**14.** Nothing can prevent a certain repetitiousness intruding here, but there is just enough local knowledge required to make it foolhardy to try to automate this from a dump of the excerpt meanings table (say). The ordering of paragraphs, as in Roget's Thesaurus, tries to proceed from solid things through to diffuse linguistic ones. But the reader of the resulting documentation page could be forgiven for thinking it a miscellany.

⟨Document and dictionary the definitions made in extension file ef 14⟩ ≡
```
erase_entries_of_this_loaded_extension(ef);
new_dictionary_entry_str("indexing", ef, "---");
⟨Document and dictionary the kinds made in extension 15⟩;
⟨Document and dictionary the objects made in extension 16⟩;
⟨Document and dictionary the kinds of value made in extension 17⟩;
⟨Document and dictionary the global variables made in extension 18⟩;
⟨Document and dictionary the enumerated constant values made in extension 19⟩;
⟨Document and dictionary the kinds of action made in extension 20⟩;
⟨Document and dictionary the actions made in extension 21⟩;
⟨Document and dictionary the verbs made in extension 22⟩;
⟨Document and dictionary the adjectival phrases made in extension 23⟩;
⟨Document and dictionary the property names made in extension 24⟩;
```
This code is used in §7.

§**15.** Off we go, then. Kinds of object:

⟨Document and dictionary the kinds made in extension 15⟩ ≡

```
world_object *wo;
int kc = 0;
LOOP_OVER(wo, world_object) {
    if ((wo != kind_kind) && (wo->kind_flag)) {
        if (lw_array[wo->creating_sentence->word_ref1].lw_source.file_of_origin ==
            ef->read_into_file) {
            kc = document_headword(DOCF, kc, ef, "Kinds", "kind",
                wo->word_ref1, wo->word_ref2);
            if ((wo->kind != kind_kind) && (wo->kind != kind_thing)) {
                STREAM_WRITE(DOCF, " (a kind of ");
                print_raw_text_to_file(wo->kind->word_ref1,
                    wo->kind->word_ref2, DOCF);
                STREAM_WRITE(DOCF, ")");
            }
            if (wo->kind == kind_kind) STREAM_WRITE(DOCF, " (a kind of object)");
        }
    }
}
if (kc != 0) STREAM_WRITE(DOCF, "<p>");
```

This code is used in §14.

§**16.** Actual objects:

⟨Document and dictionary the objects made in extension 16⟩ ≡

```
world_object *wo;
int kc = 0;
LOOP_OVER(wo, world_object) {
    if ((wo->kind_flag == FALSE) &&
        (wo->creating_sentence) && (wo->word_ref1 >=0)) {
        if (lw_array[wo->creating_sentence->word_ref1].lw_source.file_of_origin ==
            ef->read_into_file) {
            char name_of_its_kind[512];
            print_raw_text_to_string(wo->kind->word_ref1,
                wo->kind->word_ref2, name_of_its_kind);
            kc = document_headword(DOCF, kc, ef, "Physical creations", name_of_its_kind,
                wo->word_ref1, wo->word_ref2);
            STREAM_WRITE(DOCF, " (a ");
            print_raw_text_to_file(wo->kind->word_ref1, wo->kind->word_ref2, DOCF);
            STREAM_WRITE(DOCF, ")");
        }
    }
}
if (kc != 0) STREAM_WRITE(DOCF, "<p>");
```

This code is used in §14.

§**17.** Kinds of value:

⟨Document and dictionary the kinds of value made in extension 17⟩ ≡

```
int i, kc = 0;
LOOP_OVER_DESIGNED_TYPE_IDS(i) {
    int w1, w2;
    type_ID_copy_name(i, &w1, &w2);
    if (lw_array[w1].lw_source.file_of_origin == ef->read_into_file)
        kc = document_headword(DOCF, kc, ef, "Kinds of value", "kind of value", w1, w2);
}
if (kc != 0) STREAM_WRITE(DOCF, "<p>");
```

This code is used in §14.

§**18.** Global variables:

⟨Document and dictionary the global variables made in extension 18⟩ ≡

```
quantity *q;
int kc = 0;
LOOP_OVER(q, quantity)
    if ((q->word_ref1 >= 0) &&
        (qty_is_a_global_variable(q)) &&
        (lw_array[q->word_ref1].lw_source.file_of_origin == ef->read_into_file) &&
        ([[q == ... understood]] == FALSE))
        kc = document_headword(DOCF, kc, ef, "Values that vary", "value",
            q->word_ref1, q->word_ref2);
if (kc != 0) STREAM_WRITE(DOCF, "<p>");
```

This code is used in §14.

§**19.** Constants:

⟨Document and dictionary the enumerated constant values made in extension 19⟩ ≡

```
quantity *q;
int kc = 0;
LOOP_OVER(q, quantity)
    if ((q->word_ref1 >= 0) &&
        (qty_is_a_variable(q) == FALSE) &&
        (kov_is_an_enumeration(qty_kind_of_value(q))) &&
        (lw_array[q->word_ref1].lw_source.file_of_origin == ef->read_into_file))
        kc = document_headword(DOCF, kc, ef, "Values", "value",
            q->word_ref1, q->word_ref2);
if (kc != 0) STREAM_WRITE(DOCF, "<p>");
```

This code is used in §14.

§**20.** Kinds of action:

⟨Document and dictionary the kinds of action made in extension 20⟩ ≡

```
named_action_pattern *nap;
int kc = 0;
LOOP_OVER(nap, named_action_pattern)
    if (lw_array[nap->word_ref1].lw_source.file_of_origin == ef->read_into_file)
        kc = document_headword(DOCF, kc, ef, "Kinds of action", "kind of action",
            nap->word_ref1, nap->word_ref2);
if (kc != 0) STREAM_WRITE(DOCF, "<p>");
```

This code is used in §14.

§**21.** Actions:

⟨Document and dictionary the actions made in extension 21⟩ ≡

```
action_name *acn;
int kc = 0;
LOOP_OVER(acn, action_name)
    if (lw_array[acn->word_ref1].lw_source.file_of_origin == ef->read_into_file)
        kc = document_headword(DOCF, kc, ef, "Actions", "action",
            acn->word_ref1, acn->word_ref2);
if (kc != 0) STREAM_WRITE(DOCF, "<p>");
```

This code is used in §14.

§**22.** Verbs (this one we delegate):

⟨Document and dictionary the verbs made in extension 22⟩ ≡

```
lexicon_list_verbs_in_file(DOCF, ef->read_into_file, ef);
```

This code is used in §14.

§**23.** Adjectival phrases:

⟨Document and dictionary the adjectival phrases made in extension 23⟩ ≡

```
adjectival_phrase *adj;
int kc = 0;
LOOP_OVER(adj, adjectival_phrase)
    if ((adj->word_ref1 >= 0) &&
        (lw_array[adj->word_ref1].lw_source.file_of_origin == ef->read_into_file))
        kc = document_headword(DOCF, kc, ef, "Adjectives", "adjective",
            adj->word_ref1, adj->word_ref2);
if (kc != 0) STREAM_WRITE(DOCF, "<p>");
```

This code is used in §14.

§**24.** Other adjectives:

⟨Document and dictionary the property names made in extension 24⟩ ≡

```
property_name *prn;
int kc = 0;
LOOP_OVER(prn, property_name)
    if ((prn->word_ref1 >= 0) &&
        (prn_is_shown_in_index(prn)) &&
        (prn_is_used_by_i6_library_only(prn) == FALSE) &&
        (lw_array[prn->word_ref1].lw_source.file_of_origin == ef->read_into_file))
        kc = document_headword(DOCF, kc, ef, "Properties", "property",
            prn->word_ref1, prn->word_ref2);
if (kc != 0) STREAM_WRITE(DOCF, "<p>");
```

This code is used in §14.

§**25.** Finally, the utility routine which keeps count (hence `kc`) and displays suitable lists, while entering each entry in turn into the extension dictionary.

```
int document_headword(OUTPUT_STREAM, int kc, extension_file *ef, char *par_heading,
    char *category, int w1, int w2) {
    if (kc++ == 0) WRITE("%s: ", par_heading);
    else WRITE(", ");
    WRITE("<b>");
    print_raw_text_to_file(w1, w2, OUT);
    WRITE("</b>");
    new_dictionary_entry(category, ef, w1, w2);
    return kc;
}
```

§**26.** And that at last brings us to a milestone: the end of the Land of Extensions. We can return to Inform's more usual concerns.

Virtual Machines 4/vm

Purpose

I7 supports a variety of virtual machines as targets. Most source text should be independent of the target VM, but sometimes numbering is needed, and this is where any VM dependencies are decided.

4/vm.§1-4 Table of supported VMs; §5-13 Parsing VM restrictions; §14 Determining a match with the current target VM; §15 Icons for virtual machines; §16 Describing the current VM; §17-25 Displaying VM restrictions

---

Definitions

¶1. We use the term "major VM" to mean one of the families we deal with: the Z-machine, for instance, is a major VM. A "minor VM" is a specific revision of this, such as version 5 of the Z-machine. The following table describes both major and minor VMs: records with version set to −1 are major, and the rest minor.

It is assumed that major VM names are single words: happily Z-machine has always traditionally been hyphenated. We also assume their names use only the plainest ASCII characters.

¶2. We store data on both major and minor VMs in the following simple structure.

The point of multiplying version numbers by 10,000 is to allow room for sub-versions in future: Glulx in particular has version numbers like 3.4.2, which we might want to represent as 30402. (At present we don't distinguish versions of Glulx because Glulx's gestalt mechanism means that it's much easier to construct story files which can cope nicely on different Glulx interpreter versions than would be the case for the Z-machine.) Given that the multiplier is larger than 1, it is impossible for −1 to be a valid version number of any VM, so this is used as a "not a version number" value.

define `VMULT 10000`

```
typedef struct VM_identifier {
    int VM_code;                                  one of the values above
    int VM_version;                               times the VMULT, or -1 to mean generic versionless VM
    char *VM_major_name;                          or NULL if not a major VM name
    char *VM_extension;                           canonical filename extension
    char *VM_blorbed_extension;                   when blorbed up
    char *VM_name;                                text to use with author
    char *VM_image;                               filename of image for icon denoting VM
    int VM_is_32_bit;                             true or false: false means 16-bit
    int VM_matches;                               true or false: computed
} VM_identifier;
```

The structure VM_identifier is private to this section.

¶3. We keep track of how array space is used in the VM, since this is in very short supply in the Z-machine. This is done purely so that we can index helpfully on the Contents index page.

```
typedef struct VM_usage_note {
    int word_ref1, word_ref2;                    name of the structure using this array space...
    char *usage_explained;                                ...or an explanation instead
    char *usage_category;                                          e.g., "relation"
    int bytes_used;                                 number of bytes (not words) given over to this
    int each_flag;                             is this a count of how many bytes per usage of something?
    MEMORY_MANAGEMENT;
} VM_usage_note;
```

The structure VM_usage_note is private to this section.

---

§**1. Table of supported VMs.** The following data determines what VMs we know about, and how they can be inferred from the present information passed by the GUI to NI at the command line - viz., the eventual file extension. The application passes this by including among the command-line switches a pair like so:

```
    -extension ulx
```

The second word must be one of the file extensions listed in the fourth column of the table of VM data below: the comparison is made case insensitively, and any initial full stop is skipped, so ".Z6" is equivalent to "z6".

```
define Z_VM 1                                    Joel Berez and Marc Blank, 1979, and later hands
define GLULX_VM 2                                                          Andrew Plotkin, 2000
define DEFAULT_TARGET_VM 3                   if no -extension is supplied, target row 3: Z-machine v8
```

```
VM_identifier table_of_VM_data[] = {
    { Z_VM,      -1, "z-machine",  NULL, "zblorb", "Z-Machine", "vm_z.png", FALSE, FALSE },
    { Z_VM, 5*VMULT, NULL,  "z5", "zblorb", "Z-Machine version 5", "vm_z5.png", FALSE, FALSE },
    { Z_VM, 6*VMULT, NULL,  "z6", "zblorb", "Z-Machine version 6", "vm_z6.png", FALSE, FALSE },
    { Z_VM, 8*VMULT, NULL,  "z8", "zblorb", "Z-Machine version 8", "vm_z8.png", FALSE, FALSE },
    { GLULX_VM,  -1, "glulx", "ulx", "gblorb", "Glulx", "vm_glulx.png", TRUE, FALSE },
    { -1,         0, NULL, NULL, NULL, NULL, FALSE, FALSE }
};
```

§**2.** At present, we infer the target virtual machine by looking at the file extension requested at the command line.

```
int target_VM;                                   an index in the above table, or -1 if unknown
void set_VM_identifier(char *file_extension) {
    char lower_case_ext[32];
    int i;
    target_VM = -1;
    if (file_extension == NULL) { target_VM = DEFAULT_TARGET_VM; return; }
    if (file_extension[0] == '.') file_extension++;
    strcpy(lower_case_ext, file_extension);
    for (i=0; lower_case_ext[i]; i++) lower_case_ext[i] = tolower(lower_case_ext[i]);
    for (i=0; table_of_VM_data[i].VM_code >= 0; i++)
        if ((table_of_VM_data[i].VM_extension) &&
            (strcmp(lower_case_ext, table_of_VM_data[i].VM_extension) == 0)) target_VM = i;
}
```

The function set_VM_identifier is called from 14/main.

§**3.** To help Inform detect overflows, it needs to know whether integers in the target VM are 16 or 32 bits wide:

```
int target_VM_is_16_bit(void) {
    if (target_VM == -1) internal_error("target VM not set yet");
    if (table_of_VM_data[target_VM].VM_is_32_bit) return FALSE;
    return TRUE;
}
```

The function target_VM_is_16_bit is called from 5/litp, 5/lit and 7/dim.

§**4.** When releasing a blorbed story file, the file extension depends on the story file wrapped inside. (This is a dubious idea, in the opinion of the author of Inform – should not blorb be one unified wrapper? – but interpreter writers disagree.)

```
char *get_VM_blorbed_extension(void) {
    if (target_VM == -1) internal_error("target VM not set yet");
    return table_of_VM_data[target_VM].VM_blorbed_extension;
}
```

The function get_VM_blorbed_extension is called from 10/bib.

§**5. Parsing VM restrictions.** Given a word range, we see what set of virtual machines it specifies. For example, the result of calling

> for Z-machine version 5 or 8 only

is that the `VM_matches` field in the table above is set for the two minor VMs cited, and cleared for all of the others, while the `VM_matching_error_thrown` is false (since the text was valid). The same result is produced by

> for Z-machine versions 5 and 8 only

English being quirky that way.

```
define THROW_VM_MATCHING_ERROR_AND_RETURN { VM_matching_error_thrown = TRUE; return; }

int VM_matching_error_thrown = FALSE;                    an error occurred during parsing
int most_recent_major_VM;                                most recent major VM which matched, or −1
int version_can_be_inferred;                             from earlier in the word range parsed

void match_VM_against(int w1, int w2) {
    ⟨Clean the slate ready for a fresh VM parse 6⟩;
    if [[w1, w2 == for ... only --> w1, w2]] match_VM_from(w1, w2);
    else ⟨Signal a matching error, but recover by declaring every VM a valid match 7⟩;
}
```

§**6.** It is slightly lazy of this code to use global variables to preserve state through a sequence of function calls to `match_VM_from`, but sometimes a little laziness is what we deserve.

⟨Clean the slate ready for a fresh VM parse 6⟩ ≡

```
    int i;
    VM_matching_error_thrown = FALSE;
    most_recent_major_VM = -1;
    version_can_be_inferred = FALSE;
    for (i=0; table_of_VM_data[i].VM_code >= 0; i++) table_of_VM_data[i].VM_matches = FALSE;
```

This code is used in §5.

§**7.** If only some VMs match, code elsewhere has to do some finicky work. We will save it the trouble in the case where the specification did not make sense.

⟨Signal a matching error, but recover by declaring every VM a valid match 7⟩ ≡

```
int i;
for (i=0; table_of_VM_data[i].VM_code >= 0; i++) table_of_VM_data[i].VM_matches = TRUE;
THROW_VM_MATCHING_ERROR_AND_RETURN;
```

This code is used in §5.

§**8.** Given a list divided by "and" or "or", and perhaps using the serial comma, we split off into terms from left to right:

```
void match_VM_from(int w1, int w2) {
    if (is_list_divided(w1, w2, LOOK_FOR_AND + LOOK_FOR_OR)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        match_VM_from(lw1, lw2);
        match_VM_from(rw1, rw2);
        return;
    }
    ⟨Parse latest term in word range list 9⟩;
}
```

§**9.** The word "version(s)" is optional if it has been given in a previous term of the list, since the last major VM was named: this allows "versions 4 and 8" to work.

⟨Parse latest term in word range list 9⟩ ≡

```
⟨Detect major VM name, if given, and advance one word 10⟩;
⟨Give up if no major VM name found in any term of the list so far 11⟩;
if (w1 <= w2) {
    int version_specified = -1;
    if [[w1, w2 == version/versions ... --> w1, w2]] version_can_be_inferred = TRUE;
    else if (version_can_be_inferred == FALSE) THROW_VM_MATCHING_ERROR_AND_RETURN;
    if (!((w1==w2) && (vocab_test_flags(w1, NUMBER_MC)))) THROW_VM_MATCHING_ERROR_AND_RETURN;
    version_specified = VMULT * vocab_get_literal_number_value(lw_array[w1].lw_identity);
    ⟨Score a match for this specific version of the major VM, if we know about it 13⟩;
} else {
    ⟨Score a match for every known version of the major VM 12⟩;
}
```

This code is used in §8.

§**10.** The word "version" is sometimes implicit, but not after a major VM name. Thus "Glulx 3" is not allowed: it has to be "Glulx version 3".

⟨Detect major VM name, if given, and advance one word 10⟩ ≡

```
int i;
for (i=0; table_of_VM_data[i].VM_code >= 0; i++)
    if ((table_of_VM_data[i].VM_major_name) &&
        (compare_word_by_strcmp(w1, table_of_VM_data[i].VM_major_name))) {
        most_recent_major_VM = table_of_VM_data[i].VM_code;
        version_can_be_inferred = FALSE;
        w1++;
        break;
    }
```

This code is used in §9.

§**11.** The variable `VM_matching_error_thrown` may have been set either on this term or a previous one: for instance, if we are reading "Squirrel versions 4 and 7" then at the second term, "7", no major VM is named but the variable remains set from "Squirrel" having been parsed at the first term.

⟨Give up if no major VM name found in any term of the list so far 11⟩ ≡

```
if (most_recent_major_VM == -1) THROW_VM_MATCHING_ERROR_AND_RETURN;
```

This code is used in §9.

§**12.** We either make a run of matches:

⟨Score a match for every known version of the major VM 12⟩ ≡

```
int i;
for (i=0; table_of_VM_data[i].VM_code >= 0; i++)
    if (table_of_VM_data[i].VM_code == most_recent_major_VM)
        table_of_VM_data[i].VM_matches = TRUE;
```

This code is used in §9.

§**13.** ...or else we make a single match, or even none at all. This would not be an error: if the request was for "version 71 of Chipmunk", and we were unable to compile to this VM (so that no such minor VM record appeared in the table) then the situation might be that we are reading the requirements of some extension used by other people, who have a later version of Inform than us, and which does compile to that VM.

⟨Score a match for this specific version of the major VM, if we know about it 13⟩ ≡

```
int i;
for (i=0; table_of_VM_data[i].VM_code >= 0; i++)
    if ((table_of_VM_data[i].VM_code == most_recent_major_VM) &&
        (version_specified == table_of_VM_data[i].VM_version))
        table_of_VM_data[i].VM_matches = TRUE;
```

This code is used in §9.

§14. Determining a match with the current target VM. We can now answer the question: given a word range, is the current target VM a member of the set of VMs specified by the text in that word range? Only a reply of `FALSE` means definitely not.

```
int current_VM_matches_text(int w1, int w2) {
    if (target_VM == -1) internal_error("target VM not set yet");
    match_VM_against(w1, w2);
    if (VM_matching_error_thrown) return NOT_APPLICABLE;
    if (table_of_VM_data[target_VM].VM_matches) return TRUE;
    return FALSE;
}
```

The function current_VM_matches_text is called from 4/iext and 4/head.

§15. Icons for virtual machines. And everything else is cosmetic: printing, or showing icons to signify, the current VM or some set of permitted VMs. The following plots the icon associated with a given minor VM, and explicates what the icons mean:

```
void plot_VM_icon(OUTPUT_STREAM, int minor) {
    if (table_of_VM_data[minor].VM_image)
        WRITE("<img border=0 src=inform:/doc_images/%s> ",
            table_of_VM_data[minor].VM_image);
}
void write_VM_key(OUTPUT_STREAM) {
    int i;
    WRITE("Extensions compatible with specific story file formats only: ");
    for (i=0; table_of_VM_data[i].VM_code >= 0; i++) {
        if (i>0) WRITE(", ");
        plot_VM_icon(OUT, i);
        WRITE("%s", table_of_VM_data[i].VM_name);
    }
    WRITE("<p>");
}
```

The function write_VM_key is called from 4/excen.

§16. Describing the current VM.

```
void write_current_VM(OUTPUT_STREAM) {
    if (target_VM == -1) internal_error("target VM not set yet");
    WRITE("<p><hr>");
    index_anchor("STORYFILE");
    WRITE("<p>Story file format: ");
    plot_VM_icon(OUT, target_VM);
    if (table_of_VM_data[target_VM].VM_name) WRITE(table_of_VM_data[target_VM].VM_name);
    else WRITE("No name available\n");
    WRITE("<p>\n");
}
```

The function write_current_VM is called from 10/bib.

§**17. Displaying VM restrictions.** Given a word range, we parse it to set the match flags, then describe the result as concisely as we can with a row of icons.

```
void write_VM_icons(OUTPUT_STREAM, int w1, int w2) {
    int i;
    match_VM_against(w1, w2);
    ⟨Display nothing if every VM matches 18⟩;
    ⟨Display only the generic Z icon if every Z-machine VM version matches 19⟩;
    for (i=0; table_of_VM_data[i].VM_code >= 0; i++)
        if (table_of_VM_data[i].VM_matches)
            plot_VM_icon(OUT, i);
}
```

The function write_VM_icons is called from 4/excen and 4/edoc.

§**18.** To avoid the extensions directory page being plastered with gaudy but uncommunicative icons, we leave blank space if the requirements are always met. The icons are to signal exceptions.

⟨Display nothing if every VM matches 18⟩ ≡

```
    int i, everything_matches = TRUE;
    for (i=0; table_of_VM_data[i].VM_code >= 0; i++)
        if (table_of_VM_data[i].VM_matches == FALSE)
            everything_matches = FALSE;
    if (everything_matches) return;
```

This code is used in §17.

§**19.** This might happen if the user typed "for Z-machine only", but could also come about if he typed a specification naming in turn each minor version we know about, so the only way to check is to look at the match flag for each one.

⟨Display only the generic Z icon if every Z-machine VM version matches 19⟩ ≡

```
    int i, every_Z_matches = TRUE;
    for (i=0; table_of_VM_data[i].VM_code >= 0; i++)
        if ((table_of_VM_data[i].VM_code == Z_VM) &&
            (table_of_VM_data[i].VM_matches == FALSE))
            every_Z_matches = FALSE;
    if (every_Z_matches)
        ⟨Replace minor Z VMs in the match set with the single major one 20⟩;
```

This code is used in §17.

§**20.** The following operation leaves the match set in a state which does not correspond to what parsing would tell us (indeed, that's the point): so we must not use the match set again without reparsing it. But in fact, the match set is always recalculated before being used, so this is fine.

⟨Replace minor Z VMs in the match set with the single major one 20⟩ ≡

```
    int i;
    for (i=0; table_of_VM_data[i].VM_code >= 0; i++)
        if (table_of_VM_data[i].VM_code == Z_VM) {
            if (table_of_VM_data[i].VM_major_name)                    the major VM line for Z
                table_of_VM_data[i].VM_matches = TRUE;
            else
                table_of_VM_data[i].VM_matches = FALSE;                one of the minor ones
        }
```

This code is used in §19.

§**21.** The following table in the index (on the Contents page) may be useful to a few diehard Z-machine hackers, determined to squeeze the maximum out of the tiny array space available.

define NOTEWORTHY_USAGE_THRESHOLD 50 *don't mention arrays smaller than this, in bytes*

```
void note_VM_usage(char *cat, int w1, int w2, char *name, int words, int bytes, int each) {
    int b = bytes + words*((target_VM_is_16_bit())?2:4);
    if ((each == FALSE) && (b < NOTEWORTHY_USAGE_THRESHOLD)) return;
    if (b == 0) return;
    VM_usage_note *VMun = CREATE(VM_usage_note);
    VMun->word_ref1 = w1; VMun->word_ref2 = w2;
    VMun->usage_explained = name;
    VMun->usage_category = cat;
    VMun->bytes_used = b;
    VMun->each_flag = each;
}
```

The function note_VM_usage is called from 5/rel, 9/scene, 9/cot, 10/tab and 11/act.

§**22.** The explanatory note here probably ought to use the words "approximately", "incomplete" and so forth. It's really no better than a guide.

```
void index_VM_usage(void) {
    int nr = NUMBER_CREATED(VM_usage_note);
    VM_usage_note **sorted = I7_calloc(nr, sizeof(VM_usage_note *), INDEX_SORTING_MREASON);
    INDEX("<p>In a Z-machine story file, array memory can be very limited. "
        "Switching to the Glulx setting removes all difficulty, but some authors "
        "like to squeeze the very most out of the Z-machine instead. This "
        "list shows about how much array space is used by some larger items "
        "the source text has chosen to create.</p>");
    ⟨Sort the array usages 24⟩;
    ⟨Tabulate the array usages 23⟩;
    I7_free(sorted, INDEX_SORTING_MREASON);
}
```

The function index_VM_usage is called from 10/bib.

§**23.** The rows in the table mention pathetically small numbers of bytes, of course, by any rational measure.

⟨Tabulate the array usages 23⟩ ≡

```
    begin_plain_html_table(ifl);
    int i;
    VM_usage_note *VMun;
    for (i=0; i<nr; i++) {
        VMun = sorted[i];
        first_html_column(ifl, 0);
        INDEX("%s", VMun->usage_category);
        next_html_column(ifl, 0);
        if (VMun->each_flag) INDEX("each ");
        if (VMun->usage_explained)
            INDEX("%s", VMun->usage_explained);
        else if (VMun->word_ref1 >= 0)
            print_text_to_file(VMun->word_ref1, VMun->word_ref2, ifl);
        if (VMun->word_ref1 >= 0) index_link(lw_array[VMun->word_ref1].lw_source);
        next_html_column(ifl, 0);
```

```
        INDEX("%d bytes", VMun->bytes_used);
        end_html_row(ifl);
    }
    end_html_table(ifl);
```

This code is used in §22.

§**24.** As usual, we sort with the C library's `qsort`.

⟨Sort the array usages 24⟩ ≡

```
    int i = 0;
    VM_usage_note *VMun;
    LOOP_OVER(VMun, VM_usage_note) sorted[i++] = VMun;
    qsort(sorted, nr, sizeof(VM_usage_note *), compare_VM_usage_notes);
```

This code is used in §22.

§**25.** The following means the table is sorted in decreasing order of bytes used, with ties resolved by listing the first-declared item first.

```
int compare_VM_usage_notes(const void *ent1, const void *ent2) {
    const VM_usage_note *v1 = *((const VM_usage_note **) ent1);
    const VM_usage_note *v2 = *((const VM_usage_note **) ent2);
    if (v2->bytes_used != v1->bytes_used) return v2->bytes_used - v1->bytes_used;
    return v1->word_ref1 - v2->word_ref2;
}
```

Headings 4/head

Purpose

To keep track of the hierarchy of headings and subheadings found in the source text.

4/head.§5-11 The heading tree; §12 Verifying the heading tree; §13-16 Miscellaneous heading services; §17-22 Headings with extension dependencies; §23-25 World objects under each heading; §26-34 The world object search list; §35-36 Describing the heading structure, 1: to the debugging log; §37-39 Describing the heading structure, 2: to the index; §40-42 Describing the heading structure, 3: to a freestanding XML file

---

Template interpreter commands

```
5   {-callv:make_heading_tree}
17   {-callv:satisfy_heading_dependencies}
37   {-callv:index_headings}
40   {-callv:write_headings_as_xml}
```

Definitions

¶1. To take up the narrative of how Inform runs once more, we have read the source text in from the primary source file and from the extensions it requested: nothing more will be supplied from disc, and the whole combined text is arranged as a string of sentence nodes, each direct children of the root. In this section, we are concerned with HEADING nodes.

Most of these occur when the user has explicitly typed a heading such as:

```
Part VII - The Ghost of the Aragon
```

The sentence-breaker called `declare_heading` each time it found one of these, but also when a new source file started, because a file boundary is construed as beginning with a hidden "heading" of a higher rank than any other, and the sentence-breaker made a corresponding HEADING node there too. This is important because the doctrine is that each heading starts afresh with a new hierarchy of lower-order headings: thus changing the Part means we can start again with Chapter 1 if we like, and so on. Because each source file starts with an implicit super-heading, each source file gets its own independent hierarchy of Volume, and so on. But the convention is also important because we need to be able to say that every word loaded from disc ultimately falls under some heading, even if the source text as typed by the designer does not obviously have any headings in it.

The hierarchy thus runs: File (0), Volume (1), Book (2), Part (3), Chapter (4), Section (5). (The implementation below allows for even lower levels of subheading, from 6 to 9, but Inform doesn't use them.) Every run of NI declares at least two File (0) headings, representing the start of main text and the start of the Standard Rules, and these latter have a couple of dozen headings themselves, so the typical number of headings in a source text is 30 to 100.

```
define NO_HEADING_LEVELS 10
```

¶2. Although it is implicit in the parse tree already, the heading structure is not easy to deduce, and so in this section we build a much smaller tree consisting just of the hierarchy of headings. The heading tree has nodes made from the following structures:

```
typedef struct heading {
    struct parse_node *sentence_declaring;                  if any: file starts are undeclared
    struct source_location start_location;                  first word under this heading is here
    int level;                                              0 for Volume (highest) to 5 for Section (lowest)
    int indentation;                                        in a hierarchical listing
    int index_definitions_made_under_this;                  for instance, global variables made here?
    int for_release;                                        include this material in a release version?
    int omit_material;                                      if set, simply ignore all of this
    int use_with_or_without;                                if TRUE, use with the extension; if FALSE, without
    struct extension_identifier for_use_with;               e.g. "for use with ... by ..."
    int in_place_of_w1, in_place_of_w2;                     e.g. "in place of ... in ... by ..."
    struct world_object *list_of_contents;                  world objects defined under this
    struct heading *parent_heading;
    struct heading *child_heading;
    struct heading *next_heading;
    MEMORY_MANAGEMENT
} heading;
```

The structure heading is private to this section.

¶3. The headings and subheadings are formed into a tree in which each heading contains its lesser-order headings. The pseudo-heading exists to be the root of this tree; the entire text falls under it. It is not a real heading at all, and has no "level" or "indentation" as such.

```
heading pseudo_heading;                                     The entire source falls under this top-level heading
```

¶4. As an example, a sequence in the primary source text of (Chapter I, Book Two, Section 5, Chapter I, Section 1, Chapter III) would be formed up into the heading tree:

```
(the pseudo-heading)      level −1, indentation −1
    (File: Standard Rules)      level 0, indentation 0
        ...
    (File: primary source text)      level 0, indentation 0
        Chapter I      level 4, indentation 1
        Book Two      level 2, indentation 1
            Section 5      level 5, indentation 2
            Chapter I      level 4, indentation 2
                Section 1      level 5, indentation 3
            Chapter III      level 4, indentation 2
```

Note that the level of a heading is not the same thing as its depth in this tree, which we call the "indentation", and there is no simple relationship between the two numbers. Clearly we want to start at the left margin. If a new heading is subordinate to its predecessor (i.e., has higher level), we want to indent further, but by the least amount needed – a single tap step. Adjacent equal-level headings are on a par with each other and should have the same indentation. But when the new heading is lower level than its predecessor (i.e., more important) then the indentation decreases to match the last one equally important.

We can secure the last of those properties with a formal definition as follows. The level ℓ_n of a heading depends only on its wording (or source file origin), but the indentation of the nth heading, i_n, depends on $(\ell_1, \ell_2, ..., \ell_n)$, the sequence of all levels so far:

$$i_n = i_{\max\{j \mid 0 \le j < n, \ell_j < \ell_n\}} + 1$$

where $\ell_0 = i_0 = -1$, so that this set always contains 0 and is therefore not empty. We deduce that

(a) $i_1 = 0$ and thereafter $i_n \ge 0$, since ℓ_n is never negative again,

(b) if $\ell_k = \ell_{k+1}$ then $i_k = i_{k+1}$, since the set over which the maximum is taken is the same,

(c) if $\ell_{k+1} > \ell_k$, a subheading of its predecessor, then $i_{k+1} = i_k + 1$, a single tab step outward.

That establishes the other properties we wanted, and shows that i_n is indeed the number of tab steps we should be determining.

Note that to calculate i_n we do not need the whole of $(\ell_1, ..., \ell_n)$: we only need to remember the values of

$$i_{\max\{j \mid 0 \le j < n, \ell_j < K\}}$$

for each possible heading level $K = 0, 1, ..., 9$. This requires much less storage: we call it the "last indentation above level K".

This leads to the following algorithm when looking at the headings in any individual file of source text: at the top of file,

```
for (i=0; i<NO_HEADING_LEVELS; i++) last_indentation_above_level[i] = -1;
```

Then parse for headings (they have an easily recognised lexical form); each time one is found, work out its `level` as 1, ..., 5 for Volume down to Section, and call:

```
int find_indentation(int level) {
    int i, ind = last_indentation_above_level[level] + 1;
    for (i=level+1; i<NO_HEADING_LEVELS; i++)
        last_indentation_above_level[i] = ind;
    return ind;
}
```

While this algorithm is trivially equivalent to finding the depth of a heading in the tree which we are going to build anyway, it is worth noting here for the benefit of anyone writing a tool to (let's say) typeset an Inform source text with a table of contents, or provide a navigation gadget in the user interface.

¶5. The primary source text, and indeed the source text in the extensions, can make whatever headings they like: no sequence is illegal. It is not for NI to decide on behalf of the author that it is eccentric to place Section C before Section B, for instance. The author might be doing so deliberately, to put the Chariot-race before the Baths, say; and the indexing means that it will be very apparent to the author what the heading structure currently is, so mistakes are unlikely to last long. This is a classic case where Inform trying to be too clever would annoy more often than assist.

---

§**1.** The heading tree is constructed all at once, after most of the sentence-breaking is done, but since a few sentences can in principle be added later, we watch for the remote chance of further headings being added, by keeping the following flag:

```
int heading_tree_made_at_least_once = FALSE;
```

§**2.** Now, then, the routine `declare_heading` is called by the sentence-breaker each time it constructs a new HEADING node. (Note that it is not called to create the pseudo-heading, which does not come from a node.)

A level 0 heading has text (the first sentence which happens to be in the new source file), but this has no significance other than its location, and cannot contain information about releasing or about virtual machines.

```
int last_indentation_above_level[NO_HEADING_LEVELS], lial_made = FALSE;

heading *declare_heading(parse_node *PN) {
    heading *h = CREATE(heading);
    h->parent_heading = NULL; h->child_heading = NULL; h->next_heading = NULL;
    h->list_of_contents = NULL;
    h->for_release = TRUE; h->omit_material = FALSE;
    h->index_definitions_made_under_this = TRUE;
    h->use_with_or_without = NOT_APPLICABLE;
    h->in_place_of_w1 = -1; h-> in_place_of_w2 = -2;
    if ((PN == NULL) || (PN->word_ref1 < 0)) internal_error("heading at textless node");
    internal_error_if_node_type_wrong(PN, HEADING_NT);
    h->sentence_declaring = PN;
    h->start_location = lw_array[PN->word_ref1].lw_source;
    h->level = pn_int_annotation(PN, heading_level_ANNOT);
    if (h->level > 0) ⟨Parse heading text for release or other stipulations 4⟩;
    if ((h->level < 0) || (h->level >= NO_HEADING_LEVELS)) internal_error("impossible level");
    ⟨Determine the indentation from the level 3⟩;
    LOGIF(HEADINGS, "Created heading $H", h);
    if (heading_tree_made_at_least_once) make_heading_tree();
    return h;
}
```

The function declare_heading is called from 4/sent.

§**3.** This implements the indentation algorithm described above.

⟨Determine the indentation from the level 3⟩ ≡

```
    int i;
    if (lial_made == FALSE) {
        for (i=0; i<NO_HEADING_LEVELS; i++) last_indentation_above_level[i] = -1;
        lial_made = TRUE;
    }
    h->indentation = last_indentation_above_level[h->level] + 1;
    for (i=h->level+1; i<NO_HEADING_LEVELS; i++)
        last_indentation_above_level[i] = h->indentation;
```

This code is used in §2.

§**4.** A heading which is "not for release" contains material which will be omitted in the release version of a project: generally speaking, that means debugging verbs, test scripts and the like.

⟨Parse heading text for release or other stipulations 4⟩ ≡

```
int i, j, w1, w2, vm1 = -1, vm2 = -1;
[[w1, w2 <-- PN]];
if [[w1, w2 == ... not for release]]
    h->for_release = FALSE;
if [[w1, w2 == ... unindexed]]
    h->index_definitions_made_under_this = FALSE;
for (i=w1; i<w2; i++)
    if (paired_brackets(i, w2)) { vm1 = i+1; vm2 = w2-1; w2 = i-1; break; }
if (vm1 >= 0) {
    if [[vm1, vm2 == for ... only]] {
        switch (current_VM_matches_text(vm1, vm2)) {
            case TRUE: break;
            case FALSE: h->omit_material = TRUE; break;
            case NOT_APPLICABLE:
                current_sentence = PN;
                sentence_problem(_P_(C4HeadingBadVM),
                    "this heading contains a stipulation about the Setting "
                    "for story file format which I can't understand",
                    "and should be something like '(for Z-machine version 5 "
                    "or 8 only)' or '(for Glulx only)'.");
                break;
        }
    }
    if [[vm1, vm2 == for use with/without ... by ... : i]] {
        char exft[MAX_FILENAME_LENGTH], exfa[MAX_FILENAME_LENGTH];
        if [[word vm1+2 == with]] h->use_with_or_without = TRUE;
        else h->use_with_or_without = FALSE;
        print_raw_text_to_string_truncated(vm1+3, i-1, exft, MAX_FILENAME_LENGTH);
        print_raw_text_to_string_truncated(i+1, vm2, exfa, MAX_FILENAME_LENGTH);
        eid_new(&(h->for_use_with), exfa, exft, USEWITH_EIDBC);
    }
    if ([[vm1, vm2 == in place of ... in ... : j]] &&
        [[j, vm2 == in ... by ... : i]]) {
        char exft[MAX_FILENAME_LENGTH], exfa[MAX_FILENAME_LENGTH];
        h->use_with_or_without = TRUE;
        h->in_place_of_w1 = vm1+3; h->in_place_of_w2 = j-1;
        print_raw_text_to_string_truncated(j+1, i-1, exft, MAX_FILENAME_LENGTH);
        print_raw_text_to_string_truncated(i+1, vm2, exfa, MAX_FILENAME_LENGTH);
        LOG("Req is <$W> in <%s> by <%s>\n", h->in_place_of_w1, h->in_place_of_w2, exft, exfa);
        eid_new(&(h->for_use_with), exfa, exft, USEWITH_EIDBC);
    }
    if [[vm1, vm2 == not for release]]
        h->for_release = FALSE;
    if [[vm1, vm2 == unindexed]]
        h->index_definitions_made_under_this = FALSE;
}
```

This code is used in §2.

§**5. The heading tree.** The headings were constructed above as freestanding nodes (except that the pseudo-heading already existed): here, we assemble them into a tree structure. Because we want to be able to call this more than once, perhaps to make revisions if late news comes in of a new heading (see above), we begin by removing any existing relationships between the heading nodes.

```
void make_heading_tree(void) {
    heading *h;
    ⟨Reduce the whole heading tree to a pile of twigs 6⟩;
    LOOP_OVER(h, heading) {
        ⟨If h is outside the tree, make it a child of the pseudo-heading 7⟩;
        ⟨Run through subsequent equal or subordinate headings to move them downward 8⟩;
    }
    heading_tree_made_at_least_once = TRUE;
    verify_heading_tree();
}
```

The function make_heading_tree is invoked by a command in a `.i6t` template file.

§**6.** Note that the loop over headings below loops through all those which were created by the memory manager: which is to say, all of them except for the pseudo-heading, which was explicitly placed in static memory above.

⟨Reduce the whole heading tree to a pile of twigs 6⟩ ≡

```
    heading *h;
    pseudo_heading.child_heading = NULL; pseudo_heading.parent_heading = NULL;
    pseudo_heading.next_heading = NULL;
    LOOP_OVER(h, heading) {
        h->parent_heading = NULL; h->child_heading = NULL; h->next_heading = NULL;
    }
```

This code is used in §5.

§**7.** The idea of the heading loop is that when we place a heading, we also place subsequent headings of lesser or equal status until we cannot do so any longer. That means that if we reach h and find that it has no parent, it must be subordinate to no earlier heading: thus, it must be attached to the pseudo-heading at the top of the tree.

⟨If h is outside the tree, make it a child of the pseudo-heading 7⟩ ≡

```
    if (h->parent_heading == NULL)
        make_child_heading(h, &pseudo_heading);
```

This code is used in §5.

§**8.** Note that the following could be summed up as "move subsequent headings as deep in the tree as we can see they need to be from h's perspective alone". This isn't always the final position. For instance, given the sequence Volume 1, Chapter I, Section A, Chapter II, the tree is adjusted twice:

```
when h = Volume 1:          then when h = Chapter I:
Volume 1                    Volume 1
    Chapter I                   Chapter I
    Section A                       Section A
    Chapter II                  Chapter II
```

since Section A is demoted twice, once by Volume 1, then by Chapter I. (This algorithm would in principle be quadratic in the number of headings if the possible depth of the tree were unbounded – every heading might have to demote every one of its successors – but in fact because the depth is at most 9, it runs in linear time.)

⟨Run through subsequent equal or subordinate headings to move them downward 8⟩ ≡

```
heading *subseq;
for (subseq = NEXT_OBJECT(h, heading);                    start from the next heading in source
     (subseq) && (subseq->level >= h->level);             for a run with level below or equal h
     subseq = NEXT_OBJECT(subseq, heading)) {             in source declaration order
     if (subseq->level == h->level) {                    a heading of equal status ends the run...
         make_child_heading(subseq, h->parent_heading); break;   ...and becomes h's sibling
     }
     make_child_heading(subseq, h);                      all lesser headings in the run become h's children
}
```

This code is used in §5.

§**9.** The above routine, then, calls `make_child_heading` to attach a heading to the tree as a child of a given parent:

```
void make_child_heading(heading *ch, heading *pa) {
    heading *former_pa = ch->parent_heading;
    if (former_pa == pa) return;
    ⟨Detach ch from the heading tree if it is already there 10⟩;
    ch->parent_heading = pa;
    ⟨Add ch to the end of the list of children of pa 11⟩;
}
```

§**10.** If ch is present in the tree, it must have a parent, unless it is the pseudo-heading: but the latter can never be moved, so it isn't. Therefore we can remove ch by striking it out from the children list of the parent. (Any children which ch has, grandchildren so to speak, come with it.)

⟨Detach ch from the heading tree if it is already there 10⟩ ≡

```
    if (former_pa) {
        if (former_pa->child_heading == ch)
            former_pa->child_heading = ch->next_heading;
        else {
            heading *sibling;
            for (sibling = former_pa->child_heading; sibling; sibling = sibling->next_heading)
                if (sibling->next_heading == ch) {
                    sibling->next_heading = ch->next_heading;
                    break;
                }
        }
    }
    ch->next_heading = NULL;
```

This code is used in §9.

§**11.** Two cases: the new parent is initially childless, or it isn't.

⟨Add ch to the end of the list of children of pa 11⟩ ≡

```
    heading *sibling;
    if (pa->child_heading == NULL) pa->child_heading = ch;
    else
        for (sibling = pa->child_heading; sibling; sibling = sibling->next_heading)
            if (sibling->next_heading == NULL) {
                sibling->next_heading = ch;
                break;
            }
```

This code is used in §9.

§**12. Verifying the heading tree.** We have now, in effect, computed the indentation value of each heading twice, by two entirely different methods: first by the mathematical argument above, then by observing that it is the depth in the heading tree. Seeing if these two methods have given the same answer provides a convenient check on our working.

```
int heading_tree_damaged = FALSE;
void verify_heading_tree(void) {
    verify_heading_tree_recursively(&pseudo_heading, -1);
    if (heading_tree_damaged) internal_error("heading tree failed to verify");
}
void verify_heading_tree_recursively(heading *h, int depth) {
    if (h == NULL) return;
    if ((h != &pseudo_heading) && (depth != h->indentation)) {
        heading_tree_damaged = TRUE;
        LOG("$H\n*** indentation should be %d ***\n", h, depth);
    }
    verify_heading_tree_recursively(h->child_heading, depth+1);
    verify_heading_tree_recursively(h->next_heading, depth);
}
```

§**13. Miscellaneous heading services.** The first of these we have already seen in use: the sentence-breaker calls it to ask whether sentences falling under the current heading should be included in the active source text. (For instance, sentences under a heading with the disclaimer "(for Glulx only)" will not be included if the target virtual machine on this run of NI is the Z-machine.)

```
int hd_include_material(heading *h) {
    if ((h->for_release == FALSE) && (for_release == TRUE)) return FALSE;
    if (h->omit_material) return FALSE;
    return TRUE;
}

int hd_indexed(heading *h) {
    if (h == NULL) return TRUE;                    definitions made nowhere are normally indexed
    return h->index_definitions_made_under_this;
}
```

The function hd_include_material is called from 4/sent.

The function hd_indexed is called from 9/qty and 12/phin.

§**14.** Two utilities to do with the file of origin:

```
int hd_in_same_file(heading *h1, heading *h2) {
    if ((h1 == NULL) || (h2 == NULL)) return FALSE;
    if (h1->start_location.file_of_origin == h2->start_location.file_of_origin) return TRUE;
    return FALSE;
}

extension_file *hd_get_extension_containing(heading *h) {
    if ((h == NULL) || (h->start_location.file_of_origin == NULL)) return NULL;
    return sf_get_extension_corresponding(h->start_location.file_of_origin);
}
```

The function hd_in_same_file is called from 12/phin.

The function hd_get_extension_containing is called from 11/act, 12/cph and 12/phin.

§**15.** Although File (0) headings do have text, contrary to the implication of the routine here, this text is only what happens to be first in the file: it isn't a heading actually typed by the user, which is all that we are interested in for this purpose. So we send back a null word range.

```
void hd_get_text_of_heading(heading *h, int *w1, int *w2) {
    if ((h == NULL) || (h->level == 0)) { *w1 = -1; *w2 = -1; return; }
    *w1 = h->sentence_declaring->word_ref1;
    *w2 = h->sentence_declaring->word_ref2;
}
```

The function hd_get_text_of_heading is called from 9/qty, 11/act and 12/phin.

§**16.** This routine determines the (closest) heading to which a scrap of text belongs, and is important since the parsing of noun phrases is affected by that choice of heading (as we shall see): to NI, headings provide something analogous to the scope of local variables in a conventional programming language.

Because every file has a File (0) heading registered at line 1, the loop in the following routine is guaranteed to return a valid heading provided the original source location is well formed (i.e., has a non-null source file and a line number of at least 1).

```
heading *heading_of(source_location sl) {
    heading *h;
    if (sl.file_of_origin == NULL) return NULL;
    LOOP_BACKWARDS_OVER(h, heading)
        if ((sl.file_of_origin == h->start_location.file_of_origin) &&
            (sl.line_number >= h->start_location.line_number)) return h;
    internal_error("unable to determine the heading level of source material");
    return NULL;
}
```

The function heading_of is called from 9/qty, 11/act, 12/cph and 12/phin.

§**17. Headings with extension dependencies.** If the content under a heading depended on the VM not in use, or was marked not for release in a release run, we were able to exclude it just by skipping. The same cannot be done when a heading says that it should be used only if a given extension is, or is not, being used, because when the heading is created we don't yet know which extensions are included. But when the following is called, we do know that.

```
void satisfy_heading_dependencies(void) {
    heading *h;
    LOOP_OVER(h, heading)
        if (h->use_with_or_without != NOT_APPLICABLE)
            satisfy_individual_heading_dependency(h);
}
```

The function satisfy_heading_dependencies is invoked by a command in a `.i6t` template file.

§**18.** And now the code to check an individual heading's usage. This whole thing is carefully timed so that we can still afford to cut up and rearrange the parse tree on quite a large scale, and that's just what we do.

```
void satisfy_individual_heading_dependency(heading *h) {
    int loaded = FALSE;
    extension_identifier *eid = &(h->for_use_with);
    if (h->level < 1) return;
    if (no_times_EID_used_in_context(eid, LOADED_EIDBC) != 0) loaded = TRUE;
    if (pn_int_annotation(h->sentence_declaring, suppress_heading_dependencies_ANNOT)) return;
    if (h->in_place_of_w1 >= 0) {
        heading *h2; int found = FALSE;
        if (loaded == FALSE) ⟨Can't replace heading in an unincluded extension 20⟩
        else {
            LOOP_OVER(h2, heading)
                if ((h2->sentence_declaring) &&
                    (compare_word_range(h->in_place_of_w1, h->in_place_of_w2,
                        h2->sentence_declaring->word_ref1, h2->sentence_declaring->word_ref2)) &&
                    (eid_match(ef_get_eid(hd_get_extension_containing(h2)), eid))) {
                    found = TRUE;
                    if (h->level != h2->level)
```

```
                    ⟨Can't replace heading unless level matches 22⟩;
                excise_material_under(h2, NULL);
                excise_material_under(h, h2->sentence_declaring);
                break;
            }
        if (found == FALSE) ⟨Can't find heading in the given extension 21⟩;
    }
} else
    if (h->use_with_or_without != loaded) excise_material_under(h, NULL);
}
```

§**19.** To excise, we simply amend the "next" link from the heading to point to the next heading, or to null if no such exists; we repeat until we hit a heading of equal or greater importance than the original, or the end of the current extension, or the end of the whole source text. (Thus, all headings remain in the parse tree; only the material subordinate to them disappears.) If another parse node is supplied then the nodes extracted are strung onto it in sequence rather than being thrown away.

Any heading which is excised is marked so that it won't have its own dependencies checked. This clarifies several cases, and in particular ensures that if Chapter X is excised then a subordinate Section Y cannot live on by replacing something elsewhere (which would effectively delete the content elsewhere). Since subordinate headings always come later in the parse tree than higher ones, there is no risk that Section Y has already had its dependencies checked.

```
void excise_material_under(heading *h, parse_node *transfer_to) {
    int lev = h->level;
    parse_node *hpn;
    parse_node *end_of_transfer = NULL;
    if (transfer_to) end_of_transfer = transfer_to->next;
    if (h->sentence_declaring == NULL) internal_error("stipulations on a non-sentence heading");
    hpn = h->sentence_declaring;
    do {
        parse_node *pn;
        pn_annotate_int(hpn, suppress_heading_dependencies_ANNOT, TRUE);
        for (pn = hpn->next; ((pn) && (pn_get_node_type(pn) != HEADING_NT) && (pn_get_node_type(pn)
!= ENDHERE_NT)); pn = pn->next)
            if (transfer_to) { transfer_to->next = pn; transfer_to = pn; }
        hpn->next = pn;
        hpn = pn;
    } while ((hpn) && (pn_int_annotation(hpn, heading_level_ANNOT) > lev) &&
        (pn_get_node_type(hpn) != ENDHERE_NT));
    if (transfer_to) transfer_to->next = end_of_transfer;
}
```

§**20.**

⟨Can't replace heading in an unincluded extension 20⟩ ≡

```
    current_sentence = h->sentence_declaring;
    sentence_problem(_P_(C4HeadingInPlaceOfUnincluded),
        "no extension of that name is included",
        "so it is not possible to replace one of its headings with this one.");
```

This code is used in §18.

§**21.**

⟨Can't find heading in the given extension 21⟩ ≡

```
current_sentence = h->sentence_declaring;
sentence_problem(_P_(C4HeadingInPlaceOfUnknown),
    "no heading of that name appears to be in the given extension",
    "so it is not possible to replace it with this one.");
```

This code is used in §18.

§**22.**

⟨Can't replace heading unless level matches 22⟩ ≡

```
current_sentence = h->sentence_declaring;
sentence_problem(_P_(C4UnequalHeadingInPlaceOf),
    "these headings are not of the same level",
    "so it is not possible to make the replacement. (Level here means "
    "being a Volume, Book, Part, Chapter or Section: for instance, "
    "only a Chapter heading can be used 'in place of' a Chapter.)");
```

This code is used in §18.

§**23. World objects under each heading.** Every heading carries with it a linked list of the world objects created in sentences which belong to it, in the sense of the `heading_of` routine defined above. When any world object is created, the following is called to let the current sentence's heading know that it has a new friend.

But the assertion-maker giveth and the assertion-maker taketh away: blessed be the name of the assertion-maker. World objects are sometimes temporarily created and then destroyed again, and in particular this can legally happen when the current sentence is outside of any heading, in text manufactured internally.

```
int no_headless_wos = 0;
void attach_wo_to_heading(world_object *new_wo) {
    heading *h = heading_of(lw_array[current_sentence->word_ref1].lw_source);
    new_wo->under_what_heading = h;
    if (h == NULL) { no_headless_wos++; return; }
    if (h->list_of_contents == NULL) h->list_of_contents = new_wo;
    else {
        world_object *wh;
        for (wh = h->list_of_contents; wh; wh = wh->next_under_heading)
            if (wh->next_under_heading == NULL) {
                wh->next_under_heading = new_wo;
                return;
            }
    }
}
```

The function attach_wo_to_heading is called from 9/wo.

§**24.** Here a tentative world object definition has been cancelled, so that we need to strike it out of the heading's list again. (This is no longer used, in fact.)

```
void detach_wo_from_heading(world_object *moribund) {
    heading *h = moribund->under_what_heading;
    if (h) {
        moribund->under_what_heading = NULL;
        if (h->list_of_contents == moribund)
            h->list_of_contents = moribund->next_under_heading;
        else {
            world_object *item;
            for (item = h->list_of_contents; item; item = item->next_under_heading)
                if (item->next_under_heading == moribund) {
                    item->next_under_heading = moribund->next_under_heading;
                    return;
                }
        }
    } else no_headless_wos--;
}
```

§**25.** The following verification checks that every world object is listed in the object list for exactly one heading. (It abuses the `room_flag` field shabbily to do this, using it not as a flag or anything to do with rooms but as an instance count: but `room_flag` is not used until the model world is being set up, which is long after all the object creations and uncreations, when the following routine is used.) The point of the check is not so much to make sure the object lists are properly formed, as the code making those is pretty elementary: it's really a test that the source text is well-formed with everything placed under a heading, and no sentence having fallen through a crack.

```
void verify_heading_divisions(void) {
    world_object *wo; heading *h;
    int total = 0, disaster = FALSE;
    LOOP_OVER(wo, world_object) wo->room_flag = 0;
    LOOP_OVER(h, heading)
        for (wo=h->list_of_contents; wo!=NULL; wo=wo->next_under_heading)
            wo->room_flag++, total++;
    LOOP_OVER(wo, world_object) if (wo != kind_kind)
        if (wo->room_flag > 1) {
            LOG("$O occurs under %d headings\n", wo, wo->room_flag);
            disaster = TRUE;
        }
    if (total + no_headless_wos + 1 != NUMBER_CREATED(world_object)) {
        LOG("%d WOs != %d headed + %d headless + 1 kind_kind\n",
            NUMBER_CREATED(world_object), total, no_headless_wos);
        disaster = TRUE;
    }
    if (disaster) internal_error_tree_unsafe("heading contents list failed verification");
    LOOP_OVER(wo, world_object) wo->room_flag = FALSE;
}
```

The function verify_heading_divisions is called from 9/wo.

§**26. The world object search list.** Identifying noun phrases is tricky. Many plausible phrases could refer in principle to several different world objects: "east", for instance, might mean the direction or, say, "east garden". And what if the source mentions many chairs, and now refers simply to "the chair"? This problem is not so acute for nouns referring to abstractions, where we can simply forbid duplicate definitions and require an exact wording when talking about them. But for world objects – which represent the solid and often repetitive items and places of the simulated world – it cannot be ducked. We can hardly tell an Inform author to create at most one item whose name contains the word "jar", for instance.

All programming languages face similar problems. In C, for instance, a local variable named `east` will be recognised in preference to a global one of the same name (to some extent external linking provides a third level again). The way this is done is usually explained in terms of the "scope" of a definition, the part of the source for which it is valid: the winner, in cases of ambiguity, being the definition of narrowest scope which is valid at the position in question. In our terms, a stand-alone C program has a heading tree like so, with two semantically meaningful heading levels, File (0) and Routine (1), and then sublevels provided by braced blocks:

```
File
    main()
    routine1()
        interior block of a loop
        ...
    routine2()
    ...
```

The resolution of a name at a given position P is unambiguous: find the heading H to which P belongs; if the name is defined there, accept that; if not move H upwards and try again; if it is not defined even at File (0) level, issue an error: the term is undefined.

Inform is different in two respects, one trivial, the other not. The trivial difference is that an Inform name can be defined midway through the matter (though as a result of the C99 revision, ANSI C now also allows variables to be created mid-block, in fact: and some C compilers even implement this).

The big difference is that in Inform, names are always visible across headings. They can be used before being defined; Section 2 of Part II is free to mention the elephant defined in Section 7 of Part VIII, say. English text is like this: a typical essay has one great big namespace.

We resolve this by searching backwards through recent object creations in the current heading, then in the current heading level above that, and so on up to the top conceptual level of the source. Thus a "chair" in the current chapter will always have priority over any in previous chapters, and so on. However, kinds are always given priority over mere instances, in order that "door" will retain its generic meaning even if, say, "an oak door" is created.

§**27.** This means that, under every heading, the search sequence is different. So for the sake of efficiency we construct a linked list of world objects in priority order the first time we search under a new heading, then simply use that thereafter: we also keep track of the tail of this list. Sections other than this one cannot read the list itself, and use the following definition to iterate through it.

```
define LOOP_OVER_WO_SEARCH_LIST(wo)
    for (wo = wo_search_start; wo; wo = wo->next_to_search)

world_object *wo_search_start = NULL, *wo_search_finish = NULL;
```

§28. The search sequence is, in effect, a cache storing a former computation, and like all caches it can fall out of date if the circumstances change so that the same computation would now produce a different outcome. That can only happen here if any world object is created or destroyed: the assertion-maker calls the following routine to let us know.

```
heading *wo_search_list_valid_for_this_heading = NULL;                    initially it's unbuilt
void hd_disturb_headings(void) {
    wo_search_list_valid_for_this_heading = NULL;
}
```

The function hd_disturb_headings is called from 9/wo.

§29. Leaving aside the cache, then, we build a list as the kind "kind", then all the other kinds as found by recursively searching headings, then all the world objects found in the same way.

```
void construct_wo_search_list(void) {
    heading *h = NULL;
    ⟨Work out the heading from which we wish to search 30⟩;
    if (h == wo_search_list_valid_for_this_heading) return;                    rely on the cache
    LOGIF(HEADINGS, "Rebuilding WO search list from: $H\n", h);
    ⟨Start the search list containing just the kind "kind" 31⟩;
    build_wo_search_list_from(h, NULL, TRUE);                                  pass 1: add the kinds
    build_wo_search_list_from(h, NULL, FALSE);                                 pass 2: add the world objects
    ⟨Verify that the search list indeed contains every world object just once 32⟩;
    wo_search_list_valid_for_this_heading = h;
}
```

The function construct_wo_search_list is called from 9/wo.

§30. Basically, we calculate the search list from the point of view of the current sentence:

⟨Work out the heading from which we wish to search 30⟩ ≡

```
    if ((current_sentence == NULL) || (current_sentence->word_ref1 < 0))
        internal_error("cannot establish position P: there is no current sentence");
    source_location position_P = lw_array[current_sentence->word_ref1].lw_source;
    h = heading_of(position_P);
    if (h == NULL) internal_error("no heading for position P");
```

This code is used in §29.

§31. We initialise the list to consist only of the kind "kind" – which belongs to no heading: it is the unique world object created within Inform itself at start-up.

The pseudo-heading has no list of contents because all objects are created in source files, each certainly underneath a File (0) heading, so nothing should ever get that far.

⟨Start the search list containing just the kind "kind" 31⟩ ≡

```
    wo_search_start = kind_kind;
    wo_search_finish = kind_kind;
    kind_kind->next_to_search = NULL;
    pseudo_heading.list_of_contents = NULL;                    should always be true, but just in case
```

This code is used in §29.

§**32.** The potential for disaster if this algorithm should be incorrect is high, so we perform a quick count to see if everything made it onto the list and produce an internal error if not.

⟨Verify that the search list indeed contains every world object just once 32⟩ ≡

```
int c; world_object *wo;
for (c=0, wo = wo_search_start; wo != NULL; wo = wo->next_to_search) c++;
if (c != wo_get_no_wos_now_existing()) {
    LOG("Reordering failed from $H\n", h);
    LOG("%d wos created, %d in ordering\n", wo_get_no_wos_now_existing(), c);
    log_all_headings();
    LOG("Making fresh tree:\n");
    make_heading_tree();
    log_all_headings();
    internal_error_tree_unsafe("reordering of objects failed");
}
```

This code is used in §29.

§**33.** The following adds all objects under heading H to the search list, using its own list of contents, and then recurses to add all objects under subheadings of H other than the one which has just recursed up to H. With that done, we recurse up to the superheading of H.

To prove that `build_wo_search_list_from` is called exactly once for each heading in the tree, forget about the up/down orientation and consider it as a graph instead. At each node we try going to every possible other node, except the way we came (at the start of the traverse, the "way we came" being null): clearly this ensures that all of our neighbours have been visited. Since every heading ultimately depends from the pseudo-heading, the graph is connected, and therefore every heading must eventually be visited. No heading can be visited twice, because that would mean that a cycle of nodes $H_1, H_2, ..., H_i, H_1$ must exist: since we have a tree structure, there are no loops, and so $H_i = H_2$, $H_{i-1} = H_3$, and so on – we must be walking a path and then retracing our steps in reverse. That being so, there is a point where we turned back: we went from H_j to H_{j+1} to H_j again. And this violates the principle that at each node we move outwards in every direction *except* the way we came, a contradiction.

The routine looks as if it may have a large recursion depth – maybe as deep as the number of headings – but because we go downwards and then upwards, the maximum recursion depth of the routine is less than $2L + 1$ where L is the number of levels in the tree other than the pseudo-heading, which provides an upper bound of about 21, regardless of the size of the source text. The running time is linear in both the number of headings and the number of world objects in the source text.

```
void build_wo_search_list_from(heading *within, heading *way_we_came, int kinds) {
    world_object *wo; heading *subhead;
    if (within == NULL) return;
    for (wo = within->list_of_contents; wo; wo=wo->next_under_heading)
        if ((wo != kind_kind) && (wo->kind_flag == kinds))
            ⟨Add WO to the end of the search list 34⟩;
    recurse downwards through subordinate headings, other than the way we came up
    for (subhead = within->child_heading; subhead; subhead = subhead->next_heading)
        if (subhead != way_we_came)
            build_wo_search_list_from(subhead, within, kinds);
    recurse upwards to superior headings, unless we came here through a downward recursion
    if (within->parent_heading != way_we_came)
        build_wo_search_list_from(within->parent_heading, within, kinds);
}
```

§**34.** Note that `wo_search_finish` is always guaranteed non-`NULL` here, since the list was initialised containing the kind "kind", rather than being initialised empty.

⟨Add WO to the end of the search list 34⟩ ≡

```
if (wo_search_finish->next_to_search != NULL)
    internal_error("end of WO search list has frayed somehow");
wo_search_finish->next_to_search = wo;
wo_search_finish = wo_search_finish->next_to_search;
wo_search_finish->next_to_search = NULL;
```

This code is used in §33.

§**35. Describing the heading structure, 1: to the debugging log.** Finally, three ways to describe the run of headings: to the debugging log, to the index of the project, and to a freestanding XML file.

```
void log_heading(heading *h) {
    if (h==NULL) { LOG("<null heading>\n"); return; }
    if (h==&pseudo_heading) { LOG("<pseudo_heading>\n"); return; }
    LOG("H%d ", h->allocation_id);
    if (h->start_location.file_of_origin) LOG("<%s, line %d>",
        sf_get_filename(h->start_location.file_of_origin), h->start_location.line_number);
    else LOG("<nowhere>");
    LOG(" level:%d indentation:%d", h->level, h->indentation);
}
```

The function log_heading is called from 2/dl.

§**36.** And here we log the whole heading tree by recursing through it, and surreptitiously check that it is correctly formed at the same time.

```
void log_all_headings(void) {
    heading *h;
    LOOP_OVER(h, heading) LOG("$H", h);
    LOG("\n");
    log_headings_recursively(&pseudo_heading, 0);
}
void log_headings_recursively(heading *h, int depth) {
    int i;
    if (h==NULL) return;
    for (i=0; i<depth; i++) LOG("  ");
    LOG("$H\n");
    if (depth -1 != h->indentation) LOG("*** indentation should be %d ***\n", depth-1);
    log_headings_recursively(h->child_heading, depth+1);
    log_headings_recursively(h->next_heading, depth);
}
```

§37. Describing the heading structure, 2: to the index.

```
int headings_indexed = 0;
void index_headings(void) {
    INDEX("<b>Contents Page</b><p>");
    index_heading_recursively(pseudo_heading.child_heading);
    if (headings_indexed == 0) {
        INDEX("<p><i>The source text is not divided by headings.</i></p>\n");
    }
    INDEX("<hr><p>");
}
```

The function index_headings is invoked by a command in a `.i6t` template file.

§**38.** We index only headings of level 1 and up – so, not the pseudo-heading or the File (0) ones – and which are not within any extensions – so, are in the primary source text written by the user.

```
void index_heading_recursively(heading *h) {
    if (h == NULL) return;
    if ((h->level >= 1) &&
        (sf_get_extension_corresponding(h->start_location.file_of_origin) == NULL)) {
        indent to correct tab position
        open_html_paragraph(ifl, h->indentation, "tight");
        int j; for (j=0; j¡h-¿indentation; j++) INDEX("nbsp;nbsp;nbsp;nbsp;");
        write the text of the heading title
        print_raw_text_to_file(h->sentence_declaring->word_ref1,
            h->sentence_declaring->word_ref2, ifl);
        place a link to the relevant line of the primary source text
        index_link(h->start_location);
        INDEX("</p>\n");
        ⟨List all the objects and kinds created under the given heading, one tap stop deeper 39⟩;
        headings_indexed++;
    }
    index_heading_recursively(h->child_heading);
    index_heading_recursively(h->next_heading);
}
```

§**39.** We skip any objects or kinds without names (i.e., whose `creator` is null). The rest appear in italic type, and without links to source text since this in practice strews distractingly many orange berries across the page.

⟨List all the objects and kinds created under the given heading, one tap stop deeper 39⟩ ≡

```
world_object *wo;
int c = 0;
for (wo=h->list_of_contents; wo!=NULL; wo=wo->next_under_heading)
    if (wo->word_ref1 >= 0) {
        if (c++ == 0) {
            /* int j;
            for (j=0; j<h->indentation + 1; j++) INDEX("    "); */
            open_html_paragraph(ifl, h->indentation + 1, "tight");
            INDEX("<font color=\"#808080\">");
        } else INDEX(", ");
        INDEX("<i>");
        print_raw_text_to_file(wo->word_ref1, wo->word_ref2, ifl);
        INDEX("</i>");
    }
if (c > 0) INDEX("</font></p>\n");
```

This code is used in §38.

§**40. Describing the heading structure, 3: to a freestanding XML file.** This is provided as a convenience to the application using NI, which may want to have a pull-down menu or similar gadget allowing the user to jump to a given heading. This tells the interface where every heading is, thus saving it from having to parse the source.

The property list contains a single dictionary, whose keys are the numbers 0, 1, 2, ..., $n-1$, where there are n headings in all. (The pseudo-heading is not included.) A special key, the only non-numerical one, called "Application Version", contains the NI build number in its usual form: "4Q34", for instance.

```
void write_headings_as_xml(void) {
    STREAM xf_struct; STREAM *xf = &xf_struct;
    char *fn = index_filename(XML_HEADINGS_LEAFNAME, -1);
    if (STREAM_OPEN_TO_FILE(xf, fn, UTF8_ENC) == FALSE)
        fatal_error2("Can't open headings file", fn);
    write_headings_as_xml_inner(xf);
    STREAM_CLOSE(xf);
}

void write_headings_as_xml_inner(OUTPUT_STREAM) {
    heading *h;
    ⟨Write DTD indication for XML headings file 41⟩;
    WRITE("<plist version=\"1.0\"><dict>\n");
    INDENT;
    WRITE("<key>Application Version</key><string>%s</string>\n", NI_BUILD);
    LOOP_OVER(h, heading) {
        WRITE("<key>%d</key><dict>\n", h->allocation_id);
        INDENT;
        ⟨Write the dictionary of properties for a single heading 42⟩;
        OUTDENT;
        WRITE("</dict>\n");
    }
    OUTDENT;
```

```
    WRITE("</dict></plist>\n");
}
```

The function write_headings_as_xml is invoked by a command in a `.i6t` template file.

§**41.** We use a convenient Apple DTD:

⟨Write DTD indication for XML headings file 41⟩ ≡

```
    WRITE("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
        "<!DOCTYPE plist PUBLIC \"-//Apple Computer//DTD PLIST 1.0//EN\" "
        "\"http://www.apple.com/DTDs/PropertyList-1.0.dtd\">\n");
```

This code is used in §40.

§**42.** Note that a level of 0, and a title of `--`, signifies a File (0) level heading: external tools can probably ignore such records. Similarly, it is unlikely that they will ever see a record without a "Filename" key – this would mean a heading arising from text created internally within NI, which will only happen if someone has done something funny with `.i6t` files – but should this arise then the best recourse is to ignore the heading.

⟨Write the dictionary of properties for a single heading 42⟩ ≡

```
    if (h->start_location.file_of_origin)
        WRITE("<key>Filename</key><string>%s</string>\n",
            sf_get_filename(h->start_location.file_of_origin));
    WRITE("<key>Line</key><integer>%d</integer>\n", h->start_location.line_number);
    if (h->sentence_declaring != NULL) {
        WRITE("<key>Title</key><string>");
        print_raw_text_to_file(h->sentence_declaring->word_ref1,
            h->sentence_declaring->word_ref2, OUT);
        WRITE("</string>\n");
    } else {
        WRITE("<key>Title</key><string>--</string>\n");
    }
    WRITE("<key>Level</key><integer>%d</integer>\n", h->level);
    WRITE("<key>Indentation</key><integer>%d</integer>\n", h->indentation);
```

This code is used in §40.

Verb Phrases 4/verb

Purpose

To construct verb-phrase nodes in the parse tree.

Template interpreter commands

```
1   {-callv:traverse_for_verbs}
```

Definitions

¶1. At this point in the narrative, we have read files from disc, lexed the text into a stream of words, and broken this into a list of sentences; we have identified requests to include extensions, and fully acted on these, so that we can now forget about that whole complication; and we have built a tree of headings and subheadings (and file divisions) so that we have a clear map of the overall structure of the source text. Sentences intended for use only in some circumstances (for instance, when compiling for the Glulx virtual machine) have been omitted as necessary, so that we can forget about that complication, too.

This gives as much information as we can squeeze out by easily specified mechanical means: we have attacked the text at the very small scale, letters and words, and at the very large, headings and files. This zig-zag in scale will continue. In the rest of this chapter, we find the overall structure of sentences; in chapters 5, 6, and 7 we descend to the level of excerpts of two or three words at a time, and will not get back up to sentence level again until chapter 8. As for going up further to the larger structure of multi-sentence rules, that will have to wait until chapter 12.

¶2. The parse tree is currently a long, long list: each sentence is a node which is a child of the root, but no sentence has any child nodes of its own. (That is about to change.) We can divide these sentences into three:

(a) Structural sentences – headings, extension requests, extension bookends. All these have now been dealt with.

(b) Sentences inside rules: rule preambles (`ROUTINE_NT` nodes) and phrases (`COMMAND_NT`). These will not even be looked at until the second phase of compilation, after the model world has been created.

(c) Sentences with primary verbs, having node type `SENTENCE_NT`. These are the assertions: they make statements about the initial state of the model world – the existence of places and things, and their properties at the start of play – and which describe patterns of behaviour during play.

In the present section of code, then, we identify the primary verbs of assertion sentences, and deal right away with some of the easier cases, while leaving the harder ones for chapter 8 to sort out.

¶3. Every `SENTENCE_NT` node is annotated with a verb type from the enumeration below. All of the assertions which create objects and kinds, and put them into relationships with each other – a tremendous variety of possible sentences, between them making up about three-quarters of all `SENTENCE_NT` nodes in typical source – fall into one of two verb types:

define `ASSERT_VB` 10 *"The bat and ball are on the table."*
define `HAS_VB` 11 *"The silver bars have score 10."*

¶4. All other sentences are exceptional cases with much more specific wording. Though there may appear to be a great many of these, each is relatively rarely-used, and some are so exceptional that most Inform users are unaware of them. First, some verb types which we might call *notational*, since they change Inform's notation:

define USEMEANS_VB 20 *"Use American dialect means ..."*
define TRANSLATES_VB 21 *"Lighted translates into I6 as "light""*
define TRANSLATESU_VB 22 *"Black king chess piece translates into Unicode as 9818"*
define PLURAL_VB 23 *"The plural of woman is women."*
define SPECIFIES_VB 24 *"10'23 specifies a running time."*

¶5. Secondly, ways to create constructions other than objects. (Objects and kinds are constructed by standard assertions: these more abstract forms are made using special sentence forms partly to avoid misinterpreting their names as objects, and partly to allow us to use more elaborate and specific grammar for the details, as with actions.)

define DEFINED_BY_VB 30 *"The colours are defined by Table 1."*
define EPISODE_VB 31 *"The story is episode..."*
define FIGURE_VB 32 *"Figure... is the file..."*
define SOUND_VB 33 *"Sound... is the file..."*
define FILE_VB 34 *"File... is the file..."*
define CANBE_VB 35 *"A door can be open or closed."*
define NEW_ACTION_VB 36 *"Taking something is an action."*
define NEW_ACTIVITY_VB 37 *"Description is an activity."*
define NEW_VERB_VB 38 *"The verb to eat..."*
define NEW_RELATION_VB 39 *"Knowledge relates..."*

¶6. Thirdly, some relationships between these abstractions. (Whereas, again, relationships between objects are established with ordinary assertions.)

define IN_RULEBOOK_VB 40 *"The time passes rule is listed in the turn sequence rulebook."*
define NOT_IN_RULEBOOK_VB 41 *"The blossom rule is not listed in..."*
define BEGINS_WHEN_VB 42 *"Scene begins when ..."*
define ENDS_WHEN_VB 43 *"Scene ends when ..."*

¶7. Finally, the remaining verb types are all direct commands to Inform – note the imperative forms they take: Use, Understand, Include, and so forth. In a sense the whole source text is an instruction to Inform, but mostly it's a passive one: the implicit message is "make the world so that all this comes right". Here, on the other hand, the user actually speaks directly. This is a point of the design which has sometimes seemed a little doubtful – wouldn't it be more consistent for all of these sentences to be more passively worded? – but pragmatism won out: circumlocutions such as "American dialect is used." or "The story file is released along with..." are plausible enough, but

"take noun" is understood as taking the noun.

would mean a lot of important sentences being oddly punctuated with no initial capital letter, while forcing meaningless extra words, as in

The command "take noun" is understood as taking the noun.

might prove annoying. Users seem to find the directness of the imperative easier to use, at any rate, and perhaps the difference in mood helps to clarify that these are sentences rather different in implication from the usual sort.

define USE_VB 50 *"Use American dialect."*

```
define UNDERSTAND_VB 51                    "Understand "take [noun]" as taking the noun."
define TEST_VB 52                          "Test ... with "...".."
define DEBUG_VB 53                         "Include ... in the debugging log."
define DOC_VB 54                           "Document ... at ..."
define RELEASE_VB 55                       "Release along with..."
define MAP_PARAMETER_VB 56                 "Index map with ..."
```

---

§**1.** As with headings, so with SENTENCE_NT nodes: we want the ability to come back later and add some more. That means that the primary-verb-finder needs to be able to make more than one pass through. To handle this, all SENTENCE_NT nodes are annotated on creation with the "sentence unparsed" marker: we run through the top level of the parse tree, look at all nodes with this marker, parse their associated sentences, and remove the marker from them. (So, for instance, if this is run twice in quick succession, the second run-through does nothing.)

```
void traverse_for_verbs(void) {
    parse_node *PN;
    for (TREE_START(PN); PN; TREE_NEXT(PN))
        if ((pn_get_node_type(PN) == SENTENCE_NT) &&
            (pn_int_annotation(PN, sentence_unparsed_ANNOT))) {
            look_for_verb_phrase(PN);
            check_sentence_for_direction_creation(PN);
            pn_annotate_int(PN, sentence_unparsed_ANNOT, FALSE);
        }
}
```

The function traverse_for_verbs is invoked by a command in a .i6t template file.

§**2. Primary verbs.** As will be seen below, the parsing of all the exceptional cases – those which come out as anything other than ASSERT_VB or HAS_VB – is done very crudely. The first match wins, and there is no backtracking. It might reasonably be asked: why don't we handle all this with the S-grammar, the much more sophisticated parser of conditions such as "if six bricks are in the skip"? We would have to make some new verbs and relations, but we would get much better resolution of ambiguities, and could cut out some of the tedious code below. There are two answers: first, because some of these verbs handle noun phrases which are exotic in structure and could not be fitted easily into the S-grammar's noun phrase definition (consider the sentence constructing an action); second, because we expect at least some of the noun phrases in these sentences to be previously undeclared names. To parse

> The coral snake is in the green bucket.

at a time when neither snake nor bucket has been mentioned before, we really have little option but to look for "is" plus preposition, and cannot use the words either side as any support for the hypothesis that this is indeed the verb. (The main S-grammar parser would accept "NP_1 is in NP_2" only where it could satisfy itself that the two noun phrases were well-formed.) So that's why we don't use the S-grammar at this early stage. The reason we don't backtrack is that it seems to be unnecessary in practice, and anyway in principle we don't really want people to use these sentences in ways that are misleading to the eye.

The order in which the possibilities are checked is important: we must check "X is an action" before "X is Y", for instance.

```
void look_for_verb_phrase(parse_node *PN) {
    int w1, w2;
    [[w1, w2 <-- PN]];
    ⟨Maybe this is a Unicode translation sentence 3⟩;
    ⟨Maybe this is an Understand or Test sentence 4⟩;
```

```
    ⟨Maybe this is a debugging log sentence 5⟩;
    ⟨Maybe this is a Document... at... sentence 6⟩;
    ⟨Maybe this is a scene anchoring sentence 7⟩;
    ⟨Maybe this is some new notation 8⟩;
    ⟨Maybe this is a resource-declaring sentence 9⟩;
    ⟨Maybe this is an action or activity declaration 10⟩;
    ⟨Maybe this is a verb or relation declaration 11⟩;
    ⟨Maybe this is a property declaration 12⟩;
    ⟨Maybe this is a rule-positioning sentence 13⟩;
    ⟨More likely this is an assertion about objects and kinds 14⟩;
    ⟨If not, it might be an I6 translation 19⟩;
    ⟨Or possibly a use option, release instruction or index map hint 20⟩;
    ⟨Issue a problem message if a primary verb in the past tense is found 21⟩;
}
```

The function look_for_verb_phrase is called from 4/sent and 4/ofs.

§**3.** We check Unicode translations first of all, because we haven't any control over the wording of character names in the Unicode standard. Among the 12,997 definitions used in the Unicode Full Character Names extension are such choice examples as "downwards arrow from bar", "arabic hamza above", "kangxi radical use" and so forth, and we don't want to misread "from", "above", "use", and so on, as prepositions or verbs: in sentences like this one they are nouns.

As we shall see, `try_verb` looks for the given sequence of words in mid-sentence, assigning the given verb type to the sentence node and returning true if it succeeds.

⟨Maybe this is a Unicode translation sentence 3⟩ ≡

```
    if (try_verb(translates_V, into_V, unicode_V, PN, TRANSLATESU_VB, 0))
        return;
```

This code is used in §2.

§**4.** "Understand... as..." and "Test... with...":

⟨Maybe this is an Understand or Test sentence 4⟩ ≡

```
    if ([[w1, w2 == understand ...]] &&
        (try_verb(as_V, 0, 0, PN, UNDERSTAND_VB, 0))) return;
    if ([[w1, w2 == test ...]] &&
        (try_verb(with_V, 0, 0, PN, TEST_VB, 0))) return;
```

This code is used in §2.

§**5.** Two special sentence forms for toggling what does and doesn't go into the debugging log: most users do not know about this feature, but in fact it's fully enabled in the shipping version of the application.

We act immediately on these sentences, so that debugging information is available at the earliest opportunity. Following the successful call to `try_verb`, the sentence will have the not very grammatically sound subtree:

```
SENTENCE_NT "Include object creations in the debugging log"
    VERB_NT "in the debugging"
    NOUNPHRASE_NT "Include object creations"
    NOUNPHRASE_NT "log"
```

We don't care that this is a bit of a mess: we simply take the first noun phrase, strip it of its first word (necessarily "Include") and pass the text to the logging aspects code.

⟨Maybe this is a debugging log sentence 5⟩ ≡

```
if ([[w1, w2 == include ...]] &&
    (try_verb(in_V, the_V, debugging_V, PN, DEBUG_VB, 0))) {
    set_debugging_aspect(
        PN->down->next->word_ref1+1, PN->down->next->word_ref2, TRUE);
    return;
}
if ([[w1, w2 == omit ...]] &&
    (try_verb(from_V, the_V, debugging_V, PN, DEBUG_VB, 0))) {
    set_debugging_aspect(
        PN->down->next->word_ref1+1, PN->down->next->word_ref2, FALSE);
    return;
}
```

This code is used in §2.

§**6.** A special sentence intended for use only in the Standard Rules, for relating documentation symbols to section numbers of the documentation.

⟨Maybe this is a Document... at... sentence 6⟩ ≡

```
if ([[w1, w2 == document ...]] &&
    (try_verb(at_V, 0, 0, PN, DOC_VB, 0))) return;
```

This code is used in §2.

§**7.** For anchoring scenes together. The syntax for different scene endings means that the sentence form we need to recognise is a little bit worryingly general: anything in the form "...ends... when..." What saves the situation is that "when" clauses, though often attached to rules, are almost never naturally found in assertion sentences other than "Understand" ones (and we have already dealt with those).

⟨Maybe this is a scene anchoring sentence 7⟩ ≡

```
int i, j;
if (try_verb(begins_V, when_V, 0, PN, BEGINS_WHEN_VB, 0)) return;
if (try_verb(ends_V, when_V, 0, PN, ENDS_WHEN_VB, 0)) return;
if ([[w1, w2 == ... ends ... when ... : i, j]] &&
    (try_verb(when_V, 0, 0, PN, ENDS_WHEN_VB, 0))) return;
```

This code is used in §2.

§**8.** Three easy cases to dispose of:

⟨Maybe this is some new notation 8⟩ ≡
```
if (try_verb(specifies_V, 0, 0, PN, SPECIFIES_VB, 0)) return;
if (try_verb(are_V, defined_V, by_V, PN, DEFINED_BY_VB, 0)) return;
if ([[w1, w2 == the story is episode ...]] &&
    (try_verb(story_V, is_V, episode_V, PN, EPISODE_VB, 0))) return;
if ([[w1, w2 == the plural of ...]] &&
    (try_verb(is_V, 0, 0, PN, PLURAL_VB, 0))) return;
```

This code is used in §2.

§**9.** And the various external resources which can be added to a project. The syntax for files is a little more involved: the word "the" is optional, and so is a keyword giving the nature of the file.

⟨Maybe this is a resource-declaring sentence 9⟩ ≡
```
int fw1 = w1, fw2 = w2;
if ([[w1, w2 == figure ...]] &&
    (try_verb(is_V, the_V, file_V, PN, FIGURE_VB, 0))) return;
if ([[w1, w2 == sound effect ...]] &&
    (try_verb(is_V, the_V, file_V, PN, SOUND_VB, 0))) return;
if ([[w1, w2 == sound ...]] &&
    (try_verb(is_V, the_V, file_V, PN, SOUND_VB, 0))) return;
[[fw1, fw2 == the ... --> fw1, fw2]];
[[fw1, fw2 == text/binary ... --> fw1, fw2]];
if ([[fw1, fw2 == file ...]] &&
    (try_verb(is_V, called_V, 0, PN, FILE_VB, 0))) return;
```

This code is used in §2.

§**10.** We could conceivably have implemented "action" and "activity" as pseudo-kinds, and thus handled sentences like these through ordinary assertions, but it would have been a lot of fuss. So we do it the simple-minded way.

Note that activity declarations always simply end "is an activity.", thus having nothing interesting by way of an object noun phrase, whereas action declarations continue with usually extensive further text: "... is an action applying to two visible things.", say.

⟨Maybe this is an action or activity declaration 10⟩ ≡
```
if ([[w1, w2 == ... is an activity]] &&
    (try_verb(is_V, an_V, 0, PN, NEW_ACTIVITY_VB, 0))) return;
if (try_verb(is_V, an_V, action_V, PN, NEW_ACTION_VB, 0)) return;
```

This code is used in §2.

§**11.** Definitions of verbs and relations must be acted on immediately, because they may affect the way we find primary verbs in sentences which follow directly on – that is, the very next sentence might use the verb being defined here, so we can't wait.

⟨Maybe this is a verb or relation declaration 11⟩ ≡

```
if ([[w1, w2 == the verb to ...]] &&
    (try_verb(implies_V, 0, 0, PN, NEW_VERB_VB, 0))) {
    parse_new_verb(PN);
    return;
}
if ([[w1, w2 == ### relates ...]] &&
    (try_verb(to_V, 0, 0, PN, NEW_RELATION_VB, 0))) {
    parse_new_relation(PN);
    return;
}
```

This code is used in §2.

§**12.** "Can be", "can have", "is either":

⟨Maybe this is a property declaration 12⟩ ≡

```
int i;
if ([[w1, w2 == ... can ... : i]] && [[i, w2 == can have ...]]) {
    sentence_problem(_P_(C4CanHave),
        "'can have' is not a form allowed by Inform",
        "though it looks as it might be. If you want to set up a property "
        "with a limited number of options, you can say 'A room can be "
        "spacious, cramped or adequate.', for instance; if you want a "
        "property with a value, 'A room has a number called the secret "
        "code number.'. So 'can be' and 'has' are fine, but not 'can have'.");
    return;
}
if (try_verb(can_V, be_V, 0, PN, CANBE_VB, 0)) return;
if (try_verb(is_V, either_V, 0, PN, CANBE_VB, 0)) return;
```

This code is used in §2.

§**13.** Sentences giving positions in rulebooks use the preposition "listed":

⟨Maybe this is a rule-positioning sentence 13⟩ ≡

```
if (try_verb(is_V, listed_V, 0, PN, IN_RULEBOOK_VB, 0)) return;
if (try_verb(are_V, listed_V, 0, PN, IN_RULEBOOK_VB, 0)) return;
if (try_verb(is_V, not_V, listed_V, PN, NOT_IN_RULEBOOK_VB, 0)) return;
if (try_verb(are_V, not_V, listed_V, PN, NOT_IN_RULEBOOK_VB, 0)) return;
```

This code is used in §2.

§14. In some sense, the other cases are all exceptions: this is the big one, the general assertion reader, which uses the same verbs and prepositions as the fully general *S*-grammar.

It is potentially quite slow to test every word against every possible verb, even though there are typically fairly few verbs in the *S*-grammar, so we confine ourselves to words flagged in the vocabulary as being used in verbs. It's important to note that (for reasons given above) there can be no backtracking here: we will have to accept the first match we can. That makes it necessary to test these verbs in the correct order, and this is why we make three passes. "To have" has top priority, because the matter on either side may contain some surprising words, thanks to the syntax for declaring property values. "To be" is then given priority over everything else; and otherwise the leftmost valid verb phrase is accepted. The reason "to be" is given precedence is partly philosophical (it is after all the linguistic expression of the equality relation, which plays a special role in predicate calculus), but also practical. Consider:

The Fisher-Price carry cot is a container.

In this sentence, "carry" is intended as part of the subject noun phrase, but we have no way of telling that: if we didn't give "to be" priority over "to carry" we would construe this sentence as saying that a group of people called The Fisher-Price, perhaps a rock group, are carrying an object called "cot is a container", perhaps their new EP of remixed techno lullabies. (Far fewer plausible noun phrases contain "is" than contain other verbs such as "carry".)

⟨More likely this is an assertion about objects and kinds 14⟩ ≡

```
int pos, there_flag = FALSE;
if [[w1, w2 == there is/are ...]] {
    there_flag = TRUE;
    w1 += 2;
}
for (pos = w1+1; pos<w2; pos++) if ([[word pos == has/have]] && (![[word pos-1 == not]]))
    ⟨Look for a present-tense verb phrase at the current position 15⟩;
for (pos = w1+1; pos<w2; pos++) if [[word pos == is/are]]
    ⟨Look for a present-tense verb phrase at the current position 15⟩;
for (pos = w1+1; pos<w2; pos++) if (vocab_test_flags(pos, CVERB_MC))
    ⟨Look for a present-tense verb phrase at the current position 15⟩;
if (there_flag) {
    for (pos = w1+1; pos<w2; pos++) if (vocab_test_flags(pos, PREPOSITION_MC)) {
        preposition_usage *pu;
        LOOP_OVER(pu, preposition_usage) {
            int j = pu_parse_text_against(pos, w2, pu, FALSE);
            if (j >= 0) {
                binary_predicate *bp = bp_get_reversal(pu_get_meaning(pu));
                assign_verb(PN, j, w2, w1, pos-1, pos, j-1, ASSERT_VB, bp);
                ⟨Insert a RELATIONSHIP node into the parse tree 18⟩;
                return;
            }
        }
    }
    w1 -= 2; pos = w1+1; there_flag = FALSE;
    ⟨Look for a present-tense verb phrase at the current position 15⟩;
}
```

This code is used in §2.

§**15.** The loop here is taken over all valid verb phrases in the *S*-grammar, but then there are a string of exemptions. Many *S*-sentences valid in conditions are not valid in assertions, and besides, we don't have the luxury (as the *S*-sentence parser has) of knowing in advance all the possible noun phrases we might meet – so we have to be very cautious, and not accept any verb where there are good reasons to suspect that it is not meant by the user.

(1) We discard verbs in the past tense: they are not legal in assertions.

(2) We discard "is not" and "are not". For the time being, if we see something like "Velma is not a thinker" then we will parse it as "(Velma) is (not a thinker)", allowing "not a thinker" to be a noun phrase. (Assertions are generally supposed to be positive statements, not negative ones, but we don't necessarily know about this one yet. If "thinker" is an either/or property with a single possible antonym – "doer", let's say – then we want to construe this sentence as if it read "Velma is a doer". So, anyway, we allow "not" at the front of the object noun phrase, and disallow "is not" and "are not" in order that this can happen.) All other negated verb usages are parsed here – but only in order for them to run into the problem message below.

(3) We discard a verb in a relative phrase, since it is not the primary verb of the sentence: for instance, the "is" in "cat which is in the basket" will not be considered.

(4) We discard a verb if the subject noun phrase would read simply "the".

(5) We discard a verb if it is unexpectedly given in upper case form. Thus "The Glory That Is Rome is a room" will be read as "(The Glory That Is Rome) is (a room)", not "(The Glory That) is (Rome is a room)".

⟨Look for a present-tense verb phrase at the current position 15⟩ ≡
```
verb_usage *vu;
LOOP_OVER(vu, verb_usage)
    if ((vu_get_tense_used(vu) == IS_TENSE) &&
        (vu != negated_to_be) && (vu != negated_plural_to_be) &&
        (!((pos==w1+1) && [[word w1 == the]])) &&
        (unexpectedly_upper_case(pos) == FALSE)) {
        int rp = FALSE;
        if ((pos > w1) && [[word pos-1 == which/who/that]]) rp = TRUE;
        if (rp == there_flag)
            ⟨Test the words at the current position against this verb usage 16⟩;
    }
```
This code is used in §14.

§**16.** We delegate to the *S*-sentence parser to actually match the text of the verb usage.

As noted above, "to be" plays a special role because of its unique conceptual role in language and in predicate calculus: it describes or assigns the identity of things. It is less obvious why we treat "to have" differently: surely this is just another relationship? The answer is that it is indeed a relationship in sentences like "Peter has the magic flute", but not in "Peter has carrying capacity 2": we overload "to have" with two independent meanings, and this will be tricky to resolve, so it gets its own verb type for the assertion-maker to read.

⟨Test the words at the current position against this verb usage 16⟩ ≡
```
int j = vu_parse_text_against(pos, w2, vu);
if (j >= 0) {
    int subj1 = w1, subj2 = pos-1, obj1 = j, obj2 = w2;
    if (rp) subj2--;                        remove the word "which" if the verb used in a relative phrase
    ⟨Issue a problem message if the primary verb has a negative sense 17⟩;
    if (vu_get_meaning(vu) == a_has_b_predicate) {
        assign_verb(PN, subj1, subj2, obj1, obj2, pos, j-pos, HAS_VB, vu_get_meaning(vu));
    } else {
        binary_predicate *bp = vu_get_meaning(vu);
        assign_verb(PN, subj1, subj2, obj1, obj2, pos, j-pos, ASSERT_VB, vu_get_meaning(vu));
        if (bp != a_is_b_predicate)
            ⟨Insert a RELATIONSHIP node into the parse tree 18⟩;
    }
    return;
}
```
This code is used in §15.

§**17.** This catches sentences like "Timothy does not carry the ring". See above for how and why "is not" and "are not" are exempted from this for now:

⟨Issue a problem message if the primary verb has a negative sense 17⟩ ≡
```
if (vu_is_used_negatively(vu)) {
    negative_sentence_problem(_P_(C4NegatedVerb1));
    return;
}
```
This code is used in §16.

§**18.** Every primary verb other than "to be" and "to have" expresses a relationship between the objects referred to by its subject and object noun phrases, and this relationship is a binary predicate in our calculus. So, for instance, we make the transformation

```
SENTENCE_NT "The cat is on the mat"  --->  SENTENCE_NT "The cat is on the mat"
    VERB_NT "is on"                            VERB_NT "is on"
    NOUNPHRASE_NT "cat"                        NOUNPHRASE_NT "cat"
    NOUNPHRASE_NT "mat"                        RELATIONSHIP_NT "is on" = supports
                                                   NOUNPHRASE_NT "mat"
```

The sense of the binary predicate is reversed here because we are applying it to the object of the sentence not the subject: we thus turn the idea of the cat being supported by the mat into the exactly equivalent idea of the mat supporting the cat.

⟨Insert a RELATIONSHIP node into the parse tree 18⟩ ≡

```
parse_node *REL_PN = new_node(RELATIONSHIP_NT);
pn_set_relationship(REL_PN, bp_get_reversal(bp));
REL_PN->down = PN->down->next->next;
REL_PN->word_ref1 = pos; REL_PN->word_ref2 = j-1;
PN->down->next->next = REL_PN;
```

This code is used in §14,16,14,16,14,16.

§**19.** Mapping verbal names like "can't take what's already taken rule" onto I6 identifiers used in the I6 library is something done almost always by the Standard Rules, and thus so early in the source text that only standard verbs have been created. So these can afford to be low-priority sentence forms:

⟨If not, it might be an I6 translation 19⟩ ≡

```
if (try_verb(translates_V, into_V, i6_V, PN, TRANSLATES_VB, 0)) return;
```

This code is used in §2.

§**20.** And that just leaves use options, release instructions and hints for the map index. The `USE_VB` verb, uniquely, constructs a sentence subtree with only a single noun phrase: the text of that which is to be used.

```
SENTENCE_NT "Use American spelling"
    VERB_NT "use"
    NOUNPHRASE_NT "American spelling"
```

⟨Or possibly a use option, release instruction or index map hint 20⟩ ≡

```
if [[w1, w2 == use ...]] {
    parse_node *VP_PN, *NP_PN;
    if (try_verb(translates_V, as_V, 0, PN, USEMEANS_VB, 0)) return;
    VP_PN = new_node(VERB_NT);
    VP_PN->word_ref1 = PN->word_ref1; VP_PN->word_ref2 = PN->word_ref1;
    pn_annotate_int(VP_PN, verb_id_ANNOT, USE_VB);
    graft(VP_PN, PN);
    NP_PN = new_nounphrase_raw(PN->word_ref1+1, PN->word_ref2);
    graft(NP_PN, PN);
    return;
}
if ([[w1, w2 == release along with ...]] &&
    (try_verb(along_V, with_V, 0, PN, RELEASE_VB, 0))) return;
if ([[w1, w2 == index map with ...]] &&
    (try_verb(map_V, with_V, 0, PN, MAP_PARAMETER_VB, 0))) return;
```

This code is used in §2.

§**21.** This final case never matches a legal sentence: it simply hoovers up usages of past tense assertion verbs in order to give them a better Problem message than the one they will otherwise receive later on.

⟨Issue a problem message if a primary verb in the past tense is found 21⟩ ≡

```
    int pos;
    for (pos = w1+1; pos<w2; pos++) if (vocab_test_flags(pos, CVERB_MC)) {
        verb_usage *vu;
        if ((pos > w1) && [[word pos-1 == which/who/that]]) continue;
        if ((pos==w1+1) && [[word w1 == the]]) continue;
        if (unexpectedly_upper_case(pos)) continue;
        LOOP_OVER(vu, verb_usage)
            if (vu_get_tense_used(vu) != IS_TENSE) {
                int j = vu_parse_text_against(pos, w2, vu);
                if (j >= 0) {
                    LOG("Past tense verb <$W> in <$W>\n", pos, j-1, w1, w2);
                    sentence_problem(_P_(C4NonPresentTense),
                    "assertions about the initial state of play must be "
                    "given in the present tense",
                    "so 'The cat is in the basket' is fine but not 'The "
                    "cat has been in the basket'. Time is presumed to start "
                    "only when the game begins, so there is no anterior "
                    "state which we can speak of.");
                    return;
                }
            }
    }
```

This code is used in §2.

§**22. Detecting exceptional verbs.** All of the exceptional sentence forms (except `USE_VB`) test for their verbs using the routine `try_verb`, which looks for a sequence of one to three words in the middle of the sentence, optionally requiring them not to be preceded by a given "avoid" word.

```
int try_verb(vocabulary_entry *t1, vocabulary_entry *t2, vocabulary_entry *t3,
    parse_node *pn, int the_verb, vocabulary_entry *avoid_t) {
    int i, count = 1;
    i = is_sequence_intermediate(t1, t2, t3, avoid_t, pn->word_ref1, pn->word_ref2);
    if (i == -1) return FALSE;
    if (t2 != 0) count++; if (t3 != 0) count++;
    assign_verb(pn, pn->word_ref1, i-1, i+count, pn->word_ref2, i, count, the_verb, NULL);
    return TRUE;
}
```

§**23. Building verb subtrees.** Whenever any primary verb (except `USE_VB`) is discovered – either by one of the exceptional cases, using `try_verb`, or through the restricted *S*-grammar parser above – the following routine is always called. It gives the sentence node three children: verb phrase, subject noun phrase, object noun phrase.

```
SENTENCE_NT "Railway Departure begins when the player is in the train"
    VERB_NT "begins when"
    NOUNPHRASE_NT "Railway Departure"
    NOUNPHRASE_NT "the player is in the train"
```

This is an example of the simplest form of verb subtree, arising from a "raw" verb type: nothing at all is done with the noun phrases, which simply consume all the words to the left and right of the VP.

Alternatively, a verb type can be "cooked", which means that the noun phrases will be worked on a little: articles and qualifiers converted to annotations, and some sub-clauses discovered (as we shall see in the next section). For instance:

```
SENTENCE_NT "An oak table is usually a supporter"
    VERB_NT "is" certainty:LIKELY_CE
    NOUNPHRASE_NT "oak table" article:indefinite
    NOUNPHRASE_NT "supporter" article:indefinite
```

The use of qualifiers is interesting here, because again it deviates from the practice of the main *S*-grammar. We allow "usually" and such like words on either side of the VP in a sentence, but only here in assertions: not in the main *S*-grammar. This is because the assertion

A box is usually closed.

is essentially a statement about the future, not the present or the past, whereas conditions used in phrases and rules must always be determinable at once: it makes no sense to write "if a box is usually closed", since Inform cannot know what will generally happen, only what is now the case and what has been the case in the past.

§**24.** The noun phrase grammar used for NPs of cooked verbs sometimes needs some temporary context – it needs to know in particular whether it is serving the copular verb "to be" or not. The following global variable is temporarily used to save passing it endlessly as a function argument:

```
int assertion_NP_is_in_VP_for_to_be = FALSE;
```

§**25.** Other than setting this flag, we need only determine raw vs. cooked and then make the three parts of the subtree.

```
void assign_verb(parse_node *SENTENCE_PN, int subj1, int subj2, int obj1, int obj2,
    int pos, int count, int verb_type, binary_predicate *bp) {
    int raw_mode, restrict_object_to_lists = FALSE, admits_qualifiers = FALSE,
        verb_w1 = pos, verb_w2 = pos + count - 1;
    ⟨Determine whether this primary verb is raw or cooked 26⟩;
    ⟨Make the VERB node in the sentence subtree 27⟩;
    assertion_NP_is_in_VP_for_to_be = FALSE;
    if (bp == a_is_b_predicate) assertion_NP_is_in_VP_for_to_be = TRUE;
    ⟨Make the NOUNPHRASE node(s) for the subject 29⟩;
    ⟨Make the NOUNPHRASE node(s) for the object of other verbs 30⟩;
}
```

§**26.** Basically, the only cooked verbs are those whose noun phrases are likely to be actual objects or kinds.

⟨Determine whether this primary verb is raw or cooked 26⟩ ≡
```
switch(verb_type) {
    case ASSERT_VB: case HAS_VB:
        raw_mode = FALSE; restrict_object_to_lists = FALSE; admits_qualifiers = TRUE; break;
    case DEFINED_BY_VB:
        raw_mode = FALSE; restrict_object_to_lists = FALSE; admits_qualifiers = FALSE; break;
    case CANBE_VB:
        raw_mode = FALSE; restrict_object_to_lists = TRUE; admits_qualifiers = FALSE; break;
    default:
        raw_mode = TRUE; admits_qualifiers = FALSE; break;
}
```
This code is used in §25.

§**27.** The verb node has two possible annotations: the verb ID, which is the type number we have been using above, and the level of certainty, which is set in the following section (but only for cooked verbs).

⟨Make the VERB node in the sentence subtree 27⟩ ≡
```
parse_node *VP_PN = new_node(VERB_NT);
if (admits_qualifiers) ⟨Spread the verb phrase right or left to include any qualifier 28⟩;
VP_PN->word_ref1 = verb_w1; VP_PN->word_ref2 = verb_w2;
pn_annotate_int(VP_PN, verb_id_ANNOT, verb_type);
graft(VP_PN, SENTENCE_PN);
```
This code is used in §25.

§**28.** Look before and after the verb phrase to see if any qualifier is used, and if so spread the word range of the verb phrase left or right as needed; and annotate its node in the parse tree with the certainty level implied.

⟨Spread the verb phrase right or left to include any qualifier 28⟩ ≡
```
int w1, w2, certainty = UNKNOWN_CE, quals_found = 0;
[[w1, w2 <-- SENTENCE_PN]];
if (pos > w1+1) {
    certainty = try_qualifier(pos-1);
    if (certainty != UNKNOWN_CE) verb_w1--, subj2--, quals_found++;
}
if (pos < w2-1) {
    certainty = try_qualifier(pos+count);
    if (certainty != UNKNOWN_CE) verb_w2++, obj1++, quals_found++;
}
if (quals_found == 1)
    pn_annotate_int(VP_PN, verbal_certainty_ANNOT, certainty);
if (quals_found == 2)
    sentence_problem(_P_(C4TwoLikelihoods),
        "this sentence seems to have a likelihood qualification on both "
        "sides of the verb",
        "which is not allowed. 'The black door certainly is usually open' "
        "might possibly be grammatical English in some idioms, but Inform "
        "doesn't like a sentence in this shape because the 'certainly' "
        "on one side of the verb and the 'usually' on the other are "
        "rival indications of certainty.");
```
This code is used in §27.

§**29.** The subject is then just a nounphrase made from all the words to the left of the verb:

⟨Make the NOUNPHRASE node(s) for the subject 29⟩ ≡

```
parse_node *SUBJECT_NP_PN;
if (raw_mode) SUBJECT_NP_PN = new_nounphrase_raw(subj1, subj2);
else SUBJECT_NP_PN = new_nounphrase_worldly(subj1, subj2, TRUE);
graft(SUBJECT_NP_PN, SENTENCE_PN);
```

This code is used in §25.

§**30.** While the object is a nounphrase made from all the words to the right. We allow the objects of a few cooked verbs to be cooked only lightly, so to speak (rare?): a mode in which lists are broken up into individual items, but other clauses are not detached.

⟨Make the NOUNPHRASE node(s) for the object of other verbs 30⟩ ≡

```
parse_node *OBJECT_NP_PN;
if (raw_mode) OBJECT_NP_PN = new_nounphrase_raw(obj1, obj2);
else if (restrict_object_to_lists) OBJECT_NP_PN = new_nounphrase_listdivided(obj1, obj2);
else OBJECT_NP_PN = new_nounphrase_worldly(obj1, obj2, FALSE);
graft(OBJECT_NP_PN, SENTENCE_PN);
```

This code is used in §25.

§**31. Qualifiers.** Parsing of qualifiers which express varying degrees of certainty:

```
int try_qualifier(int w) {
    if [[word w == always/certainly]] return CERTAIN_CE;
    if [[word w == usually/normally]] return LIKELY_CE;
    if [[word w == rarely/seldom]] return UNLIKELY_CE;
    if [[word w == never]] return IMPOSSIBLE_CE;
    return UNKNOWN_CE;
}
```

§**32. Logging verb numbers.**

```
void log_verb_number(int verb_number) {
    switch(verb_number) {
        case ASSERT_VB: LOG("ASSERT_VB"); break;
        case BEGINS_WHEN_VB: LOG("BEGINS_WHEN_VB"); break;
        case CANBE_VB: LOG("CANBE_VB"); break;
        case DEBUG_VB: LOG("DEBUG_VB"); break;
        case DEFINED_BY_VB: LOG("DEFINED_BY_VB"); break;
        case DOC_VB: LOG("DOC_VB"); break;
        case ENDS_WHEN_VB: LOG("ENDS_WHEN_VB"); break;
        case EPISODE_VB: LOG("EPISODE_VB"); break;
        case FIGURE_VB: LOG("FIGURE_VB"); break;
        case FILE_VB: LOG("FILE_VB"); break;
        case HAS_VB: LOG("HAS_VB"); break;
        case IN_RULEBOOK_VB: LOG("IN_RULEBOOK_VB"); break;
        case MAP_PARAMETER_VB: LOG("MAP_PARAMETER_VB"); break;
        case NEW_ACTION_VB: LOG("NEW_ACTION_VB"); break;
        case NEW_ACTIVITY_VB: LOG("NEW_ACTIVITY_VB"); break;
        case NEW_RELATION_VB: LOG("NEW_RELATION_VB"); break;
        case NEW_VERB_VB: LOG("NEW_VERB_VB"); break;
```

```
        case NOT_IN_RULEBOOK_VB: LOG("NOT_IN_RULEBOOK_VB"); break;
        case PLURAL_VB: LOG("PLURAL_VB"); break;
        case RELEASE_VB: LOG("RELEASE_VB"); break;
        case SOUND_VB: LOG("SOUND_VB"); break;
        case SPECIFIES_VB: LOG("SPECIFIES_VB"); break;
        case TEST_VB: LOG("TEST_VB"); break;
        case TRANSLATESU_VB: LOG("TRANSLATESU_VB"); break;
        case UNDERSTAND_VB: LOG("UNDERSTAND_VB"); break;
        case USEMEANS_VB: LOG("USEMEANS_VB"); break;
        case USE_VB: LOG("USE_VB"); break;
        default: LOG("(number %d)", verb_number); break;
    }
}
```

The function log_verb_number is called from 2/dl.

Noun Phrases 4/noun

Purpose

To construct noun-phrase subtrees for assertion sentences found in the parse tree.

Definitions

¶1. Noun phrase nodes are built at four levels of elaboration, which we take in turn:

(NP1) Raw: where the text is entirely untouched and unannotated.
(NP2) Articled: where any initial English article is converted to an annotation.
(NP3) List-divided: where, in addition, a list is broken up into individual items.
(NP4) Worldly: where, in addition, pronouns, relative clauses establishing relationships and properties (and other grammar meaningful only for references to physical objects and kinds) are parsed.

At levels (NP1) and (NP2), a NP produces a single `NOUNPHRASE_NT` node; at level (NP3), the result is a subtree contining `NOUNPHRASE_NT` and `AND_NT` nodes; but at level (NP4) this subtree may include any of `RELATIONSHIP_NT`, `CALLED_NT`, `WITH_NT`, `AND_NT`, `KIND_NT`, `X_OF_Y_NT`, `FROM_NT`, or `NOUNPHRASE_NT`.

Because a small proportion of noun phrase subtrees is thrown away, due to backtracking on mistaken guesses at parsing of sentences, it is important that creating an (NP) should have no side-effects beyond the construction of the tree itself (and, of course, the memory used up, but we won't worry about that: the proportion thrown away really is small).

---

§1. (NP1) Raw noun phrases. There are many situations (even if all of them are individually rare) when a NP must be constructed as a raw sequence of words, with no internal structure even being considered. Two special cases are also legal: where $w_1 = w_2 = -1$, signifying an empty text arising from some earlier problem; and $w_1 = w_2 = -2$, signifying an empty text arising from a relative clause whose noun phrase is implicit rather than spelled out in words, and whose meaning is that the player is the object referred to.

```
parse_node *new_nounphrase_raw(int w1, int w2) {
    parse_node *PN;
    PN = new_node(NOUNPHRASE_NT);
    pn_annotate_int(PN, nounphrase_article_ANNOT, NO_ART);
    PN->word_ref1 = w1; PN->word_ref2 = w2;
    return PN;
}
```

The function new_nounphrase_raw is called from 2/prob3, 4/sent, 4/verb, 5/unitr, 8/sob, 8/refpt, 9/pp, 10/tab, 13/tfg and 13/gtok.

§**2.** Sometimes we want to look at the article (if any) used in a raw NP, and absorb that into annotations, removing it from the wording. For instance, in

On the table is a thing called a part of the broken box.

we want to remove the initial article from the calling-name to produce "part of the broken box". (If we handled this NP as other than raw, we might spuriously make a subtree with `RELATIONSHIP_NT` in thanks to the apparent "part of" clause.) Note that unexpectedly upper-case articles are left well alone: this is why

On the table is a thing called A Town Called Alice.

creates an object called "A Town Called Alice", not an indefinitely-articled one called "Town Called Alice". Note that articles are not removed if that would leave the text empty.

```
parse_node *fix_articles_in_raw_NP(parse_node *RAW_NP) {
    int w1, w2;
    [[w1, w2 <-- RAW_NP]];
    pn_annotate_int(RAW_NP, nounphrase_article_ANNOT, NO_ART);
    pn_annotate_int(RAW_NP, plural_reference_ANNOT, FALSE);
    if (unexpectedly_upper_case(w1) == FALSE) {
        if [[w1, w2 == some ... --> w1, w2]] {
            pn_annotate_int(RAW_NP, nounphrase_article_ANNOT, INDEF_ART);
            pn_annotate_int(RAW_NP, plural_reference_ANNOT, TRUE);
        } else if [[w1, w2 == a/an ... --> w1, w2]]
            pn_annotate_int(RAW_NP, nounphrase_article_ANNOT, INDEF_ART);
        else if [[w1, w2 == the ... --> w1, w2]]
            pn_annotate_int(RAW_NP, nounphrase_article_ANNOT, DEF_ART);
    }
    RAW_NP->word_ref1 = w1; RAW_NP->word_ref2 = w2;
    return RAW_NP;
}
```

The function fix_articles_in_raw_NP is called from 4/ofs and 10/tab.

§**3. (NP2) Articled noun phrases.** We simply make a raw NP and then apply the article-fixing routine:

```
parse_node *new_nounphrase_articled(int w1, int w2) {
    parse_node *RAW_NP = new_nounphrase_raw(w1, w2);
    return fix_articles_in_raw_NP(RAW_NP);
}
```

The function new_nounphrase_articled is called from 8/refpt, 9/map and 10/bib.

§4. **(NP3) List-divided noun phrases.** This is a list, divided by "and" and perhaps using the serial comma, of articled NPs. We build into a lopsided binary tree: thus "the lion, a witch, and some wardrobes" becomes

```
AND_NT ","
    PROPERTYLIST_NT "lion" article:definite
    AND_NT ", and"
        PROPERTYLIST_NT "witch" article:indefinite
        PROPERTYLIST_NT "wardrobe" article:indefinite pluralreference:true
```

(The binary structure is chosen since it allows us to use a simple recursion to run through possibilities, and also to preserve each connective of the text in the `AND_NT` nodes.)

```
parse_node *new_nounphrase_listdivided(int w1, int w2) {
    if (is_list_divided(w1, w2, LOOK_FOR_AND)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        parse_node *AND_PN = new_node(AND_NT);
        AND_PN->word_ref1 = left_w2 + 1; AND_PN->word_ref2 = right_w1 - 1;
        AND_PN->down = new_nounphrase_listdivided(lw1, lw2);
        AND_PN->down->next = new_nounphrase_listdivided(rw1, rw2);
        return AND_PN;
    }
    return new_nounphrase_articled(w1, w2);
}
```

The function new_nounphrase_listdivided is called from 4/verb.

§5. **(NP4) Worldly noun phrases.** The function `new_nounphrase_worldly` constructs the sometimes complicated, and certainly varied, subtrees of the parse tree making up noun phrases which refer to world objects, kinds and their properties. Each possibility is diagrammed below, but we should remember that `new_nounphrase_worldly` is recursive, so that any of the subtree diagrams below can appear in the position of any `NOUNPHRASE_NT` node in any of the diagrams.

The following shell routine simply checks an invariant on entry and exit: experience shows that this is a routine which needs careful handling, and we want to be confident about what it does.

```
parse_node *new_nounphrase_worldly(int w1, int w2, int as_subject) {
    parse_node *NEW_NP;
    if ((w1 == -1) && (w2 == -1)) return new_nounphrase_raw(-1, -1);          used for error recovery
    if ((w1 == -2) && (w2 == -2)) return new_nounphrase_raw(-2, -2);          used for implicit NPs
    if ((w1<0) || (w1 > w2)) {
        LOG("NPW improperly applied to %d, %d\n$W\n", w1, w2, w1, w2);
        internal_error("NPW improperly applied");
    }
    NEW_NP = new_nounphrase_worldly_unchecked(w1, w2, as_subject);
    if (NEW_NP->word_ref1 > NEW_NP->word_ref2) {
        LOG("NPW on <$W>, %d, %d\n$T", w1, w2, NEW_NP->word_ref1, NEW_NP->word_ref2, NEW_NP);
        internal_error("malformed NPW made");
    }
    return NEW_NP;
}
```

The function new_nounphrase_worldly is called from 4/verb, 6/treec, 8/refpt, 8/creat, 10/tab and 10/eqns.

§**6.** And now the inner routine, which tries for eight constructions in turn. The sequence of checking these is very important, because it decides which clauses are represented higher in the tree when multiple structures are present within the NP. For instance, we want to turn

[A] in a container called the flask and cap with flange

into the subtree:

```
RELATIONSHIP_NT "in" = containment
    CALLED_NT "called"
        NOUNPHRASE_NT "container" article:indefinite
        NOUNPHRASE_NT "flask and cap with flange" article:definite
```

but we also want:

[B] in a container with carrying capacity 10 and diameter 12

```
RELATIONSHIP_NT "in" = containment
    WITH_NT "with"
        NOUNPHRASE_NT "container" article:indefinite
        AND_NT "and"
            PROPERTYLIST_NT "carrying capacity 10"
            PROPERTYLIST_NT "diameter 12"
```

These two cases together force our conventions: from sentence [A] we see that initial relative clauses (in) must beat callings ("called") which must beat property clauses ("with"), while from [B] we see that property clauses must beat lists ("and"). These all have to beat "of" and "from", which seem to be about linguistically equal, because these constructions must be easily reversible, as we shall see, and the best way to ensure that is to make sure they can only appear right down close to leaves in the tree. This dictates

```
RELATIONSHIP_NT > CALLED_NT > WITH_NT > AND_NT > X_OF_Y_NT = FROM_NT
```

in the sense that a subtree construction higher in this chain will take precedence over (and therefore be higher up in the tree than) one that is lower. That leaves just the seventh construction: "kind of ...". To avoid misreading this as an "of", and to protect "called", we need

```
CALLED_NT > KIND_NT > X_OF_Y_NT
```

but otherwise we are fairly free where to put it (though the resulting trees will take different shapes in some cases if we move it around, we could write code which handled any of the outcomes about equally well). In fact, we choose to make it lowest possible, so the final precedence order is:

```
RELATIONSHIP_NT > CALLED_NT > WITH_NT > AND_NT > KIND_NT > X_OF_Y_NT = FROM_NT
```

Once all possible constructions have been recursively exhausted, every leaf we end up at is treated simply as an (NP2) – except that we allow a special article annotation for the pronouns.

```
parse_node *new_nounphrase_worldly_unchecked(int w1, int w2, int as_subject) {
    parse_node *ARTICLED_NP;
    if ((assertion_NP_is_in_VP_for_to_be) &&
        (unexpectedly_upper_case(w1) == FALSE))
        ⟨Consider initial relative clauses which establish relationships 7⟩;
    ⟨Consider a calling clause which gives a possibly tricky name 9⟩;
    ⟨Consider a with or having clause which specifies properties 10⟩;
    ⟨Consider a comma- and and-separated list of noun phrases 11⟩;
    ⟨Consider "kind" or a "kind of" clause 12⟩;
    ⟨Consider a "from" or "of" clause 13⟩;
    ARTICLED_NP = new_nounphrase_articled(w1, w2);
    if [[w1, w2 == it/he/she/they/them]]
        pn_annotate_int(ARTICLED_NP, nounphrase_article_ANNOT, IT_ART);
    return ARTICLED_NP;
}
```

§**7.** This is one of three places in the *A*-grammar where `RELATIONSHIP_NT` nodes can enter the parse tree. One was when the primary verb of a sentence implied a relationship (other than the special cases for "to have" and "to be"); the other happens here, where the verb "to be" has been used but in concert with a NP beginning with a preposition. (The third happens later when noun phrases like "east from the Drawing Room" are examined.)

It might seem grammatically odd to be parsing prepositions here, when they surely ought to belong to the VP rather than either of the NPs. But English is, in fact, odd this way: it allows declarative statements, but no other sentences, to use "subject-verb inversion". An assertion such as

> In the garden is a tortoise.

is impossible to parse with a naive grammar for relative clauses, because the "in" is miles away from its governing verb. (This quirk is called an inversion since the sentence is equivalent to the easier to construe "A tortoise is in the garden".) Subject-verb inversion can't be allowed in all cases, though; we might pedantically argue that the Yoda-like utterance

> Holding the light sabre is the young Jedi.

is grammatically correct, but if so then we have to read

> Holding Area is a room.

as a statement that a room is holding something called "Area", which then causes Inform to throw problem messages about confusion between people and rooms (since it knows that only people can hold things). This is the reason why, rather tiresomely, we need an `as_subject` flag in this routine – to tell when we're applying subject-verb inversion, and to forbid it in the case of a participle like "holding", "carrying" or "wearing".

At any rate, we may well find ourselves parsing a NP like "in the garden", whichever side of the copular verb it may be. We turn it into

```
RELATIONSHIP_NT "in" = containment
    NOUNPHRASE_NT "garden" article:definite
```

To do this, we check each preposition in the *S*-grammar. We do, however, guard against "inside of..." and "inside from...": this is because we need to handle those as map connections and not spatial containment in the normal sense. (A long-standing IF convention is that "in" and "out" are directions on a par with "north" and "south", and behave like lateral connections between rooms, but this does cause ambiguities with spatial containment, which is conceptually quite different.)

⟨Consider initial relative clauses which establish relationships 7⟩ ≡

```
    ⟨Consider initial relative clauses with implied objects 8⟩;
    if ((![[w1, w2 == inside of/from ...]]) &&
        ((as_subject == FALSE) || (vocab_test_flags(w1, ING_MC) == FALSE))) {
        preposition_usage *pu;
        LOOP_OVER(pu, preposition_usage) {
            int i = pu_parse_text_against(w1, w2, pu, FALSE);
            if ((i>w1) && (i<=w2))
                return new_rel_node(STANDARD_RELN, bp_get_reversal(pu_get_meaning(pu)), w1, i-1, i,
w2);
        }
    }
```

This code is used in §6.

§**8.** A complication is that the relationship might be with something unstated. For instance, "here" generally means a containment by the room currently being discussed; "carried" implies that the player does the carrying. These cases form stunted `RELATIONSHIP_NT` subtrees having two forms:

```
RELATIONSHIP_NT "worn" = wears    RELATIONSHIP_NT "here" inference-to-draw:PARENTAGE_HERE_INF
    NOUNPHRASE_NT <-2, -2>
```

(A nounphrase with $w_1 = w_2 = -2$ is considered to refer to the player.)

⟨Consider initial relative clauses with implied objects 8⟩ ≡

```
if [[w1, w2 == here]] return new_rel_node(PARENTAGE_HERE_RELN, NULL, w1,w2, -1,-1);
if [[w1, w2 == worn]] return new_rel_node(STANDARD_RELN, a_wears_b_predicate, w1,w2, -2,-2);
if [[w1, w2 == carried]] return new_rel_node(STANDARD_RELN, a_carries_b_predicate, w1,w2, -2,-2);
```

This code is used in §7.

§**9. Calling subtrees.** Here something like "a container called the saucepan and lid" is split into:

```
CALLED_NT "with"
    NOUNPHRASE_NT "container" article:indefinite
    NOUNPHRASE_NT "saucepan and lid" article:definite
```

Note that the called-name is treated as an articled NP, not a worldly NP, and this protects it from being split into subclauses itself – which is why the "and" in "saucepan and lid" does not result in a list subtree, in the above example.

⟨Consider a calling clause which gives a possibly tricky name 9⟩ ≡

```
int i = is_word_intermediate(called_V, w1, w2);
if ((i >= 1) && (compare_word(i-1, OPENBRACKET_V)==FALSE)) {
    parse_node *CALLED_PN = new_node(CALLED_NT);
    CALLED_PN->word_ref1 = i; CALLED_PN->word_ref2 = i;
    CALLED_PN->down = new_nounphrase_worldly(w1, i-1, FALSE);
    CALLED_PN->down->next = new_nounphrase_articled(i+1, w2);
    return CALLED_PN;
}
```

This code is used in §6.

§**10. Property subtrees.** We would normally split a NP such as "a container having carrying capacity 10" into:

```
WITH_NT "having"
    NOUNPHRASE_NT "container" article:indefinite
    PROPERTYLIST_NT "carrying capacity 10"
```

However, we watch out for brackets – since those certainly prove us wrong – and also for the text "...it with action", which is present (for instance) in the name "locking it with action" – a valid NP where the word "action" is not intended to be a property list.

⟨Consider a with or having clause which specifies properties 10⟩ ≡

```
int i;
if ((([[w1, w2 == ... with ... : i]] || [[w1, w2 == ... having ... : i]]) &&
    (![[word i+1 == CLOSEBRACKET/OPENBRACKET]]) &&
    (!([[word i-1 == it]] && [[word i+1 == action]]))) {
    parse_node *WITH_PN = new_node(WITH_NT);
    WITH_PN->word_ref1 = i; WITH_PN->word_ref2 = i;
    WITH_PN->down = new_nounphrase_worldly(w1, i-1, FALSE);
    WITH_PN->down->next = new_property_list(i+1, w2);
    return WITH_PN;
}
```

This code is used in §6.

§11. List subtrees. These look just like list-divided noun phrases (see diagram above), except that each leaf can be a general worldly noun phrase subtree.

⟨Consider a comma- and and-separated list of noun phrases 11⟩ ≡

```
if (is_list_divided(w1, w2, LOOK_FOR_AND)) {
    parse_node *AND_PN = new_node(AND_NT);
    int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
    AND_PN->down = new_nounphrase_worldly(lw1, lw2, FALSE);
    AND_PN->down->next = new_nounphrase_worldly(rw1, rw2, FALSE);
    return AND_PN;
}
```

This code is used in §6.

§12. Kind of clauses. Two very simple subtrees:

```
KIND "kind"        KIND "kind of vehicle"
                       NOUNPHRASE_NT "vehicle"
```

Note that indefinite articles are permitted before the word "kind(s)", but definite articles are not.

⟨Consider "kind" or a "kind of" clause 12⟩ ≡

```
int k1 = w1, k2 = w2;
[[k1, k2 == a/some ... --> k1, k2]];
if [[k1, k2 == kind/kinds]] {
    parse_node *KIND_NODE = new_node(KIND_NT);
    KIND_NODE->word_ref1 = k1; KIND_NODE->word_ref2 = k2;
    return KIND_NODE;
}
if [[k1, k2 == kind/kinds of ...]] {
    parse_node *KIND_NODE = new_node(KIND_NT);
    KIND_NODE->word_ref1 = k1; KIND_NODE->word_ref2 = k2;
    KIND_NODE->down = new_nounphrase_worldly(k1+2, k2, FALSE);
    return KIND_NODE;
}
```

This code is used in §6.

§13. From and of clauses. These have equal precedence, so we scan for them together, and go for the leftmost one found. They have the form:

```
FROM_NT "east from the Deer Park"
    NOUNPHRASE_NT "Deer Park" article:definite
    NOUNPHRASE_NT "east"
```

and similarly for `X_OF_Y_NT`. The perhaps contrary convention that the first child is the point of reference, while the second child is the method of reference, dates from Inform 7's earliest days: it does no harm.

```
On the shelf is Of Mice And Men and The Girl From Ipanema.
```

(Note that a routine in the Of and From section warns us off breaking "of" at certain bad positions. That only ever happens on a second attempt, where we are reconstructing the NPs for a sentence having got them wrong the first time by overzealously breaking at "of".)

⟨Consider a "from" or "of" clause 13⟩ ≡

```
int pos;
for (pos = w1+1; pos < w2; pos++) {
    int clause = NOUNPHRASE_NT;
    if (unexpectedly_upper_case(pos)) continue;
    if [[pos, w2 == in the presence of ...]] { pos += 4; continue; }
    if ([[pos, w2 == of ...]] &&
        (bad_of_break_position(pos, w1, w2) == FALSE)) clause = X_OF_Y_NT;
    if [[pos, w2 == from ...]] clause = FROM_NT;
    if (clause != NOUNPHRASE_NT) {
        parse_node *CLAUSE_NODE = new_node(clause);
        CLAUSE_NODE->word_ref1 = w1; CLAUSE_NODE->word_ref2 = w2;
        CLAUSE_NODE->down = new_nounphrase_worldly(pos+1, w2, FALSE);
        CLAUSE_NODE->down->next = new_nounphrase_worldly(w1, pos-1, FALSE);
        return CLAUSE_NODE;
    }
}
```

This code is used in §6.

§14. Relationship nodes. A modest utility routine to construct and annotation RELATIONSHIP nodes.

```
define STANDARD_RELN 0                                   the default annotation value
define PARENTAGE_HERE_RELN 1
define DIRECTION_RELN 2

parse_node *new_rel_node(int reln_type, binary_predicate *bp, int rw1, int rw2, int tw1, int tw2) {
    parse_node *REL = new_node(RELATIONSHIP_NT);
    REL->word_ref1 = rw1; REL->word_ref2 = rw2;
    if (bp) pn_set_relationship(REL, bp);
    else pn_annotate_int(REL, relationship_node_type_ANNOT, reln_type);
    if (tw1 != -1) REL->down = new_nounphrase_worldly(tw1, tw2, FALSE);
    return REL;
}
```

§15. Property lists as noun phrases. These are parsed in only the most rudimentary fashion at this early stage, because they cannot properly be understood until later (they will often refer to objects not yet created). We form lists into lopsided binary trees, so that "carrying capacity 10, dexterity 21, and skill 40" would return the following subtree:

```
AND_NT ","
    PROPERTYLIST_NT "carrying capacity 10"
    AND_NT ", and"
        PROPERTYLIST_NT "dexterity 21"
        PROPERTYLIST_NT "skill 40"
```

(See list-divided NPs above, which have a similar pattern.)

```
parse_node *new_property_list(int w1, int w2) {
    parse_node *PLIST_PN;
    if (is_list_divided(w1, w2, LOOK_FOR_AND)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        parse_node *AND_PN = new_node(AND_NT);
        AND_PN->word_ref1 = left_w2+1; AND_PN->word_ref2 = right_w1-1;
        AND_PN->down = new_property_list(lw1, lw2);
        AND_PN->down->next = new_property_list(rw1, rw2);
        return AND_PN;
    }
    PLIST_PN = new_node(PROPERTYLIST_NT);
    PLIST_PN->word_ref1 = w1; PLIST_PN->word_ref2 = w2;
    return PLIST_PN;
}
```

§16. The player is not yourself. Finally two small utilities for the noun phrase referring to the player's avatar-object. "The player" is a variable in Inform 7, not a constant, and assertions of spatial relationships cannot be made concerning variables, so if we stuck rigidly to the rules then

> The player carries an umbrella.

...would have to produce a problem message. We don't want this: the user thinks of "the player" as a constant, even though it isn't, and sentences like this one are therefore the natural thing to write. So we transform "the player" to the constant "yourself" whenever it turns up in assertions:

```
void turn_player_to_yourself(parse_node *pn) {
    if [[pn == player]] pn_make_player_noun_phrase(pn);
}
void pn_make_player_noun_phrase(parse_node *pn) {
    pn_set_node_type(pn, NOUNPHRASE_NT);
    pn_set_refers_to(pn, wo_yourself);
}
```

The function turn_player_to_yourself is called from 8/relv.

The function pn_make_player_noun_phrase is called from 8/refpt and 8/mass.

Of and From 4/ofs

Purpose

To verify that "of" and "from" subtrees in assertion sentence noun phrases are validly used, and reconstruct their sentences without them if they are not.

Template interpreter commands

```
5   {-callv:tidy_up_ofs_and_froms}
```

Definitions

¶1. At this point in the narrative, the source text has been parsed into a large parse tree in which every assertion is represented by a `SENTENCE_NT` node. This is a child of the root of the tree, and its own children form a subtree which parses it into a verb phrase and associated noun phrases. Pretty well whenever it possibly could, the noun-phrase-maker created `X_OF_Y_NT` and `FROM_NT` subtrees in these noun phrases.

But `X_OF_Y_NT` and `FROM_NT` nodes are valid only when describing a value property associated with something ("description of the cards"), or a map direction leading away from somewhere ("east from Casino Royale"), which means that the noun-phrase-maker will have been wildly overzealous in creating these. It had to be: it had no way of knowing, so early in Inform's run, which handful of the many such nodes it made were actually valid. It didn't know what the property names were, nor what the direction names were. (For chicken-and-egg reasons, it couldn't know this.) So it almost certainly made some spurious noun phrase subtrees like this one:

```
X_OF_Y_NT "worn patch of carpet"
    NOUNPHRASE_NT "carpet"
    PROPERTYLIST_NT "worn patch"
```

Now comes the reckoning. The existing parse tree may contain a few bogus nodes, but it's good enough to determine all the property and direction names. This then enables us to go back and check all the `X_OF_Y_NT` and `FROM_NT` subtrees, expunging the ones which are spurious.

---

§**1.** Mostly we can use careful parsing to distinguish usages of "of": but there is one dire case when we can't, or not at any rate without knowledge which we may not have yet. This is the case occurring when a property name itself contains the word "of", as in:

```
A woman has a text called point of view.
```

The "of" there is no trouble: but the first "of" here is a nightmare –

```
The point of view of Kathy is "All grammar is bunk."
```

We call the first "of" a "bad of break position", because we don't want a worldly noun phrase in this sentence to break into the subtree:

```
X_OF_Y_NT "point of view of Kathy"
    NOUNPHRASE_NT "Kathy"
    X_OF_Y_NT "point of view"
        NOUNPHRASE_NT "view"
        PROPERTYLIST_NT "point"
```

The only way we can tell good of-breaks from bad is by knowing all the property names: the following routine performs the necessary test, determining whether $w_1 < p < w_2$ is a good position at which to break.

(Bad breaks would also occur if "of" was inside a direction name, or inside one of the permitted pseudo-property names, but in fact this never happens with the standard configuration and at present the user cannot change that, so we needn't worry.)

```
int bad_of_break_position(int pos, int w1, int w2) {
    property_name *prn;
    LOOP_OVER(prn, property_name) {
        int i, pn1 = prn->word_ref1, pn2 = prn->word_ref2;
        for (i=pn1; i<=pn2; i++)
            if [[word i == of]]
                ⟨Can the property's name be found inside the word range with this "of" at the break position? 2⟩;
    }
    return FALSE;
}
```

The function bad_of_break_position is called from 4/noun.

§**2.** If the answer to this question is "yes", then we have a bad break:

⟨Can the property's name be found inside the word range with this "of" at the break position? 2⟩ ≡

```
    int pm1 = pos - (i-pn1), pm2 = pm1 + (pn2-pn1);
    if ((pm1 >= w1) && (pm2 <= w2) &&
        (compare_word_range(pm1, pm2, pn1, pn2))) {
            LOG("Let's not break at pos %d = $W in <$W>, given $Y.\n",
                pos, pos, pos, w1, w2, prn);
            return TRUE;
    }
```

This code is used in §1.

§**3. Tree surgery.** The following routine performs the necessary correction to expunge a bogus `X_OF_Y_NT` or `FROM_NT` construction.

```
    X_OF_Y_NT "worn patch of carpet"  --->  NOUNPHRASE_NT "worn patch of carpet"
        NOUNPHRASE_NT "carpet"
        PROPERTYLIST_NT "worn patch"
```

Since the constructions have equal precedence in the nounphrase-maker, they might be nested, so we could find something like:

```
    X_OF_Y_NT "worn patch of carpet from hell"
        FROM_NT "carpet from hell"
            NOUNPHRASE_NT "hell"
            NOUNPHRASE_NT "carpet"
        PROPERTYLIST_NT "worn patch"
```

However, if any higher-up construction was invalid, then so are any constructions lower down: so it is safe to deal with this any such tree by drastic means.

```
void expunge_X_OF_Y_subtree(parse_node *pn) {
    pn_set_node_type(pn, NOUNPHRASE_NT);
    pn->down = NULL;
    fix_articles_in_raw_NP(pn);
}
void expunge_FROM_subtree(parse_node *pn) {
    pn_set_node_type(pn, NOUNPHRASE_NT);
    pn->down = NULL;
    fix_articles_in_raw_NP(pn);
}
```

§**4.** The other operation we need is to alter these subtrees to relationships. For instance:

```
FROM_NT "east from Essex"  --->  RELATIONSHIP_NT "east from Essex" type:direction
    NOUNPHRASE_NT "Essex"            NOUNPHRASE_NT "Essex"
    NOUNPHRASE_NT "east"             NOUNPHRASE_NT "east"
```

(This is still a third form of `RELATIONSHIP_NT` subtree: one with two children.)

```
void convert_clause_to_RELATIONSHIP(parse_node *pn) {
    pn_set_node_type(pn, RELATIONSHIP_NT);
    pn_annotate_int(pn, relationship_node_type_ANNOT, DIRECTION_RELN);
}
```

§**5. Tidying up ofs and from.** In deciding whether "worn patch of carpet" is to be treated as a single noun, as a property ("worn patch") belonging to a noun ("carpet"), or as a direction ("worn patch") relative to a noun ("carpet") – consider the cases "description of carpet" and "east of the lawn" – the key thing is to understand the meaning of the subject ("worn patch"). Unfortunately, the NI model world will not be created, much less named, for quite a long time to come: so we cannot simply parse the subject as an expression. At this early stage the best we can do is to scan ahead through the parse tree to try to find direction and property declarations. We do this with a sequence of three traversals:

(1.) Look for property and direction declarations. Create the property names called for; do not actually create the directions (that will happen much later), but remember their names.
(2.) Reconstruct the parse trees of any sentences which refer to property names which themselves contain an "of" (as may happen if, for instance, there is a property called "point of view").
(3.) Look for location phrases like "east from the lawn", which we can distinguish from other uses of "from" now that we know the direction names; at the same time look for "X of Y" phrases where the "of" should not be treated as a possessive, and expunge them.

```
void tidy_up_ofs_and_froms(void) {
    verify_integrity_of_subtree(tree_root, FALSE);
    traverse_for_property_names(tree_root->down);
    traverse_for_nonbreaking_ofs(tree_root->down);
    traverse_for_FROM_NT_subtrees(tree_root->down);
}
```

The function tidy_up_ofs_and_froms is invoked by a command in a `.i6t` template file.

§**6.** The following array is used only by Traversals 1 to 3, and is how we remember direction names.

define `MAX_DIRECTIONS 100` *the Standard Rules define only 12, so this is plenty*

```
parse_node *directions_noticed[MAX_DIRECTIONS];
binary_predicate *direction_relations_noticed[MAX_DIRECTIONS];
int no_directions_noticed = 0;
```

§**7. Traversal 1.** We now come to the routine which traverses the entire tree looking for property declarations.

In this as in the subsequent traversals, we use a loop rather than recursion to span the width of the tree: otherwise NI's stack usage goes through the roof, since it might need to recurse thousands of function calls deep.

```
void traverse_for_property_names(parse_node *pn) {
    for (; pn; pn = pn->next) {
        switch(pn_get_node_type(pn)) {
            case SENTENCE_NT: current_sentence = pn; break;
            case VERB_NT:
                ⟨See if this assertion creates property names with "... has ... called ..." 10⟩;
                break;
        }
        traverse_for_property_names(pn->down);
    }
}
```

§**8.** Directions are detected in sentences having the form "D is a direction."

```
void check_sentence_for_direction_creation(parse_node *pn) {
    if (pn_get_node_type(pn) != SENTENCE_NT) return;
    if ((pn->down == NULL) || (pn->down->next == NULL) || (pn->down->next->next == NULL)) return;
    if (pn_get_node_type(pn->down) != VERB_NT) return;
    if (pn_get_node_type(pn->down->next) != NOUNPHRASE_NT) return;
    if (pn_get_node_type(pn->down->next->next) != NOUNPHRASE_NT) return;
    current_sentence = pn;
    pn = pn->down->next;
    if (![[pn->next == direction]]) return;
    if (no_directions_noticed >= MAX_DIRECTIONS) {
        limit_problem(_P_(C4TooManyDirections),
            "different directions", MAX_DIRECTIONS);
        return;
    }
    direction_relations_noticed[no_directions_noticed] =
        bp_create_sketchy_mapping_direction(pn->word_ref1, pn->word_ref2);
    directions_noticed[no_directions_noticed++] = pn;
}
```

The function check_sentence_for_direction_creation is called from 4/verb.

§**9.** And to extract that sketchy BP later, for completion when possible:

```
binary_predicate *relation_noticed(int i) {
    return direction_relations_noticed[i];
}
```

The function relation_noticed is called from 9/mapbp.

§**10.** Typical patterns here are sentences in the form "X has a K called P". The trouble is that at this stage we can't verify that X is a world object or kind (they don't exist yet), nor that K is a kind of value (because nor do they). We must simply rely on the fact that assertions cannot take this form without being property declarations – with one exception:

The taking action has a number called the hazard level.

...creates not a property name but an action variable; and the same applies to activities and rulebooks. Fortunately, the names (X) of those must always end exactly in those words, so we are saved. (Unless, of course, the kind or world object for which this property name is created also ends in one of those words. Hmmm...) We therefore look for this subtree structure:

```
SENTENCE_NT "A container has a number called volume"
    VERB_NT "has"
    NOUNPHRASE_NT "container" article:indefinite
    CALLED_NT "called"
        NOUNPHRASE_NT "number" article:indefinite
        NOUNPHRASE_NT "volume"
```

...and then extract the bottom-most, rightmost noun-phrase as the name of the new property.

⟨See if this assertion creates property names with "... has ... called ..." 10⟩ ≡

```
if ((pn_int_annotation(pn, verb_id_ANNOT) == HAS_VB)
    && (pn->next)
    && (![[pn->next == ... action/activity/rulebook]])
    && (pn->next->next)
    && (pn_get_node_type(pn->next->next) == CALLED_NT)
    && (pn->next->next->down)
    && (pn->next->next->down->next))
    force_creation_of_property_name(pn->next->next->down->next);
```

This code is used in §7.

§**11.** All we really need to do is forces its creation if it doesn't already exist...

```
void force_creation_of_property_name(parse_node *pn) {
    int w1, w2;
    [[w1, w2 <-- pn]];
    internal_error_if_node_type_wrong(pn, NOUNPHRASE_NT);
    ⟨Issue problem messages for any really unfortunately chosen property name 12⟩;
    value_property_name_ref(w1, w2);                    this lookup creates a new PRN if necessary
}
```

§**12.** ...but let's also take the opportunity to pre-empt bad property names before they can do any damage:

⟨Issue problem messages for any really unfortunately chosen property name 12⟩ ≡

```
int i;
if [[w1, w2 == a/an/the/some]] {
    sentence_problem(_P_(C4PropertyCalledArticle),
        "a property name cannot consist only of an article",
        "which this one seems to. It would lead to awful ambiguities. "
        "More likely, the end of the sentence has been lost somehow?");
    return;
}
if [[w1, w2 == presence]] {
    sentence_problem(_P_(C4PropertyCalledPresence),
        "a property name cannot consist only of the word 'presence'",
        "because this would lead to ambiguities with the rule clause "
        "'...in the presence of...' (For instance, when writing something "
        "like 'Instead of eating in the presence of the Queen: ...') "
        "The best way to fix this is probably to add another word or "
        "two to the property name: 'stage presence', say, would be fine.");
    return;
}
for (i=w1; i<=w2; i++)
    if ([[word i == COMMA]] || (vocab_test_flags(w1, TEXT_MC))) {
        sentence_problem(_P_(C4PropertyNameForbidden),
            "a property name cannot contain quoted text or a comma",
            "which this one seems to. I think I must be misunderstanding: "
            "possibly you've added a subordinate clause which I can't "
            "make sense of?");
        return;
    }
```

This code is used in §11.

§**13. Traversal 2.** At this point the parse tree may contain `X_OF_Y_NT` subtrees which were formed at bad of-breaks, and that might have prevented it from forming correct ones. Any sentence which contains both worldly noun-phrases and at least one bad of-break position must therefore be considered suspect. The simplest thing to do is to throw away the existing subtree for such a sentence and build a fresh one, very probably smaller and simpler since it will not now break at what we now know to be a bad position.

(In a few cases, where for some other reason the of-break was not taken anyway, nothing will change – for instance, "On the table is a book called My point of view" does not break at the "of" whether it's a good break or a bad one, because the "called" construction takes priority. We could with some more work avoid this, but it causes no nuisance and wastes only a negligible amount of memory.)

```
void traverse_for_nonbreaking_ofs(parse_node *pn) {
    for (; pn; pn = pn->next)
        if ((pn_get_node_type(pn) == SENTENCE_NT) &&
            (pn->down) && (pn_get_node_type(pn->down) == VERB_NT)) {
            int i, vn = pn_int_annotation(pn->down, verb_id_ANNOT);
            if (((vn == ASSERT_VB) || (vn == HAS_VB) || (vn == DEFINED_BY_VB))
                && [[pn == ... of ... : i]]
                && (bad_of_break_position(i, pn->word_ref1, pn->word_ref2))) {
                current_sentence = pn;                  (just in case any problem messages are issued)
                pn->down = NULL;                        thus cutting off and forgetting its former subtree
                look_for_verb_phrase(pn);                                  ...in order to make a new one
```

```
            }
        }
}
```

§**14. Traversal 3.** Here we look for noun phrases in the form "X from Y", which have been added to the parse tree as `FROM_NT` subtrees: for instance, "northwest from the Gazebo" comes out as:

```
FROM_NT
    PROPERTYLIST_NT "Gazebo" article:definite
    NOUNPHRASE_NT "northwest"
```

We examine all such constructs to see if the subject is a direction name (because of Traversal 1, we know all those names). If so, we convert the subtree as a `RELATIONSHIP_NT` construct, since the phrase is giving spatial information. If not, we simply expunge the construct into a `NOUNPHRASE_NT`. Either way, after Traversal 3, there are no more `FROM_NT` nodes.

This is also where we finally decide which "X of Y" subtrees are to be expunged. And the answer is: all of them, except where X is one of two special pseudo-properties (possibly with articles), a property name or a direction. And even in the latter case, the construct is converted to a `RELATIONSHIP_NT` subtree. Thus, after Traversal 3, the reference child of every `X_OF_Y_NT` subtree is a property name, `specification` or `plural`.

```
void traverse_for_FROM_NT_subtrees(parse_node *pn) {
    for (; pn; pn = pn->next) {
        traverse_for_FROM_NT_subtrees(pn->down);
        switch(pn_get_node_type(pn)) {
            case SENTENCE_NT: current_sentence = pn; break;
            case FROM_NT:
                if ((pn->down) && (pn->down->next)) {
                    int direction_found = FALSE;
                    ⟨See if the word range matches a direction name found in Traversal 1 15⟩;
                    if (direction_found == FALSE) expunge_FROM_subtree(pn);
                } else internal_error("malformed FROM_NT subtree");
                break;
            case X_OF_Y_NT:
                if ((pn->down) && (pn->down->next)) {
                    int allow_this = FALSE, direction_found = FALSE, w1, w2;
                    [[w1, w2 <-- pn->down->next]];
                    if (parse_property_name(w1, w2)) allow_this = TRUE;
                    if [[w1, w2 == specification]] allow_this = TRUE;
                    else if [[w1, w2 == plural]] allow_this = TRUE;
                    else ⟨See if the word range matches a direction name found in Traversal 1 15⟩;
                    if ((allow_this == FALSE) && (direction_found == FALSE))
                        expunge_X_OF_Y_subtree(pn);
                } else internal_error("malformed X_OF_Y_NT subtree");
                break;
        }
    }
}
```

§**15.** We do not allow direction names with unexpected capital letters because we want to allow room names to contain direction names on occasion:

The fire hydrant is in West from 47th Street.

⟨See if the word range matches a direction name found in Traversal 1 15⟩ ≡

```
int i, w1, w2;
[[w1, w2 <-- pn->down->next]];
if (unexpectedly_upper_case(w1) == FALSE)
    for (i=0; i<no_directions_noticed; i++)
        if (compare_word_range(w1, w2,
                directions_noticed[i]->word_ref1, directions_noticed[i]->word_ref2))
            direction_found = TRUE;
if (direction_found) convert_clause_to_RELATIONSHIP(pn);
```

This code is used in §14.

Rule Subtrees 4/rtree

Purpose

To tidy up COMMAND_NT nodes into a list of children under the relevant ROUTINE_NT node, and so turn each rule definition into a single subtree.

4/rtree.§3-19 Unifying Block Syntaxes; §20 Basic Structural Syntax; §21 Detect Control Structures

---

Template interpreter commands

```
1   {-callv:register_recently_lexed_phrases}
20  {-callv:create_standard_csps}
```

Definitions

¶1. Initially, the phrases (COMMAND_NT) making up a rule (ROUTINE_NT) are simply listed after it in the parse tree, but we want them to become its children: this is the only thing the *A*-grammar does with rules, which otherwise wait until Chapter 12 to be dealt with.

The single routine in this section accomplishes the regrouping: after it runs, every COMMAND_NT is a child of the ROUTINE_NT header to which it belongs.

¶2. Certain phrases are "structural": otherwise, if, repeat, while and so on. These have different expectations in terms of the layout of surrounding phrases in rule or phrase definitions, and the following structure defines the relevant behaviour. (The contents are static.)

```
typedef struct control_structure_phrase {
    struct control_structure_phrase *subordinate_to;
    int indent_subblocks;
    int body_empty_except_for_subordinates;
    int suppress_RULES_ALL_text;
    int used_at_stage;
    int requires_new_syntax;
    int allow_run_on;
    MEMORY_MANAGEMENT
} control_structure_phrase;

control_structure_phrase
    *switch_CSP = NULL,
    *if_CSP = NULL,
    *repeat_CSP = NULL,
    *while_CSP = NULL,
    *otherwise_CSP = NULL,
    *otherwise_if_CSP = NULL,
    *default_case_CSP = NULL,
    *case_CSP = NULL;
```

The structure control_structure_phrase is private to this section.

---

§**1.** This routine is used whenever new material is added. Whenever it finds a childless `ROUTINE_NT` followed by a sequence of `COMMAND_NT` nodes, it joins these in sequence as children of the `ROUTINE_NT`. Since it always acts so as to leave a non-zero number of children, and since it acts only on childless nodes, it cannot ever act on the same node twice.

```
void register_recently_lexed_phrases(void) {
    parse_node *p;
    if (problem_count > 0) return;                                  for then the tree is perhaps broken anyway
    for (p = tree_root->down; p; p=p->next)
        if ((pn_get_node_type(p) == ROUTINE_NT) && (p->down == NULL)) {
            parse_node *end_def = p;
            while ((end_def->next) && (pn_get_node_type(end_def->next) == COMMAND_NT))
                end_def = end_def->next;
            if (p == end_def) continue;                          ROUTINE_NT not followed by any COMMAND_NTs
            splice so that p->next to end_def become the children of p:
            p->down = p->next;
            p->next = end_def->next;
            end_def->next = NULL;
            unify_block_syntaxes(p);
        }
    ⟨Assert that there are indeed no loose COMMAND nodes 2⟩;
}
```

The function register_recently_lexed_phrases is invoked by a command in a `.i6t` template file and called from 5/litp, 7/dti, 9/scene and 9/madj.

§**2.** And just in case:

```
⟨Assert that there are indeed no loose COMMAND nodes 2⟩ ≡
    for (p = tree_root->down; p; p=p->next)
        if (pn_get_node_type(p) == COMMAND_NT) {
            current_sentence = p;
            internal_error("loose COMMAND node outside of rule definition");
        }
```

This code is used in §1.

§**3. Unifying Block Syntaxes.** There are now two possible syntaxes to express the structural makeup of a routine: we check which is in use for this one, and unify them into a common format in the parse tree.

```
void unify_block_syntaxes(parse_node *rout) {
    parse_node *uses_colon_syntax = NULL;
    parse_node *uses_begin_end_syntax = NULL;
    parse_node *requires_colon_syntax = NULL;
    ⟨See which block syntax is used by conditionals and loops 4⟩;
    if ((uses_colon_syntax) && (uses_begin_end_syntax)) {
        current_sentence = rout;
        quote_source(1, current_sentence);
        quote_source(2, uses_colon_syntax);
        quote_source(3, uses_begin_end_syntax);
        handmade_problem(_P_(C4BothBlockSyntaxes));
        issue_problem_segment(
            "The rule or phrase definition %1 seems to use both ways of grouping "
            "phrases together into 'if', 'repeat' and 'while' blocks at once. "
```

```
            "Inform allows two alternative forms, but they cannot be mixed in "
            "the same definition. %POne way is to end the 'if', 'repeat' or "
            "'while' phrases with a 'begin', and then to match that with an "
            "'end if' or similar. ('Otherwise' or 'otherwise if' clauses are "
            "phrases like any other, and end with semicolons in this case.) "
            "You use this begin/end form here, for instance - %3. %P"
            "The other way is to end with a colon ':' and then indent the "
            "subsequent phrases underneath, using tabs. (Note that any "
            "'otherwise' or 'otherwise if' clauses also have to end with "
            "colons in this case.) You use this indented form here - %2.");
        issue_problem_end();
        return;
    }
    if ((requires_colon_syntax) && (uses_begin_end_syntax)) {
        current_sentence = rout;
        quote_source(1, current_sentence);
        quote_source(2, requires_colon_syntax);
        handmade_problem(_P_(C4NotInOldSyntax));
        issue_problem_segment(
            "The construction %2, in the rule or phrase definition %1, "
            "is only allowed if the rule is written in the 'new' format, "
            "that is, with the phrases written one to a line with "
            "indentation showing how they are grouped together, and "
            "with colons indicating the start of such a group.");
        issue_problem_end();
        return;
    }
    if (uses_colon_syntax)
        ⟨Insert begin/end markers and check that indentation is correct 5⟩;
}
```

§**4.** Traditional I7 syntax for blocks is to place them within begin/end markers: the "begin" occurring at the end of the conditional or loop header, and the "end if", "end while", etc., as a phrase of its own at the end of the block. Newer I7 syntax (March 2008) is to use Python-style colons and indentation. Both are allowed, but not in the same routine.

⟨See which block syntax is used by conditionals and loops 4⟩ ≡

```
    parse_node *p;
    for (p = rout->down; p; p = p->next) {
        control_structure_phrase *csp =
            detect_control_structure(p->word_ref1, p->word_ref2);
        if (csp) {
            int syntax_used = pn_int_annotation(p, colon_block_command_ANNOT);
            if (syntax_used == FALSE) {                          i.e., doesn't end with a colon
                don't count "if x is 1, let y be 2" – with no block – as deciding it
                if ((csp->subordinate_to == NULL) && (![[p == ... begin]]))
                    syntax_used = NOT_APPLICABLE;
            }
            if (syntax_used != NOT_APPLICABLE) {
                if (syntax_used) {
                    if (uses_colon_syntax == NULL) uses_colon_syntax = p;
                } else {
```

```
                    if (uses_begin_end_syntax == NULL) uses_begin_end_syntax = p;
                }
            }
            if (csp->suppress_RULES_ALL_text)
                pn_annotate_int(p, suppress_debug_text_ANNOT, TRUE);
            if ((csp->requires_new_syntax) && (requires_colon_syntax == NULL))
                requires_colon_syntax = p;
        }
        if ([[p == end ###]] && (uses_begin_end_syntax == NULL))
            uses_begin_end_syntax = p;
    }
```

This code is used in §3.

§**5.** In the case where the routine uses indentation, we amend the colons at the end of block phrases to the word "begin" – thus giving them the old-style syntax ready for them to be parsed – and insert suitable "end" phrases where blocks have implicitly been closed by indentation reductions.

⟨Insert begin/end markers and check that indentation is correct 5⟩ ≡

```
    parse_node *p, *prev, *run_on_at = NULL, *first_misaligned_phrase = NULL;
    int k, indent, expected_indent = 1, indent_misalign = FALSE,
        just_opened_block = FALSE;
    the blocks open stack holds blocks currently open
    parse_node *blstack_opening_phrase[10];
    control_structure_phrase *blstack_construct[10];
    int blstack_stage[10];
    int blo_sp = 0;
    for (prev = NULL, p = rout->down, k=1; p; prev = p, p = p->next, k++) {
        control_structure_phrase *csp =
            detect_control_structure(p->word_ref1, p->word_ref2);
        ⟨Determine actual indentation of this phrase 6⟩;
        ⟨Compare actual indentation to what we expect from structure so far 7⟩;
        ⟨Insert begin marker and increase expected indentation if a block begins here 9⟩;
    }
    indent = 1;
    ⟨Try closing blocks to bring expected indentation down to match 10⟩;
    if (run_on_at) ⟨Issue problem message for run-ons within phrase definition 15⟩;
    if (indent_misalign) ⟨Issue problem message for misaligned indentation 14⟩;
```

This code is used in §3.

§**6.** Here we set `indent` to the number of tab-stops in from the margin, or to `expected_indent` if the text does not appear to be at the start of its own line in the source (because it runs on from a previous phrase, in which case we set the `run_on_at` flag: except for following on from cases in switches with a non-control-structure, which is allowed, because otherwise the lines often look silly and short).

⟨Determine actual indentation of this phrase 6⟩ ≡

```
indent = expected_indent;
if (p->word_ref1 >= 0) {
    switch (lw_array[p->word_ref1].lw_break) {
        case '\n': indent = 0; break;
        case '\t': indent = 1; break;
        case '1': indent = 1; break;
        case '2': indent = 2; break;
        case '3': indent = 3; break;
        case '4': indent = 4; break;
        case '5': indent = 5; break;
        case '6': indent = 6; break;
        case '7': indent = 7; break;
        case '8': indent = 8; break;
        case '9': indent = 9; break;
        default:
            if ((prev) && (csp == NULL)) {
                control_structure_phrase *pcsp =
                    detect_control_structure(prev->word_ref1, prev->word_ref2);
                if ((pcsp) && (pcsp->allow_run_on)) break;
            }
            if (run_on_at == NULL) run_on_at = p; break;
    }
}
```

This code is used in §5.

§**7.** We now know the `indent` level of the line as read, and also the `expected_indent` given the definition so far. If they agree, fine. If they don't agree, it isn't necessarily bad news – if each line's indentation were a function of the last, there would be no information in it, after all. Roughly speaking, when `indent` is greater than we expect, that must be wrong – it means indentation has jumped inward as if to open a new block, but blocks are opened explicitly and not by simply raising the indent. But when `indent` is less than we expect, this may simply mean that the current block(s) has or have been closed, because blocks are indeed closed implicitly just by moving the indentation back in.

⟨Compare actual indentation to what we expect from structure so far 7⟩ ≡

```
if (indent == 0) {
    ⟨Record a misalignment of indentation 13⟩;
    ⟨Record a phrase within current block 11⟩;
} else {
    if ((csp) && (csp->subordinate_to)) {
        ⟨Compare actual indentation to what we expect for an intermediate phrase 8⟩;
        just_opened_block = TRUE;
    } else {
        if (expected_indent < indent) ⟨Record a misalignment of indentation 13⟩;
        if (expected_indent > indent)
            ⟨Try closing blocks to bring expected indentation down to match 10⟩;
        expected_indent = indent;
        ⟨Record a phrase within current block 11⟩;
```

```
        }
    }
    if (expected_indent < 1) expected_indent = 1;
```

This code is used in §5.

§**8.** This is a small variation used for an intermediate phrase like "otherwise". These are required to be at the same indentation as the line which opened the block, rather than being one tab step in from there: in other words they are not deemed part of the block itself. They can also occur in "stages", which is a way to enforce one intermediate phrase only being allowed after another one – for instance, "otherwise if..." is not allowed after an "otherwise" within an "if".

⟨Compare actual indentation to what we expect for an intermediate phrase 8⟩ ≡

```
    expected_indent--;
    if (expected_indent < indent) {
        ⟨Issue problem for an intermediate phrase not matching 18⟩;
    } else {
        ⟨Try closing blocks to bring expected indentation down to match 10⟩;
        if ((blo_sp == 0) ||
            (csp->subordinate_to != blstack_construct[blo_sp-1])) {
            ⟨Issue problem for an intermediate phrase not matching 18⟩;
        } else {
            if (blstack_stage[blo_sp-1] > csp->used_at_stage)
                ⟨Issue problem for an intermediate phrase out of sequence 19⟩;
            blstack_stage[blo_sp-1] = csp->used_at_stage;
        }
    }
    expected_indent++;
```

This code is used in §7.

§**9.** In colon syntax, blocks are explicitly opened; they are only implicitly closed. Here is the opening:

If `p` is a node representing a phrase beginning a block, and we're in the colon syntax, then it is followed by a word which is the colon: thus if `p` reads "if x is 2" then the word following the "2" will be ":". We replace it with the word "begin" and stretch `p` so that it now reads "if x is 2 begin", as it would if the begin/end syntax had been used.

⟨Insert begin marker and increase expected indentation if a block begins here 9⟩ ≡

```
    if ((csp) && (csp->subordinate_to == NULL) && (pn_int_annotation(p, colon_block_command_ANNOT)))
{
        p->word_ref2++;
        change_text_of_word(p->word_ref2, "begin");
        expected_indent++;
        if (csp->indent_subblocks) expected_indent++;
        blstack_construct[blo_sp] = csp;
        blstack_stage[blo_sp] = 0;
        blstack_opening_phrase[blo_sp++] = p;
        just_opened_block = TRUE;
    }
```

This code is used in §5.

§**10.** Now for the closing of colon-syntax blocks. We know that blocks must be being closed if the indentation has jumped backwards: but it may be that many blocks are being closed at once. (It may also be that the indentation has gone awry.)

⟨Try closing blocks to bring expected indentation down to match 10⟩ ≡

```
if ((just_opened_block) &&
    (blo_sp > 0) &&
    (!(blstack_construct[blo_sp-1]->body_empty_except_for_subordinates)))
    ⟨Issue problem for an empty block 16⟩;
while (indent < expected_indent) {
    parse_node *opening;
    if (blo_sp == 0) {
        ⟨Record a misalignment of indentation 13⟩;
        indent = expected_indent;
        break;
    }
    if ((blstack_construct[blo_sp-1]->body_empty_except_for_subordinates) &&
        (expected_indent - indent == 1)) {
        indent = expected_indent;
        break;
        ⟨Issue problem for non-case in a switch 17⟩;
        indent = expected_indent;
        break;
    }
    expected_indent--;
    if (blstack_construct[blo_sp-1]->indent_subblocks) expected_indent--;
    opening = blstack_opening_phrase[--blo_sp];
    ⟨Insert end marker to match the opening of the block phrase 12⟩;
}
```

This code is used in §5,7,8,5,7,8,5,7,8.

§**11.**

⟨Record a phrase within current block 11⟩ ≡

```
if ((blo_sp > 0) &&
    (blstack_stage[blo_sp-1] == 0) &&
    (blstack_construct[blo_sp-1]->body_empty_except_for_subordinates)) {
    ⟨Issue problem for non-case in a switch 17⟩;
}
just_opened_block = FALSE;
```

This code is used in §7.

§**12.** An end marker is a phrase like “end if” which matches the “if... begin” above it: here we insert such a marker at a place where the source text indentation implicitly requires it.

⟨Insert end marker to match the opening of the block phrase 12⟩ ≡

```
parse_node *implicit_end = new_node(COMMAND_NT);
implicit_end->next = prev->next; prev->next = implicit_end;
prev = implicit_end;
implicit_end->word_ref1 = lexer_wordcount;
feed_into_lexer("end ", FALSE, FALSE);
splice_words(opening->word_ref1, opening->word_ref1);
implicit_end->word_ref2 = lexer_wordcount - 1;
```

This code is used in §10.

§**13.** Here we throw what amounts to an exception...

⟨Record a misalignment of indentation 13⟩ ≡

```
indent_misalign = TRUE;
if (first_misaligned_phrase == NULL) first_misaligned_phrase = p;
```

This code is used in §7,10,7,10,7,10.

§**14.** ...and catch it with something of a catch-all message:

⟨Issue problem message for misaligned indentation 14⟩ ≡

```
current_sentence = rout;
quote_source_eliding_begin(1, current_sentence);
quote_source_eliding_begin(2, first_misaligned_phrase);
handmade_problem(_P_(C4MisalignedIndentation));
issue_problem_segment(
    "The phrase or rule definition %1 is written using the 'colon "
    "and indentation' syntax for its 'if's, 'repeat's and 'while's, "
    "where blocks of phrases grouped together are indented one "
    "tab step inward from the 'if ...:' or similar phrase to which "
    "they belong. But the tabs here seem to be misaligned, and I can't "
    "determine the structure. The first phrase going awry in the "
    "definition seems to be %2, in case that helps. %PThis sometimes "
    "happens even when the code looks about right, to the eye, if rows "
    "of spaces have been used to indent phrases instead of tabs.");
issue_problem_end();
```

This code is used in §5.

§**15.**

⟨Issue problem message for run-ons within phrase definition 15⟩ ≡

```
current_sentence = rout;
quote_source_eliding_begin(1, current_sentence);
quote_source_eliding_begin(2, run_on_at);
handmade_problem(_P_(C4RunOnsInTabbedRoutine));
issue_problem_segment(
    "The phrase or rule definition %1 is written using the 'colon "
    "and indentation' syntax for its 'if's, 'repeat's and 'while's, "
    "but that's only allowed if each phrase in the definition "
    "occurs on its own line. So phrases like %2, which follow "
    "directly on from the previous phrase, aren't allowed.");
issue_problem_end();
```

This code is used in §5.

§**16.** It's a moot point whether the following should be incorrect syntax, but it far more often happens as an accident than anything else, and it's hard to think of a sensible use.

⟨Issue problem for an empty block 16⟩ ≡

```
LOG("$T\n", rout);
current_sentence = rout;
quote_source_eliding_begin(1, current_sentence);
quote_source_eliding_begin(2, prev);
quote_source_eliding_begin(3, p);
handmade_problem(_P_(C4EmptyIndentedBlock));
issue_problem_segment(
    "The phrase or rule definition %1 is written using the 'colon "
    "and indentation' syntax for its 'if's, 'repeat's and 'while's, "
    "where blocks of phrases grouped together are indented one "
    "tab step inward from the 'if ...:' or similar phrase to which "
    "they belong. But the phrase %2, which ought to begin a block, "
    "is immediately followed by %3 at the same or a lower indentation, "
    "so the block seems to be empty - this must mean there has been "
    "a mistake in indenting the phrases.");
issue_problem_end();
```

This code is used in §10.

§**17.**

⟨Issue problem for non-case in a switch 17⟩ ≡

```
current_sentence = rout;
quote_source_eliding_begin(1, current_sentence);
quote_source_eliding_begin(2, p);
handmade_problem(_P_(C4NonCaseInIf));
issue_problem_segment(
    "In the phrase or rule definition %1, the phrase %2 came as a "
    "surprise since it was not a case in an 'if X is...' but was "
    "instead some other miscellaneous instruction.");
issue_problem_end();
```

This code is used in §10,11,10,11,10,11.

§18.

⟨Issue problem for an intermediate phrase not matching 18⟩ ≡

```
    if (indent_misalign == FALSE) {
        current_sentence = p;
        if (csp->subordinate_to == if_CSP)
            sentence_problem(_P_(C4MisalignedOtherwise),
                "this seems to be misplaced since it is not placed within an 'if'",
                "as it must be. An 'otherwise' must be vertically underneath the "
                "'if' to which it corresponds, at the same indentation.");
        if (csp->subordinate_to == switch_CSP)
            sentence_problem(_P_(C4MisalignedCase),
                "this seems to be misplaced since it is not a case within an "
                "'if X is...'",
                "as it must be. Each case must be placed one tab stop in from "
                "the 'if X is...' to which it belongs, and the instructions "
                "for what to do in that case should be one tab stop further in "
                "still.");
    }
```

This code is used in §8.

§19.

⟨Issue problem for an intermediate phrase out of sequence 19⟩ ≡

```
    if (indent_misalign == FALSE) {
        current_sentence = p;
        sentence_problem(_P_(C4MisarrangedOtherwise),
            "this seems to be misplaced since it is out of sequence within its 'if'",
            "with an 'otherwise if...' coming after the more general 'otherwise' "
            "rather than before. (Note that an 'otherwise' or 'otherwise if' must "
            "be vertically underneath the 'if' to which it corresponds, at the "
            "same indentation.");
    }
```

This code is used in §8.

§20. **Basic Structural Syntax.** The following routine is an attempt to contain information about the basic structural phrases in one place in NI, so that if future loop constructs are added, they can fairly simply be put here.

```
control_structure_phrase *csp_new(void) {
    control_structure_phrase *csp = CREATE(control_structure_phrase);
    csp->subordinate_to = NULL;
    csp->indent_subblocks = FALSE;
    csp->body_empty_except_for_subordinates = FALSE;
    csp->suppress_RULES_ALL_text = FALSE;
    csp->used_at_stage = -1;
    csp->requires_new_syntax = FALSE;
    csp->allow_run_on = FALSE;
    return csp;
}
void create_standard_csps(void) {
    switch_CSP = csp_new();
```

```
    switch_CSP->body_empty_except_for_subordinates = TRUE;
    switch_CSP->indent_subblocks = TRUE;
    switch_CSP->requires_new_syntax = TRUE;

    if_CSP = csp_new();

    repeat_CSP = csp_new();

    while_CSP = csp_new();

    otherwise_CSP = csp_new();
    otherwise_CSP->subordinate_to = if_CSP;
    otherwise_CSP->suppress_RULES_ALL_text = TRUE;
    otherwise_CSP->used_at_stage = 1;

    otherwise_if_CSP = csp_new();
    otherwise_if_CSP->subordinate_to = if_CSP;
    otherwise_if_CSP->suppress_RULES_ALL_text = TRUE;
    otherwise_if_CSP->used_at_stage = 0;

    case_CSP = csp_new();
    case_CSP->subordinate_to = switch_CSP;
    case_CSP->suppress_RULES_ALL_text = TRUE;
    case_CSP->used_at_stage = 1;
    case_CSP->requires_new_syntax = TRUE;
    case_CSP->allow_run_on = TRUE;

    default_case_CSP = csp_new();
    default_case_CSP->subordinate_to = switch_CSP;
    default_case_CSP->suppress_RULES_ALL_text = TRUE;
    default_case_CSP->used_at_stage = 2;
    default_case_CSP->requires_new_syntax = TRUE;
    default_case_CSP->allow_run_on = TRUE;
}

int is_otherwise_if(int w1, int w2) {
    if (detect_control_structure(w1, w2) == otherwise_if_CSP) return TRUE;
    return FALSE;
}
```

The function create_standard_csps is invoked by a command in a `.i6t` template file.
The function is_otherwise_if is called from 7/cmsp.

§**21. Detect Control Structures.** And here we parse the text to see which, if any, of the above control structures it might be.

```
control_structure_phrase *detect_control_structure(int w1, int w2) {
    if [[w1, w2 == if ... is begin]] return switch_CSP;                    for problem tracking only
    if [[w1, w2 == if ... is]] return switch_CSP;
    if [[w1, w2 == if/unless ...]] return if_CSP;
    if [[w1, w2 == repeat ...]] return repeat_CSP;
    if [[w1, w2 == while ...]] return while_CSP;
    if [[w1, w2 == else/otherwise]] return otherwise_CSP;
    if [[w1, w2 == else/otherwise if/unless ...]] return otherwise_if_CSP;
    if [[w1, w2 == DASHDASH otherwise]] return default_case_CSP;
    if [[w1, w2 == DASHDASH ...]] return case_CSP;
    return NULL;
}
```

The function detect_control_structure is called from 4/sent and 7/cmsp.

Verify Parse Tree 4/verif

Purpose

To check that the parse tree has been correctly constructed, in hopes of detecting bugs in the program: this check should never fail.

Template interpreter commands

```
7   {-callv:verify_parse_tree}
```

Definitions

¶1. The parse tree is a complicated structure, arbitrarily wide and deep, and containing about 30 different node types, each subject to its own rules of usage. This is both good and bad: bad because complexity is always the enemy of program correctness, good because it gives us an independent opportunity to test a great deal of what Chapter 4's code has done. If, given every source text in the `intest` suite – some correct, some with problems – Chapter 4 always constructs a well-formed tree, we must be doing something right.

We verify the parse tree at the end of the initial parsing stage whose narrative story is told by Chapters 3 and 4. (That's why this section comes last in Chapter 4.) At this time, the parse tree is the output of the *A*-grammar parser. Nodes of one type (`FROM_NT`) which were at one time present have now been eliminated; nodes of a number of other types will not appear until later, when we work further with the tree in Chapter 8. So at present, in what we call the "initial parse tree", there can be up to 20 node types in evidence. But these are not arbitrarily arranged: for instance, a `SENTENCE_NT` node cannot legally be beneath a `NOUNPHRASE_NT` one. The collection of rules like this which the tree must satisfy is called its "invariant", and is expressed in convenient numerical form by the table below.

Note that this is verification, not an attempt to correct matters. If any test fails, NI will stop with an internal error. This should never happen.

¶2. The invariant for the usage of a node type in the initial parse tree is given by a triplet of numbers $(c_{\min}, c_{\max}, w)$ and a node type t_{parent}, where it is always true that

$$0 \le c_{\min} \le c_{\max} \le \infty.$$

(In fact, we use the enormous number `INFTY` to represent ∞.) The rule to be applied at every node is that:

(1) The weight at the node $w_{\text{node}} \neq 0$;

(2) If $w_{\text{node}} > 0$ then, except at the root, we must have $w_{\text{node}} \ge \text{abs}(w_{\text{parent}})$, where $\text{abs}(x)$ is the absolute value of x;

(3) If $t_{\text{parent}} \neq -1$ then this must be the node type of the parent;

(4) The number of children c satisfies $c_{\min} \le c \le c_{\max}$.

Rules (3) and (4) speak for themselves, and rule (1) is simply a way to mark certain node types as being illegal in the initial parse tree – these are the ones which will only be added later, in Chapter 8.

The upshot of rule (2) is that heavier nodes always occur lower down in the tree than lighter ones. For instance, an `AND_NT` node (weight 6) cannot occur anywhere in the subtree beneath an `X_OF_Y_NT` node (weight 8). A negative weight, $w < 0$, allows a node of this type to occur anywhere: but it still starts a well-formed subtree on its own terms, because it behaves to its children as if it had weight $-w$.

¶3. The following structure is used only for a row in a table of what we might call metadata about node types: information on where each node type can appear, and what restrictions apply to its use. We also store textual names for the node types here, as this is convenient for logging.

The table also includes information needed for the `allow_in_assertions` check, called by the assertion-maker of Chapter 8 as it runs. If the text asserts that, say, "The cat is in the bag." then the assertion-maker will try to equate a parse subtree for "the cat" against another for "in the bag", with node types at the roots of these subtrees being `NOUNPHRASE_NT` and `RELATIONSHIP_NT` respectively. The check makes sure that inappropriate subtrees are never accidentally fed to the assertion-maker.

```
typedef struct parse_tree_node_type {
    char *node_type_name;                    text of name of type, such as "COMMAND_NT"
    int min_children;                        minimum legal number of child nodes
    int max_children;                        maximum legal number of child nodes
    int node_weight;                         where higher weights sink to the bottom
    int required_parent_node_type;           required node type of parent, or -1
    int allow_in_assertions;                 allow this on either side of an assertion?
} parse_tree_node_type;
```

The structure parse_tree_node_type is private to this section.

¶4. The actual metadata for the node types is stored in the following array. (The order of entries must match the defined values of the `_MT` constants in the Parse Tree section.)

```
parse_tree_node_type parse_tree_node_types[];
```

---

§**1.** Note that `FROM_NT` is listed as not allowed in the initial parse tree: although it did exist in the tree for part of this chapter, the tidying up done by Of And From removed every `FROM_NT` node.

define `INFTY 1000000000` *if ever a node has more than a billion children, we are in trouble anyway*

```
parse_tree_node_type parse_tree_node_types[HIGHEST_NODE_TYPE+1] = {
    the columns are: name, max and min children, weight, required parent (if any), whether allowed in assertions
    { "(zero node type)",   0, INFTY,  0,   -1,           FALSE },
    { "ROOT_NT",            0, INFTY,  1,   -1,           FALSE },
    { "BIBLIOGRAPHIC_NT",   0,  0,     2,   ROOT_NT,      FALSE },
    { "HEADING_NT",         0,  0,     2,   ROOT_NT,      FALSE },
    { "INCLUDE_NT",         0,  0,     2,   ROOT_NT,      FALSE },
    { "BEGINHERE_NT",       0,  0,     2,   ROOT_NT,      FALSE },
    { "ENDHERE_NT",         0,  0,     2,   ROOT_NT,      FALSE },
    { "SENTENCE_NT",        0, INFTY,  2,   ROOT_NT,      FALSE },
    { "ROUTINE_NT",         0, INFTY,  2,   ROOT_NT,      FALSE },
    { "INFORM6CODE_NT",     0,  0,     2,   ROOT_NT,      FALSE },
    { "TABLE_NT",           0,  0,     2,   ROOT_NT,      FALSE },
    { "EQUATION_NT",        0,  0,     2,   ROOT_NT,      FALSE },
    { "TRACE_NT",           0,  0,     2,   ROOT_NT,      FALSE },
    { "RELATIONSHIP_NT",    0,  2,    -3,   -1,           TRUE },
    { "CALLED_NT",          2,  2,     4,   -1,           FALSE },
    { "WITH_NT",            2,  2,     5,   -1,           TRUE },
    { "AND_NT",             2,  2,     6,   -1,           TRUE },
    { "KIND_NT",            0,  1,     7,   -1,           TRUE },
    { "X_OF_Y_NT",          2,  2,     8,   -1,           TRUE },
    { "VERB_NT",            0,  0,    10,   SENTENCE_NT,  FALSE },
```

```
    { "CREATED_NT",          0,  0,    10,   -1,           TRUE },
    { "NOUNPHRASE_NT",       0,  0,    10,   -1,           TRUE },
    { "PROPERTYLIST_NT",     0,  0,    10,   -1,           TRUE },
    { "FROM_NT",             2,  2,     0,   -1,           FALSE },
    { "COMMAND_NT",          0,  0,    10,   ROUTINE_NT,   FALSE },
    { "ALLOWED_NT",          1,  1,     0,   SENTENCE_NT,  TRUE },
    { "EVERY_NT",            0, INFTY,  0,   -1,           TRUE },
    { "INSTANCE_NT",         0, INFTY,  0,   -1,           TRUE },
    { "VALUE_NT",            0, INFTY,  0,   -1,           TRUE },
    { "ACTION_NT",           0, INFTY,  0,   -1,           TRUE },
    { "ADJECTIVE_NT",        0, INFTY,  0,   -1,           TRUE },
    { "PROPERTYNOUN_NT",     0, INFTY,  0,   -1,           TRUE },
    { "PROPERTYCALLED_NT",   2,  2,     4,   -1,           FALSE },
    { "TOKEN_NT",            0, INFTY,  0,   -1,           FALSE }
};
```

§**2. Logging node types.** And also making node names available to the machinery for producing internal errors when incorrect node types are encountered, though we hope this will never be used.

```
void log_node_type(int t) {
    if ((t<1) || (t>HIGHEST_NODE_TYPE)) LOG("?%d_NT", t);
    else LOG("%s", parse_tree_node_types[t].node_type_name);
}
char *pnt_get_name(int t) {
    return parse_tree_node_types[t].node_type_name;
}
```

The function log_node_type is called from 2/dl.
The function pnt_get_name is called from 2/prob3.

§**3. Checking that the tree is even a tree.** Almost always – let's say, at the end of each cycle of the `.i6t` interpreter – the main component of the parse tree is required to be a tree. (The "main component" means "all those nodes connected, directly or indirectly, from the tree root node".) It must contain no loops, and all of its `next` and `down` links must point to valid parse nodes or else to `NULL`. Finally, every node must have a valid type number.

The following routine checks that this is so for any subtree: to check it for the whole shebang, we must call `verify_integrity_of_subtree(tree_root, f)`.

```
int tree_stats_size = 0, tree_stats_depth = 0, tree_stats_width = 0;
void verify_integrity_of_subtree(parse_node *p, int worth_logging) {
    tree_stats_size = 0; tree_stats_depth = 0; tree_stats_width = 1;
    verify_tree_integrity_recursively(p->down, p, "down", 0);
    tidy_tree_after_verification(p->down);
    if (worth_logging)
        LOGIF(VERIFICATIONS, "[Initial parse tree has %d nodes, width %d and depth %d.]\n",
            tree_stats_size, tree_stats_width, tree_stats_depth);
}
```

The function verify_integrity_of_subtree is called from 4/ofs.

§**4.** To check that we never visit the same node twice (which would indicate a loop), we will mark each visited node with an imaginary cross: that way, if we visit one with a cross on it, we know this isn't a tree. We can't afford the storage to allocate an annotation for this, so the imaginary cross is actually that the node type has been increased by the following amount.

```
define OUTRAGEOUSLY_LARGE_NODE_TYPE 10000
define MARKED_WITH_CROSS(t)
    ((t >= OUTRAGEOUSLY_LARGE_NODE_TYPE + 1) &&
        (t < OUTRAGEOUSLY_LARGE_NODE_TYPE + HIGHEST_NODE_TYPE))
```

§**5.** After verification, of course, we need to erase all these crosses:

```
void tidy_tree_after_verification(parse_node *p) {
    for (; p; p = p->next) {
        int t = pn_get_node_type(p);
        if (MARKED_WITH_CROSS(t))
            pn_set_node_type(p, t - OUTRAGEOUSLY_LARGE_NODE_TYPE);
        if (p->down) tidy_tree_after_verification(p->down);
    }
}
```

§**6.** The verification traverse is a very cautious manoeuvre: we step through the tree, testing each branch with our outstretched foot in case it might be illusory or broken. At the first sign of trouble we panic.

```
void verify_tree_integrity_recursively(parse_node *p, parse_node *from, char *way,
    int depth) {
    int t, width;
    pointer_sized_int probably_an_address = (pointer_sized_int) p;
    depth++; if (depth > tree_stats_depth) tree_stats_depth = depth;
    for (width = 0; p; p = p->next, width++) {
        if ((probably_an_address == 0) || (probably_an_address == -1)) {
            LOG("Link %s broken from:\n$P", way, from);
            internal_error_tree_unsafe("Link broken in parse tree");
        }
        t = pn_get_node_type(p);
        if (MARKED_WITH_CROSS(t)) {
            LOG("Cycle found in parse tree, found %s from:\n$P", way, from);
            internal_error_tree_unsafe("Cycle found in parse tree");
        }
        if ((t >= 1) && (t < HIGHEST_NODE_TYPE)) {
            pn_set_node_type(p, t + OUTRAGEOUSLY_LARGE_NODE_TYPE);
            tree_stats_size++;
        } else {
            LOG("Invalid node type (%d) found %s from:\n$P", t, from);
            internal_error_tree_unsafe("Link broken in parse tree");
        }
        if (p->down) verify_tree_integrity_recursively(p->down, p, "down", depth);
    }
    if (width > tree_stats_width) tree_stats_width = width;
}
```

§**7. The initial parse tree invariant.** We protect ourselves by first checking that the tree is intact as a structure: once we know the tree is safe to climb over, we can wander about counting children with impunity. If there are multiple failures, we itemise them to the debugging log, and only produce a single internal error at the end.

```
int node_errors = 0;
void verify_parse_tree(void) {
    LOGIF(VERIFICATIONS, "[Verifying initial parse tree]\n");
    if (tree_root == NULL) internal_error_tree_unsafe("Root of parse tree NULL");
    if (tree_root->down) verify_integrity_of_subtree(tree_root, TRUE);
    node_errors = 0;
    verify_initial_parse_tree_invariant(tree_root, NULL, 1);
    if (node_errors > 0) {
        LOG("[Verification failed: %d node errors]\n", node_errors);
        internal_error_tree_unsafe("The initial parse tree is broken");
    }
    LOGIF(VERIFICATIONS, "[Initial parse tree correct.]\n");
}
```

The function verify_parse_tree is invoked by a command in a `.i6t` template file.

§**8.** Note that on every call to the following routine, (i) `p` is a valid parse node; (ii) either `p` is the tree root, in which case `parent` is `NULL`, or `parent` is the unique node having `p` among its children; and (iii) `current_weight` is the absolute value of $w_{\rm parent}$.

```
void verify_initial_parse_tree_invariant(parse_node *p, parse_node *parent, int current_weight) {
    parse_node *q;
    int t = pn_get_node_type(p), children_count = 0;
    int weight_here = parse_tree_node_types[t].node_weight;
    ⟨Check rule (1) of the initial parse tree invariant 9⟩;
    ⟨Check rule (2) of the initial parse tree invariant 10⟩;
    ⟨Check rule (3) of the initial parse tree invariant 11⟩;
    if (weight_here < 0) weight_here = -weight_here;
    for (q=p->down; q; q=q->next, children_count++)
        verify_initial_parse_tree_invariant(q, p, weight_here);
    ⟨Check rule (4) of the initial parse tree invariant 12⟩;
}
```

§**9.** Rule (1): $w_{\rm node} \neq 0$.

⟨Check rule (1) of the initial parse tree invariant 9⟩ ≡

```
    if (weight_here == 0) {
        LOG("N%d is %s, which is not allowed in initial tree\n",
            p->allocation_id, parse_tree_node_types[t].node_type_name);
        LOG("$P", p);
        node_errors++;
    }
```

This code is used in §8.

§**10.** Rule (2): If $w_{\text{node}} > 0$ then, except at the root, we must have $w_{\text{node}} \geq \text{abs}(w_{\text{parent}})$.

⟨Check rule (2) of the initial parse tree invariant 10⟩ ≡

```
if ((parent) && (weight_here > 0) && (weight_here < current_weight)) {
    LOG("N%d is %s (weight %d): should not be a child of %s (weight %d)\n",
        p->allocation_id,
        parse_tree_node_types[t].node_type_name,
        parse_tree_node_types[t].node_weight,
        parse_tree_node_types[pn_get_node_type(parent)].node_type_name,
        current_weight);
    node_errors++;
    if (parent == tree_root) LOG("$P", p); else LOG("$P", parent);
}
```

This code is used in §8.

§**11.** Rule (3): If $t_{\text{parent}} \neq -1$ then this must be the node type of the parent.

⟨Check rule (3) of the initial parse tree invariant 11⟩ ≡

```
if (parent) {
    int t_parent = parse_tree_node_types[t].required_parent_node_type;
    if ((t_parent != -1) && (t_parent != pn_get_node_type(parent))) {
        LOG("N%d is a %s with parent %s not %s\n",
            p->allocation_id,
            parse_tree_node_types[t].node_type_name,
            parse_tree_node_types[pn_get_node_type(parent)].node_type_name,
            parse_tree_node_types[t_parent].node_type_name);
        if (parent == tree_root) LOG("$P", p); else LOG("$P", parent);
        node_errors++;
    }
}
```

This code is used in §8.

§**12.** Rule (4): The number of children c satisfies $c_{\min} \leq c \leq c_{\max}$.

⟨Check rule (4) of the initial parse tree invariant 12⟩ ≡

```
if (children_count < parse_tree_node_types[t].min_children) {
    LOG("N%d has %d children, but min for %s is %d:\n",
        p->allocation_id,
        children_count, parse_tree_node_types[t].node_type_name,
        parse_tree_node_types[t].min_children);
    log_subtree(p, 1);
    node_errors++;
}
if (children_count > parse_tree_node_types[t].max_children) {
    LOG("N%d has %d children, but max for %s is %d:\n",
        p->allocation_id,
        children_count, parse_tree_node_types[t].node_type_name,
        parse_tree_node_types[t].max_children);
    log_subtree(p, 1);
    node_errors++;
}
```

This code is used in §8.

§**13. The assertion-maker's invariant.** Hmm: *The Assertion-Maker's Invariant* might make a good magic-realism novel, in which an enigmatic wise man of Samarkand builds an ingenious box from camphorwood in which he traps the dreams of the people, who – However. When assertions are processed in Chapter 8, the subtrees being compared will be required to be such that their head nodes each pass this test:

```
int pnt_allow_in_assertions(parse_node *subtree) {
    int t;
    verify_integrity_of_subtree(subtree, FALSE);
    t = pn_get_node_type(subtree);
    if (parse_tree_node_types[t].allow_in_assertions) return TRUE;
    return FALSE;
}
```

The function pnt_allow_in_assertions is called from 8/mass.

§**14.** And that concludes the tree checking, and with it, the narrative story told throughout Chapters 2 and 3 which take us from the start of NI's run through to the final output of the *A*-grammar parser.

5 Meanings

`5/its`: *Introduction to Semantics.w* A general introduction to the S-parser and the data structures it makes use of.

`5/det`: *Determiners and Quantifiers.w* To create the determiners found in standard English which refer to collections of things, and to create their meanings as logical quantifiers.

`5/bp`: *Binary Predicates.w* To create and manage binary predicates, which are the underlying data structures beneath Inform's relations.

`5/rel`: *Relations.w* What Inform internally calls "binary predicates", the user calls "relations". In this section, we parse definitions of new relations and create the resulting `binary_predicate` objects.

`5/conj`: *Conjugation of Verbs.w* To define verbal and prepositional forms for relations, in different tenses and numbers.

`5/aph`: *Adjectives.w* To identify the names of all adjectives used in source text, and to organise the multiple different senses each adjective might have.

`5/litp`: *Literal Patterns.w* To manage the possible notations with which literal values can be written.

`5/em`: *Excerpt Meanings.w* To register and deregister meanings for excerpts of text as nouns, adjectives, imperative phrases and other usages.

`5/unitr`: *Unicode Translations.w* To manage the names assigned to Unicode character values.

`5/ml`: *Meaning Lists.w* To build meaning lists, which despite the name are really tree structures, showing possible interpretations of a sequence of words.

`5/arch`: *Architecture of the S-Parser.w* To describe how the S-parser is organised, and to define a kit of C macros for writing its routines.

`5/parse`: *Parse Excerpts.w* Given an excerpt `(w1,w2)`, to construct a meaning list of all registered excerpt meanings which it matches.

`5/lit`: *Parse Literals.w* To decide if the text in `(w1,w2)` is a value referred to notationally rather than by name.

`5/candd`: *Constants and Descriptions.w* To parse noun phrases in constant contexts, which specify values either explicitly or by describing them more or less vaguely.

`5/tandv`: *Type Expressions and Values.w* To parse two forms of noun: a noun phrase in a sentence, and a description of what text can be written in a given situation.

`5/varc`: *Verbal and Relative Clauses.w* To break down an excerpt into NP and VP-like clauses, perhaps with a primary verb (to make a sentence), perhaps only a relative clause (to make a more complex NP).

`5/candp`: *Conditions and Phrases.w* To parse the text of To... phrases, say phrases and conditions.

`5/mlc`: *Meaning List Conversions.w* To convert meaning lists into specifications, for the use of the rest of NI.

Introduction to Semantics 5/its

Purpose

A general introduction to the S-parser and the data structures it makes use of.

---

§**1.** At this point, the text read in by NI is now a stream of words, each of which is identified by a pointer to a `vocabulary_entry` structure in its dictionary. The words are numbered upwards from 0, and we refer to any contiguous run of words as an "excerpt", often writing `(w1, w2)` to mean the text starting with word `w1` and continuing to word `w2`. The stream of words has been divided further into sentences.

Inform has two mechanisms for making sense of this text, the A-parser and the S-parser.

(A) A stands for "assertion". For instance, "Two men are in Verona." is an assertion, telling Inform that at the start of play there are to be two previously unknown men and that they begin in the room called Verona. The A-parser handles entire sentences.

(S) S stands for "semantics", the study of how already-understood meanings correspond to excerpts of text within a sentence. The S-parser handles anything from tiny excerpts like "6" through noun phrases such as "Verona" to complicated expressions like "the number of men in Verona".

There are many similarities between the A-parser and the S-parser, partly because A makes use of S, but also because they contain parallel mechanisms which handle verbs and prepositions similarly. But there are also many differences. The A-parser will accept "On the dressing table is an amber comb." even if table and comb have never been mentioned before, whereas the S-parser can only recognise meanings already defined. On the other hand, the S-parser will accept conditions like the one in "if there are fewer than 8 men in Verona, ..." whereas the A-parser would reject the assertion "There are fewer than 8 men in Verona." as being too vague to act upon. Similarly, the A-parser works only in the present tense, whereas the S-parser can handle the past and perfect tenses. (Neither can handle any future tenses, since a computer cannot either control or definitely predict the future.)

The A-parser works by applying the S-parser to text at `parse_node` structures in the parse tree. So we will build the S-parser first, which will take up Chapters 5, 6 and 7, and won't involve the parse tree at all. We will then go back to the parse tree in Chapter 8 to write the A-parser.

§**2.** The S-parser is similar to the expression parser in a regular compiler. It is in some ways simpler because natural language tends not to form complex formulae, but in other ways more complicated, because performance issues are very significant when comparing excerpts of text, and because there are many more ambiguities to resolve.

Our aim is to turn any excerpt into a `specification` structure inside Inform. This is a universal holder for both values and descriptions of values, where "value" is interpreted very broadly. It is usually too difficult to go directly from text to a `specification`, so we use a two-stage process:

(1) parse the text to a `meaning_list` which holds all possible interpretations of it, and then
(2) convert the most likely-looking interpretation(s) to a `specification`.

Thus `meaning_list` structures are private to the S-parser, and are never seen outside Chapters 5 and 6, whereas `specification` structures appear all over Inform.

Chapter 5 divides into two halves. The first half deals with how we store details of verbs, prepositions, relations, determiners and other natural language gadgets; this provides a common English grammar which will be used both by the S-grammar and the A-grammar. The second half of Chapter 5 contains the code to turn excerpts into meaning lists, and to convert the easier cases into specifications. Chapter 6 handles predicate calculus, the form of mathematical logic used by Inform to represent the meaning of complicated sentences: and this performs the remaining difficult cases of meaning list conversion. Chapter 7 then contains all of Inform's general code for handling specifications. Chapters 6 and 7 both also provide services to the rest of Inform to help make use of the `specification` and `pcalc_prop` structures which the S-parser generates.

§**3.** Consider the following contrived example.

if Mr Fitzwilliam Darcy was carrying at least three things which are in the box, increase the score by 7;

(1) There are `excerpt_meaning` structures for "Mr Fitzwilliam Darcy" and "Mr Bingham's box", which hold the wording needed to refer to these objects. In parsing the example sentence, we connect these structures to the excerpts "Mr Fitzwilliam Darcy" – an exact match – and "the box" – an abbreviated one. The `excerpt_meaning` structures contain pointers to further `world_object` structures which represent the identities of these two tangible things, that is, Darcy and his friend's box.

(2) Another `excerpt_meaning` holds the name "score" and points it to a `quantity` structure for the relevant global variable.

(3) And a further `excerpt_meaning` holds the name "things" and points it to a `world_object` structure representing the common identity shared by all things. Inform treats individual, tangible objects such as Mr Darcy and intangible categories of objects such as thing by representing both with the same structure – `world_object`. This mirrors the way that common and proper nouns are grammatically quite similar in natural language.

(4) The final noun phrase in the above example is "7". There's no `excerpt_meaning` structure for this – it would be insanely inefficient to make such things – and instead it is parsed directly as a "literal", being converted immediately into a `specification`, of which more below.

(5) Another `excerpt_meaning` structure holds the wording "if ... , ..." and is connected to a `phrase` structure for the "if" construction. Here, the wording includes flexible-sized gaps (written "...") where excerpts should appear: the S-parser will only recognise this if the excerpts make sense in themselves. The combination of a `phrase` plus the results of parsing these gaps is stored in a structure called an `invocation`.

(6) In the example, the first gap is filled by "Mr Fitzwilliam Darcy was carrying at least three things which are in the box", which the S-parser detects as being a condition. This is translated into a `pcalc_prop` structure – a predicate-calculus proposition, that is, which is a representation in mathematical logic of the meaning of this sentence.

  (a) "was carrying" is recognised as matching wording in a `verb_usage` structure. This points to an underlying relation, stored in a `binary_predicate` structure, but combines it with an indication of tense stored in a `time_period`. Here the `binary_predicate` is the carrying relation and the `time_period` is the past tense. (The term "binary predicate" comes from logic once again; an Inform author would call the same concept a "relation".)

  (b) "are in" is recognised as a usage of the verb *to be* plus "in", which matches the wording of a `preposition_usage` structure. Here the tense derives only from the *to be* part: which is "are", so the `time_period` parsed is the present tense. This makes the `preposition_usage` a simpler business than the `verb_usage` structure – it only needs to refer to the underlying meaning, which is once again a `binary_predicate` structure, the one for the containment relation.

  (c) "which" is a word introducing a relative clause. A sentence can only have one primary verb, which in this example is "was carrying". But other verbs can exist in relative clauses, and the effect of writing "X which V Y" qualifies X by saying that any noun N matching X must also satisfy "N V Y", where V is the verb. The relative-clause construction is an example of syntax built directly into the S-parser. It doesn't come from any data structures, like the meanings of "score" or "Mr Fitzwilliam Darcy".

  (d) "at least three things" is an example of a noun phrase which has a head and a tail. The head, "at least three", is recognised as matching the wording in a `determiner` structure, "at least *number*", together with the literal number 3. Once again, the `determiner` describes textual appearances; it points to another structure, a `quantifier`, to hold the meaning. This is another logical term, and Inform's debugging log would write the resulting term as `Card>=3` ("cardinality of at least 3"). Inform only uses `determiner` structures when they quantify, that is, when they talk about a possible range of objects rather than a single item. A grammar of English would probably say that the "the" in "the box" is also grammatically a determiner, but it doesn't get a `determiner` structure in Inform.

(7) The second gap in the "if ... , ..." excerpt is "increase the score by 1", which the S-parser detects as a use of yet another `phrase`, this time referred to by the `excerpt_meaning` structure for "increase ... by ...". It's worth noting that the S-parser doesn't check types, so it would have been happy to match

"increase 2 by 1" – an impossibility. The S-parser's job is to find all possible meanings at a textual level, sometimes producing a list of options: the type-checker will winnow these out later on.

So parsing the text "if Mr Fitzwilliam Darcy was carrying at least three things which are in the box, increase the score by 7" is going to result in a mass of pointers to different structures, and we need an umbrella structure to hold this mass together. This is what the `meaning_list` is for, but as explained above, it's really only an intermediate state used while the S-parser is working.

§**4.** One obvious category of word is missing: there are no adjectives in this example. Inform currently supports many sorts of adjective – either/or properties, such as "open"; values of kinds of value which coincide with properties, such as "green" as a value of a "colour"; and adjectives defined with conditions or full phrases, such as "invisible" resulting from "Definition: a thing is invisible if...".

The S-parser treats all adjectives alike – more or less just as names. This is because "open" may mean one thing for containers and another for scenes, for example. The identification of an adjective's name with its set of possible meanings is via a structure called `adjectival_phrase`.

§**5.** To sum up. If we write "text" → structure used for parsing → structure used to hold meaning, our example is parsed like so:

(1) "Mr Fitzwilliam Darcy" → `excerpt_meaning` → `world_object`

(2) "the score" → `excerpt_meaning` → `quantity`

(3) "things" → `excerpt_meaning` → `world_object`

(4) "7" → ...none... → `specification`

(5) "if Mr Fitzwilliam Darcy was carrying at least three things which are in the box, increase the score by 7" → `excerpt_meaning` → `invocation` (incorporating a `phrase`)

(6) "Mr Fitzwilliam Darcy was carrying at least three things which are in the box" → ...many... → `pcalc_prop`
   (a) "was carrying" → `verb_usage` → `binary_predicate` plus `time_period`
   (b) "are in" → `preposition_usage` → `binary_predicate` plus `time_period`
   (c) "at least three" → `determiner` → `quantifier` plus literal number

(7) "increase the score by 7" → `excerpt_meaning` → `invocation` (incorporating a `phrase`)

(8) Adjectives like the "closed" in "three closed doors" are identified by name only, with little attempt to detect which sense is meant, so they pass straight through the S-parser as pointers to `adjectival_phrase` structures.

§**6.** To sum up further still, `excerpt_meaning` structures are used to parse simple nouns and imperative phrases, whereas other specialist structures (`preposition_usage`, `determiner`, etc.) are used to parse the hinges which hold sentences together. Once parsed, individual excerpts tend to have meanings which might be pointers to a bewildering range of structures (`world_object`, `quantifier`, `binary_predicate`, `adjectival_phrase`, etc.) but these pointers are held together inside the S-parser by a single unifying construction: the `meaning_list`. And we will eventually turn the whole thing into a `specification` for the rest of Inform to use.

Determiners and Quantifiers 5/det

Purpose

To create the determiners found in standard English which refer to collections of things, and to create their meanings as logical quantifiers.

5/det.§1-3 Creating a quantifier; §4-6 Acting on quantifiers; §7 Creating a determiner; §8-9 Parsing the determiner at the head of a noun phrase; §10-15 The built-in set

---

Template interpreter commands

```
10   {-callv:make_built_in_determiners}
```

Definitions

¶1. In logic, a "quantifier" appears at the front of a statement which can apply to many cases, and describes the quantity of cases for which the statement is true: all of them, some of them, exactly six, and so on.

When a quantifier is used, it "ranges over a domain". The domain is the set of cases. For instance, in:

> if most of the doors are open, ...

the "most of" text is parsed into a quantifier written in the debugging log as `Proportion>50%`, and the domain is the set of all doors. We then test the inner condition ("open") for the objects in the domain.

Some quantifiers apply to a proportion of the domain, and the proportion is measured with a number we will call the T-coefficient, which is measured in tenths. Thus a quantifier talking about the entire domain ("all of the doors are open") will have $T = 10$, while the "most of" example above has $T = 5$. Other quantifiers apply to an exact number, a "cardinality" in logic jargon, rather than a proportion: for instance, "three doors are open". These quantifiers have $T = -1$.

Finally, a few quantifiers apply not to the cases in the domain which passed, but to those which didn't, and those are called "complementary" (because they describe the complement of the domain set). For instance, "all but six doors are open", where the "six" describes the number of closed doors and not the number of open ones.

¶2. These different ways to describe multiple outcomes are represented in Inform by `quantifier` structures. One exists for each different meaning supported by Inform – `ForAll`, `Exists` and so forth – except that some quantifiers take a numerical parameter, and a single `quantifier` structure represents the meaning for any value of this parameter. For instance, the cardinality quantifiers `Card=3` and `Card=17` are both represented by the same quantifier structure, whose pointer is called `exactly_quantifier` below. This is the result of parsing *exactly three* doors or *exactly 17* containers, for instance, where the parameter is 3 or 17 respectively.

```
typedef struct quantifier {
    char *operator;                          I6 operator to compare successes against the threshold
    int T_coefficient;                                                          see above
    int is_complementary;                    tests the complement of the set, not the set of matches
    int can_be_used_in_now;                                  can be asserted true or false using "now"
    int can_be_used_in_assertions;                                  can be used in assertion sentences
    struct quantifier *negated_quant;                                the logically converse determiner
    char *log_text;                          to be used in the debugging log when logging propositions
    MEMORY_MANAGEMENT
} quantifier;
```

The structure quantifier is private to this section.

¶3. The built-in set of 16 quantifiers, arranged in eight pairs, is as follows:

```
quantifier
    *for_all_quantifier = NULL,    *not_for_all_quantifier = NULL,
    *exists_quantifier = NULL,     *not_exists_quantifier = NULL,
    *all_but_quantifier = NULL,    *not_all_but_quantifier = NULL,
    *almost_all_quantifier = NULL, *almost_no_quantifier = NULL,
    *most_quantifier = NULL,       *under_half_quantifier = NULL,
    *at_least_quantifier = NULL,   *more_than_quantifier = NULL,
    *at_most_quantifier = NULL,    *less_than_quantifier = NULL,
    *exactly_quantifier = NULL,    *other_than_quantifier = NULL;
```

¶4. Whereas "quantifier" is a term from mathematical logic, "determiner" is a term from linguistics which approximately – but only approximately – means the same thing.

The determiner is the part of a noun phrase, always its head, which gives counting information to be combined with a common noun. Thus *the* clock, *seven* seals, *almost all of the* open doors, and so on. When a determiner appears to refer to a range of objects rather than a single item, NI translates it into a quantifier. Thus *the* clock is not parsed into a quantifier, but *all but three* rooms is.

The same quantifier can have several different verbal forms. For instance, *each* container and *every* container mean the same thing: both apply the `ForAll` quantifier to containers. These different verbal forms are stored in the `determiner` structure, and each one points to the `quantifier` structure which is its meaning.

```
typedef struct determiner {
    int allows_prefixed_not;                          can the word "not" come before this?
    struct vocabulary_entry *det_word1;               or null (e.g. for a bare number "N")
    struct vocabulary_entry *det_word2;                                              or null
    int takes_number;                                 does a number follow? (e.g. for "at least N"
    struct quantifier *quantifier_meant;                                  meaning of this quantifier
    char *index_text;                                 used in the Phrasebook index lexicon
    MEMORY_MANAGEMENT
} determiner;
```

The structure determiner is private to this section.

---

§**1. Creating a quantifier.** At present, there's only the built-in set, and no method exists to create new quantifiers from the source text or the template files, but what follows is written so that it would be fairly easy to add this ability.

```
quantifier *quant_new(char *op, int T, int is_comp, char *text) {
    quantifier *quant = CREATE(quantifier);
    quant->operator = op;
    quant->T_coefficient = T; quant->is_complementary = is_comp;
    quant->can_be_used_in_now = FALSE;
    quant->can_be_used_in_assertions = FALSE;
    quant->negated_quant = NULL;
    quant->log_text = text;
    return quant;
}
```

§**2.** That fills out the whole structure except for the negation pointers, and to ensure that these always occur in matched pairs, these are set here.

A little explanation may be useful about what we mean by negation. In traditional logic, the basic quantifiers "for all" and "there exists" are dual to each other in that they are related by a sort of negation: "there does not exist an open door" means the same as "all doors are closed", and so on. Thus

`Not ( ForAll x: P(x) )` is equivalent to `Exists x: Not(P(x))`

That isn't what we mean here. If Q and NQ are a quantifier and its negation in our sense, then:

`Not ( Q x: P(x) )` is equivalent to `NQ x: P(x)`

Why do we do this? There are several reasons. First, we are using a richer set of quantifiers than traditional logic provides, and most of these have natural negations which we were going to be creating anyway – so we may as well exploit that. Second, we are going to try to represent propositions using as much conjunction ("and") and as little disjunction ("or") as possible. Consider what effect de Morgan's laws have if we simplify:

```
Not ( ForAll x: closed(x) and locked(x) and lockable(x) )
```

in the traditional way: we obtain

```
Exists x: Not(closed(x)) or Not(locked(x)) or Not(lockable(x))
```

which introduces disjunction ("or") in just the way we don't want. By simply regarding `NotAll` as a quantifier in its own right, we obtain something much easier to handle:

```
NotAll x: closed(x) and locked(x) and lockable(x)
```

This is why we will be creating quantifiers `NotAll` and `DoesNotExist` – the negations of `ForAll` and `ThereExists` – even though they might seem puzzlingly redundant from a traditional logic point of view.

```
void quants_negate_each_other(quantifier *qx, quantifier *qy) {
    qx->negated_quant = qy; qy->negated_quant = qx;
}
quantifier *quant_get_negation(quantifier *quant) {
    return quant->negated_quant;
}
```

The function quant_get_negation is called from 6/simp.

§**3.** Logging a quantifier:

```
void log_quantifier(quantifier *quant, int parameter) {
    if (quant == NULL) { LOG("<NULL-QUANTIFIER>"); return; }
    LOG(quant->log_text, parameter);
}
```

The function log_quantifier is called from 5/ml and 6/aprop.

§**4. Acting on quantifiers.** When compiling code to test a proposition which includes a quantifier, we need to test the cases in the domain set to see how many of them qualify and how many do not. These counts are stored in local variables called `qcy_0`, `qcn_0` and so on: `qcn` means "quantifier count number" and is the size of the domain set, while `qcy` is the number of "yes" cases. Thus if the original source text read:

> if most of the closed doors are locked, ...

`qcy_0` will be the number of closed doors which turned out to be locked and `qcn_0` the total number of closed doors. (The indices `_0`, `_1`, ..., are used because the same routine may have to compile code to test several quantifiers.)

The following routine compiles an I6 condition to test whether the tallies are acceptable for the given quantifier. In the example above, the quantifier is `Proportion>50%`, and compiles to the test:

```
qcy_0 > 5*qcn_0/10
```

(It looks a little wasteful to multiply by 5 and then divide by 10, but I6 will fold that out in eventual code generation. When the proportion is 0/10ths or 10/10ths, though, we do generate simpler code, mostly so that the resulting I6 is more legible.)

```
void quant_compile_test(OUTPUT_STREAM, quantifier *quant, int index,
    int quantification_parameter) {
    int TC = quant->T_coefficient;
    switch (TC) {
        case -1:
            if (quant->is_complementary)
                WRITE("qcy_%d %s qcn_%d-%d",
                    index, quant->operator, index, quantification_parameter);
            else
                WRITE("qcy_%d %s %d",
                    index, quant->operator, quantification_parameter);
            break;
        case 10:
            WRITE("qcy_%d %s qcn_%d", index, quant->operator, index);
            break;
        case 0:
            WRITE("qcy_%d %s 0", index, quant->operator);
            break;
        default:
            WRITE("qcy_%d %s %d*qcn_%d/10", index, quant->operator, TC, index);
            break;
    }
}
```

The function quant_compile_test is called from 6/cdefp.

§**5.** "Now" is the Inform way to assert that a proposition should now be made true. Many quantifiers obstruct this, by introducing too much vagueness. For instance, "now three doors are open" is dangerously vague because it doesn't say which doors are to be made open; similarly "now most of the coins are in the box". On the other hand, "now all the coins are in the box" is fine, because there's no ambiguity. The `can_be_used_in_now` flag for a quantifier shows whether it can be asserted in "now" like this.

```
int quant_is_now_assertable(quantifier *quant) {
    return quant->can_be_used_in_now;
}
```

The function quant_is_now_assertable is called from 6/aprop.

§**6.** Not every proposition can be used in assertion sentences, either, and again it's the quantifiers which cause the trouble. For instance, "Not every room is dark." gives Inform too little to act on. Which room(s) should it make lighted?

```
int quant_can_be_used_in_assertions(quantifier *quant) {
    return quant->can_be_used_in_assertions;
}
```

The function quant_can_be_used_in_assertions is called from 8/refpt.

§**7. Creating a determiner.** Again, at present there's only the built-in set, but we want to keep our options open.

```
determiner *det_new(int not, vocabulary_entry *ve1, vocabulary_entry *ve2,
    int num, quantifier *quant, char *text) {
    determiner *det = CREATE(determiner);
    det->det_word1 = ve1; det->det_word2 = ve2;
    det->takes_number = num;
    det->allows_prefixed_not = not;
    det->quantifier_meant = quant;
    if (quant == NULL) internal_error("created meaningless quantifier");
    det->index_text = text;
    if (text)
        lexicon_new_entry_with_details(
            -1, -1, MISCELLANEOUS_LEXE, ve1, ve2, "determiner", text);
    return det;
}
```

§**8. Parsing the determiner at the head of a noun phrase.** We run through the possible determiners in creation order, choosing the first which matches. The following returns -1 if nothing was found, or else the first word number after the determiner words, and in that case it also writes a pointer to the quantifier meant to `*which_quant` and the parameter value to `*which_P`.

```
int quant_parse_against_text(int w1, int w2, int *which_P, quantifier **which_quant) {
    determiner *det;
    int not_flag = FALSE;
    if [[w1, w2 == not ... --> w1, w2]] not_flag = TRUE;
    *which_P = -1; *which_quant = NULL;
    LOOP_OVER(det, determiner) {
        int x1;
        if ((not_flag) && (det->allows_prefixed_not == FALSE)) continue;
        x1 = det_parse_against_text(w1, w2, det, which_P);
        if (x1 < 0) continue;
        if (not_flag) *which_quant = det->quantifier_meant->negated_quant;
        else *which_quant = det->quantifier_meant;
        [[x1, w2 == the ... --> x1, w2]];
        return x1;
    }
    return -1;
}
```

The function quant_parse_against_text is called from 5/candd.

§**9.** We attempt to see if the word range begins with (or consists of) text which refers to the given determiner, returning the first word past this text and also (where appropriate) setting the number specified. For instance, for "at least three doors are open" and the `at_least_determiner` we would return the word "doors" and set `which_P` to 3.

Note that "of", if used, must be followed by "the". Thus "three of the doors are open" will work, but "three of doors are open" will not. This reduces misunderstandings when objects have names like "three of clubs", meaning a single playing card.

```
int det_parse_against_text(int x1, int w2, determiner *det, int *which_P) {
    int parameter = -1;
    if (x1 < 0) return -1;
    if (det->det_word1) {
        if (compare_word(x1++, det->det_word1) == FALSE) return -1;
        if (x1 > w2) return -1;
    }
    if (det->det_word2) {
        if (compare_word(x1++, det->det_word2) == FALSE) return -1;
        if (x1 > w2) return -1;
    }
    if (det->takes_number) {
        if ((is_a_cardinal(x1, x1) == FALSE) || (unexpectedly_upper_case(x1))) return -1;
        x1++;
        if (x1 > w2) return -1;
        parameter = last_literal_evaluated;
    }
    if [[x1, w2 == of ...]] {
        if (compare_word(x1+1, the_V)) x1++;
        else return -1;
    }
    *which_P = parameter;
    return x1;
}
```

§**10. The built-in set.** We now construct both the tidy logical world of 16 quantifiers in matched pairs, and also a higgledy-piggledy world of 20 English-language determiners referring to them. There are four broad families which we take in turn.

```
void make_built_in_determiners(void) {
    ⟨Make traditional quantification determiners 11⟩;
    ⟨Make complement comparison determiners 12⟩;
    ⟨Make proportion determiners 13⟩;
    ⟨Make cardinality quantification determiners 14⟩;
}
```

The function make_built_in_determiners is invoked by a command in a `.i6t` template file.

§11. As discussed above, the two traditional quantifiers in logic are "for all" and "there exists", usually written in mathematical notation as ∀ and ∃, but we also need to create their negation quantifiers. So we end up with four: `ForAll`, `NotAll`, `Exists` and `DoesNotExist`.

The for-all quantifier can be used in assertions for a slightly oddball reason: it's how the source text makes assemblies. For instance,

> A nose is part of every person.

The "every" is parsed as a use of `ForAll`. Strictly speaking this sentence should be read as creating a single nose which would be shared by all of the people. But the presence of a `ForAll` quantifier in an assertion causes the A-parser to interpret the sentence differently, and to create a fresh nose for each person. (There are some restrictions on the use of `ForAll` in this way, but they are enforced in the A-parser: our part here is simply to authorise `ForAll` in assertions.)

Something which English allows, but Inform does not, is the use of "all" in a way which also specifies a cardinality. For instance, the following condition:

> if all six doors are open, ...

is an attempt to use a determiner which Inform does not possess – "all" plus number. We don't allow this because if there happen to be eight doors, say, the condition would be meaningless.

It's an example of the irregularity of English that you can say "not every door is open" but would never say "not each door is open". In all other respects "each" and "every" are synonymous in the S-parser.

⟨Make traditional quantification determiners 11⟩ ≡

```
for_all_quantifier     = quant_new("==", 10, FALSE, "ForAll");
not_for_all_quantifier = quant_new("<", 10, FALSE, "NotAll");
exists_quantifier      = quant_new(">", 0, FALSE, "Exists");
not_exists_quantifier  = quant_new("==", 0, FALSE, "DoesNotExist");

for_all_quantifier->can_be_used_in_now = TRUE;
for_all_quantifier->can_be_used_in_assertions = TRUE;
not_exists_quantifier->can_be_used_in_now = TRUE;

quants_negate_each_other(for_all_quantifier, not_for_all_quantifier);
quants_negate_each_other(exists_quantifier, not_exists_quantifier);

det_new(TRUE, all_V, NULL, FALSE, for_all_quantifier,
    "used in conditions: 'if all of the doors are open'");
det_new(FALSE, each_V, NULL, FALSE, for_all_quantifier,
    "- see </i>all<i>");
det_new(TRUE, every_V, NULL, FALSE, for_all_quantifier,
    "- see </i>all<i>, and can also be used in generalisations such as "
    "'A nose is part of every person.'");
det_new(FALSE, no_V, NULL, FALSE, not_exists_quantifier,
    "opposite of 'all': 'if no door is open...'");
det_new(FALSE, none_V, NULL, FALSE, not_exists_quantifier,
    "opposite of 'all of': 'if none of the doors is open...'");
det_new(FALSE, some_V, NULL, FALSE, exists_quantifier, NULL);
```

This code is used in §10.

§**12.** Here $T = -1$, because we are counting actual numbers of matches rather than a proportion of matches. But these quantifiers count downwards from the total: thus "all but six" means there have to be exactly $S - 6$ matching items, where S is the total available. The only logical negation for this quantifier would be "other than $S - 6$", which is too unnatural a construction to have any natural English paraphrase, so we do not make a `determiner *` structure pointing to it. But we create it in order that the built-in quantifiers all occur in negation pairs.

⟨Make complement comparison determiners 12⟩ ≡

```
all_but_quantifier     = quant_new("==", -1, TRUE, "AllBut%d");
not_all_but_quantifier = quant_new("~=", -1, TRUE, "NotAllBut%d");

quants_negate_each_other(all_but_quantifier, not_all_but_quantifier);

det_new(FALSE, all_V, but_V, TRUE, all_but_quantifier,
    "used to count things: 'all but three containers'");
det_new(FALSE, all_V, except_V, TRUE, all_but_quantifier,
    "- see </i>all except<i>");
```

This code is used in §10.

§**13.** Here the T-coefficient, measuring the proportion needed, has $0 < T < 10$.

We don't support the determiner "half", as in, "if half the doors are open", because it's ambiguous as to whether it means exactly half or half-or-more.

⟨Make proportion determiners 13⟩ ≡

```
almost_all_quantifier  = quant_new(">=", 8, FALSE, "Proportion>=80%%");
almost_no_quantifier   = quant_new("<",  2, FALSE, "Proportion<20%%");
most_quantifier        = quant_new(">",  5, FALSE, "Proportion>50%%");
under_half_quantifier  = quant_new("<=", 5, FALSE, "Proportion<=50%%");

quants_negate_each_other(almost_all_quantifier, almost_no_quantifier);
quants_negate_each_other(most_quantifier, under_half_quantifier);

det_new(FALSE, almost_V, all_V, FALSE, almost_all_quantifier,
    "used in conditions: true if 80 percent or more possibilities work");
det_new(FALSE, almost_V, no_V, FALSE, almost_no_quantifier,
    "used in conditions: true if fewer than 20 percent of possibilities work");
det_new(FALSE, most_V, NULL, FALSE, most_quantifier,
    "used in conditions: true if a simple majority of possibilities work");
det_new(FALSE, under_V, half_V, FALSE, under_half_quantifier,
    "used in conditions: true if fewer than half of possibilities work");
```

This code is used in §10.

§**14.** The usefulness of cardinality quantifiers in logic as applied to linguistics seems to be an observation due to Barwise and Cooper. They are a natural generalisation of the for-all and there-exists quantifiers, and again come in matched pairs.

The bare number determiner, as in "six doors are open", is perhaps a little ambiguous in English. We read it as "at least six doors are open", in distinction to "exactly six doors are open". This is why the at-least quantifier is allowed in assertions: the assertion sentence "Four coins are in the strongbox." is read as containing the `Card>=4` quantifier, not `Card=4` one. The advantage of this is that two assertions in a row, such as

> Four coins are in the strongbox. Two coins are in the strongbox.

can combine to put six coins in the strongbox, rather than having to be read as contradictory. (It may look improbable that anyone would ever write that, but of course the two assertions need not be adjacent in the source text. One might be in an extension, for instance.)

⟨Make cardinality quantification determiners 14⟩ ≡

```
    at_least_quantifier    = quant_new(">=", -1, FALSE, "Card>=%d");
    at_most_quantifier     = quant_new("<=", -1, FALSE, "Card<=%d");
    exactly_quantifier     = quant_new("==", -1, FALSE, "Card=%d");
    less_than_quantifier   = quant_new("<",  -1, FALSE, "Card<%d");
    more_than_quantifier   = quant_new(">",  -1, FALSE, "Card>%d");
    other_than_quantifier  = quant_new("~=", -1, FALSE, "Card~=%d");

    at_least_quantifier->can_be_used_in_assertions = TRUE;

    quants_negate_each_other(at_least_quantifier, less_than_quantifier);
    quants_negate_each_other(at_most_quantifier, more_than_quantifier);
    quants_negate_each_other(exactly_quantifier, other_than_quantifier);

    det_new(FALSE, at_V, least_V, TRUE, at_least_quantifier,
        "used to count things: 'at least five doors'");
    det_new(FALSE, at_V, most_V, TRUE, at_most_quantifier,
        "- see </i>at least<i>");
    det_new(FALSE, exactly_V, NULL, TRUE, exactly_quantifier,
        "whereas 'if two doors are open' implicitly means 'if at least two "
        "doors are open', 'if exactly two...' makes the count precise");
    det_new(TRUE, fewer_V, than_V, TRUE, less_than_quantifier,
        "pedantic way to say </i>less than<i> when counting");
    det_new(TRUE, less_V, than_V, TRUE, less_than_quantifier,
        "- see </i>more than<i>");
    det_new(TRUE, more_V, than_V, TRUE, more_than_quantifier,
        "used to count things: 'more than three rooms'");
    det_new(FALSE, other_V, than_V, TRUE, other_than_quantifier, NULL);

    det_new(FALSE, NULL, NULL, TRUE, at_least_quantifier, NULL);
```

This code is used in §10.

§**15.** The following question is relevant when simplifying propositions:

```
int quant_requires_at_least_one_true_case(quantifier *quant, int parameter) {
    if (quant == exists_quantifier) return TRUE;
    if (((quant == exactly_quantifier) || (quant == at_least_quantifier)) &&
        (parameter > 0)) return TRUE;
    if ((quant == more_than_quantifier) && (parameter >= 0)) return TRUE;
    if ((quant == other_than_quantifier) && (parameter == 0)) return TRUE;
    return FALSE;
}
```

Binary Predicates 5/bp

Purpose

To create and manage binary predicates, which are the underlying data structures beneath Inform's relations.

Template interpreter commands

```
24  {-callv:make_built_in_relations}
25  {-callv:make_further_built_in_relations}
```

Definitions

¶1. A "binary predicate" (the term comes from logic) is a property B such that for any combination x and y, and at any given moment at run-time, $B(x, y)$ is either true or false.

Examples used in Inform include equality, where $EQ(x, y)$ is true if and only if $x = y$, and containment, where $C(x, y)$ is true if and only if the thing x is inside the room or container y. (EQ does not change during play, but C does.) A fairly large set of binary predicates is built into Inform, and the user is allowed to create more with sentences like

> Lock-fitting relates one thing (called the matching key) to various things.

In the Inform documentation, binary predicates are called "relations". The code to parse "relates" sentences and construct the binary predicate implied can be found in the next section, "Relations.w".

Binary predicates are of central importance because they allow complex sentences to be written which talk about more than one thing at a time, with some connection between them. In excerpts like "an animal inside something" or "a man who wears the top hat", the meanings of the two connecting pieces of text – "inside" and "who wears" – are (pointers to) binary predicates: the containment relation and the wearing relation.

Inform is rich in ways to create relations, and consequently the BPs are many and varied. They turn up in one-off examples but also in whole families. Still, despite the variation, what they share in common is greater yet, and so a single `binary_predicate` structure is used to represent them all.

¶2. The values x and y to which a binary predicate B can apply are called its "terms". For some relations, the source text gives these names:

> Lock-fitting relates one thing (called the matching key) to various things.

Here the x term has the name "matching key", whereas the y term is anonymous. (More often, both are anonymous.) Internally the terms are not named but are numbered 0 and 1: so we should really write $B(x_0, x_1)$ rather than $B(x, y)$.

¶3. Different BPs apply to different sorts of terms: for instance, the numerical less-than comparison applies to numbers, whereas containment applies to things. The two terms need not have the same domain: the "wearing" relation, as seen in

```
Harry Smythe wears the tweed waistcoat.
```

is a binary predicate $W(x_0, x_1)$ such that x_0 ranges across people and x_1 ranges across things.

Inform represents this by allowing each BP to have *either* a designated kind of object, *or* a designated kind of value, *or* no restriction at all, for each term. (In practice, even the unrestricted terms have limitations, but which are enforced by special code in the type-checker to handle special predicates such as equality. For instance, $EQ(x, y)$ can be tested for any values x and y of the same kind, so the two terms in effect constrain each other.)

In the S-parser, type-checking is used to make sure the source text doesn't try to test or assert $B(x, y)$ for any x or y which don't fit, so that

```
if 1 wears "Hello there", ...
```

will be rejected. Whereas in the A-parser, these restrictions are used to infer information about otherwise unknown quantities: so writing

```
Harry Smythe wears the tweed waistcoat.
```

causes the A-parser to force the Harry Smythe object to be of kind "person", and the tweed waistcoat of kind "thing".

¶4. Some BPs are such that $B(x, y)$ can be true for more or less any combination of x and y. Those can take a lot of storage and it is difficult to perform any reasoning about them, because knowing that $B(x, y)$ is true doesn't give you any information about $B(x, z)$. For instance, the BP created by

```
Suspicion relates various people to various people.
```

is stored at run-time in a bitmap of P^2 bits, where P is the number of people, and searching it ("if anyone suspects Harry") requires exhaustive loops, which incur some speed overhead as well.

But other BPs have special properties restricting the circumstances in which they are true, and in those cases we want to capitalise on that. "Contains" is an example of this. A single thing y can be (directly) inside only one other thing x at a time, so that if we know $C(x, y)$ and $C(w, y)$ then we can deduce that $x = w$. We write this common value as $f_0(y)$, the only possible value for term 0 given that term 1 is y. Another way to say this is that the only possible pairs making C true have the form $C(f_0(y), y)$.

And similarly for term 1. If we write T for the "on top of" relation then it turns out that there is a function f_1 such that the only cases where T is true have the form $T(x, f_1(x))$. Here $f_1(x)$ is the thing which directly supports x.

Containment has an f_0 but not an f_1 function; "on top of" has an f_1 but not an f_0. Many BPs (like "suspicion" above) have neither.

Note that if B does have an f_0 function then its reversal R has an identical f_1 function, and vice versa.

¶5. We never in fact need to calculate the value of $f_0(y)$ from y during compilation – only at run-time. So we store the function $f_0(y)$ as what is called an "I6 schema", basically a piece of I6 source code with a place-holder where y is to be inserted. In the case of containment, the schema is written

$$f_0(*1) = \texttt{ContainerOf}(*1)$$

and what this means is that we can calculate $f_0(y)$ from an object y at run-time by calling the `ContainerOf` function, which tells us what container (if any) is at present directly containing y.

¶6. To sum up, each term of a BP can specify: a name, a kind of object or a kind of value, and an f_i function. Every one of these details is optional. They are gathered together in a sub-structure called `bp_term_details`.

The `kind_when_ready` field is needed temporarily because of a timing constraint, the details of which are gone into below.

```
typedef struct bp_term_details {
    int word_ref1, word_ref2;                    "(called...)" name, if any exists
    struct world_object *implies_kind;           terms necessarily of a given kind?
    int kind_when_ready;                         temp. version of implies_kind used before kind creation
    struct kind_of_value *implies_kov;           terms necessarily of a given data type?
    struct i6_schema *function_of_other;         the function f_0 or f_1 as above
} bp_term_details;
```

The structure bp_term_details is shared with 9/vpbp and 10/libp.

¶7. Given any binary predicate B, we may wish to do some or all of the following at run-time:

(a) Test whether or not $B(x, y)$ is true at run-time. Here Inform needs to compile an I6 condition.

(b) Assert that $B(x, y)$ is true in the assertion sentences of the source text. Inform will need to remember all pairs x, y for which B has been asserted so that it can compile this information as the original state of the I6 data structure containing the current state of B.

(c) Set $B(x, y)$ true, or false, at run-time. Here Inform needs to compile I6 code which will modify that data structure.

Some BPs provide an I6 schema to achieve (a), others provide (a) and (b), while a happy few provide all of (a), (b), (c).

The variety of BPs is such that different BPs use very different run-time mechanisms. Some relations compile elaborate routines to test (a), some look at parents or chidren in the I6 object tree, some look at I6 property values, others look inside bitmaps. The actual work is often done by routines in the I6 template, which are called by code generated by the I6 schema for (a); and similarly for (b) and (c).

¶8. Each BP has a partner which we call its "reversal". If B is the original and R is its reversal, then $B(x, y)$ is true if and only if $R(y, x)$ is true. Reversals sometimes occur quite naturally in English language. "To wear" is the reversal of "to be worn by". "Contains" is the reversal of being "inside". (Though not every BP has an interesting reversal. The reversal of "is" – equality – looks much the same as the original, because $x = y$ if and only if $y = x$.)

The following sentences express the same fact:

> The ball is inside the trophy case.
>
> The trophy case contains the ball.

...but when we parse them into their meanings, we could easily lose sight that they are saying the same thing, because they involve different BPs:

> `inside(ball, trophy case)` and `contains(trophy case, ball)`

It's usually a bad idea for any computer program to represent the same conceptual idea in more than one way. So for every pair of BPs X and Y which are each other's reversal, Inform designates one as being "the right way round" and the other as being "the wrong way round". Whenever a sentence's meaning involves a BP which is "the wrong way round", Inform swaps over the terms and replaces the BP by its reversal, which is "the right way round". That makes it much easier to recognise when pairs of sentences like the one above are duplicating each other's meanings.

This is purely an internal implementation trick. There's no natural sense in language or mathematics in which "contains" is the right way round and "inside" the wrong way round.

¶9. We can finally now declare the epic BP structure.

```
typedef struct binary_predicate {
    int relation_family;                                    one of the *_KBP constants defined below
    int form_of_relation;                                   one of the Relation_* constants defined below
    struct vocabulary_entry *relation_name;                 (if any): must be one single word
    struct parse_node *bp_created_at;                       where declared in the source text
    char debugging_log_name[MAX_WORD_LENGTH+10];            used when printing propositions to the debug log
    struct bp_term_details term_details[2];                 term 0 is the left term, 1 is the right
    struct binary_predicate *reversal;                      the R such that R(x,y) iff B(y,x)
    int right_way_round;                                    was this BP created directly? or is it a reversal of another?
    how to compile code which tests or forces this BP to be true or false:
    struct i6_schema *test_function;                        I6 schema for (a) testing B(x,y)...
    int condition_defn_w1, condition_defn_w2;                ...unless this I7 condition is used instead
    struct i6_schema *make_true_function;                   I6 schema for (b) "now B(x,y)"
    struct i6_schema *make_false_function;                  I6 schema for (c) "now not(B(x,y))"
    for use in the A-parser:
    int arbitrary;                                          allow source to assert B(x,y) for any arbitrary pairs x,y
    struct property_name *set_property;                     asserting B(x,v) sets this prop. of x to v
    int property_pending_w1, property_pending_w2;           temp. version used until props created
    int set_inference;                                      unless -1, asserting B in the A-parser makes an inference of this type
    int relates_values_not_objects;                         true if either term is necessarily a value...
    struct inference *knowledge_about_bp;                   ...and if so, here's the list of known assertions
    for optimisation of run-time code:
    struct property_name *i6_storage_property;              provides run-time storage
    int allow_function_simplification;                      allow Inform to make use of any f_i functions?
    int fast_route_finding;                                 use fast rather than slow route-finding algorithm?
    char *loop_parent_optimisation_proviso;                 if not NULL, optimise loops using object tree
    char *loop_parent_optimisation_ranger;                  if not NULL, routine iterating through contents
    details, filled in for right-way-round BPs only, for particular kinds of BP:
    int a_listed_in_predicate;                              (if right way) was this generated from a table column?
    struct property_name *same_property;                    (if right way) if a "same property as..."
    struct property_name *comparative_property;             (if right way) if a comparative adjective
    int comparison_sign;                                    ...and +1 or -1 according to sign of definition
    int *equivalence_partition;                             (if right way) partition array of equivalence classes
    struct world_object *direction_object;                  (if right way) if spatial between rooms
    MEMORY_MANAGEMENT
} binary_predicate;
```

The structure binary_predicate is shared with 5/rel, 9/spabp, 9/mapbp, 9/cmpbp, 9/vpbp and 10/libp.

¶10. This seems a good point to lay out a classification of all of the BPs existing within Inform. Broadly, they divide into two: the ones explicitly created by the source text, in sentences like

Admiration relates various people to various people.

These are called "explicit". The others are "implicit" and are either created automatically soon after Inform starts up, or else are created as a consequence of something else being created by the source text, such as

Definition: A woman is tall if her height is 68 or more.

which implicitly creates a "taller than" relation. All explicit BPs are constructed in the "Relations.w" section of the source code.

define `EQUALITY_KBP 1` *there is exactly one of these: the $x = y$ predicate*
define `QUASINUMERIC_KBP 2` *the inequality comparison $\leq$, $<$ and so on*
define `SPATIAL_KBP 3` *a relation associated with a map connection*
define `MAP_CONNECTING_KBP 4` *a relation associated with a map connection*
define `PROPERTY_SETTING_KBP 5` *a relation associated with a value property*
define `PROPERTY_SAME_KBP 6` *another relation associated with a value property*
define `PROPERTY_COMPARISON_KBP 7` *another relation associated with a value property*
define `LISTED_IN_KBP 8` *a relation for indirect table lookups, one for each column name*
define `PROVISION_KBP 9` *a relation for specifying which objects provide which properties*
define `EXPLICIT_KBP 100` *defined explicitly in the source text; the others are all implicit*

¶11. The following constants are used to identify the "form" of a BP (in that the `form_of_relation` field of any BP always equals one of these and never changes). These constant names (and values) exactly match a set of constants compiled into every I6 program created by Inform, so they can be used freely both in the Inform source code and also in the I6 template layer.

define `Relation_Implicit -1` *all implicit BPs have this form, and all others are explicit*
define `Relation_OtoO 1` *one to one: "R relates one K to one K"*
define `Relation_OtoV 2` *one to various: "R relates one K to various K"*
define `Relation_VtoO 3` *various to one: "R relates various K to one K"*
define `Relation_VtoV 4` *various to various: "R relates various K to various K"*
define `Relation_Sym_OtoO 5` *symmetric one to one: "R relates one K to another"*
define `Relation_Sym_VtoV 6` *symmetric various to various: "R relates K to each other"*
define `Relation_Equiv 7` *equivalence relation: "R relates K to each other in groups"*
define `Relation_ByRoutine 8` *relation tested by a routine: "R relates K to L when (some condition)"*

¶12. That completes the catalogue of the one-off cases, and we can move on to the five families of implicit relations which correspond to other structures in the source text.

¶13. The second family of implicit relations corresponds to any property which has been given as the meaning of a verb, as in the example

The verb to weigh (it weighs, they weigh, it is weighing) implies the weight property.

This implicitly constructs a relation $W(p, w)$ where p is a thing and w a weight.

¶14. The third family corresponds to defined adjectives which perform a numerical comparison in a particular way, as here:

Definition: A woman is tall if her height is 68 or more.

This implicitly constructs a relation $T(x, y)$ which is true if and only if woman x is taller than woman y.

¶15. The fourth family corresponds to value properties, so that

A door has a number called street number.

implicitly constructs a relation $SN(d_1, d_2)$ which is true if and only if doors d_1 and d_2 have the same street number.

¶16. The fifth family corresponds to names of table columns. If any table includes a column headed "eggs per clutch" then that will implicitly construct a relation $LEPC(n, T)$ which is true if and only if the number n is listed as one of the eggs-per-clutch entries in the table T, where T has to be one of the tables which has a column of this name.

---

§**1. Creating term details.** Five, count them, five ways to set up the details of a term in a BP. The first is `bptd_blank()`, which gives Inform no information at all about what a term signifies.

```
bp_term_details bptd_blank(void) {
    bp_term_details bptd;
    bptd.word_ref1 = -1; bptd.word_ref2 = -1;
    bptd.kind_when_ready = -1;
    bptd.implies_kind = NULL; bptd.implies_kov = NULL;
    bptd.function_of_other = NULL;
    return bptd;
}
```

The function bptd_blank is called from 9/spabp, 9/mapbp, 9/provr, 9/cmpbp, 9/vpbp and 10/libp.

§**2.** `bptd_new()` is for general use once Inform's initialisation is finished:

```
bp_term_details bptd_new(world_object *k, kind_of_value *kov) {
    bp_term_details bptd = bptd_blank();
    if ((k == NULL) && (kov == NULL)) internal_error("BP term without domain");
    if ((k != NULL) && (kov != NULL)) internal_error("BP term with ambiguous domain");
    bptd.implies_kind = k; bptd.implies_kov = kov;
    return bptd;
}
```

The function bptd_new is called from 10/qnbp and 10/libp.

§**3.** And there is also a fuller version:

```
bp_term_details bptd_full_new(world_object *k, kind_of_value *kov, int c1, int c2, i6_schema *f) {
    bp_term_details bptd = bptd_new(k, kov);
    bptd.word_ref1 = c1; bptd.word_ref2 = c2;
    bptd.function_of_other = f;
    return bptd;
}
```

The function bptd_full_new is called from 5/rel.

§**4.** `bptd_early_new()` is for use early on in Inform's run. We potentially have a vicious circle, because
(a) we need fundamental kinds like "thing" and "room" in order to specify the domains for built-in relations like "containment",
(b) we need built-in relations like "containment" in order to specify the meanings of prepositional usages like "to be inside",
(c) we need to have defined usages like "to be inside" in order to break up sentences in the parse tree and identify their primary verbs and noun phrases, but
(d) we need a parse tree already in place before we can act on sentences like "A room is a kind of object." which create the fundamental kinds.

We break the deadlock at (a) by specifying the domains of the built-in relations using the following constants written into the `kind_when_ready` field. These amount to promisory notes that we will copy in pointers to the actual kinds later on, at some point after stage (d) when they exist.

```
define kind_thing_WR 1
define kind_supporter_WR 2
define kind_person_WR 3
define kind_room_WR 4
```

```
bp_term_details bptd_early_new(int kwr, i6_schema *fn) {
    bp_term_details bptd = bptd_blank();
    bptd.kind_when_ready = kwr;
    bptd.function_of_other = fn;
    return bptd;
}
```

The function bptd_early_new is called from 9/spabp.

§**5.** Lastly, a catch-all way to define a "this must be a thing" term: it works whether the kind "thing" exists yet, or not.

```
bp_term_details bptd_thing(void) {
    bp_term_details bptd = bptd_blank();
    bptd.kind_when_ready = kind_thing_WR;
    bptd.implies_kind = kind_thing;                    which might still be equal to NULL at this point
    return bptd;
}
```

The function bptd_thing is called from 9/vpbp.

§**6.** And here is where we redeem those "when ready" IOU notes. This catches everything, because the only `bp_term_details` structures in existence (well, other than as local variables in routines in this section) are the ones inside `binary_predicate` structures.

```
void write_kind_references_into_BPs(void) {
    binary_predicate *bp;
    LOOP_OVER(bp, binary_predicate) {
        int i;
        for (i=0; i<2; i++) {
            switch(bp->term_details[i].kind_when_ready) {
                case kind_thing_WR:     bp->term_details[i].implies_kind = kind_thing; break;
                case kind_supporter_WR: bp->term_details[i].implies_kind = kind_supporter; break;
                case kind_person_WR:    bp->term_details[i].implies_kind = kind_person; break;
                case kind_room_WR:      bp->term_details[i].implies_kind = kind_room; break;
            }
        }
    }
}
```

The function write_kind_references_into_BPs is called from 9/wo.

§**7.** The table of relations in the index needs this. All those IOUs will be redeemed long since.

```
void bp_term_details_index(bp_term_details *bptd) {
    if (bptd->implies_kind) {
        print_text_to_file(bptd->implies_kind->word_ref1,
            bptd->implies_kind->word_ref2, ifl);
    } else if (bptd->implies_kov) {
        char term_name_buffer[500];
        copy_kov_to_string(bptd->implies_kov, term_name_buffer, FALSE);
        INDEX("%s", term_name_buffer);
    } else INDEX("--");
}
```

The function bp_term_details_index is called from 5/rel.

§**8.** The following routine adds the given BP term as a call parameter to the routine currently being compiled, deciding that something is an object if its kind indications are all blank, but verifying that the value supplied matches the specific necessary kind of object if there is one.

```
void bptd_add_as_call_parameter(OUTPUT_STREAM, ph_stack_frame *phsf, bp_term_details bptd) {
    int nv;
    kind_of_value *kov = kova(OBJECT_TY);
    if (bptd.implies_kov) kov = bptd.implies_kov;
    nv = phsf_add_call_parameter(phsf, bptd.word_ref1, bptd.word_ref2, kov);
    if (bptd.implies_kind)
        WRITE("  if (~~(t_%d ofclass %s)) rfalse;\n",
            nv, wo_get_I6_representation(bptd.implies_kind));
}
```

The function bptd_add_as_call_parameter is called from 12/phsf.

§**9. Making the equality relation.** As we shall see below, BPs are almost always created in matched pairs. There is one and only one exception to this rule: the equality predicate where $EQ(x, y)$ if $x = y$. Equality plays a special role in the system of logic we'll be using in Chapter 6. Since $x = y$ and $y = x$ are exactly equivalent, it is safe to make EQ its own reversal; this makes it impossible for equality to occur "the wrong way round" in any proposition, even one which is not yet fully simplified.

There is no fixed domain to which x and y belong: equality can be used whenever x and y belong to the same domain. Thus "if the score is 12" and "if the location is the Pantheon" are both valid uses of EQ, where x and y are numbers in the former case and rooms in the latter. It will take special handling in the type-checker (Chapter 7) to achieve this effect. For now, we give EQ entirely blank term details.

```
binary_predicate *make_equality(void) {
    binary_predicate *bp = make_single_BP(EQUALITY_KBP,
        bptd_blank(), bptd_blank(),
        "is", 0, NULL, NULL, NULL, equality_V);
    bp->reversal = bp; bp->right_way_round = TRUE;
    return bp;
}
```

The function make_equality is called from 6/equal.

§**10. Making a pair of relations.** Every other BP belongs to a matched pair, in which each is the reversal of the other, but only one is designated as being "the right way round". The left-hand term of one behaves like the right-hand term of the other, and vice versa.

The BP which is the wrong way round is never used in compilation, because it will long before that have been reversed, so we only fill in details of how to compile the BP for the one which is the right way round.

```
binary_predicate *make_pair_of_BPs(int family,
    bp_term_details left_term, bp_term_details right_term,
    char *name, char *namer, int inference_type, property_name *pn,
    i6_schema *mtf, i6_schema *tf, vocabulary_entry *word) {
    binary_predicate *bp, *bpr;
    char namer_constructed[MAX_WORD_LENGTH+10];
    if (name == NULL) name = "nameless";
    if (namer == NULL) { sprintf(namer_constructed, "%s-r", name); namer = namer_constructed; }
    bp  = make_single_BP(family, left_term, right_term, name, inference_type, pn, mtf, tf, word);
    bpr = make_single_BP(family, right_term, left_term, namer, inference_type, NULL, NULL, NULL, NULL);
    bp->reversal = bpr; bpr->reversal = bp;
    bp->right_way_round = TRUE; bpr->right_way_round = FALSE;
    if (word)
        register_reworded_meaning(MISCELLANEOUS_MC, RELATION_SMC, word, relation_V, -1, -1, 0,
            STORE_POINTER_binary_predicate(bp));
    return bp;
}
```

The function make_pair_of_BPs is called from 9/spabp, 9/mapbp, 9/provr, 9/cmpbp, 9/vpbp, 10/qnbp and 10/libp.

§**11.** When the source text declares new relations, it turns out to be convenient to make their BPs in a two-stage process: to make sketchy, mostly-blank BP structures for them early on – but getting their names registered – and then fill in the correct details later. This is where such sketchy pairs are made:

```
void make_sketchy_pair_of_BPs(vocabulary_entry *ve, int f) {
    char *name = vocab_get_exemplar(ve, FALSE);
    binary_predicate *bp =
        make_pair_of_BPs(EXPLICIT_KBP,
        bptd_blank(), bptd_blank(), name, NULL, -1, NULL, NULL, NULL, ve);
    bp->form_of_relation = f;
    bp->reversal->form_of_relation = f;
}
```

The function make_sketchy_pair_of_BPs is called from 5/rel.

§**12. BP construction.** The following routine should only ever be called from the two above: provided we stick to that, we ensure the golden rule that *every BP has a reversal and a BP equals its reversal if and only if it is the equality relation.*

It looks a little asymmetric that the "make true function" schema `mtf` is an argument here, but the "make false function" isn't. That's because it happens that the implicit relations defined in this section of code generally do support making-true, but don't support making-false, so that such an argument would always be `NULL` in practice.

```
binary_predicate *make_single_BP(int family,
    bp_term_details left_term, bp_term_details right_term,
    char *name, int inference_type, property_name *pn,
    i6_schema *mtf, i6_schema *tf, vocabulary_entry *word) {
    binary_predicate *bp = CREATE(binary_predicate);
    bp->relation_family = family;
    bp->form_of_relation = Relation_Implicit;
    bp->relation_name = word;
    bp->bp_created_at = current_sentence;
    truncated_strcpy(bp->debugging_log_name, name, MAX_WORD_LENGTH+10);
    bp->term_details[0] = left_term; bp->term_details[1] = right_term;
    the reversal and the right_way_round field must be set by the caller
    for use in code compilation
    bp->test_function = tf;
    bp->condition_defn_w1 = -1; bp->condition_defn_w2 = -1;
    bp->make_true_function = mtf;
    bp->make_false_function = NULL;
    for use by the A-parser
    bp->arbitrary = FALSE;
    bp->set_property = NULL;
    bp->property_pending_w1 = -1; bp->property_pending_w2 = -1;
    bp->set_inference = inference_type;
    bp->relates_values_not_objects = FALSE;
    bp->knowledge_about_bp = NULL;
    for optimisation of run-time code
    bp->i6_storage_property = pn;
    bp->allow_function_simplification = TRUE;
    bp->fast_route_finding = FALSE;
    bp->loop_parent_optimisation_proviso = NULL;
```

```
    bp->loop_parent_optimisation_ranger = NULL;
    details for particular kinds of relation
    bp->a_listed_in_predicate = FALSE;
    bp->same_property = NULL;
    bp->comparative_property = NULL;
    bp->comparison_sign = 0;
    bp->equivalence_partition = NULL;
    bp->direction_object = NULL;
    return bp;
}
```

§13. BP and term logging.

```
void log_bp_term_details(bp_term_details *bptd, int i) {
    LOG("  function(%d): $i\n", i, bptd->function_of_other);
    if (bptd->word_ref1 >= 0) LOG("  term %d is '$W'\n", i, bptd->word_ref1, bptd->word_ref2);
    if (bptd->implies_kind) LOG("  term %d implies kind $O\n", i, bptd->implies_kind);
    if (bptd->implies_kov) LOG("  term %d implies KOV $u\n", i, bptd->implies_kov);
}
void log_binary_predicate(binary_predicate *bp) {
    int i;
    if (bp == NULL) { LOG("<null-BP>\n"); return; }
    LOG("BP%d <%s> - %s way round - %s\n",
        bp->allocation_id, bp->debugging_log_name, bp->right_way_round?"right":"wrong",
        bp_form_to_text(bp));
    for (i=0; i<2; i++) log_bp_term_details(&bp->term_details[i], i);
    LOG("  test: $i\n", bp->test_function);
    LOG("  make true: $i\n", bp->make_true_function);
    LOG("  make false: $i\n", bp->make_false_function);
    LOG("  storage property: $Y\n", bp->i6_storage_property);
}
```

The function log_binary_predicate is called from 2/dl.

§14. Relation names. A useful little routine to spot the names of relations. This is only used when there is good reason to suspect that the word in question is the name of a relation, so its relative inefficiency does not matter.

```
binary_predicate *parse_relation_name(int w1) {
    binary_predicate *bp;
    LOOP_OVER(bp, binary_predicate)
        if (compare_word(w1, bp->relation_name))
            return bp;
    return NULL;
}
char *bp_get_log_name(binary_predicate *bp) {
    return bp->debugging_log_name;
}
```

The function parse_relation_name is called from 5/rel, 5/conj and 13/gtok.
The function bp_get_log_name is called from 5/rel, 5/conj, 6/aprop and 8/refpt.

§15. Miscellaneous access routines.

```
int bp_get_form_of_relation(binary_predicate *bp) {
    return bp->form_of_relation;
}
int bp_is_explicit_with_runtime_storage(binary_predicate *bp) {
    if (bp->right_way_round == FALSE) bp = bp->reversal;
    if (bp->form_of_relation == Relation_Implicit) return FALSE;
    if (bp->form_of_relation == Relation_ByRoutine) return FALSE;
    return TRUE;
}
char *bp_form_to_text(binary_predicate *bp) {
    switch(bp->form_of_relation) {
        case Relation_Implicit: return "Relation_Implicit";
        case Relation_OtoO: return "Relation_OtoO";
        case Relation_OtoV: return "Relation_OtoV";
        case Relation_VtoO: return "Relation_VtoO";
        case Relation_VtoV: return "Relation_VtoV";
        case Relation_Sym_OtoO: return "Relation_Sym_OtoO";
        case Relation_Sym_VtoV: return "Relation_Sym_VtoV";
        case Relation_Equiv: return "Relation_Equiv";
        case Relation_ByRoutine: return "Relation_ByRoutine";
        default: return "formless-BP";
    }
}
```

The function bp_get_form_of_relation is called from 9/exrel and 13/gtok.
The function bp_is_explicit_with_runtime_storage is called from 9/exrel.
The function bp_form_to_text is called from 5/rel.

§16. Details of the terms:

```
world_object *bp_term_kind(binary_predicate *bp, int t) {
    return bp->term_details[t].implies_kind;
}
kind_of_value *bp_term_kind_of_value(binary_predicate *bp, int t) {
    return bp->term_details[t].implies_kov;
}
i6_schema *bp_get_term_as_function_of_other(binary_predicate *bp, int t) {
    return bp->term_details[t].function_of_other;
}
```

The function bp_term_kind is called from 5/rel, 6/simp, 6/tcpr, 6/asp, 8/creat and 13/gtok.
The function bp_term_kind_of_value is called from 5/rel, 6/sconv, 6/simp, 6/tcpr, 8/creat, 9/exrel and 13/gtok.
The function bp_get_term_as_function_of_other is called from 6/term, 6/simp and 6/cind.

§**17.** Reversing:

```
binary_predicate *bp_get_reversal(binary_predicate *bp) {
    return bp->reversal;
}
int bp_is_the_wrong_way_round(binary_predicate *bp) {
    if (bp->right_way_round == FALSE) return TRUE;
    return FALSE;
}
```

The function bp_get_reversal is called from 4/verb, 4/noun, 5/conj, 6/simp, 8/refpt, 8/relv, 9/prop and 13/gtok.
The function bp_is_the_wrong_way_round is called from 6/simp, 6/tcpr, 8/relv and 13/gtok.

§**18.** For compiling code from conditions:

```
i6_schema *bp_get_test_function(binary_predicate *bp) {
    return bp->test_function;
}
int bp_can_be_made_true_at_runtime(binary_predicate *bp) {
    if ((bp->make_true_function) ||
        (bp->reversal->make_true_function)) return TRUE;
    return FALSE;
}
```

The function bp_get_test_function is called from 13/gtok.
The function bp_can_be_made_true_at_runtime is called from 9/spabp and 9/exrel.

§**19.** For the A-parser. The real code is all elsewhere; note that the `assertions` field, which is used only for relations between values rather than objects, is a linked list. (Information about objects is stored in linked lists pointed to from the `world_object` structure in question; that can't be done if an assertion is about values, so they are stored under the relation itself.)

```
int bp_allow_arbitrary_assertions(binary_predicate *bp) {
    return bp->arbitrary;
}
int bp_get_kind_of_inference(binary_predicate *bp) {
    return bp->set_inference;
}
int bp_relates_values_not_objects(binary_predicate *bp) {
    return bp->relates_values_not_objects;
}
inference **bp_get_inferences(binary_predicate *bp) {
    return &(bp->knowledge_about_bp);
}
```

The function bp_allow_arbitrary_assertions is called from 9/exrel and 9/prop.
The function bp_get_kind_of_inference is called from 9/spabp.
The function bp_relates_values_not_objects is called from 5/rel, 8/mass and 8/relv.
The function bp_get_inferences is called from 9/inf.

§**20.** For use when optimising code.

```
property_name *bp_get_i6_storage_property(binary_predicate *bp) {
    return bp->i6_storage_property;
}
int bp_allows_function_simplification(binary_predicate *bp) {
    return bp->allow_function_simplification;
}
```

The function bp_get_i6_storage_property is called from 9/exrel and 13/gtok.
The function bp_allows_function_simplification is called from 6/simp.

§**21.** The predicate-calculus engine (Chapter 6) compiles much better loops if we can help it by providing an I6 schema of a loop header solving the following problem:

Loop a variable v (in the schema, `*1`) over all possible x such that $R(x, t)$, for some fixed t (in the schema, `*1`).

If we can't do this, it will still manage, but by the brute force method of looping over all x in the left domain of R and testing every possible $R(x, t)$.

```
int bp_write_optimised_loop_schema(i6_schema *sch, binary_predicate *bp) {
    if (bp == NULL) return FALSE;
    ⟨Try loop ranger optimisation 22⟩;
    ⟨Try loop parent optimisation subject to a proviso 23⟩;
    return FALSE;
}
```

The function bp_write_optimised_loop_schema is called from 6/cdefp.

§**22.** Some relations R provide a "ranger" routine, `R`, which is such that `R(t)` supplies the first "child" of t and `R(t, n)` supplies the next "child" after n. Thus `R` iterates through some linked list of all the objects x such that $R(x, t)$.

⟨Try loop ranger optimisation 22⟩ ≡

```
    if (bp->loop_parent_optimisation_ranger) {
        sch_write_to_existing_2s(sch,
            "for (*1=%s(*2): *1: *1=%s(*2,*1))",
            bp->loop_parent_optimisation_ranger,
            bp->loop_parent_optimisation_ranger);
        return TRUE;
    }
```

This code is used in §21.

§**23.** Other relations make use of the I6 object tree, in cases where $R(x, t)$ is true if and only if t is an object which is the parent of x in the I6 object tree *and* some routine associated with R, called its proviso `P`, is such that `P(x) == t`. For example, $worn - by(x, t)$ is true iff t is the parent of x and `WearerOf(x) == t`. The proviso ensures that we don't falsely pick up, say, items carried by t which aren't being worn, or aren't even clothing.

⟨Try loop parent optimisation subject to a proviso 23⟩ ≡

```
if (bp->loop_parent_optimisation_proviso) {
    sch_write_to_existing_1s(sch,
        "objectloop (*1 in *2) if (%s(*1)==parent(*1))",
        bp->loop_parent_optimisation_proviso);
    return TRUE;
}
```

This code is used in §21.

§**24. The built-in BPs.** Here we create spatial relationships, numerical comparisons and a few others: all of the BPs in the "exceptional one-off cases" part of the classification above. This happens very early in compilation.

```
void make_built_in_relations(void) {
    EQUALITY_KBP_create_initial_stock();
    PROVISION_KBP_create_initial_stock();
    QUASINUMERIC_KBP_create_initial_stock();
    SPATIAL_KBP_create_initial_stock();
    MAP_CONNECTING_KBP_create_initial_stock();
    PROPERTY_SETTING_KBP_create_initial_stock();
    PROPERTY_SAME_KBP_create_initial_stock();
    PROPERTY_COMPARISON_KBP_create_initial_stock();
    LISTED_IN_KBP_create_initial_stock();
    EXPLICIT_KBP_create_initial_stock();
}
```

The function make_built_in_relations is invoked by a command in a `.i6t` template file.

§**25. Other property-based relations.**

```
void make_further_built_in_relations(void) {
    EQUALITY_KBP_create_second_stock();
    PROVISION_KBP_create_second_stock();
    QUASINUMERIC_KBP_create_second_stock();
    SPATIAL_KBP_create_second_stock();
    MAP_CONNECTING_KBP_create_second_stock();
    PROPERTY_SETTING_KBP_create_second_stock();
    PROPERTY_SAME_KBP_create_second_stock();
    PROPERTY_COMPARISON_KBP_create_second_stock();
    LISTED_IN_KBP_create_second_stock();
    EXPLICIT_KBP_create_second_stock();
}
```

The function make_further_built_in_relations is invoked by a command in a `.i6t` template file.

§**26.**

define DECLINE_TO_MATCH 1000 *not one of the three legal* *_MATCH *values*
define NEVER_MATCH_SAYING_WHY_NOT 1001 *not one of the three legal* *_MATCH *values*

```
int bp_typecheck(binary_predicate *bp,
        kind_of_value **kovs_of_terms, kind_of_value **kovs_required, tc_problem_kit *tck) {
    int result = DECLINE_TO_MATCH;
    switch (bp->relation_family) {
        case EQUALITY_KBP: result = EQUALITY_KBP_typecheck(bp, kovs_of_terms, kovs_required, tck); break;
        case PROVISION_KBP: result = PROVISION_KBP_typecheck(bp, kovs_of_terms, kovs_required, tck);
break;
        case QUASINUMERIC_KBP: result = QUASINUMERIC_KBP_typecheck(bp, kovs_of_terms, kovs_required,
tck); break;
        case SPATIAL_KBP: result = SPATIAL_KBP_typecheck(bp, kovs_of_terms, kovs_required, tck); break;
        case MAP_CONNECTING_KBP: result = MAP_CONNECTING_KBP_typecheck(bp, kovs_of_terms, kovs_required,
tck); break;
        case PROPERTY_SETTING_KBP: result = PROPERTY_SETTING_KBP_typecheck(bp, kovs_of_terms, kovs_required,
tck); break;
        case PROPERTY_SAME_KBP: result = PROPERTY_SAME_KBP_typecheck(bp, kovs_of_terms, kovs_required,
tck); break;
        case PROPERTY_COMPARISON_KBP: result = PROPERTY_COMPARISON_KBP_typecheck(bp, kovs_of_terms,
kovs_required, tck); break;
        case LISTED_IN_KBP: result = LISTED_IN_KBP_typecheck(bp, kovs_of_terms, kovs_required, tck);
break;
        case EXPLICIT_KBP: result = EXPLICIT_KBP_typecheck(bp, kovs_of_terms, kovs_required, tck); break;
        default: internal_error("typechecked unknown KBP");
    }
    return result;
}

int bp_assert(binary_predicate *bp,
        world_object *wo0, specification *spec0, world_object *wo1, specification *spec1) {
    int success = FALSE;
    switch (bp->relation_family) {
        case EQUALITY_KBP: success = EQUALITY_KBP_assert(bp, wo0, spec0, wo1, spec1); break;
        case PROVISION_KBP: success = PROVISION_KBP_assert(bp, wo0, spec0, wo1, spec1); break;
        case QUASINUMERIC_KBP: success = QUASINUMERIC_KBP_assert(bp, wo0, spec0, wo1, spec1); break;
        case SPATIAL_KBP: success = SPATIAL_KBP_assert(bp, wo0, spec0, wo1, spec1); break;
        case MAP_CONNECTING_KBP: success = MAP_CONNECTING_KBP_assert(bp, wo0, spec0, wo1, spec1); break;
        case PROPERTY_SETTING_KBP: success = PROPERTY_SETTING_KBP_assert(bp, wo0, spec0, wo1, spec1);
break;
        case PROPERTY_SAME_KBP: success = PROPERTY_SAME_KBP_assert(bp, wo0, spec0, wo1, spec1); break;
        case PROPERTY_COMPARISON_KBP: success = PROPERTY_COMPARISON_KBP_assert(bp, wo0, spec0, wo1,
spec1); break;
        case LISTED_IN_KBP: success = LISTED_IN_KBP_assert(bp, wo0, spec0, wo1, spec1); break;
        case EXPLICIT_KBP: success = EXPLICIT_KBP_assert(bp, wo0, spec0, wo1, spec1); break;
        default: internal_error("asserted unknown KBP");
    }
    return success;
}

i6_schema *bp_get_i6_schema(int task, binary_predicate *bp, annotated_i6_schema *asch) {
    int success = FALSE;
    switch (bp->relation_family) {
        case EQUALITY_KBP: success = EQUALITY_KBP_compile(task, bp, asch); break;
```

```
        case PROVISION_KBP: success = PROVISION_KBP_compile(task, bp, asch); break;
        case QUASINUMERIC_KBP: success = QUASINUMERIC_KBP_compile(task, bp, asch); break;
        case SPATIAL_KBP: success = SPATIAL_KBP_compile(task, bp, asch); break;
        case MAP_CONNECTING_KBP: success = MAP_CONNECTING_KBP_compile(task, bp, asch); break;
        case PROPERTY_SETTING_KBP: success = PROPERTY_SETTING_KBP_compile(task, bp, asch); break;
        case PROPERTY_SAME_KBP: success = PROPERTY_SAME_KBP_compile(task, bp, asch); break;
        case PROPERTY_COMPARISON_KBP: success = PROPERTY_COMPARISON_KBP_compile(task, bp, asch); break;
        case LISTED_IN_KBP: success = LISTED_IN_KBP_compile(task, bp, asch); break;
        case EXPLICIT_KBP: success = EXPLICIT_KBP_compile(task, bp, asch); break;
        default: internal_error("compiled unknown KBP");
    }
    if (success == FALSE) {
        switch(task) {
            case TEST_ATOM_TASK: asch->schema = bp->test_function; break;
            case NOW_ATOM_TRUE_TASK: asch->schema = bp->make_true_function; break;
            case NOW_ATOM_FALSE_TASK: asch->schema = bp->make_false_function; break;
        }
    }
    return asch->schema;
}
void bp_describe_for_problems(binary_predicate *bp, char *problem_buffer) {
    int success = FALSE;
    switch (bp->relation_family) {
        case EQUALITY_KBP: success = EQUALITY_KBP_describe_for_problems(bp, problem_buffer); break;
        case PROVISION_KBP: success = PROVISION_KBP_describe_for_problems(bp, problem_buffer); break;
        case QUASINUMERIC_KBP: success = QUASINUMERIC_KBP_describe_for_problems(bp, problem_buffer);
break;
        case SPATIAL_KBP: success = SPATIAL_KBP_describe_for_problems(bp, problem_buffer); break;
        case MAP_CONNECTING_KBP: success = MAP_CONNECTING_KBP_describe_for_problems(bp, problem_buffer);
break;
        case PROPERTY_SETTING_KBP: success = PROPERTY_SETTING_KBP_describe_for_problems(bp, problem_buffer);
break;
        case PROPERTY_SAME_KBP: success = PROPERTY_SAME_KBP_describe_for_problems(bp, problem_buffer);
break;
        case PROPERTY_COMPARISON_KBP: success = PROPERTY_COMPARISON_KBP_describe_for_problems(bp, problem_buffer);
break;
        case LISTED_IN_KBP: success = LISTED_IN_KBP_describe_for_problems(bp, problem_buffer); break;
        case EXPLICIT_KBP: success = EXPLICIT_KBP_describe_for_problems(bp, problem_buffer); break;
        default: internal_error("found unknown KBP");
    }
    if (success == NOT_APPLICABLE) return;
    if (success == FALSE) {
        if (bp->relation_name) {
            strcat(problem_buffer, "the ");
            sprintf(problem_buffer+strlen(problem_buffer), "%s",
                vocab_get_exemplar(bp->relation_name, FALSE));
        } else strcat(problem_buffer, "a");
        strcat(problem_buffer, " relation");
    }
    strcat(problem_buffer, " (between ");
    if (bp_term_kind(bp, 0)) strcat(problem_buffer, "an object");
    else if (bp_term_kind_of_value(bp, 0))
        copy_kov_to_string(bp_term_kind_of_value(bp, 0),
```

```
            problem_buffer+strlen(problem_buffer), FALSE);
    else strcat(problem_buffer, "an object");
    strcat(problem_buffer, " and ");
    if (bp_term_kind(bp, 1)) strcat(problem_buffer, "an object");
    else if (bp_term_kind_of_value(bp, 1))
        copy_kov_to_string(bp_term_kind_of_value(bp, 1),
            problem_buffer+strlen(problem_buffer), FALSE);
    else strcat(problem_buffer, "an object");
    strcat(problem_buffer, ")");
}
```

The function bp_typecheck is called from 6/tcpr.

The function bp_assert is called from 6/asp.

The function bp_get_i6_schema is called from 6/atoms.

The function bp_describe_for_problems is called from 2/prob2.

Relations 5/rel

Purpose

What Inform internally calls "binary predicates", the user calls "relations". In this section, we parse definitions of new relations and create the resulting `binary_predicate` objects.

Template interpreter commands

```
25  {-array:relation_metadata}
29  {-callv:compile_runtime_relation_storage}
42  {-callv:compile_defined_relations}
```

---

§1. Creation, Stage I. The creation of relations happens in two stages. First, when the parse tree is being organised into sentences, we call the following routine the moment a relation definition has been found. (This is important because it may affect the parsing of subsequent sentences in the source text.) The predicate we make is initially sketchy: but by existing, and having a name, it can be used in subsequent verb definitions, and then subsequent sentences using those newly defined verbs can be properly parsed all during the same run-through of the parse tree.

```
void parse_new_relation(parse_node *PN) {
    int rw1 = PN->down->next->word_ref1;
    if (parse_relation_name(rw1)) {
        sentence_problem(_P_(C5RelationExists),
            "that relation already exists",
            "and cannot have its definition amended now.");
        return;
    }
    make_sketchy_pair_of_BPs(lw_array[rw1].lw_identity, Relation_OtoO);
}
```

The function parse_new_relation is called from 4/verb.

§2. Creation, Stage II. In the second stage, which is reached during the first traverse of sentences to work through the assertions, we parse the specification of the relation properly and complete the BP structure. (In the interim period, the name of the BP is really the only thing that has been used.)

Altogether, the Inform user is allowed to define some eight different forms of relation. The code below is an attempt to find whatever common ground can be found from these different outcomes, but inevitably ends up splitting into cases.

```
sentence_handler NEW_RELATION_SH_handler =
    { SENTENCE_NT, NEW_RELATION_VB, 1, parse_new_relation_further };

void parse_new_relation_further(parse_node *PN) {
    int rw = PN->down->next->word_ref1;                                  word number of relation name
    binary_predicate *bp = parse_relation_name(rw), *bpr = bp->reversal;
    property_name *prn = NULL;                                  used for run-time storage of this relation
    char *i6_prn_name = NULL;                                          the I6 identifier for this property
    world_object *storage_k = NULL;                          what kind of object, if any, might be stored in it
    int left_unique = NOT_APPLICABLE,                                  TRUE for one, FALSE for various,
        right_unique = NOT_APPLICABLE,                                 ...or NOT_APPLICABLE for unspecified
        condition_w1 = -1, condition_w2 = -1,                                text of test condition if any
        symmetric = FALSE,                                                        a symmetric relation?
        equivalence = FALSE,                                    an equivalence ("in groups") relation?
        rvno = FALSE,                                                      relate values not objects?
        frf = FALSE;                                                          use fast route-finding?
    if (bp == NULL) internal_error("BP in relation not initially parsed");
    ⟨Parse the classification variables and use them to fill in the BP term details 3⟩;
    if (rvno) { bp->relates_values_not_objects = TRUE; bpr->relates_values_not_objects = TRUE; }
    if (frf) { bp->fast_route_finding = TRUE; bpr->fast_route_finding = TRUE; }
    if (prn) { bp->i6_storage_property = prn; bpr->i6_storage_property = prn; }
    if (condition_w1 >= 0) ⟨Complete as a relation-by-routine BP 22⟩
    else if (equivalence) ⟨Complete as an equivalence-relation BP 21⟩
    else if (left_unique) {
        if (right_unique) {
            if (symmetric) ⟨Complete as a symmetric one-to-one BP 19⟩
            else ⟨Complete as an asymmetric one-to-one BP 15⟩;
        } else ⟨Complete as a one-to-various BP 16⟩;
    } else {
        if (right_unique) ⟨Complete as a various-to-one BP 17⟩
        else if (symmetric) ⟨Complete as a symmetric various-to-various BP 20⟩
        else ⟨Complete as an asymmetric various-to-various BP 18⟩;
    }
    bpr->form_of_relation = bp->form_of_relation;
    LOGIF(RELATION_DEFINITIONS, "Defined the binary predicate:\n$2\n", bp);
}
```

§**3. The parsing phase.** Our aims here are:

(i) to decide if the definition is valid, and reject it with a suitable problem message if not, returning from the current routine;
(ii) to fill in the classification variables `left_unique`, `symmetric`, etc., as defined above;
(iii) to choose a property which will provide run-time storage for this relation, if it needs any; and
(iii) to set `bp->term_details[0]` and `...[1]` with the kinds, names and logical properties of the two terms of the BP being defined.

⟨Parse the classification variables and use them to fill in the BP term details 3⟩ ≡

```
int f1 = PN->down->next->word_ref1 + 2;                    left term declaration, before "to"
int f2 = PN->down->next->word_ref2;
int s1 = PN->down->next->next->word_ref1;                  right term declaration, after "to"
int s2 = PN->down->next->next->word_ref2;
world_object *left_k = NULL, *right_k = NULL;                              kind requirement
kind_of_value *left_kov = NULL, *right_kov = NULL;                  kind of value requirement
int left_cw1 = -1, left_cw2 = -1,                                    left term "calling" name
    right_cw1 = -1, right_cw2 = -1;                                 right term "calling" name
LOGIF(RELATION_DEFINITIONS,
    "Relation definition of $W: left term: '$W', right term: '$W'\n",
        rw, rw, f1, f2, s1, s2);
⟨Find term multiplicities and use of fast route-finding 5⟩;
⟨Detect use of a condition for a test-only relation 6⟩;
⟨Detect callings for the terms of the relation 7⟩;
⟨Detect use of symmetry in definition of second term 8⟩;
⟨Vet the use of callings for the terms of the relation 9⟩;
⟨Work out the kinds of the terms in the relation 10⟩;
if (left_unique == NOT_APPLICABLE) left_unique = FALSE;
if (right_unique == NOT_APPLICABLE) right_unique = FALSE;
if (condition_w1 == -1) {
    ⟨Police restrictions on some combinations 11⟩;
    ⟨Determine property used for run-time storage 12⟩;
}
⟨Fill in the BP term details based on the left- and right- variables 4⟩;
```

This code is used in §2.

§**4.** The left- and right- local variables above provide us with convenient aliases for the entries which will end up in the `bp_term_details` structures attached to the BP: this is where we put them back.

For the meaning of functions f_0 and f_1, see "Binary Predicates.w". The idea here is this: suppose we have a relation of objects where the only true outcomes have the form $B(f_0(y), y)$. At run-time we store the identity of the counterpart object $f_0(y)$ in the `prn` property of the original object y.

And we similarly construct an f_1 function if the only true outcomes have the form $B(x, f_1(x))$.

⟨Fill in the BP term details based on the left- and right- variables 4⟩ ≡

```
i6_schema *f0 = NULL, *f1 = NULL;
bp_term_details left_bptd, right_bptd;
if (i6_prn_name) {
    if (left_unique) f0 = sch_new_1("(*1.%s)", i6_prn_name);
    else if (right_unique) f1 = sch_new_1("(*1.%s)", i6_prn_name);
}
left_bptd = bptd_full_new(left_k, left_kov, left_cw1, left_cw2, f0);
right_bptd = bptd_full_new(right_k, right_kov, right_cw1, right_cw2, f1);
```

```
    bp->term_details[0] = left_bptd; bp->term_details[1] = right_bptd;
    bpr->term_details[0] = right_bptd; bpr->term_details[1] = left_bptd;
```

This code is used in §3.

§**5.** We set word ranges for the condition (if any) and the callings (if any), whittling down the word ranges for the left and right specifications if these are clipped away, and also look at the multiplicities.

⟨Find term multiplicities and use of fast route-finding 5⟩ ≡

```
    if [[s1, s2 == ... with fast route_finding --> s1, s2]] frf = TRUE;
    if [[f1, f2 == one ... --> f1, f2]]     left_unique = TRUE;
    if [[f1, f2 == various ... --> f1, f2]] left_unique = FALSE;
    if [[s1, s2 == one ... --> s1, s2]]     right_unique = TRUE;
    if [[s1, s2 == various ... --> s1, s2]] right_unique = FALSE;
    if (frf && (left_unique != FALSE) && (right_unique != FALSE)) {
        sentence_problem(_P_(C5FRFUnavailable),
            "fast route-finding is only possible with various-to-various "
            "relations",
            "though this doesn't matter because with other relations the "
            "standard route-finding algorithm is efficient already.");
        return;
    }
```

This code is used in §3.

§**6.** When a relation is said to hold depending on a condition to be tested at run-time, it is meaningless to tell Inform anything about the uniqueness of terms in the domain: a relation might be one-to-one at the start of play but become various-to-various later on, as the outcomes of these tests change. So we reject any such misleading syntax.

⟨Detect use of a condition for a test-only relation 6⟩ ≡

```
    int i;
    if [[s1, s2 == ... when ... : i --> s1, s2 ... condition_w1, condition_w2]] {
        if ((left_unique != NOT_APPLICABLE) || (right_unique != NOT_APPLICABLE)) {
            sentence_problem(_P_(C5OneOrVariousWithWhen),
                "this relation is a mixture of different syntaxes",
                "and must be simplified. If it is going to specify 'one' or "
                "'various' then it cannot also say 'when' the relation holds.");
            return;
        }
    }
```

This code is used in §3.

§**7.** Callings are used to give names to the terms on each side of the relation, e.g.,

Lock-fitting relates one thing (called the matching key) to various things.

⟨Detect callings for the terms of the relation 7⟩ ≡

```
if (word_range_ends_with_calling(f1, f2, &f1, &f2, &left_cw1, &left_cw2)) {
    if (brackets_unbalanced(left_cw1, left_cw2))
        ⟨Issue problem message for malformed relation definition 13⟩;
}
if (word_range_ends_with_calling(s1, s2, &s1, &s2, &right_cw1, &right_cw2)) {
    if (brackets_unbalanced(right_cw1, right_cw2))
        ⟨Issue problem message for malformed relation definition 13⟩;
}
```

This code is used in §3.

§**8.** The second term can be given in several special ways to indicate symmetry between the two terms. This is more than a declaration that the left and right terms belong to the same domain set (though that is true): it says that $R(x, y)$ is true if and only if $R(y, x)$ is true.

⟨Detect use of symmetry in definition of second term 8⟩ ≡

```
int specified_one = left_unique;
if [[s1, s2 == another]] {
    symmetric = TRUE; left_unique = TRUE; right_unique = TRUE;
}
if [[s1, s2 == each other]] {
    symmetric = TRUE; left_unique = FALSE; right_unique = FALSE;
}
if [[s1, s2 == each other in groups]] {
    symmetric = TRUE; left_unique = FALSE; right_unique = FALSE; equivalence = TRUE;
}
if ((specified_one == TRUE) && (left_unique == FALSE)) {
    sentence_problem(_P_(C5BothOneAndMany),
        "the left-hand term in this relation seems to be both 'one' thing "
        "and also many things",
        "given the mention of 'each other'. Try removing the 'one'.");
    return;
}
```

This code is used in §3.

§**9.** To give a name to one term implies some degree of uniqueness about it. But that only makes sense if there is indeed some uniqueness involved, because otherwise it is unclear what the name refers to. Who is "the greeter of the Queen of Sheba" given the following definition?

> Acquaintance relates various people (called the greeter) to various people.

Because of that, callings are only allowed in certain circumstances. An exception is made – that is, they are always allowed – where the relation tests a given condition, because then the names identify the terms, e.g.,

> Divisibility relates a number (called N) to a number (called M) when the remainder after dividing M by N is 0.

Here the names "N" and "M" unambiguously refer to the terms being tested at this moment, and have no currency beyond that context.

⟨Vet the use of callings for the terms of the relation 9⟩ ≡

```
if (condition_w1 == -1) {
    if ((left_unique == FALSE) && (left_cw1 >= 0)) {
        sentence_problem(_P_(C5CantCallLeft),
            "the left-hand term of this relation is not unique",
            "so you cannot assign a name to it using 'called'.");
        return;
    }
    if ((right_unique == FALSE) && (right_cw1 >= 0)) {
        sentence_problem(_P_(C5CantCallRight),
            "the right-hand term of this relation is not unique",
            "so you cannot assign a name to it using 'called'.");
        return;
    }
    if ((left_cw1 >= 0) && (right_cw1 >= 0)) {
        sentence_problem(_P_(C5CantCallBoth),
            "the relation has to have the same name on both sides",
            "so it should be 'called' something even-handed: for instance, "
            "'Marriage relates one person to another (called the spouse).' "
            "rather than 'Marriage relates one man (called the husband) to "
            "one woman (called the wife).'");
        return;
    }
    if ((symmetric == FALSE) && (left_unique) && (right_unique) && (right_cw1 >= 0)) {
        sentence_problem(_P_(C5OneToOneMiscalled),
            "with a one-to-one relation which is not symmetrical "
            "only the left-hand item can be given a name using 'called'",
            "so this needs rephrasing to name the left in terms of the right "
            "rather than vice versa. For instance, 'Transmission relates "
            "one remote to one gadget (called the target).' should be "
            "rephrased as 'Transmission relates one gadget (called the "
            "target) to one remote.' It will then be possible to talk about "
            "'the gadget of' any given remote.");
        return;
    }
}
```

This code is used in §3.

§**10.** At the end of this paragraph of code, on each side either the `k` or the `kov` is set to a non-NULL pointer, but not both. In the event of a KOV, it is an enumerated KOV with a known finite domain. (If a relation will involve run-time storage, then it must have a fixed-size and fairly small domain set for its terms: relating a number to a number would be impossible. A relation defined in terms of a condition, which is test-only, doesn't suffer from this restriction.)

⟨Work out the kinds of the terms in the relation 10⟩ ≡

```
if (parse_relation_term_type(f1, f2, &left_k, &left_kov, "left") == FALSE) return;
if (symmetric) {
    right_k = left_k; right_kov = left_kov;
} else {
    if (parse_relation_term_type(s1, s2, &right_k, &right_kov, "right") == FALSE) return;
}
if ((left_kov) || (right_kov)) rvno = TRUE;
if (condition_w1 == -1) {
    if ((left_kov) &&
        (check_finite_range(left_kov) == FALSE)) return;
    if ((right_kov) && (symmetric == FALSE) &&
        (check_finite_range(right_kov) == FALSE)) return;
}
```

This code is used in §3.

§**11.** Not everything permitted by the grammar of these declarations is actually allowed: there are a number of pragmatic restrictions.

⟨Police restrictions on some combinations 11⟩ ≡

```
if ((left_k == NULL) || (right_k == NULL))
    if ((left_unique) || (right_unique)) {
        sentence_problem(_P_(C5ValueRelationNotV2V),
            "at present relations among values must be various-to-various",
            "so for instance 'Shading relates various things to various colours' "
            "is allowed, but not 'Shading relates various things to one colour'. "
            "(The latter would be better done with a property anyhow.)");
        return;
    }
if (equivalence) {
    if (left_k == NULL) {
        sentence_problem(_P_(C5ValueRelationInGroups),
            "at present relations in groups are restricted to objects",
            "such as things or rooms, and not values.");
        return;
    }
}
```

This code is used in §3.

§**12.** All forms of relation we can produce from here use an I6 property for run-time storage (though different forms of relation use it differently). We use the calling, if any, to name this property: if there are no callings, then it gets a name like "concealment relates", and is omitted from the index.

⟨Determine property used for run-time storage 12⟩ ≡
```
if (left_cw1 >= 0) {
    prn = typed_property_name_ref(left_cw1, left_cw2, left_kov);
    if (prn == NULL) return;
} else if (right_cw1 >= 0) {
    prn = typed_property_name_ref(right_cw1, right_cw2, right_kov);
    if (prn == NULL) return;
} else {
    if (brackets_unbalanced(rw, rw+1))
        ⟨Issue problem message for malformed relation definition 13⟩;
    prn = typed_property_name_ref(rw, rw+1, kova(OBJECT_TY));
    if (prn == NULL) return;
    prn_exclude_from_index(prn);
}
i6_prn_name = prn_get_i6_identifier(prn);
storage_k = left_k;
if (left_unique) storage_k = right_k;
else if (right_unique) storage_k = left_k;
```

This code is used in §3.

§**13.** Something of a catch-all problem message:

⟨Issue problem message for malformed relation definition 13⟩ ≡
```
sentence_problem(_P_(C5RelMalformed),
    "this relation's definition is malformed",
    "and should follow these models: 'Concealment relates "
    "one thing to various things' or 'Knowledge relates "
    "various people to various things'.");
return;
```

This code is used in §7,12,7,12,7,12.

§**14. The completion phase.** At this point the BP is filled in except for: its form; the schemas for testing, asserting true and asserting false; the run-time storage property to be used, if any; and any fields which are specific to the form in question. Anyway, there are eight possible forms of explicit BP, so here are eight paragraphs creating them.

§**15.** The `Relation_OtoO` case, or one to one: "R relates one K to one K".

Such a relation consumes run-time storage of $5D$ bytes on the Z-machine and $14D$ bytes on Glulx, where D is the size of the domain...

⟨Complete as an asymmetric one-to-one BP 15⟩ ≡
```
bp->form_of_relation = Relation_OtoO;
prop_true_in_world_model_about(prop_to_provide_property(prn), storage_k, NULL);
prn_now_stores_1to1_relation(prn);
bp->make_true_function = sch_new_1("Relation_Now1to1(*2,%s,*1)", i6_prn_name);
bp->make_false_function = sch_new_1("Relation_NowN1toV(*2,%s,*1)", i6_prn_name);
```

This code is used in §2.

§**16.** The `Relation_OtoV` case, or one to various: "R relates one K to various K".

⟨Complete as a one-to-various BP 16⟩ ≡
```
bp->form_of_relation = Relation_OtoV;
prop_true_in_world_model_about(prop_to_provide_property(prn), storage_k, NULL);
bp->make_true_function = sch_new_1("*2.%s = *1", i6_prn_name);
bp->make_false_function = sch_new_1("Relation_NowN1toV(*2,%s,*1)", i6_prn_name);
```
This code is used in §2.

§**17.** The `Relation_VtoO` case, or various to one: "R relates various K to one K".

⟨Complete as a various-to-one BP 17⟩ ≡
```
bp->form_of_relation = Relation_VtoO;
prop_true_in_world_model_about(prop_to_provide_property(prn), storage_k, NULL);
bp->make_true_function = sch_new_1("*1.%s = *2", i6_prn_name);
bp->make_false_function = sch_new_1("Relation_NowN1toV(*1,%s,*2)", i6_prn_name);
```
This code is used in §2.

§**18.** The `Relation_VtoV` case, or various to various: "R relates various K to various K".

⟨Complete as an asymmetric various-to-various BP 18⟩ ≡
```
bp->form_of_relation = Relation_VtoV;
bp->arbitrary = TRUE;
bp->test_function = sch_new_1d("(Relation_TestVtoV(*1,V2V_Bitmap_%d,*2,false))",
    bp->allocation_id);
bp->make_true_function = sch_new_1d("(Relation_NowVtoV(*1,V2V_Bitmap_%d,*2,false))",
    bp->allocation_id);
bp->make_false_function = sch_new_1d("(Relation_NowNVtoV(*1,V2V_Bitmap_%d,*2,false))",
    bp->allocation_id);
```
This code is used in §2.

§**19.** The `Relation_Sym_OtoO` case, or symmetric one to one: "R relates one K to another".

⟨Complete as a symmetric one-to-one BP 19⟩ ≡
```
bp->form_of_relation = Relation_Sym_OtoO;
prop_true_in_world_model_about(prop_to_provide_property(prn), storage_k, NULL);
prn_now_stores_1to1_relation(prn);
bp->make_true_function = sch_new_1("Relation_NowS1to1(*2,%s,*1)", i6_prn_name);
bp->make_false_function = sch_new_1("Relation_NowSN1to1(*2,%s,*1)", i6_prn_name);
```
This code is used in §2.

§**20.** The `Relation_Sym_VtoV` case, or symmetric various to various: "R relates K to each other".

⟨Complete as a symmetric various-to-various BP 20⟩ ≡
```
bp->form_of_relation = Relation_Sym_VtoV;
bp->arbitrary = TRUE;
bp->test_function = sch_new_1d("(Relation_TestVtoV(*1,V2V_Bitmap_%d,*2,true))",
    bp->allocation_id);
bp->make_true_function = sch_new_1d("(Relation_NowVtoV(*1,V2V_Bitmap_%d,*2,true))",
    bp->allocation_id);
bp->make_false_function = sch_new_1d("(Relation_NowNVtoV(*1,V2V_Bitmap_%d,*2,true))",
    bp->allocation_id);
```
This code is used in §2.

§**21.** The `Relation_Equiv` case, or equivalence relation: "R relates K to each other in groups".

⟨Complete as an equivalence-relation BP 21⟩ ≡

```
bp->form_of_relation = Relation_Equiv;
bp->arbitrary = TRUE;
prop_true_in_world_model_about(prop_to_provide_property(prn), storage_k, NULL);
bp->test_function = sch_new_2("(*1.%s == *2.%s)", i6_prn_name, i6_prn_name);
bp->make_true_function = sch_new_1("Relation_NowEquiv(*1,%s,*2)", i6_prn_name);
bp->make_false_function = sch_new_1("Relation_NowNEquiv(*1,%s,*2)", i6_prn_name);
```

This code is used in §2.

§**22.** The `Relation_ByRoutine` case, or relation tested by a routine: "R relates K to L when (some condition)".

⟨Complete as a relation-by-routine BP 22⟩ ≡

```
bp->form_of_relation = Relation_ByRoutine;
bp->test_function = sch_new_1d("(Relation_%d(*1,*2))", bp->allocation_id);
bp->condition_defn_w1 = condition_w1; bp->condition_defn_w2 = condition_w2;
```

This code is used in §2.

§**23. Parsing utilities.** A term is specified either as a kind, or as the name of a kind of value.

```
int parse_relation_term_type(int w1, int w2, world_object **k, kind_of_value **kov, char *side) {
    specification *spec;
    *k = parse_world_object(w1, w2, TRUE); *kov = NULL;
    if (*k) return TRUE;
    spec = parse_expression(w1, w2, TYPE_EXPCON);
    if (spec_is_generic_CONSTANT(spec)) {
        *kov = spec_get_kind_of_value(spec);
        return TRUE;
    }
    quote_source(1, current_sentence);
    quote_words(2, w1, w2);
    quote_text(3, side);
    handmade_problem(_P_(C5RelatedKindsUnknown));
    issue_problem_segment(
        "In the relation definition %1, I am unable to understand the %3-hand "
        "side -- I was expecting that %2 would be either the name of a kind, "
        "or the name of a kind of value, but it wasn't either of those.");
    issue_problem_end();
    return FALSE;
}
```

§**24.** A modest utility, to check for a case we forbid because of the prohibitive (or anyway unpredictable) run-time storage it would imply.

```
int check_finite_range(kind_of_value *kov) {
    if (kov_is_an_enumeration(kov)) return TRUE;
    if (kov == NULL) return TRUE;                                    to recover from earlier problems
    if ((is_kova(kov, OBJECT_TY)) || (is_kova(kov, ANY_VALUE_TY)))
    sentence_problem(_P_(C5RangeOverlyBroad),
        "relations aren't allowed to range over all 'objects' or all 'values'",
        "as these are too broad. A relation has to be between two kinds of "
        "object, or kinds of value. So 'Taming relates various people to "
        "various animals' is fine, because 'people' and 'animals' both mean "
        "kinds of object, but 'Wanting relates various objects to various "
        "values' is not allowed.");
    else sentence_problem(_P_(C5ValueRangeInfinite),
        "relations involving values can only be used with new kinds of value "
        "having a known set of possible values (unless they are defined using "
        "a condition)",
        "so for instance 'Tallying relates various things to various numbers.' "
        "is disallowed because 'number' is a kind of value with an unlimited "
        "range of possible values; but 'Colour is a kind of value. The colours "
        "are red, blue and green. Painting relates various things to various "
        "colours.' is allowed. (And if we use a condition, anything goes: "
        "for instance, 'Divisibility relates a number (called N) to a number "
        "(called M) when remainder after dividing M by N is 0.' defines a "
        "relation between numbers.)");
    return FALSE;
}
```

§**25. The relation metadata array at run-time.** The template layer needs to be able to tell, given a value of KOV "abstract-relation", what form of relation it represents, where its run-time storage lives in memory and so on. This information is wrapped up in the `relation_metadata` array, which consists of three-word records plus a `NULL` terminator. An "abstract-relation" value at run-time is the allocation ID number of the `binary_predicate` structure representing the relation: it follows that data on abstract-relation `R` can be found at the I6 location

```
relation_metadata-->(3*R)
```

Field 0 depends on the form of the relation (see below). Field 1 is the form number. Field 2 is the text of the sentence used to define the relation – this is so that the RELATIONS testing command can generate legible output.

```
void compile_relation_metadata_array(OUTPUT_STREAM) {
    binary_predicate *bp;
    WRITE("Array relation_metadata -->\n"); INDENT;
    LOOP_OVER(bp, binary_predicate) {
        ⟨Write the storage field of the relation metadata array 26⟩;
        if (bp->right_way_round == FALSE) WRITE(" (%d) \"", Relation_Implicit);
        else WRITE(" %s \"", bp_form_to_text(bp));
        if (bp->form_of_relation == Relation_Implicit) WRITE("%s", bp_get_log_name(bp));
        else print_raw_text_to_file(bp->bp_created_at->word_ref1,
            bp->bp_created_at->word_ref2, OUT);
        WRITE("\"\n");
    }
```

```
    WRITE("NULL;\n");
    OUTDENT;
}
```

The function compile_relation_metadata_array is invoked by a command in a `.i6t` template file.

§**26.** Field 0 has different meanings for different families of BPs:

⟨Write the storage field of the relation metadata array 26⟩ ≡

```
    binary_predicate *dbp = bp;
    if (bp->right_way_round == FALSE) dbp = bp->reversal;
    switch(dbp->form_of_relation) {
        case Relation_Implicit:                                    Field 0 is not used
            WRITE("0");                                  which is not the same as NULL, unlike in C
            break;
        case Relation_OtoO:
        case Relation_OtoV:
        case Relation_VtoO:
        case Relation_Sym_OtoO:
        case Relation_Equiv:                          Field 0 is the property used for run-time storage
            WRITE("%s", prn_get_i6_identifier(dbp->i6_storage_property));
            break;
        case Relation_VtoV:
        case Relation_Sym_VtoV:                  Field 0 is the bitmap array used for run-time storage
            WRITE("V2V_Bitmap_%d", dbp->allocation_id);
            break;
        case Relation_ByRoutine:                      Field 0 is the routine used to test the relation
            WRITE("Relation_%d", dbp->allocation_id);
            break;
        default:
            internal_error("Binary predicate with unknown structural type");
    }
```

This code is used in §25.

§**27. The bitmap for various-to-various relations.** It is unavoidable that a general V-to-V relation will take at least LR bits of storage, where L is the size of the left domain and R the size of the right domain. (A symmetric V-to-V relation needs only a little over $LR/2$ bits, though in practice we don't want the nuisance of this memory saving.) Cheaper implementations would only be possible if we could guarantee that the relation would have some regularity, or would be sparse, but we can't guarantee any of that. Our strategy will therefore be to store these LR bits in the most direct way possible, with as little overhead as possible: in a bitmap.

§**28.** The following code compiles a stream of bits into a sequence of 16-bit I6 constants written in hexadecimal, padding out with 0s to fill any incomplete word left at the end. The first bit of the stream becomes the least significant bit of the first word of the output.

```
int word_compiled = 0, bit_counter = 0, words_compiled;

void begin_bit_stream(OUTPUT_STREAM) {
    word_compiled = 0; bit_counter = 0; words_compiled = 0;
}

void compile_bit(OUTPUT_STREAM, int b) {
    word_compiled += (b << bit_counter);
    bit_counter++;
    if (bit_counter == 16) {
        WRITE("$%04x ", word_compiled);
        words_compiled++;
        word_compiled = 0; bit_counter = 0;
    }
}

void end_bit_stream(OUTPUT_STREAM) {
    while (bit_counter != 0) compile_bit(OUT, 0);
}
```

§**29.** As was implied above, the run-time storage for a various to various relation whose BP has allocation ID number `X` is an I6 word array called `V2V_Bitmap_X`. This begins with a header of 8 words and is then followed by a bitmap.

```
define MAX_RIGHT_DOMAIN_SIZE 10000

void compile_runtime_relation_storage(OUTPUT_STREAM) {
    binary_predicate *bp;
    LOOP_OVER(bp, binary_predicate)
        if ((bp->right_way_round) &&
            ((bp->form_of_relation == Relation_VtoV) ||
                (bp->form_of_relation == Relation_Sym_VtoV))) {
            int left_count = 0, right_count = 0, words_used = 0, bytes_used = 0;
            ⟨Index the left and right domains and calculate their sizes 30⟩;
            WRITE("Array V2V_Bitmap_%d --> ", bp->allocation_id);
            ⟨Compile header information in the V-to-V structure 31⟩;
            if ((left_count > 0) && (right_count > 0))
                ⟨Compile bitmap pre-initialised to the V-to-V relation at start of play 34⟩;
            WRITE(";\n");
            if ((left_count > 0) && (right_count > 0))
                ⟨Allocate a zeroed-out memory cache for relations with fast route-finding 32⟩;
            note_VM_usage("relation",
                bp->bp_created_at->word_ref1, -1,
                vocab_get_exemplar(bp->relation_name, FALSE),
                words_used, bytes_used, FALSE);
        }
}
```

The function compile_runtime_relation_storage is invoked by a command in a `.i6t` template file.

§**30.** We calculate numbers L and R, and index the items being related, so that the possible left values are indexed $0, 1, 2, ..., L-1$ and the possible right values $0, 1, 2, ..., R-1$. Note that in a relation such as

Roominess relates various things to various containers.

the same object (if a container) might be in both the left and right domains, and be indexed differently on each side: it might be thing number 11 but container number 6, for instance.

L and R are stored in the variables `left_count` and `right_count`. If the left domain contains objects, the index of a member `wo` is stored in `wo->relation_indices[0]`; if the right domain does, then in `wo->relation_indices[1]`. If the domain set is an enumerated kind of value, no index needs to be stored, because the values are already enumerated $1, 2, 3, ..., N$ for some N. The actual work in this is done by the routine `relation_range` (below).

⟨Index the left and right domains and calculate their sizes 30⟩ ≡

```
left_count = relation_range(bp, 0);
right_count = relation_range(bp, 1);
if (right_count >= MAX_RIGHT_DOMAIN_SIZE) internal_error("Far too many relatees");
```

This code is used in §29.

§**31.** See "Relations.i6t" in the template layer for details.

⟨Compile header information in the V-to-V structure 31⟩ ≡

```
if ((bp_term_kind(bp, 0)) && (left_count > 0))
    WRITE("IK_%d ", bp_term_kind(bp, 0)->allocation_id);
else WRITE("0 ");
if ((bp_term_kind(bp, 1)) && (right_count > 0))
    WRITE("IK_%d ", bp_term_kind(bp, 1)->allocation_id);
else WRITE("0 ");
WRITE("\n  %d ! Number of left instances\n", left_count);
WRITE("  %d ! Number of right instances\n  ", right_count);
WRITE("  %s ! To print left instances\n",
    (bp_term_kind(bp, 0))?"PrintShortName":
        (kov_get_name_of_printing_rule(bp_term_kind_of_value(bp, 0))));
WRITE("  %s ! To print right instances\n  ",
    (bp_term_kind(bp, 1))?"PrintShortName":
        (kov_get_name_of_printing_rule(bp_term_kind_of_value(bp, 1))));
WRITE("  true ! Cache broken flag\n");
if ((left_count > 0) && (right_count > 0))
    WRITE("  V2V_Route_Cache_%d ! Cache array (if any)\n", bp->allocation_id);
else
    WRITE("  0 ! No cache array needed\n");
words_used += 8;
```

This code is used in §29.

§**32.** Fast route finding is available only where the left and right domains are equal, and even then, only when the user asked for it. If so, we allocate LR bytes as a cache if $L = R < 256$, and LR words otherwise. The cache is initialised to all-zeros, which saves an inordinate amount of nuisance, and this is why the "cache broken" flag is initially set in the header above: it forces the template layer to generate the cache when first used.

⟨Allocate a zeroed-out memory cache for relations with fast route-finding 32⟩ ≡

```
if ((bp->fast_route_finding) &&
    ((bp_term_kind(bp, 0) == bp_term_kind(bp, 1)) && (left_count == right_count))) {
    if (left_count < 256) {
        WRITE("Array V2V_Route_Cache_%d -> %d;\n",
            bp->allocation_id, 2*left_count*left_count);
        bytes_used += 2*left_count*left_count;
    } else {
        WRITE("Array V2V_Route_Cache_%d --> %d;\n",
            bp->allocation_id, 2*left_count*left_count);
        words_used += 2*left_count*left_count;
    }
} else {
    WRITE("Constant V2V_Route_Cache_%d = 0;\n", bp->allocation_id);
}
```

This code is used in §29.

§**33.** The following macro `wo` conveniently iterates through the objects in domain `i` of the relation defined by `bp`: it should only be used if the relevant domain set does indeed contain objects.

```
define LOOP_OVER_BP_DOMAIN(wo, bp, i)
    LOOP_OVER(wo, world_object)
        if ((wo->kind_flag == FALSE) && (wo_of_kind(wo, bp_term_kind(bp, i))))
```

§**34.** Now to assemble the bitmap. We do this by calling out to the world-model code to ask it what pairs (x, y) are such that assertions have declared that $B(x, y)$ must be true. Because the world-model code stores inferences about objects differently from those about values, we call one department when x and y are both objects, but another if either one is a value.

It would be convenient if the world-model code could feed us the necessary information in exactly the right order, but life is not that kind. On the other hand it would save the world model code some trouble if we built the entire bitmap in memory, so that it could send the pairs (x, y) in any order at all, but that's a little wasteful. We compromise and build the bitmap one row at a time, requiring us to store a whole row, but allowing the world-model code to send the pairs in that row in any order. We do this by first calling either `begin_collating_vpairs` or `begin_collating_wopairs`, and then calling either `send_vpairs_in_row` or `send_wopairs_in_row` for each row (i.e., each x value) in turn. The world-model code then calls `send_related_index` for every y it knows of where $B(x, y)$ is true.

⟨Compile bitmap pre-initialised to the V-to-V relation at start of play 34⟩ ≡

```
char row_flags[MAX_RIGHT_DOMAIN_SIZE];
if (bp_relates_values_not_objects(bp))
    begin_collating_vpairs(bp, left_count, right_count);
else
    begin_collating_wopairs(bp, left_count, right_count);
begin_bit_stream(OUT);
if (bp_relates_values_not_objects(bp)) {
    int i, j;
    for (i=0; i<left_count; i++) {
```

```
            for (j=0; j<right_count; j++) row_flags[j] = 0;
            send_vpairs_in_row(bp, i, row_flags);
            for (j=0; j<right_count; j++) compile_bit(OUT, row_flags[j]);
        }
    } else {
        world_object *wo; int j;
        ⟨Include some comments to make the bitmap slightly easier for humans to decipher 36⟩;
        LOOP_OVER_BP_DOMAIN(wo, bp, 0) {
            for (j=0; j<right_count; j++) row_flags[j] = 0;
            send_wopairs_in_row(bp, wo, row_flags);
            for (j=0; j<right_count; j++) compile_bit(OUT, row_flags[j]);
        }
    }
    end_bit_stream(OUT);
    words_used += words_compiled;
```

This code is used in §29.

§**35.** See above. (The world-model code does nothing with `row_flags`, simply passing it back on to us as a pointer.)

```
void send_related_index(binary_predicate *bp, int i, char *row_flags) {
    row_flags[i] = 1;
}
```

The function send_related_index is called from 9/cot.

§**36.** The following does nothing functional.

⟨Include some comments to make the bitmap slightly easier for humans to decipher 36⟩ ≡

```
    LOOP_OVER_BP_DOMAIN(wo, bp, 0)
        WRITE("  ! Left-count %d = %s\n",
            wo->relation_indices[0], wo_get_I6_representation(wo));
    LOOP_OVER_BP_DOMAIN(wo, bp, 1)
        WRITE("  ! Right-count %d = %s\n",
            wo->relation_indices[1], wo_get_I6_representation(wo));
```

This code is used in §34.

§**37.** Lastly on this: the way we count and index the left (`index=0`) or right (1) domain. If the domain set is an enumerated kind of value, there's no indexing required, because the values at run-time are already $1, 2, 3, ..., N$, where N is the result of calling `kov_get_highest_valid_value_as_integer`.

But if the domain set is a kind of object, there's no regularity in the values at run-time, and so we assign each object an index in the domain set, counting upwards from 0 (in order of creation).

```
int relation_range(binary_predicate *bp, int index) {
    int t = 0;
    if (bp_term_kind(bp, index)) {
        world_object *wo;
        LOOP_OVER(wo, world_object)
            wo->relation_indices[index] = -1;
        LOOP_OVER_BP_DOMAIN(wo, bp, index) {
            wo->relation_indices[index] = t++;
        }
    } else {
        kind_of_value *kov = bp_term_kind_of_value(bp, index);
        if (kov_has_named_constant_values(kov) == FALSE) return -1;
        t = kov_get_highest_valid_value_as_integer(kov);
    }
    return t;
}
```

§**38. The partition for an equivalence relation.** An equivalence relation E is such that $E(x, x)$ for all x, such that $E(x, y)$ if and only if $E(y, x)$, and such that $E(x, y)$ and $E(y, z)$ together imply $E(x, z)$: the properties of being reflexive, symmetric and transitive. The relation constructed by a sentence like

Alliance relates people to each other in groups.

is to be an equivalence relation. This means we need to ensure first that the original state of the relation, resulting from assertions such as...

The verb to be allied to implies the alliance relation. Louis is allied to Otto. Otto is allied to Helene.

...satisfies the reflexive, symmetric and transitive properties; and then also that these properties are maintained at run-time when the situation changes as a result of executing phrases such as

now Louis is allied to Gustav;

We use the same solution both in the compiler and at run-time, which is to exploit an elementary theorem about ERs. Let E be an equivalence relation on the members of a set S (say, the set of people in Central Europe). Then there is a unique way to divide up S into a "partition" of subsets called "equivalence classes" such that:

(a) every member of S is in exactly one of the classes,
(b) none of the classes is empty, and
(c) $E(x, y)$ is true if and only if x and y belong to the same class.

Conversely, given any partition of S (i.e., satisfying (a) and (b)), there is a unique equivalence relation E such that (c) is true. In short: possible states of an equivalence relation on a set correspond exactly to possible ways to divide it up into non-empty, non-overlapping pieces.

We therefore store the current state not as some list of which pairs (x, y) for which $E(x, y)$ is true, but instead as a partition of the set S. We store this as a function $p : S \to \{1, 2, 3, ...\}$ such that x and y belong in the same class – or to put it another way, such that $E(x, y)$ is true – if and only if $p(x) = p(y)$. When we are assembling the initial state, the function p is an array of integers whose address is stored in the `bp->equivalence_partition` field of the BP structure. It is then compiled into the storage properties of the I6 objects concerned. For instance, if we have `p44_alliance` as the storage property for the "alliance"

relation, then `O31_Louis.p44_alliance` and `O32_Otto.p44_alliance` will be set to the same partition number. The template routines which set and remove alliance then maintain the collective values of the `p44_alliance` property, keeping it always a valid partition function for the relation.

§**39.** We calculate the initial partition by starting with the sparsest possible equivalence relation, $E(x, y)$ if and only if $x = y$, where each member is related only to itself. (This is the equality relation.) The partition function here is given by $p(x)$ equals the allocation ID number for object x, plus 1. Since all objects have distinct IDs, $p(x) = p(y)$ if and only if $x = y$, which is what we want. But note that the objects in S may well not have contiguous ID numbers. This doesn't matter to us, but it means p may look less tidy than we expect.

For instance, suppose there are five people: Sophie, Ryan, Daisy, Owen and the player, with a "helping" equivalence relation. We might then generate the initial partition:

$$p(P) = 12, p(S) = 23, p(R) = 25, p(D) = 26, p(O) = 31.$$

```
void equivalence_relations_make_singleton_partitions(int domain_size) {
    binary_predicate *bp;
    LOOP_OVER(bp, binary_predicate)
        if ((bp->form_of_relation == Relation_Equiv) && (bp->right_way_round)) {
            int i;
            int *partition_array = I7_calloc(domain_size, sizeof(int), PARTITION_MREASON);
            for (i=0; i<domain_size; i++) partition_array[i] = i+1;
            bp->equivalence_partition = partition_array;
        }
}
```

The function equivalence_relations_make_singleton_partitions is called from 9/cot.

§**40.** The A-parser, over in Chapter 9, has meanwhile been reading in facts about the helping relation:

> Sophie helps Ryan. Daisy helps Ryan. Owen helps the player.

And it feeds these facts to us one at a time. It tells us that $A(S, R)$ has to be true by calling the routine below for the helping relation with the ID numbers of Sophie and Ryan as arguments. Sophie is currently in class number 23, Ryan in class 25. We merge these two classes so that anybody whose class number is 25 is moved down to have class number 23, and so:

$$p(P) = 12, p(S) = 23, p(R) = 23, p(D) = 26, p(O) = 31.$$

Similarly we now merge Daisy's class with Ryan's:

$$p(P) = 12, p(S) = 23, p(R) = 23, p(D) = 23, p(O) = 31.$$

And Owen's with the player's:

$$p(P) = 12, p(S) = 23, p(R) = 23, p(D) = 23, p(O) = 12.$$

This leaves us with the final partition where the two equivalence classes are

$$\{\text{player}, \text{Owen}\} \quad \{\text{Sophie}, \text{Daisy}, \text{Ryan}\}.$$

As mentioned above, it might seem "tidy" to renumber these classes 1 and 2 rather than 12 and 23, but there's really no need and we don't bother.

Note that the A-parser does not allow negative assertions about equivalence relations to be made:

Daisy does not help Ryan.

While we could try to accommodate this (using the same method we use at run-time to handle "now Daisy does not help Ryan"), it would only invite users to set up these relations in a stylistically poor way.

```
void equivalence_relation_merge_classes(binary_predicate *bp,
    int domain_size, int ix1, int ix2) {
    if (bp->form_of_relation != Relation_Equiv)
        internal_error("attempt to merge classes for a non-equivalence relation");
    if (bp->right_way_round == FALSE) bp = bp->reversal;
    int *partition_array = bp->equivalence_partition;
    if (partition_array == NULL)
        internal_error("attempt to use null equivalence partition array");
    int little, big;                                                    or, The Fairies' Parliament
    big = partition_array[ix1]; little = partition_array[ix2];
    if (big == little) return;
    if (big < little) { int swap = little; little = big; big = swap; }
    int i;
    for (i=0; i<domain_size; i++)
        if (partition_array[i] == big)
            partition_array[i] = little;
}
```

The function equivalence_relation_merge_classes is called from 9/cot.

§**41.** Once that process has completed, the code in Chapter 9 which compiles the initial state of the I6 object tree calls the following routine to ask it to fill in the (let's say) `p63_helping` property for each person in turn.

```
void equivalence_relations_add_properties(OUTPUT_STREAM, world_object *wo) {
    binary_predicate *bp;
    LOOP_OVER(bp, binary_predicate)
        if ((bp->form_of_relation == Relation_Equiv) && (bp->right_way_round) &&
            (wo->kind_flag == FALSE) && (wo_of_kind(wo, bp_term_kind(bp, 1))))
            WRITE(" with %s %d\n",
                prn_get_i6_identifier(bp->i6_storage_property),
                equivalence_relation_get_class(bp, wo->allocation_id));
}
int equivalence_relation_get_class(binary_predicate *bp, int ix) {
    if (bp->form_of_relation != Relation_Equiv)
        internal_error("attempt to merge classes for a non-equivalence relation");
    if (bp->right_way_round == FALSE) bp = bp->reversal;
    int *partition_array = bp->equivalence_partition;
    if (partition_array == NULL)
        internal_error("attempt to use null equivalence partition array");
    return partition_array[ix];
}
```

The function equivalence_relations_add_properties is called from 9/cot.

§42. Generating routines to test relations by condition. When a relation has to be tested as a condition, we can't simply embed that condition as the I6 schema for "test relation": it might very well need local variables, the table row-choosing variables, etc., to evaluate. It has to be tested in its own context. So we generate a routine called `Relation_X`, where `X` is the allocation ID number of the BP, which takes two parameters `t_0` and `t_1` and returns true or false according to whether or not $R(\texttt{t\_0}, \texttt{t\_1})$.

This is where those routines are compiled.

```
void compile_defined_relations(OUTPUT_STREAM) {
    binary_predicate *bp;
    LOOP_OVER(bp, binary_predicate)
        if ((bp->form_of_relation == Relation_ByRoutine) && (bp->right_way_round)) {
            char rname[32];
            sprintf(rname, "Relation_%d", bp->allocation_id);
            current_sentence = bp->bp_created_at;
            WRITE("! Routine to decide if %s(t_0, t_1)\n", bp_get_log_name(bp));
            compile_condition_routine(OUT, rname,
                bp->condition_defn_w1, bp->condition_defn_w2,
                bp->term_details[0], bp->term_details[1]);
        }
}
```

The function compile_defined_relations is invoked by a command in a `.i6t` template file.

§43. Indexing relations. A brief table of relations appears on the Phrasebook Index page.

```
void index_table_of_relations(void) {
    binary_predicate *bp;
    begin_plain_html_table(ifl);
    LOOP_OVER(bp, binary_predicate)
        if (bp->right_way_round) {
            char *type = NULL, *left = NULL, *right = NULL;
            switch (bp->relation_family) {
                case EQUALITY_KBP: type = "equality"; left = "<i>any</i>"; right = left; break;
                case QUASINUMERIC_KBP: type = "numeric"; break;
                case SPATIAL_KBP: type = "spatial"; break;
                case MAP_CONNECTING_KBP: type = "map"; left = "room/door"; right = left; break;
                case PROVISION_KBP: type = "provision"; left = "<i>any</i>"; right = "property"; break;
                case EXPLICIT_KBP:
                    switch (bp->form_of_relation) {
                        case Relation_OtoO: type = "one-to-one"; break;
                        case Relation_OtoV: type = "one-to-various"; break;
                        case Relation_VtoO: type = "various-to-one"; break;
                        case Relation_VtoV: type = "various-to-various"; break;
                        case Relation_Sym_OtoO: type = "one-to-another"; break;
                        case Relation_Sym_VtoV: type = "various-to-each-other"; break;
                        case Relation_Equiv: type = "in groups"; break;
                        case Relation_ByRoutine: type = "defined"; break;
                    }
                    break;
            }
            if ((type == NULL) || (bp->relation_name == NULL)) continue;
            first_html_column(ifl, 0);
            INDEX("%s", vocab_get_exemplar(bp->relation_name, FALSE));
```

```
            if (bp->bp_created_at) index_link(lw_array[bp->bp_created_at->word_ref1].lw_source);
            next_html_column(ifl, 0);
            if (type) INDEX("%s", type); else INDEX("--");
            next_html_column(ifl, 0);
            if (left) INDEX("%s", left);
            else bp_term_details_index(&(bp->term_details[0]));
            next_html_column(ifl, 0);
            if (right) INDEX("%s", right);
            else bp_term_details_index(&(bp->term_details[1]));
            end_html_row(ifl);
        }
    end_html_table(ifl);
}
```

The function index_table_of_relations is called from 2/lexi.

Conjugation of Verbs 5/conj

Purpose

To define verbal and prepositional forms for relations, in different tenses and numbers.

5/conj.§1-3 Verb usages; §4-6 Preposition usages; §7-8 Parsing source text against verb and preposition usages; §9 Debug log; §10 Index tabulation; §11 The built-in verbs and prepositions; §12 Registration of to be; §13-17 Registration of regular verbs; §18-21 Registration of to be able to; §22 Late registration of prepositions comparing properties; §23 Parsing new verb declarations; §24-26 I: Semantics; §27-33 IIa: Syntax of a new verb usage; §34-39 IIb: Syntax of a new prepositional usage

---

Template interpreter commands

```
9   {-callv:debug_verbs}
11   {-callv:make_built_in_verbs}
```

Definitions

¶1. A single English verb, such as "to contain", produces numerous `verb_usage` objects, since we have one for each combination of tense, number and negation – "contains", "had not contained", etc.

```
define MAX_WORDS_IN_USAGE 5

typedef struct verb_usage {
    struct vocabulary_entry *vu_text[MAX_WORDS_IN_USAGE];      some may be null
    int negated_form_of_verb;                                  is this a negated form?
    int tensed;                                                one of the four tense values
    struct binary_predicate *vu_meaning;                       the BP is the root meaning of the verb
    struct lexicon_entry *under_main_verb;                     for use when indexing
    struct parse_node *where_vu_created;                       for use if problem messages needed
    MEMORY_MANAGEMENT
} verb_usage;

verb_usage *regular_to_be = NULL;                              "is"
verb_usage *negated_to_be = NULL;                              "is not"
verb_usage *negated_plural_to_be = NULL;                       "are not"
```

The structure verb_usage is private to this section.

¶2. We use the term "preposition" pretty loosely here: for instance, "in" counts (which is fair enough) but so does "contained by". Essentially, these are forms of words which, when modifying the verb "to be", assert a relationship. (Thus "had not been contained by" couples "to be" in a particular tense with "contained by".)

Ordinarily a preposition, coupled with "to be", overly asserts a binary predicate. "X is carried by Y", for instance, clearly asserts a relationship between two things, X and Y. But in some cases the past participle of a verb can be used adjectivally with the second term, Y, left unspoken: for instance, "X is carried". The `implies_player` flag below asserts that given form is allowable adjectivally with Y filled in as the player.

```
typedef struct preposition_usage {
    struct vocabulary_entry *pu_text[MAX_WORDS_IN_USAGE];          some may be null
    int implies_player;                                  for prepositions used adjectivally
    struct property_name *is_same_property_as;              for "is the same P as" preps
    struct binary_predicate *pu_meaning;                          again, the root meaning
    struct parse_node *where_pu_created;                 for use if problem messages needed
    int negated_sense;                    if the sense of the sentence is implicitly negated
    int allow_unexpected_upper_case;                    for preps like "in Cahoots With"
    int implicitly_behave_as_noun;                                for preps like "part of"
    MEMORY_MANAGEMENT
} preposition_usage;
```

The structure preposition_usage is private to this section.

---

§**1. Verb usages.** The meaning of a verb, or of *to be* plus a preposition, is more complicated than a simple relation $R(a, b)$. It is R together with an indication of when R needs to be true (the *tense* of the verb), and of whether it in fact needs to be false (or *negated*). Each combination, and each possible wording, can have its own `verb_usage` structure. For instance, some 14 `verb_usage` structures exist which all ultimately correspond to the same equality predicate.

```
parse_node *set_where_created = NULL;
verb_usage *register_vu(vocabulary_entry *ve1, vocabulary_entry *ve2,
    vocabulary_entry *ve3, vocabulary_entry *ve4, vocabulary_entry *ve5,
    int negated, int tensed, binary_predicate *root) {
    verb_usage *vu = CREATE(verb_usage);
    vu->vu_text[0] = ve1; vu->vu_text[1] = ve2; vu->vu_text[2] = ve3;
    vu->vu_text[3] = ve4; vu->vu_text[4] = ve5;
    vu->negated_form_of_verb = negated; vu->tensed = tensed;
    vu->vu_meaning = root;
    vu->under_main_verb = current_main_verb;
    vu->where_vu_created = set_where_created;
    vocab_set_flags(ve1, CVERB_MC);
    return vu;
}
```

§**2.** The following is not especially fast, but is only a convenience used when creating verbs, so it runs only a handful of times in Inform's run.

```
verb_usage *find_vu(vocabulary_entry *ve1, vocabulary_entry *ve2,
    vocabulary_entry *ve3, vocabulary_entry *ve4, vocabulary_entry *ve5) {
    verb_usage *vu;
    LOOP_OVER(vu, verb_usage) {
        if (vu->vu_text[0] != ve1) continue;
        if (vu->vu_text[1] != ve2) continue;
        if (vu->vu_text[2] != ve3) continue;
        if (vu->vu_text[3] != ve4) continue;
        if (vu->vu_text[4] != ve5) continue;
        return vu;
    }
    return NULL;
}
```

§**3.** And some access routines:

```
binary_predicate *vu_get_meaning(verb_usage *vu) {
    return vu->vu_meaning;
}
int vu_get_tense_used(verb_usage *vu) {
    return vu->tensed;
}
int vu_is_used_negatively(verb_usage *vu) {
    return vu->negated_form_of_verb;
}
```

The function vu_get_meaning is called from 4/verb, 5/varc, 5/mlc and 6/sconv.
The function vu_get_tense_used is called from 2/prob3, 4/verb, 5/mlc and 6/sconv.
The function vu_is_used_negatively is called from 4/verb and 6/sconv.

§**4. Preposition usages.** And now almost the same code again as for verb usages.

```
preposition_usage *register_pu(vocabulary_entry *ve1, vocabulary_entry *ve2,
    vocabulary_entry *ve3, vocabulary_entry *ve4, vocabulary_entry *ve5,
    int assumes_player, binary_predicate *root) {
    preposition_usage *pu = CREATE(preposition_usage);
    pu->pu_text[0] = ve1; pu->pu_text[1] = ve2; pu->pu_text[2] = ve3;
    pu->pu_text[3] = ve4; pu->pu_text[4] = ve5;
    pu->pu_meaning = root; pu->implies_player = assumes_player;
    pu->is_same_property_as = NULL;
    pu->where_pu_created = set_where_created;
    pu->negated_sense = FALSE;
    pu->allow_unexpected_upper_case = FALSE;
    pu->implicitly_behave_as_noun = FALSE;
    vocab_set_flags(ve1, PREPOSITION_MC);
    return pu;
}
```

The function register_pu is called from 9/mapbp.

§**5.** And once again a slowish but rarely-needed search.

```
preposition_usage *find_pu(vocabulary_entry *ve1, vocabulary_entry *ve2,
    vocabulary_entry *ve3, vocabulary_entry *ve4, vocabulary_entry *ve5) {
    preposition_usage *pu;
    LOOP_OVER(pu, preposition_usage) {
        if (pu->pu_text[0] != ve1) continue;
        if (pu->pu_text[1] != ve2) continue;
        if (pu->pu_text[2] != ve3) continue;
        if (pu->pu_text[3] != ve4) continue;
        if (pu->pu_text[4] != ve5) continue;
        return pu;
    }
    return NULL;
}
```

§**6.** And again some access routines:

```
int pu_implies_player(preposition_usage *pu) {
    return pu->implies_player;
}
int pu_implicitly_negates(preposition_usage *pu) {
    return pu->negated_sense;
}
binary_predicate *pu_get_meaning(preposition_usage *pu) {
    return pu->pu_meaning;
}
int pu_implicitly_behaves_as_noun(preposition_usage *pu) {
    return pu->implicitly_behave_as_noun;
}
```

The function pu_implies_player is called from 5/varc.
The function pu_implicitly_negates is called from 6/sconv.
The function pu_get_meaning is called from 4/verb, 4/noun and 6/sconv.
The function pu_implicitly_behaves_as_noun is called from 5/varc.

§**7. Parsing source text against verb and preposition usages.** Given a particular VU, and a word range `w1` to `w2`, we test whether the range begins with but does not consist only of the text of the VU. We return the first word after the VU text if it does (which will therefore be a word number still inside the range), or −1 if it doesn't.

```
int vu_parse_text_against(int w1, int w2, verb_usage *vu) {
    int k, i;
    for (i=0, k=w1; i<MAX_WORDS_IN_USAGE; i++)
        if (vu->vu_text[i]) {
            if ((k==w2) || (vu->vu_text[i] != lw_array[k].lw_identity)) return -1;
            k++;
        }
    return k;
}
```

The function vu_parse_text_against is called from 2/prob3, 4/verb and 5/varc.

§**8.** More complicatedly, the same test for a prepositional usage. Sometimes we allow prepositions like "the same weight as", which refer to properties, and sometimes we don't. If we do, the bogus word `STROKE_V` is used to mark where the property name should appear in the text of the usage; this value is chosen since it can't ever occur in a preposition written by the user, being used otherwise only to mark paragraph breaks.

Note that, unlike the corresponding test for VUs, this one allows the usage to exactly fill the text `w1` to `w2` supplied, and in that case we return `w2+1`. This is needed because of prepositional usages which behave as if adjectives, like "carried", where the second term is missing because it is understood as meaning the player: "The briefcase is carried."

```
int pu_parse_text_against(int w1, int w2, preposition_usage *pu,
    int allow_sap) {
    int k, i;
    if ((pu->is_same_property_as) && (allow_sap == FALSE)) return -1;
    for (i=0, k=w1; i<MAX_WORDS_IN_USAGE; i++)
        if (pu->pu_text[i] == STROKE_V) {
            int pw1 = pu->is_same_property_as->word_ref1;
            int pw2 = pu->is_same_property_as->word_ref2;
            if (k + pw2 - pw1 > w2) return -1;
            if (compare_word_range(pw1, pw2, k, k+pw2-pw1) == FALSE) return -1;
            k += pw2 - pw1 + 1;
        } else if (pu->pu_text[i]) {
            if ((k > w2) || (pu->pu_text[i] != lw_array[k].lw_identity)) return -1;
            if ((pu->allow_unexpected_upper_case == FALSE) &&
                (unexpectedly_upper_case(k))) return -1;
            k++;
        }
    if ((k > w2) && (pu->implies_player == FALSE)) return -1;
    return k;
}
```

The function pu_parse_text_against is called from 4/verb, 4/noun and 5/varc.

§**9. Debug log.** The following dumps the entire stock of registered verb and preposition usages to the debugging log.

```
void log_verb_usage(verb_usage *vu) {
    int i;
    LOG("VU: ");
    for (i=0; i<MAX_WORDS_IN_USAGE; i++)
        if (vu->vu_text[i]) LOG("%s ", vocab_get_exemplar(vu->vu_text[i], FALSE));
    if (vu->negated_form_of_verb) LOG("(negated) ");
    log_tense_number(vu->tensed);
    LOG(" -> %s", bp_get_log_name(vu->vu_meaning));
}
void log_preposition_usage(preposition_usage *pu) {
    int i;
    LOG("PU: ");
    for (i=0; i<MAX_WORDS_IN_USAGE; i++)
        if (pu->pu_text[i]) LOG("%s ", vocab_get_exemplar(pu->pu_text[i], FALSE));
    if (pu->implies_player) LOG("(implies player) ");
    if (pu->is_same_property_as) LOG("(same $Y) ", pu->is_same_property_as);
    LOG("-> %s", bp_get_log_name(pu->pu_meaning));
}
void debug_verbs(void) {
    verb_usage *vu;
    preposition_usage *pu;
    LOG("The current S-grammar has the following verb and preposition usages:\n");
    LOOP_OVER(vu, verb_usage) {
        log_verb_usage(vu);
        LOG("\n");
    }
    LOOP_OVER(pu, preposition_usage) {
        log_preposition_usage(pu);
        LOG("\n");
    }
}
```

The function log_verb_usage is called from 5/ml.

The function log_preposition_usage is called from 5/ml.

The function debug_verbs is invoked by a command in a `.i6t` template file.

§10. Index tabulation. The following produces the table of verbs in the Phrasebook Index page.

```
void tabulate_verb(lexicon_entry *lex, int tense, char *tensename) {
    verb_usage *vu; int f = TRUE;
    LOOP_OVER(vu, verb_usage)
        if ((vu->under_main_verb == lex) && (vu_is_used_negatively(vu) == FALSE)
            && (vu_get_tense_used(vu) == tense)) {
            vocabulary_entry *lastword = vu->vu_text[4];
            if (lastword == NULL) lastword = vu->vu_text[3];
            if (lastword == NULL) lastword = vu->vu_text[2];
            if (lastword == NULL) lastword = vu->vu_text[1];
            if (lastword == NULL) lastword = vu->vu_text[0];
            if (f) INDEX("<p class=\"hang\">  <i>%s:</i> ",
                tensename);
            else INDEX("; ");
            f = FALSE;
            if (lastword == by_V) INDEX("B "); else INDEX("A ");
            if (vu->vu_text[0]) INDEX("%s ", vocab_get_exemplar(vu->vu_text[0], FALSE));
            if (vu->vu_text[1]) INDEX("%s ", vocab_get_exemplar(vu->vu_text[1], FALSE));
            if (vu->vu_text[2]) INDEX("%s ", vocab_get_exemplar(vu->vu_text[2], FALSE));
            if (vu->vu_text[3]) INDEX("%s ", vocab_get_exemplar(vu->vu_text[3], FALSE));
            if (vu->vu_text[4]) INDEX("%s ", vocab_get_exemplar(vu->vu_text[4], FALSE));
            if (lastword == by_V) INDEX("A"); else INDEX("B");
        }
    if (f == FALSE) INDEX("</p>\n");
}
```

The function tabulate_verb is called from 2/lexi.

§11. The built-in verbs and prepositions. Although we have created numerous built-in relations, very few verbs and prepositions are built in: those are created, instead, early on in the Standard Rules. (See Appendix A.) To justify the exceptions:

(a) "To be" is both grammatically irregular and also plays a special role in terms of having as its meaning the equality predicate, which behaves unlike all others. The syntax available to the Standard Rules can't construct this.

(b) "To have" is more surprising, given that there's nothing grammatically irregular about it. It's defined here just in order to come before the definition of "to be", because definition order occasionally breaks ties in parsing ambiguous sentences. (Because "to have" is sometimes used as an auxiliary verb, we can indeed find both in the same sentence – "Fred has been in the hut".) So although the Standard Rules could create "to have", they couldn't create it before the creation of "to be".

(c) Note that numerical comparisons are handled by two methods. Verbally, they are prepositions: "less than", for instance, is combined with "to be", giving us "A is less than B" and similar forms. These wordy forms are therefore defined as prepositional usages and created as such in the Standard Rules. But we also permit the use of the familiar mathematical symbols `<`, `>`, `<=` and `>=`. Inform treats these as verbs without tense, so registers them as verb usages, but without the full conjugation given to a conventional verb; and they are also excluded from the lexicon in the Phrasebook index, being notation rather than words. (This is why the variable `current_main_verb` is cleared.)

```
void make_built_in_verbs(void) {
    set_where_created = NULL;
    register_regular_verb(has_V, have_V, had_V, had_V, NULL, a_has_b_predicate, FALSE);
    register_to_be();
```

```
    current_main_verb = NULL;
    register_vu(LT_V, 0, 0, 0, 0, FALSE, IS_TENSE, a_lt_b_predicate);
    register_vu(GT_V, 0, 0, 0, 0, FALSE, IS_TENSE, a_gt_b_predicate);
    register_vu(LE_V, 0, 0, 0, 0, FALSE, IS_TENSE, a_le_b_predicate);
    register_vu(GE_V, 0, 0, 0, 0, FALSE, IS_TENSE, a_ge_b_predicate);
}
```

The function make_built_in_verbs is invoked by a command in a `.i6t` template file.

§**12. Registration of to be.** The English verb *to be* is the only one irregular enough to need its own special code to be created.

```
void register_to_be(void) {
    lexicon_new_main_verb(be_V, VERB_LEXE);
    negated_to_be = register_vu(is_V, not_V, 0, 0, 0, TRUE, IS_TENSE, a_is_b_predicate);
    regular_to_be = register_vu(is_V, 0, 0, 0, 0, FALSE, IS_TENSE, a_is_b_predicate);
    negated_plural_to_be = register_vu(are_V, not_V, 0, 0, 0, TRUE, IS_TENSE, a_is_b_predicate);
    register_vu(are_V, 0, 0, 0, 0, FALSE, IS_TENSE, a_is_b_predicate);
    register_vu(was_V, not_V, 0, 0, 0, TRUE, WAS_TENSE, a_is_b_predicate);
    register_vu(was_V, 0, 0, 0, 0, FALSE, WAS_TENSE, a_is_b_predicate);
    register_vu(were_V, not_V, 0, 0, 0, TRUE, WAS_TENSE, a_is_b_predicate);
    register_vu(were_V, 0, 0, 0, 0, FALSE, WAS_TENSE, a_is_b_predicate);
    register_vu(has_V, not_V, been_V, 0, 0, TRUE, HASBEEN_TENSE, a_is_b_predicate);
    register_vu(has_V, been_V, 0, 0, 0, FALSE, HASBEEN_TENSE, a_is_b_predicate);
    register_vu(have_V, not_V, been_V, 0, 0, TRUE, HASBEEN_TENSE, a_is_b_predicate);
    register_vu(have_V, been_V, 0, 0, 0, FALSE, HASBEEN_TENSE, a_is_b_predicate);
    register_vu(had_V, not_V, been_V, 0, 0, TRUE, HADBEEN_TENSE, a_is_b_predicate);
    register_vu(had_V, been_V, 0, 0, 0, FALSE, HADBEEN_TENSE, a_is_b_predicate);
}
```

§**13. Registration of regular verbs.** English forms verbs with reasonable regularity, and the following routine takes the standard conjugation of a verb (he likes, they like, etc.) and forms all its possible usages, both alone and in combination with the auxiliary verbs *to have* and *to do*. (*To have* also exists as a verb in its own right, and indeed can be used as its own auxiliary: "they had had...". As a primary verb, it is defined by the Standard Rules.)

```
void register_regular_verb(vocabulary_entry *present_singular,
    vocabulary_entry *present_plural, vocabulary_entry *past,
    vocabulary_entry *past_participle, vocabulary_entry *participle,
    binary_predicate *root, int participle_adjectival) {
    lexicon_new_main_verb(present_plural, VERB_LEXE);
    if (past_participle)
        ⟨To have plus past participle makes present and past perfect forms 14⟩;
    if (present_plural)
        ⟨To do plus infinitive makes auxiliary forms 15⟩;
    if (present_singular) register_vu(present_singular, 0, 0, 0, 0, FALSE, IS_TENSE, root);
    if (present_plural) register_vu(present_plural, 0, 0, 0, 0, FALSE, IS_TENSE, root);
    if (past) register_vu(past, 0, 0, 0, 0, FALSE, WAS_TENSE, root);
    if (past_participle)
        ⟨Make prepositional usages arising from past participle 16⟩;
    if (participle)
        ⟨Make prepositional usage arising from participle 17⟩;
}
```

§**14.** English forms the perfect tenses using *to have* as an auxiliary verb. (For instance, "has not fetched", "had fetched".)

⟨To have plus past participle makes present and past perfect forms 14⟩ ≡

```
register_vu(has_V, not_V, past_participle, 0, 0, TRUE, HASBEEN_TENSE, root);
register_vu(have_V, not_V, past_participle, 0, 0, TRUE, HASBEEN_TENSE, root);
register_vu(had_V, not_V, past_participle, 0, 0, TRUE, HADBEEN_TENSE, root);
register_vu(has_V, past_participle, 0, 0, 0, FALSE, HASBEEN_TENSE, root);
register_vu(have_V, past_participle, 0, 0, 0, FALSE, HASBEEN_TENSE, root);
register_vu(had_V, past_participle, 0, 0, 0, FALSE, HADBEEN_TENSE, root);
```

This code is used in §13.

§**15.** Because the verb is regular, its present plural is the same as its infinitive. (So the word "fetch" is the same in "They fetch" and "to fetch".)

⟨To do plus infinitive makes auxiliary forms 15⟩ ≡

```
register_vu(do_V, not_V, present_plural, 0, 0, TRUE, IS_TENSE, root);
register_vu(does_V, not_V, present_plural, 0, 0, TRUE, IS_TENSE, root);
register_vu(did_V, not_V, present_plural, 0, 0, TRUE, WAS_TENSE, root);
register_vu(do_V, present_plural, 0, 0, 0, FALSE, IS_TENSE, root);
register_vu(does_V, present_plural, 0, 0, 0, FALSE, IS_TENSE, root);
register_vu(did_V, present_plural, 0, 0, 0, FALSE, WAS_TENSE, root);
```

This code is used in §13.

§**16.** These prepositional usages reverse the sense of the relation. For instance, "to be fetched by" means the reversal of "to fetch". For a few verbs, the participle can also be used adjectivally with the implied understanding that the player is the missing term in the relation: for instance, "The hat is carried" is understood as if it were "The hat is carried by the player."

⟨Make prepositional usages arising from past participle 16⟩ ≡

```
binary_predicate *bpr = bp_get_reversal(root);
register_pu(past_participle, by_V, 0, 0, 0, FALSE, bpr);
if (participle_adjectival) register_pu(past_participle, 0, 0, 0, 0, TRUE, bpr);
```

This code is used in §13.

§**17.** With the present participle the meaning is back the right way around: for instance, "to be fetching" has the same meaning as "to fetch". At any rate, Inform's linguistic model is not subtle enough to distinguish the difference, in terms of a continuous rather than instantaneous process, which a human reader might be aware of.

⟨Make prepositional usage arising from participle 17⟩ ≡

```
register_pu(participle, 0, 0, 0, 0, FALSE, root);
```

This code is used in §13.

§**18. Registration of to be able to.** Finally, verbs taking the auxiliary *to be able to*: for instance, "to be able to follow".

```
void register_to_be_able_to_verb(vocabulary_entry *infinitive,
    vocabulary_entry *past_participle, binary_predicate *root) {
    if (infinitive == NULL) internal_error("Registered verb without infinitive");
    lexicon_new_main_verb(infinitive, ABLE_VERB_LEXE);
    if (past_participle)
        ⟨Have ever plus past participle for additional perfect forms 19⟩;
    if (past_participle)
        ⟨Can be plus past participle for imperfect forms 20⟩;
    register_vu(cannot_V, infinitive, 0, 0, 0, TRUE, IS_TENSE, root);
    register_vu(can_V, infinitive, 0, 0, 0, FALSE, IS_TENSE, root);
    register_vu(can_V, not_V, infinitive, 0, 0, TRUE, IS_TENSE, root);
    register_vu(could_V, not_V, infinitive, 0, 0, TRUE, WAS_TENSE, root);
    register_vu(could_V, infinitive, 0, 0, 0, FALSE, WAS_TENSE, root);
    ⟨Able or unable to plus infinitive or past participle for prepositional forms 21⟩;
}
```

§**19.** The following looks slightly questionable. After all, the past perfect form for "to be able to grasp" ought to be "had not been able to grasp", not "had never grasped". But "had not been able to grasp" is already created by the prepositional usage "able to grasp" made below; this one is an extra. It's here primarily because of "to be able to see", which is a curiosity of English. Seeing is so instantaneous that being able to see something and actually seeing it are more or less the same thing, and language takes note of this. So while "had never grasped" seems different in meaning from "had not been able to grasp", "had never seen" and "had not been able to see" are much closer.

⟨Have ever plus past participle for additional perfect forms 19⟩ ≡

```
    register_vu(has_V, never_V, past_participle, 0, 0, TRUE, HASBEEN_TENSE, root);
    register_vu(have_V, never_V, past_participle, 0, 0, TRUE, HASBEEN_TENSE, root);
    register_vu(had_V, never_V, past_participle, 0, 0, TRUE, HADBEEN_TENSE, root);
    register_vu(has_V, ever_V, past_participle, 0, 0, FALSE, HASBEEN_TENSE, root);
    register_vu(have_V, ever_V, past_participle, 0, 0, FALSE, HASBEEN_TENSE, root);
    register_vu(had_V, ever_V, past_participle, 0, 0, FALSE, HADBEEN_TENSE, root);
```

This code is used in §18.

§**20.** For instance, "cannot be seen by", which reverses the relation: "X cannot be seen by Y" has the same meaning as "Y cannot see X".

⟨Can be plus past participle for imperfect forms 20⟩ ≡

```
    binary_predicate *bpr = bp_get_reversal(root);
    register_vu(cannot_V, be_V, past_participle, by_V, 0, TRUE, IS_TENSE, bpr);
    register_vu(can_V, not_V, be_V, past_participle, by_V, TRUE, IS_TENSE, bpr);
    register_vu(can_V, be_V, past_participle, by_V, 0, FALSE, IS_TENSE, bpr);
    register_vu(could_V, not_V, be_V, past_participle, by_V, TRUE, WAS_TENSE, bpr);
    register_vu(could_V, be_V, past_participle, by_V, 0, FALSE, WAS_TENSE, bpr);
```

This code is used in §18.

§**21.** "To be able to see", "to be unable to see", "to be able to be seen by", "to be unable to be seen by".

⟨Able or unable to plus infinitive or past participle for prepositional forms 21⟩ ≡

```
register_pu(able_V, to_V, infinitive, 0, 0, FALSE, root);
register_pu(unable_V, to_V, infinitive, 0, 0, FALSE, root)->negated_sense = TRUE;
if (past_participle) {
    binary_predicate *bpr = bp_get_reversal(root);
    register_pu(able_V, to_V, be_V, past_participle, by_V, FALSE, bpr);
    register_pu(unable_V, to_V, be_V, past_participle, by_V, FALSE, bpr)
        ->negated_sense = TRUE;
}
```

This code is used in §18.

§**22. Late registration of prepositions comparing properties.** The following routines, used only when all the properties have been created, make suitable comparatives ("bigger than", etc.) and prepositional usages to test property-equality ("the same height as").

```
void register_comparative_preposition(int wn, binary_predicate *root) {
    set_where_created = current_sentence;
    register_pu(lw_array[wn].lw_identity, than_V, 0, 0, 0, FALSE, root);
}
void register_same_property_as_preposition(binary_predicate *root) {
    set_where_created = current_sentence;
    preposition_usage *pu =
        register_pu(the_V, same_V, STROKE_V, as_V, 0, FALSE, root);
    pu->is_same_property_as = bp_get_same_as_property(root);
}
```

The function register_comparative_preposition is called from 9/cmpbp.
The function register_same_property_as_preposition is called from 9/vpbp.

§**23. Parsing new verb declarations.** In addition to the built-in stock, new verbs can be declared from the source text. This is where such text is parsed and acted upon. Before we get here, the sentence is known to have the form:

> The verb to ... implies ...

and the parse node has two children, for the text before and after the primary verb "implies".

```
void parse_new_verb(parse_node *PN) {
    int w1, w2, rw1, rw2, parts1, parts2, i;
    binary_predicate *bp = NULL;
    int to_be_able_to = FALSE;
    set_where_created = current_sentence;
    [[rw1, rw2 <-- PN->down->next->next]];
    ⟨Find the underlying relation of the new verb or preposition 24⟩;
    [[w1, w2 <-- PN->down->next]];
    [[w1, w2 == the verb ... --> w1, w2]];
    parts1 = -1; parts2 = -1;
    if [[w1, w2 == ... OPENBRACKET ... CLOSEBRACKET : i --> w1, w2 ... parts1, parts2]] ;
    if [[w1, w2 == to be]] {
        sentence_problem(_P_(C5VerbRedefinesBe),
            "the verb 'to be' cannot be defined again",
```

```
            "though prepositional forms of it can ('The verb to be alongside ...').");
        return;
    } else if [[w1, w2 == to be able to ...]] {
        w1 = w1 + 3;                                    to skip over the words "to be able"
        to_be_able_to = TRUE;
        ⟨Definition makes a new verb usage 27⟩;
    } else if [[w1, w2 == to be ... --> w1, w2]] {
        ⟨Definition makes a new prepositional usage 34⟩;
    } else if [[w1, w2 == to ...]] {
        to_be_able_to = FALSE;
        ⟨Definition makes a new verb usage 27⟩;
    } else internal_error("Verb without infinitive");
}
```

The function parse_new_verb is called from 4/verb.

§24. I: Semantics.

⟨Find the underlying relation of the new verb or preposition 24⟩ ≡

```
    int reversal_flag = FALSE;
    [[rw1, rw2 == the ... --> rw1, rw2]];
    if [[rw1, rw2 == relation]] {
        sentence_problem(_P_(C5VerbRelationVague),
            "that's too vague",
            "calling a relation simply 'relation'.");
        return;
    }
    if [[rw1, rw2 == ... property --> rw1, rw2]]
        ⟨Find the BP corresponding to setting this property 25⟩
    else if [[rw1, rw2 == ... relation --> rw1, rw2]]
        ⟨Find the BP corresponding to this named relation 26⟩
    else {
        sentence_problem(_P_(C5VerbUnknownMeaning),
            "new verbs can only be defined in terms of existing properties "
            "or relations",
            "so the last word should be 'property' or 'relation': thus "
            "'...implies the possession relation' is an example of a valid "
            "definition, this being one of the relations built into Inform.");
        return;
    }
```

This code is used in §23.

§25. The following perhaps looks odd. Why do we delegate to `make_set_property_BP` the business of parsing the property name? What happens if the user typed

> The verb to be mystified by implies the arfle barfle gloop property.

when there is no property of that name? The answer is that we can't check this yet, because verb definitions are read long before properties come into existence. The check will be made later on, and the BP generated by the following call is really only provisional until it is. So for now absolutely any non-empty word range is accepted as the property name.

⟨Find the BP corresponding to setting this property 25⟩ ≡

```
    bp = make_set_property_BP(rw1, rw2);
```

This code is used in §24.

§26.

⟨Find the BP corresponding to this named relation 26⟩ ≡
```
if [[rw1, rw2 == reversed ... --> rw1, rw2]] reversal_flag = TRUE;
if (rw2 > rw1) {
    sentence_problem(_P_(C5VerbRelationLong),
        "that's too long for a relation",
        "whose name has to be a single word followed by 'relation'.");
    return;
}
bp = parse_relation_name(rw1);
if (bp == NULL) {
    sentence_problem(_P_(C5VerbRelationUnknown),
        "new verbs can only be defined in terms of existing relations",
        "all of which have names ending 'relation': thus '...implies the "
        "possession relation' is an example of a valid definition, this "
        "being one of the relations built into Inform.");
    return;
}
if (reversal_flag) bp = bp_get_reversal(bp);
```

This code is used in §24.

§27. IIa: Syntax of a new verb usage.

⟨Definition makes a new verb usage 27⟩ ≡
```
int can_be_used_adjectivally = FALSE;
vocabulary_entry *infinitive = lw_array[w1+1].lw_identity;
vocabulary_entry *present_singular = NULL;
vocabulary_entry *present_plural = NULL;
vocabulary_entry *past = NULL;
vocabulary_entry *past_participle = NULL;
vocabulary_entry *participle = NULL;
⟨Reject with a problem message if verb has no auxiliary and is not conjugated 28⟩;
⟨Parse the parts of speech supplied for the verb 29⟩;
⟨Reject with a problem message if this verb has been defined before 32⟩;
⟨Register the new verb 33⟩;
```

This code is used in §23.

§28.

⟨Reject with a problem message if verb has no auxiliary and is not conjugated 28⟩ ≡
```
if ((parts1 < 0) && (to_be_able_to == FALSE)) {
    sentence_problem(_P_(C5VerbWithoutParts),
        "new verbs can't be defined with infinitive alone",
        "and should have all parts supplied: for instance, 'The verb "
        "to sport (he sports, they sport, he sported, it is sported, "
        "he is sporting) ...'");
    return;
}
```

This code is used in §27.

§**29.** We read the parts of speech as a comma-separated list of individual parts (but we don't allow "and" or "or" to divide this list: only commas).

At the end, if no present plural is supplied, we may as well use the infinitive for that – the two are the same in most regular English verbs ("to sleep", "they sleep") even if not irregular ones ("to be", "they are").

⟨Parse the parts of speech supplied for the verb 29⟩ ≡

```
int more_to_read = TRUE;
while (more_to_read) {
    int c1 = parts1, c2 = parts2, i;
    if [[c1, c2 == ... COMMA ... : i --> c1, c2 ... parts1, parts2]]
        more_to_read = TRUE;
    else
        more_to_read = FALSE;
    ⟨Parse the part of speech in this clause 30⟩;
}
if (present_plural == NULL) present_plural = infinitive;
```

This code is used in §27.

§**30.** Note that the suffix "-ing" is used to distinguish the present participle ("he is grabbing") from the past ("he is grabbed").

⟨Parse the part of speech in this clause 30⟩ ≡

```
int number = 0;
if [[c1, c2 == he/she/it ... --> c1, c2]] number = 1;
else if [[c1, c2 == they ... --> c1, c2]] number = 2;
else ⟨Give up on verb definition as malformed 31⟩;
int is_a_participle = FALSE;
if [[c1, c2 == is ... --> c1, c2]] is_a_participle = TRUE;
if [[c1, c2 == ... OPENBRACKET adjectival CLOSEBRACKET --> c1, c2]]
    can_be_used_adjectivally = TRUE;
if (c1 != c2) ⟨Give up on verb definition as malformed 31⟩;
if (is_a_participle) {
    if (vocab_test_flags(c1, ING_MC)) participle = lw_array[c1].lw_identity;
    else past_participle = lw_array[c1].lw_identity;
} else {
    if (number == 2) {
        if (present_plural) {
            sentence_problem(_P_(C5PresentPluralTwice),
                "the present plural has been given twice",
                "since two of the principal parts of this verb begin "
                "with 'they'.");
        }
        present_plural = lw_array[c1].lw_identity;
    } else {
        if (present_singular) past = lw_array[c1].lw_identity;
        else present_singular = lw_array[c1].lw_identity;
    }
}
```

This code is used in §29.

§**31.** A catch-all problem message:

⟨Give up on verb definition as malformed 31⟩ ≡
```
sentence_problem(_P_(C5VerbMalformed),
    "this verb's definition is malformed",
    "and should have its principal parts supplied like so: 'The verb "
    "to sport (he sports, they sport, he sported, it is sported, "
    "he is sporting) ...' As in this example, the verb must be a "
    "single word, so 'to sport' is allowed but not 'to run over'.");
return;
```

This code is used in §30.

§**32.** As the slightly evasive wording of these problem messages suggests, it's not so much a matter of the new verb coinciding with an old one as overlapping with it. We possibly reject more than we need to, but it seems wise to be cautious.

⟨Reject with a problem message if this verb has been defined before 32⟩ ≡
```
verb_usage *duplicate_vu = NULL;
if (to_be_able_to)
    duplicate_vu = find_vu(able_V, to_V, infinitive, 0, 0);
else
    duplicate_vu = find_vu(present_singular, 0, 0, 0, 0);
if (duplicate_vu) {
    if (duplicate_vu->where_vu_created)
        two_sentences_problem(_P_(C5DuplicateVerbs1),
            duplicate_vu->where_vu_created,
            "this gives us two definitions of what appears to be the same verb",
            "or at least has the same infinitive form.");
    else
        sentence_problem(_P_(C5DuplicateVerbs2),
            "this verb definition appears to clash with a built-in verb",
            "a table of which can be seen on the Phrasebook index.");
    return;
}
```

This code is used in §27.

§**33.** And the happy ending:

⟨Register the new verb 33⟩ ≡
```
if (to_be_able_to)
    register_to_be_able_to_verb(infinitive, past_participle, bp);
else
    register_regular_verb(present_singular, present_plural, past,
        past_participle, participle, bp, can_be_used_adjectivally);
```

This code is used in §27.

§34. IIb: Syntax of a new prepositional usage. This runs along much the same lines.

⟨Definition makes a new prepositional usage 34⟩ ≡
```
int unexpected_upper_casing_used = FALSE, noun_like = FALSE;
vocabulary_entry *word_one = NULL, *word_two = NULL, *word_three = NULL;
⟨Reject with a problem message if preposition is conjugated 35⟩;
⟨Find the one to three words making up the preposition 36⟩;
⟨Determine if unexpected upper casing is used in preposition 37⟩;
⟨Reject with a problem message if this preposition has been defined before 38⟩;
⟨Register the new preposition 39⟩;
```

This code is used in §23.

§35. This funny little problem message is the price we pay for blurring grammar in the syntax provided for users. They shouldn't really use the same sentence form to define prepositions as they do to define verbs: but it's easy to remember, and convenient, and no alternative seems better, so we go along with it, and allow

The verb to be beneath implies ...

even though this is not the definition of a verb at all, and it is only "beneath" which is being defined. Prepositions do not inflect in English when used in different tenses or when negated, so there's no conjugation involved, and we need to reject any attempt – even though it would be perfectly valid if a verb were being defined.

⟨Reject with a problem message if preposition is conjugated 35⟩ ≡
```
if (parts1 >= 0) {
    sentence_problem(_P_(C5PrepositionConjugated),
        "the principal parts of 'to be' are known already",
        "so should not be spelled out again as part of the instructions "
        "for this new preposition.");
    return;
}
```

This code is used in §34.

§36.

⟨Find the one to three words making up the preposition 36⟩ ≡
```
if (w2>w1+2) {
    sentence_problem(_P_(C5PrepositionLong),
        "prepositions can only be up to three words long",
        "so 'to be very roughly alongside' is allowed, but 'to be "
        "frankly far away from' is not.");
    return;
}
if [[w1, w2 == part/parts of]] noun_like = TRUE;
word_one = lw_array[w1].lw_identity;
if (w2>w1) word_two = lw_array[w1+1].lw_identity;
if (w2>w1+1) word_three = lw_array[w1+2].lw_identity;
```

This code is used in §34.

§**37.** Casing problems are acutely problematic with prepositions, because so many locations have names which begin with them – "Under Milkwood", "Inside the Machine", "On Top of Old Smoky". Our best way to avoid confusion is to read prepositions as such only when they do not unexpectedly jump into upper case, i.e., to distinguish between the meanings of

X is in Bahrain. Y is In Bahrain.

according to the unexpected capital I in the second "In". But just occasionally people do want to define prepositions which genuinely involve an unexpected upper-case letter; and those we flag for special treatment, since otherwise they could never be parsed successfully.

⟨Determine if unexpected upper casing is used in preposition 37⟩ ≡

```
if (unexpectedly_upper_case(w1)) unexpected_upper_casing_used = TRUE;
if ((w2>w1) && (unexpectedly_upper_case(w1+1))) unexpected_upper_casing_used = TRUE;
if ((w2>w1+1) && (unexpectedly_upper_case(w1+2))) unexpected_upper_casing_used = TRUE;
```

This code is used in §34.

§**38.** And once again:

⟨Reject with a problem message if this preposition has been defined before 38⟩ ≡

```
preposition_usage *duplicate_pu = NULL;
duplicate_pu = find_pu(word_one, word_two, word_three, 0, 0);
if (duplicate_pu) {
    if (duplicate_pu->where_pu_created)
        two_sentences_problem(_P_(C5DuplicateVerbs3),
            duplicate_pu->where_pu_created,
            "this gives us two definitions of what appears to be the "
            "same prepositional form",
            "or at least has the same words in.");
    else
        sentence_problem(_P_(BelievedImpossible),                    none are built in nowadays
            "this definition appears to clash with a built-in "
            "prepositional form",
            "a list of which can be seen on the Phrasebook index.");
    return;
}
```

This code is used in §34.

§**39.** And the happy ending:

⟨Register the new preposition 39⟩ ≡

```
preposition_usage *pu = register_pu(word_one, word_two, word_three, 0, 0, FALSE, bp);
pu->allow_unexpected_upper_case = unexpected_upper_casing_used;
pu->implicitly_behave_as_noun = noun_like;
```

This code is used in §34.

Adjectives 5/aph

Purpose

To identify the names of all adjectives used in source text, and to organise the multiple different senses each adjective might have.

Definitions

¶1. Chapter 5's tour of linguistic categories has already taken in determiners, verbs and prepositions. Next we cover adjectives, which will turn out to be simpler; subsequent sections will cover nouns, which will turn out not to be so easy.

Adjectives are simpler than verbs since they define unary rather than binary predicates. The word "open" applies to only one term – logically, we regard it as *open*(x), whereas a verb like "suspects" would appear in formulae as *suspects*(x, y).

But they are more complicated than verbs in that Inform allows multiple definitions of the same adjective. For instance, two of the senses of "empty" in the Standard Rules are:

> Definition: a text is empty rather than non-empty if it is "".
>
> Definition: a table-name is empty rather than non-empty if the number of filled rows in it is 0.

(Which also defines two of the senses of "non-empty", another adjective.) The clause *empty*(x) can be fully understood only when we know what kind of value x has; for a text, the first sense applies, and for a table-name, the second.

Instances of the `adjectival_phrase` structure correspond exactly to names of adjectives. Thus a single `adjectival_phrase` exists for "empty", and another for "non-empty". The structure records just one thing: a linked list of `adjective_meaning` structures, of which there is one for each sense. (In all, the Standard Rules define six senses of "empty", each one getting its own `adjective_meaning` structure.)

We eventually need to sort this list of definitions into logical priority order – so that a definition applying to just Count Dracula precedes one applying to men, which in turn precedes one applying to things. (Priority order is irrelevant when two senses apply to domains with no overlap, as in the case of texts and table-names.) It's convenient and costs little memory to keep the sorted list as a second linked list.

```
typedef struct adjectival_phrase {
    int word_ref1, word_ref2;                                  text of what is defined
    struct adjective_meaning *possible_meanings;               list of definitions in order given
    struct adjective_meaning *sorted_meanings;                 the same list sorted into logical order
    MEMORY_MANAGEMENT
} adjectival_phrase;
```

The structure adjectival_phrase is private to this section.

¶2. Each individual sense of an adjective, then, has its own `adjective_meaning` structure, which we define next. It consists of some logistical data to keep its place in the linked lists (see above), some data to specify its domain (see below), some indexing data which is not very important, and then the crucial part: its "detailed meaning".

The general model is that adjective meanings come in different "kinds", for which specific code is scattered across Inform. In each case, the `detailed_meaning` points to an appropriate data structure, and specialised routines are called to create and use the adjective.

We can also specify that the meaning implied by this pointer is to be understood reversely: that the adjective is the negation of the one specified. This enables "non-empty" for texts (say) to be defined identically with "empty" for texts, but with the `meaning_parity` flag set to `FALSE` rather than `TRUE`.

```
define CONDITION_KADJ 1                                  defined by a condition in I7 source text
define PHRASE_KADJ 2                                     defined by an explicit but nameless rule
define I6_ROUTINE_KADJ 3                                 defined by a named I6 routine
define I6_CONDITION_KADJ 4                               defined by an explicit I6 schema
define MEASUREMENT_KADJ 5                                defined by numerical comparison with a property value
define ENUMERATIVE_KADJ 6                                defined by a property like "colour" with named values
define EORP_KADJ 7                                       defined by an either/or property like "closed"

typedef struct adjective_meaning {
    int index_w1, index_w2;                              text to use in the Phrasebook index
    struct parse_node *defined_at;                       from what sentence this came (if it did)
    struct adjectival_phrase *owning_adjective;          of which this is a definition
    struct adjective_meaning *next_meaning;              next in order of definition
    struct adjective_meaning *next_sorted;               next in logically sorted order
    int definition_domain_w1, definition_domain_w2;      domain to which defn applies
    struct kind_of_value *domain_kov;                    resolved into a KOV
    struct world_object *domain_wo;                      used only for specific objects as domain
    int problems_thrown;                                 complaining about the domain of this adjective
    int meaning_parity;                                  meaning understood positively?
    struct adjective_meaning *am_negated_from;           if explicitly constructed as such
    int adjective_form;                                  one of the *_KADJ constants: see below
    general_pointer detailed_meaning;                    to the relevant structure
    int task_via_support_routine[NO_ADJECTIVE_TASKS + 1];
    struct i6_schema i6s_to_transfer_to_SR[NO_ADJECTIVE_TASKS + 1];   where TRUE
    struct i6_schema i6s_for_runtime_task[NO_ADJECTIVE_TASKS + 1];    where TRUE
    int am_ready_flag;                                   optional flag to mark whether schemas prepared yet
    int defined_already;                                 temporary workspace used when compiling support routines
    MEMORY_MANAGEMENT
} adjective_meaning;
```

The structure adjective_meaning is private to this section.

¶3. What are adjectives for? Since an adjective is a unary predicate, it can be thought of as an assignment from its domain set to the set of two possibilities: true, false. Thus one sense of "open" maps doors to true if they are currently open, false if they are closed.

There are altogether five things we might want to do with an adjective:

(1) Test whether it is true at any given point during play.
(2) Assert that it is true at the start of play.
(3) Assert that it is false at the start of play.
(4) Assert that it is now to be true from this point on during play.
(5) Assert that it is now to be false from this point on during play.

We do not need to test whether it is false, since we need only test whether it is true and negate the result.

Adjectives for which all five of these operations can be carried out are the exception rather than the rule. "Open" is an example:

[1] if the marble door is open, ...

[2] The marble door is open.

[3] The marble door is not open.

[4] now the marble door is open;

[5] now the marble door is not open;

Every adjective in practice supports (1), testing for truth, but this is not required by the code below. Many adjectives – properly speaking, many senses of an adjective – only support testing: "empty" in the sense of texts, for instance.

Of the five possibilities, (1), (4) and (5) happen at run-time. These are called "tasks" and are identified by the following constants. While in theory an adjective's handling code can compile anything it likes to carry out these tasks, in practice most are defined by providing an I6 schema, which is why the `adjective_meaning` structure contains these – see below.

```
define NO_ADJECTIVE_TASKS 3
define TEST_ADJECTIVE_TASK 1                    test if currently true
define NOW_ADJECTIVE_TRUE_TASK 2                assert now true
define NOW_ADJECTIVE_FALSE_TASK 3               assert now false
```

¶4. For indexing (only) we need to run through the definitions of a given adjectival phrase in sorted order, so:

```
define LOOP_OVER_SORTED_MEANINGS(aph, am)
    for (am = aph_get_sorted_definition_list(aph); am; am=am->next_sorted)
```

¶5. Two adjectival phrases are special, and are needed by the Creator to ensure that newly made objects have the right naming.

```
adjectival_phrase *plural_named_aph = NULL;
adjectival_phrase *proper_named_aph = NULL;
```

---

§**1. Adjectival phrases.** These are registered as excerpt meanings with the `ADJECTIVE_MC` meaning code, so parsing a word range to match an adjective is easy. By construction there is only one `adjectival_phrase` for any given excerpt of text, so the following is unambiguous:

```
adjectival_phrase *aph_parse(int w1, int w2) {
    meaning_list *ml = SP_excerpt(ADJECTIVE_MC, w1, w2);
    if (ml == NULL) return NULL;
    return RETRIEVE_POINTER_adjectival_phrase(em_data(ml_meaning(ml)));
}
```

The function aph_parse is called from 12/cinv.

§**2.** Given a word range, we return its corresponding APH, creating it if necessary:

```
adjectival_phrase *aph_from_word_range(int w1, int w2) {
    adjectival_phrase *aph;
    aph = aph_parse(w1, w2);
    if (aph) return aph;
    aph = CREATE(adjectival_phrase);
    aph->word_ref1 = w1; aph->word_ref2 = w2;
    if [[w1, w2 == plural_named]] plural_named_aph = aph;
    if [[w1, w2 == proper_named]] proper_named_aph = aph;
    aph->possible_meanings = NULL; aph->sorted_meanings = NULL;
    if ((w1 == w2) && (vocab_test_flags(w1, ARTICLE_MC))) {
        sentence_problem(_P_(C5ArticleAsAdjective),
            "a defined adjective cannot consist only of an article such as "
            "'a' or 'the'",
            "since this will lead to parsing ambiguities.");
    } else
    register_excerpt_meaning(ADJECTIVE_MC, 0, w1, w2, STORE_POINTER_adjectival_phrase(aph));
    LOGIF(QUANTITY_CREATIONS, "Created adjectival phrase: $a\n", aph);
    return aph;
}
```

§**3. The list of definitions.** The following assigns a new meaning to a given word range: we find the appropriate APH (creating if necessary) and then add the new meaning to the end of its unsorted meaning list.

```
adjectival_phrase *am_declare(adjective_meaning *am, int w1, int w2) {
    adjectival_phrase *aph = aph_from_word_range(w1, w2);
    adjective_meaning *aml = aph->possible_meanings;
    if (aml == NULL) aph->possible_meanings = am;
    else {
        while (aml->next_meaning) aml = aml->next_meaning;
        aml->next_meaning = am;
    }
    am->next_meaning = NULL;
    am->owning_adjective = aph;
    return aph;
}
```

The function am_declare is called from 9/qty, 9/prop, 9/madj and 12/def.

§**4.** Once declared, an AM stays with the same APH for the whole of Inform's run, and it can only be declared once. So every AM belongs to one and only one APH, which we can read off as follows:

```
adjectival_phrase *am_get_aph(adjective_meaning *am) {
    return am->owning_adjective;
}
```

The function am_get_aph is called from 9/prop.

§**5.** After meanings have been declared, a typical APH will have a disordered "possible meaning" list and an empty "sorted meaning" list. The following sorts the possibles list into the sorted list.

```
void aph_sort_meanings(adjectival_phrase *aph) {
    if (aph == NULL) internal_error("tried to sort meanings for null APH");
    aph->sorted_meanings = am_list_sort(aph->possible_meanings);
}
adjective_meaning *am_list_sort(adjective_meaning *unsorted_head) {
    adjective_meaning *sorted_head = NULL;
    adjective_meaning *am, *am2;
    for (am = unsorted_head; am; am = am->next_meaning)
        if (am->domain_kov == NULL)
            am_set_definition_domain(am, FALSE);
    for (am = unsorted_head; am; am = am->next_meaning) {
        if (sorted_head == NULL) {
            sorted_head = am;
            am->next_sorted = NULL;
        } else {
            adjective_meaning *lastdef = NULL;
            for (am2 = sorted_head; am2; am2 = am2->next_sorted) {
                if (compare_ams(am, am2) == 1) {
                    if (lastdef == NULL) {
                        sorted_head = am;
                        am->next_sorted = am2;
                    } else {
                        lastdef->next_sorted = am;
```

```
                    am->next_sorted = am2;
                }
                break;
            }
            if (am2->next_sorted == NULL) {
                am2->next_sorted = am;
                am->next_sorted = NULL;
                break;
            }
            lastdef = am2;
        }
    }
}
return sorted_head;
}
```

§**6.** And voilà, the result can be read here:

```
adjective_meaning *aph_get_sorted_definition_list(adjectival_phrase *aph) {
    return aph->sorted_meanings;
}
```

§**7. Logging.** To identify an adjective in the debugging log:

```
void log_adjectival_phrase(adjectival_phrase *aph) {
    if (aph == NULL) { LOG("<null adjectival phrase>"); return; }
    if (logging_to_I6_text) LOG("'$W'", aph->word_ref1, aph->word_ref2);
    else LOG("A%d'$W'", aph->allocation_id, aph->word_ref1, aph->word_ref2);
}
```

The function log_adjectival_phrase is called from 2/dl and 6/aprop.

§**8.** And here we log the unsorted meaning list.

```
void aph_log_meanings(adjectival_phrase *aph) {
    adjective_meaning *am;
    int n;
    if (aph == NULL) { LOG("<null-APH>\n"); return; }
    for (n=1, am = aph->possible_meanings; am; n++, am = am->next_meaning)
        LOG("%d: $W ($u; $O)\n", n, am->index_w1, am->index_w2,
            am->domain_kov, am->domain_wo);
}
```

§**9. Individual meanings.** So you want to define a new meaning for an adjective? Here's the procedure:

(1) Call `am_new` to create it. The `form` should be one of the `*_KADJ` constants, and the `details` should contain a pointer to the data structure it uses. The word range is used for indexing only.
(2) Call `am_declare` to associate it with a given adjective name, and thus have it added to the possible meanings list of the appropriate APH.
(3) Give it a domain of definition (see below).
(4) Optionally, give it explicit I6 schemas for testing and asserting (see below) – this makes coding what the adjective compiles to much easier.

```
adjective_meaning *am_new(int form, general_pointer details, int ix1, int ix2) {
    int i;
    adjective_meaning *am = CREATE(adjective_meaning);
    am->defined_at = current_sentence;
    am->index_w1 = ix1; am->index_w2 = ix2;
    am->owning_adjective = NULL;
    am->next_meaning = NULL;
    am->next_sorted = NULL;
    am->definition_domain_w1 = -1; am->definition_domain_w2 = -1;
    am->domain_kov = NULL; am->domain_wo = NULL;
    am->adjective_form = form;
    am->detailed_meaning = details;
    am->defined_already = FALSE;
    am->problems_thrown = 0;
    am->meaning_parity = TRUE;
    am->am_ready_flag = FALSE;
    for (i=1; i<=NO_ADJECTIVE_TASKS; i++) {
        am->task_via_support_routine[i] = NOT_APPLICABLE;
        sch_write_to_existing(&(am->i6s_for_runtime_task[i]), "");
        sch_write_to_existing(&(am->i6s_to_transfer_to_SR[i]), "");
    }
    am->am_negated_from = NULL;
    return am;
}
```

The function am_new is called from 9/qty, 9/prop, 9/madj and 12/def.

§**10.** Negating an AM. If you want to define an adjective as the logical negation of an existing one, take any AM which has been through stages (1) to (4) and then apply `am_negate` to create a new AM. Then use `am_declare` to associate this with a (presumably different) name, but there's no need to specify its I6 schemas or its domain – those are inherited.

```
adjective_meaning *am_negate(adjective_meaning *am) {
    int i;
    adjective_meaning *neg = CREATE(adjective_meaning);
    neg->defined_at = current_sentence;
    neg->index_w1 = am->index_w1; neg->index_w2 = am->index_w2;
    neg->owning_adjective = NULL;
    neg->next_meaning = NULL;
    neg->next_sorted = NULL;
    neg->definition_domain_w1 = am->definition_domain_w1;
    neg->definition_domain_w2 = am->definition_domain_w2;
    neg->domain_kov = am->domain_kov; neg->domain_wo = am->domain_wo;
    neg->adjective_form = am->adjective_form;
```

```
    neg->detailed_meaning = am->detailed_meaning;
    neg->defined_already = FALSE;
    neg->problems_thrown = 0;
    neg->am_ready_flag = FALSE;
    neg->meaning_parity = (am->meaning_parity)?FALSE:TRUE;
    for (i=1; i<=NO_ADJECTIVE_TASKS; i++) {
        int j = i;
        if (i == NOW_ADJECTIVE_TRUE_TASK) j = NOW_ADJECTIVE_FALSE_TASK;
        if (i == NOW_ADJECTIVE_FALSE_TASK) j = NOW_ADJECTIVE_TRUE_TASK;
        neg->task_via_support_routine[j] = am->task_via_support_routine[i];
        neg->i6s_for_runtime_task[j] = am->i6s_for_runtime_task[i];
        sch_write_to_existing(&(neg->i6s_to_transfer_to_SR[j]), "");
    }
    neg->am_negated_from = am;
    return neg;
}

int am_get_form(adjective_meaning *am) {
    if (am == NULL) return -1;
    return am->adjective_form;
}
```

The function am_negate is called from 12/def.
The function am_get_form is called from 12/def.

§**11. The domain of validity.** Every AM has a clearly defined range of values or objects to which it applies. This can be either a single object, a kind of object, or a kind of value. (It cannot be a single value.)

There are two fields specifying the domain, `domain_kov` and `domain_wo`. Two are needed because a single object is allowed as a domain of an adjective even though it cannot on its own be a KOV. So

(a) either the domain is expressed exactly by `domain_kov` and `domain_wo` is null, or
(b) the domain is a single object (not a kind) in `domain_wo` and `domain_kov` holds the smallest KOV which contains this, that is, the KOV corresponding to its kind.

For example, "odd" for numbers has `domain_kov` equal to `kova(NUMBER_TY)` and `domain_wo` null, while the sense of "odd" created by

> Mrs Elspeth Spong can be odd, eccentric or mildly dotty.

would have `domain_wo` equal to the world object for Mrs Spong and `domain_kov` equal to `kovko(kind_women)`, representing the set of all women.

§**12.** In comparing and testing domains, we use two different levels of matching: weak and strong.

Strong checking makes an exact match, but weak checking blurs the definitions so that two domains are counted as equal if they are close enough that run-time type checking can be used to tell them apart.

In general, any two base KOVs are different even in weak checking – "scene" and "number", for instance. On the other hand, "list of scenes" weakly matches "list of numbers", and "container" weakly matches "animal". As this last example shows, two domains can be completely disjoint and still make a weak match.

```
int domain_weak_match(kind_of_value *kov1, kind_of_value *kov2) {
    if (kov_I6_ID(kov1) == kov_I6_ID(kov2)) return TRUE;
    return FALSE;
}
```

§**13.** Whereas the following makes a strict check of whether a given object is within the domain of an adjective meaning.

```
int domain_wo_compare(world_object *wo, adjective_meaning *am) {
    if (wo == NULL) return TRUE;
    if (am->domain_wo) { if (wo == am->domain_wo) return TRUE; return FALSE; }
    if (kovko_get_kind(am->domain_kov) == NULL) return TRUE;
    if (wo->kind == NULL) return TRUE;
    if ((wo->kind_flag) && (wo == kovko_get_kind(am->domain_kov))) return TRUE;
    if (wo_of_kind(wo, kovko_get_kind(am->domain_kov))) return TRUE;
    return FALSE;
}
```

§**14. Specifying the domain of a new AM.** In principle the domain should be set as soon as the AM is created, but in practice some AMs – those coming from properties – might need to be created very early in Inform's run, at a time when objects and kinds of object do not exist. For those cases, an alternative is to give a word range – "a number", say, or "a container" – and if necessary this is left until later on in the run to parse. (For "a number", it wouldn't be necessary; for "a container", it would.)

To set the domain, call exactly one of the following three routines:

```
void am_set_domain_from_word_range(adjective_meaning *am, int w1, int w2) {
    am->domain_kov = NULL; am->domain_wo = NULL;
    am->definition_domain_w1 = w1; am->definition_domain_w2 = w2;
    am_set_definition_domain(am, TRUE);
}
void am_set_domain_from_wo(adjective_meaning *am, world_object *wo) {
    am->domain_kov = NULL; am->domain_wo = NULL;
    am->definition_domain_w1 = -1; am->definition_domain_w2 = -1;
    if (wo == NULL) am->domain_kov = kova(OBJECT_TY);
    else if (wo->kind_flag) am->domain_kov = kovko(wo);
    else if (wo->kind) { am->domain_kov = kovko(wo->kind); am->domain_wo = wo; }
    else { am->domain_kov = kova(OBJECT_TY); am->domain_wo = wo; }
}
void am_set_domain_from_kov(adjective_meaning *am, kind_of_value *kov) {
    am->domain_kov = kov; am->domain_wo = NULL;
    am->definition_domain_w1 = -1; am->definition_domain_w2 = -1;
}
```

The function am_set_domain_from_word_range is called from 9/madj and 12/def.

The function am_set_domain_from_wo is called from 9/qty.

The function am_set_domain_from_kov is called from 9/qty and 9/prop.

§**15.** And we can read the main domain thus:

```
kind_of_value *am_get_domain(adjective_meaning *am) {
    return am->domain_kov;
}
```

The function am_get_domain is called from 9/prop.

§**16.** In the case where the domain is declared as a word range, the following routine eventually converts it to the correct form. In effect, this is a lazy evaluation trick – the routine is called just before the domain is actually needed.

```
void am_set_definition_domain(adjective_meaning *am, int early) {
    if (am->domain_kov) return;
    if (am->definition_domain_w1 < 0) internal_error("undeclared domain KOV for AM");
    specification *supplied = parse_expression(
        am->definition_domain_w1, am->definition_domain_w2, DESCRIPTIVE_TYPE_EXPCON);
    ⟨Reject domain of adjective if meaning of supplied text is unknown 17⟩;
    ⟨Reject domain of adjective unless a kind of value or description of objects 18⟩;
    kind_of_value *kov = NULL;
    if (family_is(supplied, CONDITION_FMY)) {
        if (spec_get_described_kind(supplied)) kov = kovko(spec_get_described_kind(supplied));
        else if (spec_get_described_kov(supplied)) kov = spec_get_described_kov(supplied);
        else kov = kova(OBJECT_TY);
        ⟨Reject domain of adjective if it is a set of objects which may vary in play 19⟩;
    } else if (family_is(supplied, VALUE_FMY)) kov = kov_when_VALUE_is_evaluated(supplied);
    if (kov == NULL) internal_error("KOVless definition");
    am->domain_kov = kov;
    am->domain_wo = wo_of_CONSTANT_OBJECT_if_any(supplied);
    if (am->domain_wo) am->domain_kov = kovko(am->domain_wo->kind);
}
```

§**17.** Note that we throw only one problem message per AM, as otherwise duplication can't be avoided.

⟨Reject domain of adjective if meaning of supplied text is unknown 17⟩ ≡

```
    if (spec_is_UNKNOWN(supplied)) {
        if ((early) || (am->problems_thrown++ > 0)) return;
        LOG("Domain text: $W\n", am->definition_domain_w1, am->definition_domain_w2);
        current_sentence = am->defined_at;
        adjective_problem(_P_(C5AdjDomainUnknown),
            am->index_w1, am->index_w2, am->definition_domain_w1, am->definition_domain_w2,
            "this isn't a thing, a kind of thing or a kind of value",
            "and indeed doesn't have any meaning I can make sense of.");
        return;
    }
```

This code is used in §16.

§**18.** Similarly:

⟨Reject domain of adjective unless a kind of value or description of objects 18⟩ ≡

```
if ((spec_is_actual_CONSTANT(supplied)) &&
    (spec_describes_an_object_vaguely_or_exactly(supplied) == FALSE)) {
    if ((early) || (am->problems_thrown++ > 0)) return;
    LOG("Domain text: $S\n", supplied);
    current_sentence = am->defined_at;
    adjective_problem(_P_(C5AdjDomainSurreal),
        am->index_w1, am->index_w2, am->definition_domain_w1, am->definition_domain_w2,
        "this isn't allowed as the domain of a definition",
        "since adjectives like this can be applied only to specific things, "
        "kinds of things or kinds of values: so 'Definition: a door is ajar "
        "if...' is fine, because a door is a kind of thing, and 'Definition: "
        "a number is prime if ...' is fine too, but 'Definition: 5 is prime "
        "if ...' is not allowed.");
    return;
}
```

This code is used in §16.

§**19.** And a final possible objection:

⟨Reject domain of adjective if it is a set of objects which may vary in play 19⟩ ≡

```
if (spec_is_qualified_DESCRIPTION(supplied)) {
    if (am->problems_thrown++ > 0) return;
    current_sentence = am->defined_at;
    adjective_problem(_P_(C5AdjDomainSlippery),
        am->index_w1, am->index_w2, am->definition_domain_w1, am->definition_domain_w2,
        "this is slippery",
        "because it can change during play. Definitions can only be "
        "made in cases where it's clear for any given value or object "
        "what definition will apply. For instance, 'Definition: a "
        "door is shiny if ...' is fine, but 'Definition: an open "
        "door is shiny if ...' is not allowed - Inform wouldn't know "
        "whether or not to apply it to the Big Blue Door (say), since "
        "it would only apply some of the time.");
    return;
}
```

This code is used in §16.

§20. Comparing domains of validity. In order to sort AMs into logical precedence order, we rely on the following routine, which like `strcmp` returns a positive number to favour the first term, a negative to favour the second, and zero if they are equally good. Note that zero is only in fact returned when the two AMs compared are one and the same – we want to ensure that there is one and only one possible sorted state for any given list of AMs.

Suppose the adjectives A_1 and A_2 have domain sets D_1 and D_2. Then:

(i) If $D_1 \subseteq D_2$ and $D_1 \neq D_2$, then A_1 precedes A_2.
(ii) If $D_2 \subseteq D_1$ and $D_2 \neq D_1$, then A_2 precedes A_1.
(iii) If $D_1 = D_2$ or if $D_1 \cap D_2 = \emptyset$ then whichever was created earlier in Inform's run will take precedence.

Those are the only possibilities; the range of possible domains is set up so that there can never be an interesting Venn diagram of overlaps between them.

Unlike our weak domain tests above, this is a strict test.

```
int compare_ams(adjective_meaning *am1, adjective_meaning *am2) {
    kind_of_value *kov1, *kov2;
    int c1, c2;
    if (am1 == am2) return 0;
    if ((am1->domain_wo) && (am2->domain_wo == NULL)) return 1;
    if ((am1->domain_wo == NULL) && (am2->domain_wo)) return -1;
    kov1 = am1->domain_kov; kov2 = am2->domain_kov;
    c1 = can_we_cast_kovs(kov1, kov2); c2 = can_we_cast_kovs(kov2, kov1);
    if ((c1 == ALWAYS_MATCH) && (c2 != ALWAYS_MATCH)) return 1;
    if ((c1 != ALWAYS_MATCH) && (c2 == ALWAYS_MATCH)) return -1;
    return am2->allocation_id - am1->allocation_id;
}
```

§21. Checking an adjective's applicability. If the source tries to apply the word "open", say, to a given value or object X, when does that make sense?

We can only find out by checking every possible meaning of "open" to see if it can accommodate the kind of value of X. But this time we use weak checking, and make it weaker still since a null KOV is taken to mean "any object", either in the AM's definition – which can happen if we are very early in Inform's run – or because the caller doesn't actually know the kind of value of X. (In other words, adjectives tend to assume they apply to objects rather than other values.) This means we will accept some logically impossible outcomes – we would say that it's acceptable to apply "open" to an animal, say – but that is actually a good thing. It means that "list of open things" or "something open" are allowed. Source text such as:

```
The labrador puppy is an open animal.
```

will successfully parse, but then result in higher-level problem messages in Chapter 9. The following does compile:

```
now the labrador puppy is open;
```

but results in a run-time problem message when it executes.

It makes no difference what order we check the AMs in, so we can use the unsorted list, which is helpful since we may need to call this routine early in the run when sorting cannot yet be done.

```
int aph_applicable_to(adjectival_phrase *aph, kind_of_value *kov) {
    adjective_meaning *am;
    for (am = aph->possible_meanings; am; am = am->next_meaning) {
        if (am->domain_kov == NULL) am_set_definition_domain(am, FALSE);
        if (is_kova(am->domain_kov, OBJECT_TY)) {
            if (kov == FALSE) return TRUE;
```

```
            if (is_kova(kov, OBJECT_TY)) return TRUE;
        } else {
            if ((kov) && (is_kova(kov, OBJECT_TY) == FALSE) &&
                (can_we_cast_kovs(kov, am->domain_kov) == ALWAYS_MATCH))
                return TRUE;
        }
    }
    return FALSE;
}
```

The function aph_applicable_to is called from 5/candd and 6/tcpr.

§**22. Broad applicability tests.** Does a given APH have any interpretation as an enumerated property value, or an either/or property? If so we return the earliest known.

```
quantity *aph_has_ENUMERATIVE_meaning(adjectival_phrase *aph) {
    adjective_meaning *am;
    for (am = aph->possible_meanings; am; am = am->next_meaning)
        if (am->adjective_form == ENUMERATIVE_KADJ)
            return RETRIEVE_POINTER_quantity(am->detailed_meaning);
    return NULL;
}
property_name *aph_has_EORP_meaning(adjectival_phrase *aph) {
    adjective_meaning *am;
    for (am = aph->possible_meanings; am; am = am->next_meaning)
        if (am->adjective_form == EORP_KADJ)
            return RETRIEVE_POINTER_property_name(am->detailed_meaning);
    return NULL;
}
```

The function aph_has_ENUMERATIVE_meaning is called from 5/mlc, 6/term, 7/tc, 8/refpt, 10/tab and 13/tfg.
The function aph_has_EORP_meaning is called from 5/mlc, 6/term, 6/prop, 7/tc, 8/refpt, 8/imp and 13/tfg.

§23. Asserting the initial state. All that domain-checking machinery means we can begin to use an adjective.

Suppose an assertion sentence in the source text claims that, in the initial state of things, what the adjective tests is true. For example:

> The ormolu clock is fixed in place.

The S-parser, finding that this sentence is syntactically reasonable, identifies "fixed in place" as an adjective, and stores a pointer to its APH structure, but goes no further. Later on, the A-parser, working through sentences like this, works out that the adjective is to be applied to the world object "ormolu clock", whose KOV is "thing"; and that the sentence asserts a truth, not a falsity. It then calls the following routine, with `parity` equal to `TRUE`.

What happens is that the list of definitions for "fixed in place" is strictly checked in logical precedence order, and that `am_assert` is eventually called on the logically narrowest definition which the "ormolu clock" matches. (That will probably be the definition for the "fixed in place" either/or property for things, unless someone has given the adjective some special meaning unique to the clock.)

The following routine therefore acts as a junction-box, deciding which sense of the adjective is to be applied. We return `TRUE` if we were able to find a definition which could be asserted and which the clock matched, and `FALSE` if there was no definition which applied, or if none of those which did could be asserted for it.

```
int aph_assert(adjectival_phrase *aph, kind_of_value *kov_domain,
    world_object *wo_to_assert_on, specification *val_to_assert_on, int parity) {
    adjective_meaning *am;
    aph_sort_meanings(aph);
    for (am = aph->sorted_meanings; am; am = am->next_sorted) {
        if (domain_weak_match(kov_domain, am->domain_kov) == FALSE) continue;
        if (domain_wo_compare(wo_to_assert_on, am) == FALSE) continue;
        if (am_assert(am, wo_to_assert_on, val_to_assert_on, parity)) return TRUE;
    }
    return FALSE;
}
```

The function aph_assert is called from 6/asp.

§24. Testing and asserting in play. That takes care of two of the uses for an adjective, and now the other three: testing, making true and making false in play. We won't be there when the story file is played, of course, so what we have to do is to compile code to perform the test or force the state.

In fact what we do is to supply an I6 schema, which for this purpose is simply the text of I6 code in which the escape `*1` represents the value to which the adjective is applied. In the example of "open" for containers, we might choose:

```
if the sack is open, ...  -->   (Adj_53_t1_v61(*1))
now the sack is open; ...  -->   Adj_53_t2_v61(*1)
now the sack is not open; ...  -->   Adj_53_t3_v61(*1)
```

These schemas call an I6 routine called a "support routine". The names here are schematic: "open" on this run was APH number 53, the run-time tasks to perform were task 1, task 2 and task 3, and the sense of the adjective was the one applying to (weak) domain 61 – which in this run was the value of `OBJECT_TY`. In other words, these are routines to "test open in the sense of objects", "now open in the sense of objects", and "now not open in the sense of objects".

If we make a choice like that, then we say that the task is provided "via a support routine". We need not do so: for instance,

```
if the Entire Game is happening, ...  -->  (scene_status->(*1 - 1)==1)
```

is an example where the sense of "happening" for scenes can be tested directly using a schema, without calling a support routine. And clearly support routines only put off the problem, because we will also have

to compile the routine itself. So why use them? The answer is that in complicated situations where run-time type checking is needed, they avoid duplication of code, and can make repeated use of the `*1` value without repeating any side-effects produced by the calculation of this value. They also make the code simpler for human eyes to read.

§**25.** When an AM has been declared, the provider can choose to set an I6 schema for it, for any of the tasks, immediately; or can wait and do it later; or can choose not to do it, and instead provide code which generates a suitable schema on the fly. If at whatever stage the provider does set an I6 schema for a task, it should call the following.

Note that any AM working on objects always has to go via a support routine – this is because, thanks to weak domain-checking, there may be run-time type-checking code to apply. In other cases, the provider can choose to go via a support routine or not.

```
i6_schema *am_set_i6_schema(adjective_meaning *am, int T, int via_support) {
    kind_of_value *kov = am->domain_kov;
    if (kov == NULL) kov = kova(OBJECT_TY);
    if (is_kova(kov, OBJECT_TY)) via_support = TRUE;
    am->task_via_support_routine[T] = via_support;
    return &(am->i6s_for_runtime_task[T]);
}
```

The function am_set_i6_schema is called from 9/qty, 9/prop, 9/madj and 12/def.

§**26.** When Inform's code-generator needs to compile one of the tasks, then, it calls the following to obtain the correct I6 schema.

Note that the `task_via_support_routine` values are not flags: they can be `TRUE` (allowed, done via support routine), `FALSE` (allowed, done directly) or `NOT_APPLICABLE` (the task certainly can't be done). If none of the applicable meanings for the adjective are able to perform the task at run-time, we return `NULL` as our schema, and the code-generator will use that to issue a suitable problem message.

```
i6_schema *aph_get_i6_schema(adjectival_phrase *aph, kind_of_value *kov_domain,
    int T) {
    adjective_meaning *am;
    if (kov_domain == NULL) kov_domain = kova(OBJECT_TY);
    aph->sorted_meanings = am_list_sort(aph->possible_meanings);
    for (am = aph->sorted_meanings; am; am = am->next_sorted) {
        if (am->domain_kov == NULL) am_set_definition_domain(am, FALSE);
        if (domain_weak_match(kov_domain, am->domain_kov) == FALSE) continue;
        am_compiling_soon(am, T);
        switch (am->task_via_support_routine[T]) {
            case FALSE: return &(am->i6s_for_runtime_task[T]);
            case TRUE:
                if (sch_empty(&(am->i6s_to_transfer_to_SR[T])))
                    sch_write_to_existing_3d(&(am->i6s_to_transfer_to_SR[T]),
                        "Adj_%d_t%d_v%d(*1)",
                        aph->allocation_id, T, kov_I6_ID(am->domain_kov));
                return &(am->i6s_to_transfer_to_SR[T]);
        }
    }
    return NULL;
}
```

The function aph_get_i6_schema is called from 6/atoms.

§**27.** The following instructs an AM to use a support routine to handle a given task.

```
void am_pass_task_to_support_routine(adjective_meaning *am, int T) {
    am_set_i6_schema(am, T, TRUE);
}
```

The function am_pass_task_to_support_routine is called from 9/madj and 12/def.

§**28.** Some kinds of adjective find it useful to do some preparation work just before first compilation, but only once. For those, the ready flag is available:

```
int am_get_ready_flag(adjective_meaning *am) {
    return am->am_ready_flag;
}
void am_set_ready_flag(adjective_meaning *am) {
    am->am_ready_flag = TRUE;
}
```

The function am_get_ready_flag is called from 9/prop.
The function am_set_ready_flag is called from 9/prop.

§**29. Support routines.** Using these is only passing the buck: and the buck stops here.

The following utility is used to loop through the sorted meaning list, skipping over any which have been dealt with already.

```
adjective_meaning *am_list_next_domain_kov(adjective_meaning *am, kind_of_value **kov, int T) {
    while ((am) && ((am->defined_already) || (am_compile(am, T, NULL, NULL) == FALSE)))
        am = am->next_sorted;
    if (am == NULL) return NULL;
    *kov = am->domain_kov;
    return am->next_sorted;
}
```

§**30.** And this is where we do the iteration. The idea is that one adjective definition routine is defined (for each task number) which covers all of the weakly-domain-equal definitions for the same adjective. Thus one routine might handle "detailed" for rulebooks, and another might handle "detailed" for all of its meanings associated with objects – possibly many AMs.

```
void aph_compile_support_code(OUTPUT_STREAM) {
    ⟨Ensure, just in case, that domains exist and are sorted on 31⟩;
    int T;
    for (T=1; T<=NO_ADJECTIVE_TASKS; T++) {
        adjectival_phrase *aph;
        LOOP_OVER(aph, adjectival_phrase) {
            adjective_meaning *am;
            for (am = aph->possible_meanings; am; am = am->next_meaning)
                am->defined_already = FALSE;
            for (am = aph->sorted_meanings; am; ) {
                kind_of_value *kov = NULL;
                am = am_list_next_domain_kov(am, &kov, T);
                if (kov)
                    ⟨Compile adjective definition for this atomic kind of value 32⟩;
            }
        }
    }
}
```

The function aph_compile_support_code is called from 12/cs.

§**31.** It's unlikely that we have got this far without the domains for the AMs having been established, but certainly possible. We need the domains to be known in order to sort.

⟨Ensure, just in case, that domains exist and are sorted on 31⟩ ≡

```
adjectival_phrase *aph;
LOOP_OVER(aph, adjectival_phrase) {
    adjective_meaning *am;
    for (am = aph->possible_meanings; am; am = am->next_meaning) {
        am_set_definition_domain(am, FALSE);
        am->defined_already = FALSE;
    }
    aph->sorted_meanings = am_list_sort(aph->possible_meanings);
}
```

This code is used in §30.

§**32.** The following is a standard way to compile a one-off routine; we create a new stack frame, compile the code of the routine, then the header for it, and then fold the stack frame up again.

⟨Compile adjective definition for this atomic kind of value 32⟩ ≡

```
LOGIF(QUANTITY_CREATIONS, "Compiling support code for $W applying to $u, task %d\n",
    aph->word_ref1, aph->word_ref2, kov, T);
ph_stack_frame *phsf;
phsf = phsf_create_nonphrase_stack_frame();
OUT = begin_compiling_phrase(OUT);
begin_code_blocks();
⟨Add an it-variable to represent the value or object in the domain 33⟩;
INDENT;
if (problem_count == 0) am_list_compile(aph->sorted_meanings, OUT, phsf, kov, T);
WRITE("rfalse;\n");
OUTDENT; WRITE("];\n\n");
enable_its = -1;
OUT = write_routine_header();
WRITE("[ Adj_%d_t%d_v%d ! meaning of \"", aph->allocation_id, T, kov_I6_ID(kov));
print_raw_text_within_i6_literal(OUT, aph->word_ref1, aph->word_ref2);
WRITE("\"\n");
copy_compiled_phrase();
end_code_blocks();
phsf_remove_nonphrase_stack_frame();
```

This code is used in §30.

§**33.** The stack frame has just one call parameter: the value x which might, or might not, be such that *adjective*(x) is true. We allow this to be called "it", though it can also have a calling name in some cases (see below).

Clearly it ought to have the KOV which defines the domain – so it's a rulebook if the domain is all rulebooks, and so on – but it doesn't always do so. The exception is that it is bogusly given the KOV "number" if the adjective is being defined only by I6 routines. This is done to avoid compiling very inefficient code from the Standard Rules. For instance, the SR reads, in slightly simplified form:

Definition: an indexed text is empty if I6 routine "INDEXED_TEXT_TY_Empty" says so.

rather than the more obvious:

Definition: an indexed text is empty if it is not "".

Both of these definitions work. But if the routine defining "empty" for indexed text is allowed to act on an indexed text variable, Inform needs to compile code which acts on block values held on the memory heap at run-time. That means it needs to compile a memory heap; and that costs 8K or so of storage, making large Z-machine games which don't need indexed text or lists impossible to fit into the 64K array space limit. (There's also a benefit even if we do need a heap; the adjective can act on a direct pointer to the structure, and no time is wasted allocating memory and copying the block value first.)

⟨Add an it-variable to represent the value or object in the domain 33⟩ ≡

```
int conceal_kov = TRUE;
adjective_meaning *am;
for (am = aph->sorted_meanings; am; am = am->next_sorted)
    if ((am->adjective_form != I6_ROUTINE_KADJ) && (domain_weak_match(kov, am->domain_kov)))
        conceal_kov = FALSE;
if (conceal_kov)
    phsf_add_it_variable(phsf, -1, -1, kova(NUMBER_TY));
else
    phsf_add_it_variable(phsf, -1, -1, kov);
enable_its = 0;                                    the local variable number referred to by "its"
```

This code is used in §32.

§**34.** We run through possible meanings of the APH which share the current weak domain, and compile code which performs the stronger part of the domain test at run-time. In practice, at present the only weak domain which might have multiple definitions is "object", but that may change.

```
void am_list_compile(adjective_meaning *list_head, OUTPUT_STREAM,
    ph_stack_frame *phsf, kind_of_value *kov, int T) {
    adjective_meaning *am;
    for (am = list_head; am; am = am->next_sorted)
        if ((am_compile(am, T, NULL, NULL)) && (domain_weak_match(kov, am->domain_kov))) {
            current_sentence = am->defined_at;
            char cond[1024];
            if (is_kova(am->domain_kov, OBJECT_TY) == FALSE)
                cond[0] = 0;
            else if (am->domain_wo)
                sprintf(cond, "t_0 == %s",
                    wo_get_I6_representation(am->domain_wo));
            else if (kovko_get_kind(am->domain_kov))
                sprintf(cond, "t_0 ofclass %s",
                    wo_get_I6_representation(kovko_get_kind(am->domain_kov)));
            else
                sprintf(cond, "t_0");
```

```
            if (cond[0] != 0) WRITE("  if (%s) ", cond);
            WRITE("return ");
            if ((am->meaning_parity == FALSE) && (T == TEST_ADJECTIVE_TASK)) WRITE("(~~(");
            am_compile(am, T, OUT, phsf);
            if ((am->meaning_parity == FALSE) && (T == TEST_ADJECTIVE_TASK)) WRITE("))");
            WRITE(";\n");
            am->defined_already = TRUE;
        }
}
```

§**35. Kinds of adjectives.** This is where `inweb`'s use of C rather than `C++` or Python as a base language becomes a little embarrassing: we really want to have seven or eight subclasses of an "adjective" class, and provide a group of methods. Instead we simulate this with the following clumsy code. (More elegant code using pointers to functions would trip up `inweb`'s structure-element usage checking.)

To define a new kind of adjective, first allocate it a new `*_KADJ` constant (see above). Then declare functions to handle the following methods.

§**36.** 1. `*_KADJ_parse`. This enables the kind of adjective to claim a definition which the user has explicitly written, like so:

> Definition: A ... (called ...) is ... if ...

In place of the ellipses are the adjective name, domain name, condition text and (optionally) also the calling name. The routine should return a pointer to the AM it creates, if it does want to claim the definition; and `NULL` if it doesn't want it. `sense` is either 1, meaning that "if" was used (the condition has positive sense); or -1, meaning that it was "unless" (a negative sense); or 0, meaning that instead of a condition, a rule was supplied. (Most kinds of adjective will only claim if the sense is 1; some never claim at all.)

```
adjective_meaning *am_parse(parse_node *q,
    int sense,
    int adj_name_w1, int adj_name_w2,
    int dom_name_w1, int dom_name_w2,
    int cond_w1, int cond_w2,
    int called_w1, int called_w2) {
    adjective_meaning *am = NULL;
    if (am == NULL) am = EORP_KADJ_parse(q, sense, adj_name_w1, adj_name_w2,
        dom_name_w1, dom_name_w2, cond_w1, cond_w2, called_w1, called_w2);
    if (am == NULL) am = ENUMERATIVE_KADJ_parse(q, sense, adj_name_w1, adj_name_w2,
        dom_name_w1, dom_name_w2, cond_w1, cond_w2, called_w1, called_w2);
    if (am == NULL) am = MEASUREMENT_KADJ_parse(q, sense, adj_name_w1, adj_name_w2,
        dom_name_w1, dom_name_w2, cond_w1, cond_w2, called_w1, called_w2);
    if (am == NULL) am = I6_CONDITION_KADJ_parse(q, sense, adj_name_w1, adj_name_w2,
        dom_name_w1, dom_name_w2, cond_w1, cond_w2, called_w1, called_w2);
    if (am == NULL) am = I6_ROUTINE_KADJ_parse(q, sense, adj_name_w1, adj_name_w2,
        dom_name_w1, dom_name_w2, cond_w1, cond_w2, called_w1, called_w2);
    if (am == NULL) am = PHRASE_KADJ_parse(q, sense, adj_name_w1, adj_name_w2,
        dom_name_w1, dom_name_w2, cond_w1, cond_w2, called_w1, called_w2);
    if (am == NULL) am = CONDITION_KADJ_parse(q, sense, adj_name_w1, adj_name_w2,
        dom_name_w1, dom_name_w2, cond_w1, cond_w2, called_w1, called_w2);
    return am;
}
```

The function am_parse is called from 12/def.

§**37.** 2. `*_KADJ_compiling_soon`. This warns the adjective that it will shortly be needed in compilation, that is, that code will soon be compiled which uses it. This advance warning is an opportunity to compile a schema for the adjective at the last minute, but there is no obligation. There is also no return value.

```
void am_compiling_soon(adjective_meaning *am, int T) {
    switch (am->adjective_form) {
        case CONDITION_KADJ: CONDITION_KADJ_compiling_soon(am,
            RETRIEVE_POINTER_definition(am->detailed_meaning), T); break;
        case I6_CONDITION_KADJ: I6_CONDITION_KADJ_compiling_soon(am,
            RETRIEVE_POINTER_definition(am->detailed_meaning), T); break;
        case I6_ROUTINE_KADJ: I6_ROUTINE_KADJ_compiling_soon(am,
            RETRIEVE_POINTER_definition(am->detailed_meaning), T); break;
        case PHRASE_KADJ: PHRASE_KADJ_compiling_soon(am,
            RETRIEVE_POINTER_definition(am->detailed_meaning), T); break;
        case MEASUREMENT_KADJ: MEASUREMENT_KADJ_compiling_soon(am,
            RETRIEVE_POINTER_measurement_definition(am->detailed_meaning), T); break;
        case ENUMERATIVE_KADJ: ENUMERATIVE_KADJ_compiling_soon(am,
            RETRIEVE_POINTER_quantity(am->detailed_meaning), T); break;
        case EORP_KADJ: EORP_KADJ_compiling_soon(am,
            RETRIEVE_POINTER_property_name(am->detailed_meaning), T); break;
    }
}
```

§**38.** 3. `*_KADJ_compile`. We should now *either* compile code which, in the given stack frame and writing code to the given file handle, carries out the given task for the adjective, and return `TRUE`; *or* return `FALSE` to tell Inform that the task is impossible.

Note that if an adjective has defined a schema to handle the task, then its `*_KADJ_compile` is not needed and not consulted.

```
int am_compile(adjective_meaning *am, int T, OUTPUT_STREAM, ph_stack_frame *phsf) {
    am_compiling_soon(am, T);
    ⟨Use the I6 schema instead to compile the task, if one exists 39⟩;
    switch (am->adjective_form) {
        case CONDITION_KADJ: return CONDITION_KADJ_compile(
            RETRIEVE_POINTER_definition(am->detailed_meaning), T, OUT, phsf);
        case I6_ROUTINE_KADJ: return I6_ROUTINE_KADJ_compile(
            RETRIEVE_POINTER_definition(am->detailed_meaning), T, OUT, phsf);
        case I6_CONDITION_KADJ: return I6_CONDITION_KADJ_compile(
            RETRIEVE_POINTER_definition(am->detailed_meaning), T, OUT, phsf);
        case PHRASE_KADJ: return PHRASE_KADJ_compile(
            RETRIEVE_POINTER_definition(am->detailed_meaning), T, OUT, phsf);
        case MEASUREMENT_KADJ: return MEASUREMENT_KADJ_compile(
            RETRIEVE_POINTER_measurement_definition(am->detailed_meaning), T, OUT, phsf);
        case ENUMERATIVE_KADJ: return ENUMERATIVE_KADJ_compile(
            RETRIEVE_POINTER_quantity(am->detailed_meaning), T, OUT, phsf);
        case EORP_KADJ: return EORP_KADJ_compile(
            RETRIEVE_POINTER_property_name(am->detailed_meaning), T, OUT, phsf);
    }
    internal_error("unknown KADJ code");
    return FALSE;
}
```

§**39.** We expand the I6 schema, placing the "it" variable – a nameless call parameter which is always local variable number 0 for this stack frame – into `*1`.

⟨Use the I6 schema instead to compile the task, if one exists 39⟩ ≡

```
if (sch_empty(&(am->i6s_for_runtime_task[T])) == FALSE) {
    specification *it_var = new_LOCAL_VARIABLE_spec(-1, -1, 0);
    pcalc_term it_term = term_new_constant(it_var);
    WRITE("(");
    sch_expand(&(am->i6s_for_runtime_task[T]), OUT, &it_term, NULL);
    WRITE(")");
    return TRUE;
}
```

This code is used in §38.

§**40.** 4. `*_KADJ_assert`. We should now *either* take action to ensure that the adjective will hold (or not hold, according to `parity`) for the given object or value; *or* return `FALSE` to tell Inform that this cannot be asserted, which will trigger a problem message.

```
int am_assert(adjective_meaning *am, world_object *wo_to_assert_on,
    specification *val_to_assert_on, int parity) {
    switch (am->adjective_form) {
        case CONDITION_KADJ: return CONDITION_KADJ_assert(
            RETRIEVE_POINTER_definition(am->detailed_meaning),
            wo_to_assert_on, val_to_assert_on, parity);
        case I6_CONDITION_KADJ: return I6_CONDITION_KADJ_assert(
            RETRIEVE_POINTER_definition(am->detailed_meaning),
            wo_to_assert_on, val_to_assert_on, parity);
        case I6_ROUTINE_KADJ: return I6_ROUTINE_KADJ_assert(
            RETRIEVE_POINTER_definition(am->detailed_meaning),
            wo_to_assert_on, val_to_assert_on, parity);
        case PHRASE_KADJ: return PHRASE_KADJ_assert(
            RETRIEVE_POINTER_definition(am->detailed_meaning),
            wo_to_assert_on, val_to_assert_on, parity);
        case MEASUREMENT_KADJ: return MEASUREMENT_KADJ_assert(
            RETRIEVE_POINTER_measurement_definition(am->detailed_meaning),
            wo_to_assert_on, val_to_assert_on, parity);
        case ENUMERATIVE_KADJ: return ENUMERATIVE_KADJ_assert(
            RETRIEVE_POINTER_quantity(am->detailed_meaning),
            wo_to_assert_on, val_to_assert_on, parity);
        case EORP_KADJ: return EORP_KADJ_assert(
            RETRIEVE_POINTER_property_name(am->detailed_meaning),
            wo_to_assert_on, val_to_assert_on, parity);
    }
    internal_error("unknown KADJ code");
    return FALSE;
}
```

§**41.** 5. `*_KADJ_index`. This should print a description of the adjective to the index, for use in the Phrasebook lexicon. Note that it is only needed where the AM has been constructed positively, that is, it is not needed if the AM was made as a negation of something else.

Note also that if the AM was defined with any indexing text then that will be printed if the routine does nothing better.

```
void am_print_to_index(adjective_meaning *am) {
    int rv;
    ⟨Index the domain of validity of the AM 42⟩;
    if (am->am_negated_from) {
        INDEX(" opposite of </i>");
        print_raw_text_to_file(am->am_negated_from->owning_adjective->word_ref1,
            am->am_negated_from->owning_adjective->word_ref2, ifl);
        INDEX("<i>");
    } else {
        switch (am->adjective_form) {
            case CONDITION_KADJ: rv = CONDITION_KADJ_index(
                RETRIEVE_POINTER_definition(am->detailed_meaning)); break;
            case I6_CONDITION_KADJ: rv = I6_CONDITION_KADJ_index(
                RETRIEVE_POINTER_definition(am->detailed_meaning)); break;
            case I6_ROUTINE_KADJ: rv = I6_ROUTINE_KADJ_index(
                RETRIEVE_POINTER_definition(am->detailed_meaning)); break;
            case PHRASE_KADJ: rv = PHRASE_KADJ_index(
                RETRIEVE_POINTER_definition(am->detailed_meaning)); break;
            case MEASUREMENT_KADJ: rv = MEASUREMENT_KADJ_index(
                RETRIEVE_POINTER_measurement_definition(am->detailed_meaning)); break;
            case ENUMERATIVE_KADJ: rv = ENUMERATIVE_KADJ_index(
                RETRIEVE_POINTER_quantity(am->detailed_meaning)); break;
            case EORP_KADJ: rv = EORP_KADJ_index(
                RETRIEVE_POINTER_property_name(am->detailed_meaning)); break;
            default: rv = FALSE; break;
        }
        if ((rv == FALSE) && (am->index_w1 >= 0))
            print_raw_text_to_file(am->index_w1, am->index_w2, ifl);
    }
    index_link(lw_array[am->index_w1].lw_source);
}
```

The function am_print_to_index is called from 2/lexi.

§**42.** This is supposed to imitate dictionaries, distinguishing meanings by concisely showing their usage. Thus "empty" would have indexed entries prefaced "(of a rulebook)", "(of an activity)", and so on.

⟨Index the domain of validity of the AM 42⟩ ≡

```
if ((am->domain_wo) ||
    ((am->domain_kov) && (kov_compare(am->domain_kov, kova(OBJECT_TY)) == FALSE))) {
    INDEX("(of </i>");
    if (am->domain_wo)
        print_raw_text_to_file(am->domain_wo->word_ref1,
            am->domain_wo->word_ref2,
            ifl);
    else if (kovko_get_kind(am->domain_kov))
        print_raw_text_to_file(kovko_get_kind(am->domain_kov)->word_ref1,
            kovko_get_kind(am->domain_kov)->word_ref2,
            ifl);
    else {
        char kovname[1024]; kovname[0] = 0;
        copy_kov_to_string(am->domain_kov, kovname, FALSE);
        print_literal_string_to_file(ifl, kovname);
    }
    INDEX("<i>) ");
}
```

This code is used in §41.

Literal Patterns 5/litp

Purpose

To manage the possible notations with which literal values can be written.

5/litp.§1-3 Creating patterns, tokens and elements; §4-12 Listing LPs; §13 Optional break points; §14-23 Matching an LP in the source text; §24-30 Matching an LP at run-time; §31-36 Indexing literal patterns for a given data type; §37-44 Printing values in an LP's notation to the index at compile-time; §45-53 Printing the I6 variable `value` out in an LP's notation at run-time; §54-72 Creating new literal patterns; §73-75 I7 phrases to print values in specified ways; §76-78 I7 phrases to pack and unpack the value

---

Template interpreter commands

```
74  {-callv:define_named_printing_phrases}
```

Definitions

¶1. Literals are excerpts of text which explicitly specify values, rather than by indirectly naming them. "15" is a literal but "the score" is not, even if it happens to have the value 15. Inform contains specific code to parse literals such as

```
-314, "Morocco leather", 10:41 PM
```

(see "Lexer.w" and "Parsing Literals.w"), but it also has a system allowing the user to create new notations for quasi-numerical kinds of value. For example,

```
16:9 specifies an aspect ratio.
```

establishes a new notation for writing literal aspect ratios. This notation is stored internally as a "literal pattern", and the code to handle it is found in this section. (No other sorts of literal are parsed here.) Each kind of value has a linked list of literal notations which can specify it, if any. We sometimes need to iterate through the this list, and can do so with the following macro:

```
define LITERAL_FORMS_LOOP(lp, data_type)
    for (lp = data_type_list_of_literal_forms(data_type); lp;
        lp=lp->next_for_this_kov)
```

¶2. LPs with just a single numerical part to them (like "20 yards" rather than "16:9") are of special interest for holding scientific measurements, and we provide elaborate extra features for this form of LP.

In particular the single number is represented in decimal notation but is scaled and/or offset. Each such LP has an "overall offset" O and an "overall multiplier" M such that a value written k is stored as the I6 value $Mk + O$.

A given KOV can have many different LPs to represent it, and this is especially convenient for physics – it means we can give ways to describe mass (a kind of value) in grams, kilograms or tonnes (all literal patterns with different O and M values). Among these, one LP is special and is called the "benchmark" for the KOV – it is the default notation, the one considered most natural, and other LPs for the same KOV are scaled relative to this. For instance, the benchmark for mass might be the notation "1 kg"; the notations "1 g" and "1 tonne" would then be scaled down by 1000, or scaled up by 1000, as compared with the benchmark.

¶3. Syntactically, a literal pattern is a series of "tokens", of which more below. Some tokens are simply fixed lettering or wording, but others represent positions where "elements" – the numerical parts – appear. In the "16:9" notation, there are three tokens – an element token, a fixed colon, and another element token. At least one of the tokens has to be a element token.

```
define MAX_ELEMENTS_PER_LITERAL 16
define MAX_TOKENS_PER_LITERAL 100

typedef struct literal_pattern {
    struct kind_of_value *kov_specified;                    the data type of the result: i.e., what it specifies
    struct literal_pattern *next_for_this_kov;                          continuing list for this data type
    int prototype_w1, prototype_w2;                                    where the prototype specification is
    int no_lp_tokens;                                                   number of tokens in specification
    struct literal_pattern_token lp_tokens[MAX_TOKENS_PER_LITERAL];
    int no_lp_elements;                                                              how many tokens are numbers
    struct literal_pattern_element lp_elements[MAX_ELEMENTS_PER_LITERAL];
    int number_signed;                                          for instance -10 cm would be allowed if this is set
    used when we have a sequence of alternative notations for the same unit
    int primary_alternative;                                                         first of a set of alternatives?
    struct literal_pattern *next_alternative_lp;                        continuing list of alternatives
    int singular_form_only;                                          print using this notation only for 1 unit
    int plural_form_only;                                  print using this notation for 2 units, 0.5 units, etc.
    int overall_offset;                                                                the O value described above
    int overall_multiplier;                                                            the M value described above
    used in the process of defining patterns and calculating their O and M values
    int scaled_at;                                                        defined with this scaling factor, e.g., 100
    int scaled_up;                                      or, scaled up from the benchmark by this factor, e.g., 1000
    int scaled_down;                                                        or, scaled down from it similarly
    int equivalent_unit;                                                  is this just an equivalent to another LP?
    int benchmark;                                                         is this the benchmark LP for its KOV?
    int last_resort;                          is this the last possible LP to use when printing a value of the KOV?
    int marked_for_printing;                                                  used in compiling printing routines
    MEMORY_MANAGEMENT
} literal_pattern;
```

The structure literal_pattern is private to this section.

¶4. There are three sorts of token: character, word and element. Each token can be a whole word, or only part of a word. For instance, in

28kg net specifies a weight.

we have a sequence of four tokens: an element token, marked as beginning a word; a character token `k`; a character token `g`; and a word token `net`, which necessarily begins a word. Word boundaries in the source text must match those in the specification, so this notation does not match the text "41 kg net", for instance.

```
define WORD_LPT 1
define CHARACTER_LPT 2
define ELEMENT_LPT 3

typedef struct literal_pattern_token {
    int new_word_at;                                                        does token start a new word?
    int lpt_type;                                                      one of the three constants defined above
    char token_char;                                                CHARACTER_LPT only; the character to match
    int token_wn;                                          WORD_LPT only; word number in source text of the prototype
} literal_pattern_token;
```

The structure literal_pattern_token is private to this section.

¶5. A value notated this way is like an old-school Pascal packed integer, where a small data structure was joined into a single word of data. For instance, in the "16:9" example, $e_0 : e_1$ would be stored as $e_0 r_1 + e_1$ where $r_1 = 10$ is one more than the maximum value of e_1. So "4:3" would be stored as $4 \cdot (9 + 1) + 3 = 43$. More formally, we call the numbers in such a literal its "elements". In the case of "16:9", there are two elements, $e_0 = 16$ and $e_1 = 9$. The general formula is:

$$N = \sum_{i=0}^{n-1} e_i \cdot \prod_{j>i} r_j$$

where $(e_0, e_1, ..., e_{n-1})$ are the values and r_j, the "range", is the constraint such that $0 \leq e_j < r_j$. Note that r_0 is never required, since e_0 is constrained in size only by the need for N to fit into a single virtual machine integer. The value

$$m_i = \prod_{j>i} r_j$$

is called the "multiplier", and note that $m_{n-1} = 1$. Conversely,

$$e_i = \begin{cases} N/m_0 & \text{if } i = 0, \\ N/m_i \bmod r_i & \text{otherwise.} \end{cases}$$

The rightmost element e_{n-1} is the least significant numerically.

```
typedef struct literal_pattern_element {
    int element_index;                    /* the value i placing this within its LP, where 0 ≤ i < n */
    int element_range;                    /* the value r_i for this LP */
    int element_multiplier;               /* the value m_i for this LP */
    int word_ref1, word_ref2;             /* if we define a name for the element */
    int without_leading_zeros;            /* normally without? */
    int element_optional;                 /* can we truncate the LP here? */
    int preamble_optional;                /* if so, can we lose the preamble as well? */
} literal_pattern_element;
```

The structure literal_pattern_element is private to this section.

¶6. For the sake of printing, we can specify which notation is to be used in printing a value back. For instance,

1 tonne (in tonnes, singular) specifies a mass scaled up by 1000.

assigns the name "in tonnes" to this notation for writing a mass. There can be several notation associated with "in tonnes":

2 tonnes (in tonnes, plural) specifies a mass scaled up by 1000.

and hence the linked list of LPs associated with a single "literal pattern name". Moreover, a given kind of value can support multiple named notations; mass might also support "in kilograms" and "in grams", for instance.

```
typedef struct literal_pattern_name {
    int word_ref1, word_ref2;             /* name for this notation, if any; e.g. "in centimetres" */
    struct literal_pattern *can_use_this_lp;  /* list of LPs used under this name */
    struct literal_pattern_name *next;    /* other names for the same KOV */
    int lpn_compiled_already;
    MEMORY_MANAGEMENT
} literal_pattern_name;
```

The structure literal_pattern_name is private to this section.

§1. Creating patterns, tokens and elements.

```
literal_pattern *lp_new(kind_of_value *kov, int spw1, int spw2) {
    literal_pattern *lp = CREATE(literal_pattern);
    lp->plural_form_only = FALSE;
    lp->singular_form_only = FALSE;
    lp->kov_specified = kov;
    lp->prototype_w1 = spw1; lp->prototype_w2 = spw2;
    lp->next_for_this_kov = NULL;
    lp->primary_alternative = FALSE;
    lp->next_alternative_lp = NULL;
    lp->no_lp_elements = 0;
    lp->no_lp_tokens = 0;
    lp->number_signed = FALSE;
    lp->overall_offset = 1;
    lp->overall_multiplier = 1;
    lp->scaled_at = 1;
    lp->scaled_up = 1;
    lp->scaled_down = 1;
    lp->equivalent_unit = FALSE;
    lp->benchmark = FALSE;
    return lp;
}
```

§2.

```
literal_pattern_token lpt_new(int t, int nw) {
    literal_pattern_token lpt;
    lpt.new_word_at = nw;
    lpt.lpt_type = t;
    lpt.token_char = 0;
    lpt.token_wn = -1;
    return lpt;
}
```

§3.

```
literal_pattern_element lpe_new(int i, int r, int sgn) {
    literal_pattern_element lpe;
    if (i == 0) lpe.element_range = -1; else lpe.element_range = r;
    lpe.element_multiplier = 1;
    lpe.element_index = i;
    lpe.word_ref1 = -1; lpe.word_ref2 = -1;
    lpe.preamble_optional = FALSE;
    lpe.element_optional = FALSE;
    lpe.without_leading_zeros = FALSE;
    return lpe;
}
```

§**4. Listing LPs.** A routine to append a LP to the linked list of LPs for a given KOV. But it's a little more involved because this is where we calculate the scale factors which relate LPs in the list, and also because we need to keep the list in a particular order.

```
literal_pattern *lp_list_add(literal_pattern *list_head, literal_pattern *new_lp) {
    if (list_head == NULL) ⟨Begin a new list with just the new LP in it 5⟩
    else ⟨Add the new LP to the existing list 6⟩;
    ⟨Correct the "last resort" flags in the list of LPs 8⟩;
    ⟨Automatically enable signed literals if there are scaled LPs in the list 9⟩;
    return list_head;
}
```

The function lp_list_add is called from 7/data.

§**5.** When the new LP is the first one, it can only be scaled in absolute terms: "scaled at", which specifies its M value.

```
⟨Begin a new list with just the new LP in it 5⟩ ≡
    if ((new_lp->scaled_up != 1) || (new_lp->scaled_down != 1) || (new_lp->equivalent_unit))
        sentence_problem(_P_(C5LPCantScaleYet),
            "this tries to scale up or down a value which so far has no point of "
            "reference to scale from",
            "which is impossible.");
    new_lp->overall_multiplier = new_lp->scaled_at;
    list_head = new_lp;
```

This code is used in §4.

§**6.** But if other LPs already exist, then absolute scalings are forbidden. The new LP must be scaled up or down relative to existing notations, or pegged equivalent to an exact value.

```
⟨Add the new LP to the existing list 6⟩ ≡
    if (new_lp->scaled_at != 1) {
        if (new_lp->primary_alternative)
            sentence_problem(_P_(C5LPCantScaleTwice),
                "this tries to specify the scaling for a kind of value whose "
                "scaling is already established",
                "which is impossible.");
        new_lp->scaled_at = 1;
    }
    literal_pattern *lp, *benchmark_lp = NULL;
    for (lp = list_head; lp; lp = lp->next_for_this_kov)
        if (lp->benchmark) benchmark_lp = lp;
    if (new_lp->equivalent_unit) new_lp->overall_multiplier = new_lp->scaled_up;
    else if (benchmark_lp) ⟨Calculate the multiplier for the LP relative to the benchmark 7⟩
    else new_lp->overall_multiplier = 1;
    list_head = lp_list_add_inner(list_head, new_lp);
```

This code is used in §4.

§**7.** An equivalent unit exactly specifies its M-value relative to the benchmark's M-value, which we call B. If it is scaled up by k, then $M = kB$ and there's no problem (ignoring overflows, anyway).

If scaled down then we want $M = B/k$, but that may be impossible since we're performing integer division and k might not divide B. We make it possible by increasing every M-value in the list by a factor of $k/\gcd(B, k)$, thus making B divisible by k. To keep the proportions right we must increase M for the new LP by the same factor, so we now want

$$M = (B/k) \cdot (k/\gcd(B, k)) = B/\gcd(B, k)$$

and this, of course, is an integer division which is always possible, since the gcd is by construction a factor of B.

⟨Calculate the multiplier for the LP relative to the benchmark 7⟩ ≡

```
int B = benchmark_lp->overall_multiplier;
if (new_lp->scaled_up != 1)
    new_lp->overall_multiplier = B * new_lp->scaled_up;
if (new_lp->scaled_down != 1) {
    int k = new_lp->scaled_down;
    int g = gcd(B, k);
    literal_pattern *lp;
    for (lp = list_head; lp; lp = lp->next_for_this_kov)
        if (lp->equivalent_unit == FALSE)
            lp->overall_multiplier = (k/g)*(lp->overall_multiplier);
    new_lp->overall_multiplier = B/g;
}
```

This code is used in §6.

§**8.** Within the list, exactly one LP is marked with the `last_resort` flag: the last one not marked as an equivalent unit. (You can only be equivalent to something already there, so it's not possible for all the LPs in the list to be equivalent.)

⟨Correct the "last resort" flags in the list of LPs 8⟩ ≡

```
literal_pattern *lp, *last_resorter = NULL;
for (lp = list_head; lp; lp = lp->next_for_this_kov) {
    lp->last_resort = FALSE;
    if (lp->equivalent_unit == FALSE) last_resorter = lp;
}
if (last_resorter) last_resorter->last_resort = TRUE;
```

This code is used in §4.

§**9.** Inform is ordinarily a bit picky about not allowing negative values within these notations, unless they have explicitly been defined to allow it. That makes sense for basically combinatorial notations (room 1 to room 64, say) but would be a nonsense for scientific measurements where we intend to perform arithmetic. So:

⟨Automatically enable signed literals if there are scaled LPs in the list 9⟩ ≡

```
    int scalings_exist = FALSE;
    literal_pattern *lp;
    for (lp = list_head; lp; lp = lp->next_for_this_kov)
        if (lp->overall_multiplier != 1)
            scalings_exist = TRUE;
    if (scalings_exist)
        for (lp = list_head; lp; lp = lp->next_for_this_kov)
            lp->number_signed = TRUE;
```

This code is used in §4.

§**10.** The actual insertion of the new LP into the list is carried out here, and is complicated by the fact that we need to keep these in a special order.

```
literal_pattern *lp_list_add_inner(literal_pattern *list_head, literal_pattern *new_lp) {
    literal_pattern *lp, *lp_prev;
    new_lp->next_for_this_kov = NULL;
    if (list_head == NULL) return new_lp;
    lp = list_head; lp_prev = NULL;
    while (lp) {
        if (lp_precedes(new_lp, lp)) {
            new_lp->next_for_this_kov = lp;
            if (lp_prev) lp_prev->next_for_this_kov = new_lp;
            else list_head = new_lp;
            return list_head;
        }
        lp_prev = lp;
        lp = lp->next_for_this_kov;
    }
    lp_prev->next_for_this_kov = new_lp;
    return list_head;
}
```

§**11.** Large M-values come before small ones; plural forms for the same M-value come before singular ones; and otherwise an earlier-defined LP comes before a later one.

```
int lp_precedes(literal_pattern *A, literal_pattern *B) {
    if (A->overall_multiplier > B->overall_multiplier) return TRUE;
    if (A->overall_multiplier < B->overall_multiplier) return FALSE;
    if ((A->plural_form_only) && (B->plural_form_only == FALSE)) return TRUE;
    if ((A->plural_form_only == FALSE) && (B->plural_form_only == TRUE)) return FALSE;
    if (A->allocation_id < B->allocation_id) return TRUE;
    return FALSE;
}
```

§**12.** This returns the multiplier of the benchmark, which is important for performing multiplications (see "Dimensions" in Chapter 7).

```
int data_type_scale_factor(int data_type) {
    literal_pattern *lp;
    LITERAL_FORMS_LOOP(lp, data_type)
        if (lp->benchmark)
            return lp->overall_multiplier;
    return 1;
}
```

The function data_type_scale_factor is called from 7/kov.

§**13. Optional break points.** Sometimes the pattern allows later numerical elements to be skipped, in which case they are understood to be 0.

```
int at_optional_break_point(literal_pattern *lp, int ec, int tc) {
    if ((ec<lp->no_lp_elements) &&                    i.e., if there are still numerical elements to supply
        (lp->lp_elements[ec].element_optional) &&                                  but which are optional
        ((lp->lp_elements[ec].preamble_optional) ||     and either the preamble tokens are also optional
            (lp->lp_tokens[tc].lpt_type == ELEMENT_LPT)))                  or we're at the number token
        return TRUE;
    return FALSE;
}
```

§**14. Matching an LP in the source text.** Given an excerpt `(w1, w2)`, we try to parse it as a constant value written in the LP notation: if it passes, we return the kind of value, and if not we return `NULL`.

```
int waive_lp_overflows = FALSE;
int last_LP_problem_at = -1;
kind_of_value *match_literal_pattern(literal_pattern *lp, int w1, int w2, int *found) {
    int matched_number = 0, overflow_16_bit_flag = FALSE, overflow_32_bit_flag = FALSE;
    literal_pattern_element *sign_used_at = NULL, *element_overflow_at = NULL;
    if the excerpt is longer than the maximum length of such a notation, give up quickly:
    if (w2-w1 > lp->prototype_w2-lp->prototype_w1) return NULL;
    ⟨Try to match the excerpt against the whole prototype or up to an optional break 15⟩;
    if (sign_used_at) {
        ⟨Check that a negative number can be used in this notation 20⟩;
        matched_number = -matched_number;
    }
    if (waive_lp_overflows == FALSE) {
        if (element_overflow_at) ⟨Report a problem because one element in the notation overflows 22⟩;
        ⟨Check that the value found lies within the range which the VM can hold 21⟩;
    }
    *found = matched_number;
    return lp->kov_specified;
}
```

The function match_literal_pattern is called from 5/lit.

§**15.** Scanning the tokens one at a time. The scan position is represented as a word number `wn` together with a character position within the word, `wpos`. The `wpos` value −1 means that word `wn` has not yet been started.

⟨Try to match the excerpt against the whole prototype or up to an optional break 15⟩ ≡

```
int tc, wn = w1, wpos = -1, ec = 0, matched_scaledown = 1;
char *wd = lw_array[w1].lw_text;
for (tc=0; tc<lp->no_lp_tokens; tc++) {
    if (wn > w2) {
        if ((wpos == -1)                                    i.e., if we are cleanly at a word boundary
            && (at_optional_break_point(lp, ec, tc))) break;
        return NULL;
    }
    switch (lp->lp_tokens[tc].lpt_type) {
        case WORD_LPT: ⟨Match a fixed word token within a literal pattern 16⟩; break;
        case CHARACTER_LPT: ⟨Match a character token within a literal pattern 17⟩; break;
        case ELEMENT_LPT: ⟨Match an element token within a literal pattern 18⟩; break;
        default: internal_error("unknown literal pattern token type");
    }
}
if (wpos >= 0) return NULL;                                 we need to end cleanly, not in mid-word
if (wn <= w2) return NULL;                                  and we need to have used up all of the excerpt
if (lp->overall_multiplier % matched_scaledown != 0)
    ⟨Report a problem because not enough accuracy is available 23⟩;
long long int max_16_bit = 32767LL, max_32_bit = 2147483647LL;
matched_number = matched_number*(lp->overall_multiplier/matched_scaledown) + lp->overall_offset;
if (matched_number > max_16_bit) overflow_16_bit_flag = TRUE;
if (matched_number > max_32_bit) overflow_32_bit_flag = TRUE;
```

This code is used in §14.

§**16.** A word token matches an exact word (but allowing for variation in casing).

⟨Match a fixed word token within a literal pattern 16⟩ ≡

```
if (wpos >= 0) return NULL;                    if we're still in the middle of the last word, we must fail
if (compare_words(wn, lp->lp_tokens[tc].token_wn) == FALSE) return NULL;
wn++;
```

This code is used in §15.

§**17.** A character token matches only a single character – note the case insensitivity here, because of the use of `tolower`.

⟨Match a character token within a literal pattern 17⟩ ≡

```
if (wpos == -1) { wpos = 0; wd = lw_array[wn].lw_text; }        start parsing the interior of a word
if (tolower(wd[wpos++]) != tolower(lp->lp_tokens[tc].token_char)) return NULL;
if (wd[wpos] == 0) { wn++; wpos = -1; }                         and stop parsing the interior of a word
```

This code is used in §15.

§**18.** There are three different sorts of overflow:

(1) The calculation of the packed value exceeding the range which an integer can store on a 16-bit virtual machine;
(2) Ditto, but on a 32-bit virtual machine; and
(3) One of the numerical elements inside the notation being given out of range.

We report none of these as a problem immediately – only if the pattern would otherwise match.

The following assumes that `long long int` is at least 64-bit, so that it can hold any 32-bit integer multiplied by 10, and also any product of two 32-bit numbers. This is true for all modern `gcc` implementations and is required by C99, but was not required by C90, so it is just possible that this could cause trouble on unusual platforms.

⟨Match an element token within a literal pattern 18⟩ ≡

```
literal_pattern_element *lpe = &(lp->lp_elements[ec++]);                fetch details of next number
if (wpos == -1) { wpos = 0; wd = lw_array[wn].lw_text; }                start parsing the interior of a word
long long int tot = 0, max_32_bit, max_16_bit;
int digits_found = 0, point_at = -1;
if (wd[wpos] == '-') { sign_used_at = lpe; wpos++; }
max_16_bit = 32767LL; if (sign_used_at) max_16_bit = 32768LL;
max_32_bit = 2147483647LL; if (sign_used_at) max_32_bit = 2147483648LL;
while ((isdigit(wd[wpos])) ||
    ((wd[wpos] == '.') && (lp->overall_multiplier > 1) && (point_at == -1))) {
    if (wd[wpos] == '.') { point_at = digits_found; wpos++; continue; }
    tot = 10*tot + (wd[wpos++] - '0');
    if (tot > max_16_bit) overflow_16_bit_flag = TRUE;
    if (tot > max_32_bit) overflow_32_bit_flag = TRUE;
    digits_found++;
}
if ((point_at == 0) || (point_at == digits_found)) return NULL;
if (digits_found == 0) return NULL;
while ((point_at > 0) && (point_at < digits_found)) {
    matched_scaledown *= 10; point_at++;
}
if ((tot >= lpe->element_range) && (lpe->element_index > 0)) element_overflow_at = lpe;
tot = (lpe->element_multiplier)*tot;
if (tot > max_16_bit) overflow_16_bit_flag = TRUE;
if (tot > max_32_bit) overflow_32_bit_flag = TRUE;
tot = matched_number + tot;
if (tot > max_16_bit) overflow_16_bit_flag = TRUE;
if (tot > max_32_bit) overflow_32_bit_flag = TRUE;
matched_number = (int) tot;
if (wd[wpos] == 0) { wn++; wpos = -1; }                    and stop parsing the interior of a word
```

This code is used in §15.

§**19.** Problem messages here have a tendency to be repeated, in some situations, which is annoying. So we have a mechanism to suppress duplicates:

```
define ISSUING_LP_PROBLEM
    if (last_LP_problem_at == w1) return NULL;
    last_LP_problem_at = w1;
```

§**20.**

⟨Check that a negative number can be used in this notation 20⟩ ≡

```
if (sign_used_at->element_index != 0) {
    ISSUING_LP_PROBLEM;
    sentence_problem(_P_(C5NegationInternal),
        "a negative number can't be used in the middle of a constant",
        "and the minus sign makes it look as if that's what you are "
        "trying here.");
    return NULL;
}
if (lp->number_signed == FALSE) {
    ISSUING_LP_PROBLEM;
    sentence_problem(_P_(C5NegationForbidden),
        "the minus sign is not allowed here",
        "since this is a kind of value which only allows positive "
        "values to be written.");
    return NULL;
}
```

This code is used in §14.

§**21.** The out of range problem messages:

⟨Check that the value found lies within the range which the VM can hold 21⟩ ≡

```
if (overflow_32_bit_flag) {
    ISSUING_LP_PROBLEM;
    sentence_problem(_P_(C5EvenOverflow-G),
        "you use a literal specification to make a value which is too large",
        "even for a story file compiled with the Glulx setting. (You can "
        "see the size limits for each way of writing a value on the Kinds "
        "page of the Index.)");
    return NULL;
}
if ((overflow_16_bit_flag) && (target_VM_is_16_bit())) {
    ISSUING_LP_PROBLEM;
    sentence_problem(_P_(C5ZMachineOverflow),
        "you use a literal specification to make a value which is too large",
        "at least with the Settings for this project as they currently are. "
        "(Change to Glulx to be allowed to use much larger numbers; "
        "meanwhile, you can see the size limits for each way of writing a "
        "value on the Kinds page of the Index.)");
    return NULL;
}
```

This code is used in §14.

§**22.** The more specific problem of an internal overflow:

⟨Report a problem because one element in the notation overflows 22⟩ ≡

```
int max = element_overflow_at->element_range - 1;
quote_source(1, current_sentence);
quote_words(2, w1, w2);
quote_words(3, lp->prototype_w1, lp->prototype_w2);
quote_number(4, &max);
switch (element_overflow_at->element_index) {
    case 0: quote_text(5, "first"); break;
    case 1: quote_text(5, "second"); break;
    case 2: quote_text(5, "third"); break;
    case 3: quote_text(5, "fourth"); break;
    case 4: quote_text(5, "fifth"); break;
    case 5: quote_text(5, "sixth"); break;
    case 6: quote_text(5, "seventh"); break;
    case 7: quote_text(5, "eighth"); break;
    case 8: quote_text(5, "ninth"); break;
    case 9: quote_text(5, "tenth"); break;
    default: quote_text(5, "eventual"); break;
}
handmade_problem(_P_(C5ElementOverflow));
issue_problem_segment(
    "In the sentence %1, you use the notation '%2' to write a constant value. "
    "But the notation was specified as '%3', which means that the %5 numerical "
    "part should range between 0 and %4.");
issue_problem_end();
return NULL;
```

This code is used in §14.

§**23.**

⟨Report a problem because not enough accuracy is available 23⟩ ≡

```
quote_source(1, current_sentence);
quote_words(2, w1, w2);
quote_words(3, lp->prototype_w1, lp->prototype_w2);
handmade_problem(_P_(C5LPTooLittleAccuracy));
issue_problem_segment(
    "In the sentence %1, you use the notation '%2' to write a constant value. "
    "But to store that, I would need greater accuracy than this kind of "
    "value has - see the Kinds page of the Index for the range it has.");
issue_problem_end();
return NULL;
```

This code is used in §15.

§24. Matching an LP at run-time. The following routine compiles an I6 general parsing routine (GPR) to match typed input in the correct notation. It amounts to printing out a version of the above routine, but ported to I6, and with the token loop "rolled out" so that no `tc` and `ec` variables are needed at run-time, and simplified by having the numerical overflow detection removed. (It's a little slow to perform that check within the VM.)

Properly speaking this is not an entire GPR, but only a segment of one, and we should compile code which allows execution to reach the end if and only if we fail to make a match.

```
void gpr_for_lp(OUTPUT_STREAM, literal_pattern *lp) {
    int label = lp->allocation_id;
    int tc, ec;
    comment_use_of_lp(OUT, lp);
    WRITE("wpos = 0; mid_word = false; matched_number = 0;\n");
    for (tc=0, ec=0; tc<lp->no_lp_tokens; tc++) {
        int lookahead = -1;
        if ((tc+1<lp->no_lp_tokens) && (lp->lp_tokens[tc+1].lpt_type == CHARACTER_LPT))
            lookahead = lp->lp_tokens[tc+1].token_char;
        WRITE("if (mid_word == false) {\n"); INDENT;
        WRITE("cur_word = NextWordStopped(); wn--;\n");
        WRITE("if (cur_word == -1) ");
        if (at_optional_break_point(lp, ec, tc))
            WRITE("jump Succeeded_LP_%d;\n", label);
        else
            WRITE("jump Failed_LP_%d;\n", label);
        OUTDENT; WRITE("}\n");
        switch (lp->lp_tokens[tc].lpt_type) {
            case WORD_LPT: ⟨Compile I6 code to match a fixed word token within a literal pattern 25⟩; break;
            case CHARACTER_LPT: ⟨Compile I6 code to match a character token within a literal pattern 26⟩; break;
            case ELEMENT_LPT: ⟨Compile I6 code to match an element token within a literal pattern 27⟩; break;
            default: internal_error("unknown literal pattern token type");
        }
    }
    WRITE(".Succeeded_LP_%d;\n", label);
    WRITE("if (mid_word) jump Failed_LP_%d;\n", label);
    WRITE("if (sgn<0) matched_number = -1*matched_number;\n");
    WRITE("parsed_number = matched_number;\n");
    char offset_text[16]; sprintf(offset_text, "%d+x", lp->overall_offset);
    carefully_scale_and_add(OUT, "parsed_number", lp->overall_multiplier, offset_text, label);
    WRITE("if (parser_trace >= 3) { print \"  [parsed value \", parsed_number, \" by: ");
    print_text_to_file(lp->prototype_w1, lp->prototype_w2, OUT);
    WRITE("]^\"; }\n");
    WRITE("return GPR_NUMBER;\n");
    WRITE(".Failed_LP_%d;\n", label);
}
```

The function gpr_for_lp is called from 13/gprv.

§25.

⟨Compile I6 code to match a fixed word token within a literal pattern 25⟩ ≡
```
WRITE("if (mid_word) jump Failed_LP_%d;\n", label);
WRITE("if (cur_word ~= ");
isn_compile_dictionary_word(OUT, lw_array[lp->lp_tokens[tc].token_wn].lw_text, FALSE);
WRITE(") jump Failed_LP_%d;\n", label);
WRITE("wn++;\n");
```

This code is used in §24.

§26.

⟨Compile I6 code to match a character token within a literal pattern 26⟩ ≡
```
⟨Compile I6 code to enter mid-word parsing if not already in it 28⟩;
WRITE("if (cur_addr->wpos++ ~= '%c') jump Failed_LP_%d;\n",
    lp->lp_tokens[tc].token_char, label);
⟨Compile I6 code to exit mid-word parsing if at end of a word 29⟩;
```

This code is used in §24.

§27.

⟨Compile I6 code to match an element token within a literal pattern 27⟩ ≡
```
literal_pattern_element *lpe = &(lp->lp_elements[ec++]);
⟨Compile I6 code to enter mid-word parsing if not already in it 28⟩;
WRITE("sgn = 1; tot = 0; f = false;\n");
if ((lp->number_signed) && (ec == 0))
    WRITE("if (cur_addr->wpos == '-') { sgn = -1; wpos++; }\n");
else
    WRITE("if (cur_addr->wpos == '-') jump Failed_LP_%d;\n", label);
WRITE("while ((wpos < cur_len) && (DigitToValue(cur_addr->wpos)>=0)) {\n"); INDENT;
WRITE("f = DigitToValue(cur_addr->wpos);\n");
carefully_scale_and_add(OUT, "tot", 10, "f", label);
WRITE("f = true;\n");
WRITE("wpos++;\n");
OUTDENT; WRITE("}\n");
WRITE("if (f == false) jump Failed_LP_%d;\n", label);
if (lpe->element_index > 0)
    WRITE("if (tot >= %d) jump Failed_LP_%d;\n",
        lpe->element_range, label);
carefully_scale_and_add(OUT, "tot", lpe->element_multiplier, "matched_number", label);
WRITE("matched_number = tot;\n");
if (lp->overall_multiplier > 1) {
    WRITE("if (wpos == cur_len) {\n"); INDENT;
    WRITE("wn++;\n");
    WRITE("cur_word = NextWordStopped();\n");
    WRITE("wn = wn - 2;\n");
    WRITE("if (cur_word == THEN1__WD) {\n"); INDENT;
    WRITE("wn = wn + 2;\n");
    WRITE("mid_word = false;\n");
    ⟨Compile I6 code to enter mid-word parsing if not already in it 28⟩;
    WRITE("for (x = 0, f = %d; (f>1) && (f%%10 == 0) && (DigitToValue(cur_addr->wpos) >= 0);"
        "f = f/10) {\n", lp->overall_multiplier); INDENT;
    carefully_scale_and_add(OUT, "x", 1, "DigitToValue(cur_addr->wpos)*f/10", label);
```

```
        WRITE("wpos++;\n");
        OUTDENT; WRITE("}\n");
        OUTDENT; WRITE("}\n");
        OUTDENT; WRITE("}\n");
    }
    ⟨Compile I6 code to exit mid-word parsing if at end of a word 29⟩;
```

This code is used in §24.

§**28.**

⟨Compile I6 code to enter mid-word parsing if not already in it 28⟩ ≡

```
    WRITE("if (mid_word == false) {\n"); INDENT;
    WRITE("mid_word = true;\n");
    WRITE("wpos = 0;\n");
    WRITE("cur_addr = WordAddress(wn);\n");
    WRITE("cur_len = WordLength(wn);\n");
    OUTDENT; WRITE("}\n");
```

This code is used in §26,27,26,27,26,27.

§**29.**

⟨Compile I6 code to exit mid-word parsing if at end of a word 29⟩ ≡

```
    WRITE("if (wpos == cur_len) {\n"); INDENT;
    if ((lookahead != -1) && (lookahead != '.'))
        WRITE("jump Failed_LP_%d; ! Lookahead <%d>\n", label, lookahead);
    WRITE("wn++; mid_word = false;\n");
    OUTDENT; WRITE("}\n");
```

This code is used in §26,27,26,27,26,27.

§**30.** The following routine performs the operation $v \mapsto kv + \ell$, where $0 \leq \ell < k$ and $k > 1$, but carefully checking that v does not overflow the virtual machine's integer size limit in the process. k is a constant known at compile time, but ℓ is not known until run-time.

```
void carefully_scale_and_add(OUTPUT_STREAM, char *var, int scale_factor, char *to_add, int label) {
    long long int max;
    if (scale_factor > 1) {
        if (target_VM_is_16_bit()) max = 32767LL; else max = 2147483647LL;
        WRITE("if (sgn == 1) {\n"); INDENT;
        WRITE("if ((%s > %d) || ((%s == %d) && (%s > %d)))\n",
            var, (int) (max/scale_factor), var, (int) (max/scale_factor),
            to_add, (int) (max%scale_factor)); INDENT;
        WRITE("jump Failed_LP_%d;\n", label);
        OUTDENT; OUTDENT; WRITE("} else {\n"); INDENT;
        max++;
        WRITE("if ((%s > %d) || ((%s == %d) && (%s > %d)))\n",
            var, (int) (max/scale_factor), var, (int) (max/scale_factor),
            to_add, (int) (max%scale_factor)); INDENT;
        WRITE("jump Failed_LP_%d;\n", label);
        OUTDENT; OUTDENT; WRITE("}\n");
    }
    WRITE("%s = %d*%s + %s;\n", var, scale_factor, var, to_add);
}
```

§31. Indexing literal patterns for a given data type. (A data type is an ID number used to identify base kinds of value, or KOV constructors – though the latter can never occur here.)

```
void index_literal_patterns(int data_type) {
    literal_pattern *lp, *benchmark_lp = NULL;
    int B = 1, scalings_exist = FALSE;
    LITERAL_FORMS_LOOP(lp, data_type) {
        if (lp->benchmark) {
            benchmark_lp = lp;
            B = lp->overall_multiplier;
        }
        if (lp->overall_multiplier != 1) scalings_exist = TRUE;
    }
    ⟨Index the list of possible LPs for the KOV, not counting equivalents 32⟩;
    ⟨Index the list of possible LPs for the KOV, only counting equivalents 33⟩;
    ⟨Index the possible names for these notations, as ways of printing them back 34⟩;
}
```

The function index_literal_patterns is called from 7/kix.

§32. Each entry in this list is, in principle, a list all by itself – of alternatives such as "1 tonne" vs "2 tonnes", which aren't different enough to be listed separately. Of these exactly one is the "primary" alternative.

⟨Index the list of possible LPs for the KOV, not counting equivalents 32⟩ ≡

```
    int f = FALSE;
    LITERAL_FORMS_LOOP(lp, data_type)
        if ((lp->primary_alternative) && (lp->equivalent_unit == FALSE)) {
            if (f) INDEX("<br>");
            else INDEX("<i>Written as:</i><br>");
            if ((scalings_exist) && (benchmark_lp)) {
                index_lp_possibilities(lp, benchmark_lp);
            } else {
                print_raw_text_to_file(lp->prototype_w1, lp->prototype_w2, ifl);
            }
            f = TRUE;
        }
```

This code is used in §31.

§33.

⟨Index the list of possible LPs for the KOV, only counting equivalents 33⟩ ≡

```
    int f = FALSE;
    LITERAL_FORMS_LOOP(lp, data_type)
        if ((lp->primary_alternative) && (lp->equivalent_unit)) {
            if (f) INDEX("<br>");
            else INDEX("<br><i>With these equivalent units:</i><br>");
            index_lp_possibilities(lp, benchmark_lp);
            f = TRUE;
        }
```

This code is used in §31.

§**34.**

⟨Index the possible names for these notations, as ways of printing them back 34⟩ ≡

```
int f = FALSE;
literal_pattern_name *lpn;
LOOP_OVER(lpn, literal_pattern_name)
    if (lpn->word_ref1 >= 0) {
        LITERAL_FORMS_LOOP(lp, data_type) {
            literal_pattern_name *lpn2;
            for (lpn2 = lpn; lpn2; lpn2 = lpn2->next)
                if (lp == lpn2->can_use_this_lp) {
                    if (f) INDEX("; ");
                    else {
                        INDEX("<br><i>Can be printed back:</i><br>");
                        INDEX("    ");
                    }
                    f = TRUE;
                    print_raw_text_to_file(lpn->word_ref1, lpn->word_ref2, ifl);
                    goto NextLPN;
                }
        }
        NextLPN: ;
    }
```

This code is used in §31.

§**35.** And here we list of alternatives followed by the relationship this notation has to the benchmark, e.g., "where 1 tonne = 1000 kg".

```
void index_lp_possibilities(literal_pattern *lp, literal_pattern *benchmark_lp) {
    int B = benchmark_lp->overall_multiplier;
    INDEX("    ");
    index_lp_possibility(lp, benchmark_lp);
    if (lp->equivalent_unit) {
        INDEX(" <i>where</i> ");
        lp_index_value_specific(lp, lp->overall_multiplier);
        INDEX(" = ");
        lp_index_value_specific(benchmark_lp, lp->overall_multiplier);
    } else {
        if (lp->overall_multiplier < B) {
            INDEX(" <i>where</i> ");
            lp_index_value_specific(lp, B);
            INDEX(" = ");
            lp_index_value_specific(benchmark_lp, B);
        }
        if (lp->overall_multiplier > B) {
            INDEX(" <i>where</i> ");
            lp_index_value_specific(lp, lp->overall_multiplier);
            INDEX(" = ");
            lp_index_value_specific(benchmark_lp, lp->overall_multiplier);
        }
    }
}
```

§**36.** This is where the list of alternatives, "1 tonne" followed by "2 tonnes", say, is produced:

```
void index_lp_possibility(literal_pattern *lp, literal_pattern *benchmark_lp) {
    if (lp == benchmark_lp) INDEX("<b>");
    if (lp->plural_form_only)
        lp_index_value_specific(lp, 2*lp->overall_multiplier);
    else
        lp_index_value_specific(lp, lp->overall_multiplier);
    if (lp == benchmark_lp) INDEX("</b>");
    if (lp->next_alternative_lp) {
        INDEX(" <i>or</i> ");
        index_lp_possibility(lp->next_alternative_lp, benchmark_lp);
    }
}
```

§**37. Printing values in an LP's notation to the index at compile-time.** This front-end routine chooses the most appropriate notation to use when indexing a given value. For instance, a mass of 1000000 is best expressed as "1 tonne", not "1000000 grams".

```
void lp_index_value(literal_pattern *lp_list, int v) {
    literal_pattern *lp;
    literal_pattern *lp_possibility = NULL;
    int k = 0;
    for (lp = lp_list; lp; lp = lp->next_for_this_kov) {
        if (v == 0) {
            if (lp->benchmark) {
                lp_index_value_specific(lp, v); return;
            }
        } else {
            if ((lp->primary_alternative) && (lp->equivalent_unit == FALSE)) {
                if ((lp_possibility == NULL) || (lp->overall_multiplier != k)) {
                    lp_possibility = lp;
                    k = lp->overall_multiplier;
                }
                if ((v-lp->overall_offset) >= (lp->overall_multiplier)) {
                    lp_index_value_specific(lp, v); return;
                }
            }
        }
    }
    if (lp_possibility) lp_index_value_specific(lp_possibility, v);
    else lp_index_value_specific(lp_list, v);
}
```

The function lp_index_value is called from 7/dim and 7/kix.

§**38.** Here we index the benchmark value. Pursuing our example of mass, if the benchmark is 1 kilogram, then the following indexes the value 1000 in kilograms, resulting in "1 kg". (This will always effectively look like "1 something", whatever the something is.)

```
void data_type_index_benchmark_value(int data_type) {
    literal_pattern *lp;
    LITERAL_FORMS_LOOP(lp, data_type)
        if (lp->benchmark) {
            lp_index_value_specific(lp, lp->overall_multiplier);
            return;
        }
    INDEX("1");
}
```

The function data_type_index_benchmark_value is called from 7/dim.

§**39.** We are rather formal when printing values to the index, so we choose not to make use of optional truncation.

```
void lp_index_value_specific(literal_pattern *lp, int v) {
    int tc, ec, remainder;
    remainder = (v - lp->overall_offset)%(lp->overall_multiplier);
    v = (v - lp->overall_offset)/(lp->overall_multiplier);
    for (tc=0, ec=0; tc<lp->no_lp_tokens; tc++) {
        if ((tc>0) && (lp->lp_tokens[tc].new_word_at)) INDEX(" ");
        switch (lp->lp_tokens[tc].lpt_type) {
            case WORD_LPT: ⟨Index a fixed word token within a literal pattern 40⟩; break;
            case CHARACTER_LPT: ⟨Index a character token within a literal pattern 41⟩; break;
            case ELEMENT_LPT: ⟨Index an element token within a literal pattern 42⟩; break;
            default: internal_error("unknown literal pattern token type");
        }
    }
}
```

§**40.** We parse in a case-insensitive way, but print back case-sensitively – note that the following uses the raw text of the word.

⟨Index a fixed word token within a literal pattern 40⟩ ≡

```
    if (tc > 0) INDEX(" ");
    print_literal_string_to_file(ifl, lw_array[lp->lp_tokens[tc].token_wn].lw_rawtext);
```

This code is used in §39.

§**41.**

⟨Index a character token within a literal pattern 41⟩ ≡

```
    html_char_out(ifl, lp->lp_tokens[tc].token_char);
```

This code is used in §39.

§**42.**

⟨Index an element token within a literal pattern 42⟩ ≡

```
    literal_pattern_element *lpe = &(lp->lp_elements[ec]);
    if (ec == 0) INDEX("%d", v/(lpe->element_multiplier));
    else {
        char *prototype = "%d";
        if ((lp->lp_tokens[tc].new_word_at == FALSE) && (lpe->without_leading_zeros == FALSE))
            prototype = leading_zero_prototype(lpe->element_range);
        INDEX(prototype, (v/(lpe->element_multiplier)) % (lpe->element_range));
    }
    if (ec == 0) ⟨Index the fractional part of the value 43⟩;
    ec++;
```

This code is used in §39.

§**43.**

⟨Index the fractional part of the value 43⟩ ≡

```
int ranger = 1;
while (lp->overall_multiplier > ranger) ranger = ranger*10;
remainder = remainder*(ranger/lp->overall_multiplier);
while ((remainder > 0) && ((remainder % 10) == 0)) {
    ranger = ranger/10; remainder = remainder/10;
}
if (remainder > 0) {
    INDEX(".");
    INDEX(leading_zero_prototype(ranger), remainder);
}
```

This code is used in §42.

§**44.** Please don't mention the words "logarithm" or "shift". It works fine.

```
char *leading_zero_prototype(int range) {
    if (range > 1000000000) return "%010d";
    if (range > 100000000) return "%09d";
    if (range > 10000000) return "%08d";
    if (range > 1000000) return "%07d";
    if (range > 100000) return "%06d";
    if (range > 10000) return "%05d";
    if (range > 1000) return "%04d";
    if (range > 100) return "%03d";
    if (range > 10) return "%02d";
    return "%d";
}
```

§45. Printing the I6 variable `value` out in an LP's notation at run-time.

```
void printing_routine_for_lp(OUTPUT_STREAM, literal_pattern *lp_list) {
    literal_pattern_name *lpn;
    literal_pattern *lp;
    int k;
    WRITE("which rem ran;\n"); INDENT;
    LOOP_OVER(lpn, literal_pattern_name) {
        if (lpn->word_ref1 >= 0) {
            k = 0;
            for (lp = lp_list; lp; lp = lp->next_for_this_kov)
                lp->marked_for_printing = FALSE;
            literal_pattern_name *lpn2;
            for (lpn2 = lpn; lpn2; lpn2 = lpn2->next)
                for (lp = lp_list; lp; lp = lp->next_for_this_kov)
                    if (lp == lpn2->can_use_this_lp) {
                        k++; lp->marked_for_printing = TRUE;
                    }
            if (k > 0) {
                WRITE("! The named notation: ");
                print_text_to_file(lpn->word_ref1, lpn->word_ref2, OUT);
                WRITE("\n");
                WRITE("if (which == %d) {\n", lpn->allocation_id + 1);
                ⟨Compile code to jump to the correct printing pattern 46⟩;
                WRITE("}\n\n");
            }
        }
    }
    for (lp = lp_list; lp; lp = lp->next_for_this_kov) {
        if ((lp->primary_alternative) && (lp->equivalent_unit == FALSE)) {
            k++; lp->marked_for_printing = TRUE;
        } else lp->marked_for_printing = FALSE;
    }
    ⟨Compile code to jump to the correct printing pattern 46⟩;
    WRITE("return;\n");
    for (lp = lp_list; lp; lp = lp->next_for_this_kov) {
        ⟨Print according to this particular literal pattern 47⟩;
        WRITE("return;\n");
    }
}
```

The function printing_routine_for_lp is called from 9/rsdt.

§46.

⟨Compile code to jump to the correct printing pattern 46⟩ ≡

```
int benchmark_id = -1;
for (lp = lp_list; lp; lp = lp->next_for_this_kov)
    if (lp->marked_for_printing)
        if (lp->benchmark) benchmark_id = lp->allocation_id;
if (benchmark_id >= 0)
    WRITE("if (value == 0) jump Use_LP_%d;\n", benchmark_id);
literal_pattern *last_lp = NULL, *last_singular = NULL, *last_plural = NULL;
for (lp = lp_list; lp; lp = lp->next_for_this_kov) {
    if (lp->marked_for_printing) {
        char *op = ">="; last_lp = lp;
        if (lp->singular_form_only) { last_singular = lp; op = "=="; }
        if (lp->plural_form_only) { last_plural = lp; op = ">"; }
        WRITE("if ((value - %d) %s %d) jump Use_LP_%d;\n",
            lp->overall_offset, op, lp->overall_multiplier, lp->allocation_id);
    }
}
if (last_lp) {
    if ((last_lp->singular_form_only) &&
        (last_plural) &&
        (last_plural->overall_offset == last_lp->overall_offset) &&
        (last_plural->overall_multiplier == last_lp->overall_multiplier))
        WRITE("jump Use_LP_%d;\n", last_plural->allocation_id);
    WRITE("jump Use_LP_%d;\n", last_lp->allocation_id);
}
```

This code is used in §45.

§47.

⟨Print according to this particular literal pattern 47⟩ ≡

```
int tc, ec=0, oc=0;
WRITE("\n");
comment_use_of_lp(OUT, lp);
WRITE(".Use_LP_%d;\n", lp->allocation_id);
if (lp->overall_offset != 0) WRITE("value = value - %d;\n", lp->overall_offset);
if (lp->overall_multiplier != 1) {
    WRITE("rem = value%%%d;\n", lp->overall_multiplier);
    WRITE("value = value/%d;\n", lp->overall_multiplier);
}
for (tc=0; tc<lp->no_lp_tokens; tc++) {
    if (lp->lp_elements[ec].preamble_optional)
        ⟨Truncate the printed form here if subsequent numerical parts are zero 52⟩;
    if ((tc>0) && (lp->lp_tokens[tc].new_word_at))
        WRITE("print \" \";\n");
    switch (lp->lp_tokens[tc].lpt_type) {
        case WORD_LPT: ⟨Compile I6 code to print a fixed word token within a literal pattern 48⟩; break;
        case CHARACTER_LPT: ⟨Compile I6 code to print a character token within a literal pattern 49⟩; break;
        case ELEMENT_LPT: ⟨Compile I6 code to print an element token within a literal pattern 50⟩; break;
        default: internal_error("unknown literal pattern token type");
    }
}
```

This code is used in §45.

§48.

⟨Compile I6 code to print a fixed word token within a literal pattern 48⟩ ≡

```
WRITE("print \"");
isn_compile_string(OUT, lw_array[lp->lp_tokens[tc].token_wn].lw_rawtext, 0);
WRITE("\";\n");
```

This code is used in §47.

§49.

⟨Compile I6 code to print a character token within a literal pattern 49⟩ ≡

```
char tiny_string[2];
tiny_string[0] = lp->lp_tokens[tc].token_char; tiny_string[1] = 0;
WRITE("print \"");
isn_compile_string(OUT, tiny_string, 0);
WRITE("\";\n");
```

This code is used in §47.

§50.

⟨Compile I6 code to print an element token within a literal pattern 50⟩ ≡

```
literal_pattern_element *lpe = &(lp->lp_elements[ec]);
if (lpe->element_optional)
    ⟨Truncate the printed form here if subsequent numerical parts are zero 52⟩;
oc = ec + 1;
if (ec == 0) {
    if (lp->number_signed)
        WRITE("if ((value<0) && (value/%d == 0)) print \"-\";\n", lpe->element_multiplier);
    WRITE("print value/%d;\n", lpe->element_multiplier);
    if (lp->number_signed) WRITE("if (value<0) value=-value;\n");
} else {
    if ((lp->lp_tokens[tc].new_word_at == FALSE) &&
        (lpe->without_leading_zeros == FALSE)) {
        int pow = 1;
        for (pow = 1000000000; pow>1; pow = pow/10)
            if (lpe->element_range > pow)
                WRITE("if ((value/%d)%%%d < %d) print \"0\";\n",
                    lpe->element_multiplier, lpe->element_range, pow);
    }
    WRITE("print (value/%d)%%%d;\n", lpe->element_multiplier, lpe->element_range);
}
if (ec == 0) ⟨Compile I6 code to print the fractional part of the value 51⟩;
ec++;
```

This code is used in §47.

§**51.** We need to print a decimal approximation to the fraction `rem/ran`.

⟨Compile I6 code to print the fractional part of the value 51⟩ ≡

```
    int remm = 1, ran = 1;
    while (lp->overall_multiplier > ran) ran = ran*10;
    remm = ran/lp->overall_multiplier;
    WRITE("if (rem > 0) {\n"); INDENT;
    WRITE("print \".\";\n");
    if (remm != 1) WRITE("rem = rem*%d;\n", remm);
    WRITE("ran = %d;\n", ran);
    WRITE("while ((rem %% 10 == 0) && (rem > 0)) { rem=rem/10; ran=ran/10; }\n");
    WRITE("while (rem < ran/10) { print \"0\"; ran=ran/10; }\n");
    WRITE("print rem;\n");
    OUTDENT; WRITE("}\n");
```

This code is used in §50.

§**52.**

⟨Truncate the printed form here if subsequent numerical parts are zero 52⟩ ≡

```
    if (oc == ec) {
        if (ec == 0)
            WRITE("if ((value/%d) == 0) rtrue;\n",
                lp->lp_elements[ec].element_multiplier);
        else
            WRITE("if ((value/%d)%%%d == 0) rtrue;\n",
                lp->lp_elements[ec].element_multiplier,
                lp->lp_elements[ec].element_range);
        oc = ec + 1;
    }
```

This code is used in §47,50,47,50,47,50.

§**53.**

```
void comment_use_of_lp(OUTPUT_STREAM, literal_pattern *lp) {
    WRITE("! ");
    print_text_to_file(lp->prototype_w1, lp->prototype_w2, OUT);
    WRITE(", with offset=%d, multiplier=%d\n", lp->overall_offset, lp->overall_multiplier);
    WRITE("\n");
}
```

§**54. Creating new literal patterns.** We create new LPs during traverse 1 of the parse tree. This imposes two timing constraints:

(a) The specification sentence must come after the sentence creating the kind of value being specified; but
(b) It must come before any sentences using constants written in this notation.

In practice both constraints seem to be accepted by users as reasonable, and this causes no trouble.

```
sentence_handler SPECIFIES_SH_handler =
    { SENTENCE_NT, SPECIFIES_VB, 1, new_literal_specification };

void new_literal_specification(parse_node *pn) {
    int ltw1, ltw2, spw1, spw2;
    [[spw1, spw2 <-- pn->down->next]];                    the text of the notation
    [[ltw1, ltw2 <-- pn->down->next->next]];              the name of the KOV specified
    new_literal_specification_list(spw1, spw2, ltw1, ltw2, NULL);
}
```

§**55.** One can define LPs with a list of alternatives, of which the first is the "primary alternative" and said to be the "owner" of the rest. For instance:

> 1 tonne (in metric units, in tonnes, singular) or 2 tonnes (in metric units,
> in tonnes, plural) specifies a mass scaled up by 1000.

Here we call `new_literal_specification_list` on "1 tonne" with no owner, and then on "2 tonnes" with the "1 tonne" LP as its owner.

```
literal_pattern *new_literal_specification_list(int spw1, int spw2, int ltw1, int ltw2,
    literal_pattern *lp_main) {
    if (is_list_divided(spw1, spw2, LOOK_FOR_OR)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        lp_main = new_literal_specification_list(lw1, lw2, ltw1, ltw2, lp_main);
        return new_literal_specification_list(rw1, rw2, ltw1, ltw2, lp_main);
    }
    return new_literal_specification_inner(spw1, spw2, ltw1, ltw2, lp_main);
}
```

§**56.** So here we parse a single LP.

```
define LP_SCALED_UP 1
define LP_SCALED_DOWN -1
define LP_SCALED_AT 2

literal_pattern *new_literal_specification_inner(int spw1, int spw2, int ltw1, int ltw2,
    literal_pattern *owner) {
    int i, scaled = 1, scaled_dir = LP_SCALED_UP, offset = 0;
    int notation_name_w1 = -1, notation_name_w2 = -1, parts_w1 = -1, parts_w2 = -1;
    int equivalent_w1 = -1, equivalent_w2 = -1;
    int offset_w1 = -1, offset_w2 = -1;
    kind_of_value *kov;
    literal_pattern *lp;
    ⟨Parse and remove notation name at the end of the pattern 58⟩;
    ⟨Parse and remove optional clauses at the end of the specification 59⟩;
    ⟨Check that the new notation does not overlap with that of any existing LP 62⟩;
    ⟨Find the kind of value being specified by the new LP 63⟩;
    ⟨Deal with the case where a dimensionality rule is being specified 64⟩;
```

```
    ⟨Check that the KOV is acceptable as the owner of a LP 65⟩;
    ⟨Check that any other value mentioned as an equivalent or scaled equivalent has the right KOV 61⟩;
    lp = lp_new(kov, spw1, spw2);
    lp->overall_offset = offset;
    if (owner == NULL) lp->primary_alternative = TRUE;
    if (equivalent_w1 >= 0) lp->equivalent_unit = TRUE;
    switch (scaled_dir) {
        case LP_SCALED_UP: lp->scaled_up = scaled; break;
        case LP_SCALED_DOWN: lp->scaled_down = scaled; break;
        case LP_SCALED_AT: lp->scaled_at = scaled; break;
    }
    lp_set_notations(lp, notation_name_w1, notation_name_w2);
    ⟨Break down the specification text into tokens and elements 66⟩;
    ⟨Calculate the multipliers for packing the elements into a single integer 69⟩;
    if (kov_list_of_literal_forms(kov) == NULL) lp->benchmark = TRUE;
    kov_add_literal_pattern(kov, lp);
    if (parts_w1 >= 0) {
        ⟨Work through parts text to assign names to the individual elements 70⟩;
        ⟨Check that any notes to do with optional elements are mutually compatible 71⟩;
        define_packing_phrases(lp, ltw1, ltw2);
    }
    if (owner == NULL) owner = lp;
    else ⟨Add this new alternative to the list belonging to our owner 57⟩;
    return owner;
}
```

§57.

⟨Add this new alternative to the list belonging to our owner 57⟩ ≡

```
    literal_pattern *alt = owner;
    while ((alt) && (alt->next_alternative_lp)) alt = alt->next_alternative_lp;
    alt->next_alternative_lp = lp;
```

This code is used in §56.

§58.

⟨Parse and remove notation name at the end of the pattern 58⟩ ≡

```
    int i;
    [[spw1, spw2 == ... OPENBRACKET ... CLOSEBRACKET : i --> spw1, spw2 ... notation_name_w1, notation_name_w2]]
;
```

This code is used in §56.

§59.

⟨Parse and remove optional clauses at the end of the specification 59⟩ ≡

```
    int clause = TRUE;
    while (clause) {
        clause = FALSE;
        ⟨Look for an optional clause attached to the specification 60⟩;
    }
```

This code is used in §56.

§60.

⟨Look for an optional clause attached to the specification 60⟩ ≡

```
if ([[ltw1, ltw2 == ... with ... : i]] &&
    [[i, ltw2 == with parts ... --> parts_w1, parts_w2]]) {
    ltw2 = i-1; clause = TRUE;
}
if (vocab_test_flags(ltw2, NUMBER_MC)) {
    int v = vocab_get_literal_number_value(lw_array[ltw2].lw_identity);
    if [[ltw1, ltw2 == ... scaled up by ### --> ltw1, ltw2]] {
        scaled = v; scaled_dir = LP_SCALED_UP; clause = TRUE;
    } else
    if [[ltw1, ltw2 == ... scaled down by ### --> ltw1, ltw2]] {
        scaled = v; scaled_dir = LP_SCALED_DOWN; clause = TRUE;
    } else
    if [[ltw1, ltw2 == ... scaled at ### --> ltw1, ltw2]] {
        scaled = v; scaled_dir = LP_SCALED_AT; clause = TRUE;
    }
}
if ([[ltw1, ltw2 == ... offset ... : i]] &&
    [[i, ltw2 == offset by ... --> offset_w1, offset_w2]]) {
    ltw2 = i-1; clause = TRUE;
}
if ([[ltw1, ltw2 == ... equivalent ... : i]] &&
    [[i, ltw2 == equivalent to ... --> equivalent_w1, equivalent_w2]]) {
    ltw2 = i-1; clause = TRUE;
}
```

This code is used in §59.

§61.

⟨Check that any other value mentioned as an equivalent or scaled equivalent has the right KOV 61⟩ ≡

```
if (equivalent_w1 >= 0) {
    if (kov_compare(is_a_literal(equivalent_w1, equivalent_w2), kov)) {
        scaled_dir = LP_SCALED_UP; scaled = last_literal_evaluated;
    } else {
        sentence_problem_with_note(_P_(C5BadLPEquivalent),
            "the equivalent value needs to be a constant of the same kind "
            "of value as you are specifying",
            "and this seems not to be.",
            "Note that you can only use notations specified in sentences "
            "before the current one.");
    }
}
if (offset_w1 >= 0) {
    if (kov_compare(is_a_literal(offset_w1, offset_w2), kov)) {
        offset = last_literal_evaluated;
    } else {
        sentence_problem_with_note(_P_(C5BadLPOffset),
            "the offset value needs to be a constant of the same kind "
            "of value as you are specifying",
            "and this seems not to be.",
```

```
            "Note that you can only use notations specified in sentences "
            "before the current one.");
    }
}
```

This code is used in §56.

§**62.** We parse the specification text as if it were a constant value, hoping for the result `NULL` – so that it doesn't already mean something else. During this process, we waive checking of numerical overflows in matching an LP: this is done so that

3/13 specifies a bar. 2/19 specifies a foo.

reports "2/19" as a duplicate using the following problem message, but does not throw a problem message as being a bar which is out of range (because in the bar notation, the number after the slash can be at most 13, so that 19 is illegal).

⟨Check that the new notation does not overlap with that of any existing LP 62⟩ ≡

```
waive_lp_overflows = TRUE; kov = is_a_literal(spw1, spw2); waive_lp_overflows = FALSE;
if (kov) {
    quote_source(1, current_sentence);
    quote_kov(2, kov);
    quote_words(3, spw1, spw2);
    handmade_problem(_P_(C5DuplicateUnitSpec));
    issue_problem_segment(
        "In the sentence %1, it looks as if you intend to give a new meaning "
        "to expressions like '%3', but this is already something I "
        "recognise - specifying %2 - so a more distinctive specification "
        "must be chosen.");
    issue_problem_end();
    return owner;
}
```

This code is used in §56.

§**63.** For the present, at least, LPs can only be used to specify a limited range of possible values:

⟨Find the kind of value being specified by the new LP 63⟩ ≡

```
specification *spec = parse_expression(ltw1, ltw2, TYPE_EXPCON);
if (spec_is_actual_CONSTANT(spec)) {
    sentence_problem(_P_(C5LPNotKOV),
        "you can only specify ways to write kinds of value",
        "as created with sentences like 'A weight is a kind of value.'");
    return owner;
}
kov = spec_get_kind_of_value(spec);
```

This code is used in §56.

§**64.** We react to the word "times", in the specification, only when the matter either side of it can be recognised as something; this in principle makes it possible to define LPs containing the word, though in fact that is very unlikely to end happily, given that "times" will usually be interpreted as multiplication when parsing values in any case.

⟨Deal with the case where a dimensionality rule is being specified 64⟩ ≡

```
int left_w1, left_w2, right_w1, right_w2;
if [[spw1, spw2 == ... times ... : i --> left_w1, left_w2 ... right_w1, right_w2]] {
    specification *left_spec = parse_expression(left_w1, left_w2, TYPE_EXPCON);
    specification *right_spec = parse_expression(right_w1, right_w2, TYPE_EXPCON);
    if ((spec_is_UNKNOWN(left_spec) == FALSE) && (spec_is_UNKNOWN(right_spec) == FALSE)) {
        if ((spec_is_generic_CONSTANT(left_spec)) && (spec_is_generic_CONSTANT(right_spec)))
            dim_set_multiplication(
                spec_get_kind_of_value(left_spec), spec_get_kind_of_value(right_spec), kov);
        else
            sentence_problem(_P_(C5MultiplyingNonKOVs),
                "only kinds of value can be multiplied here",
                "and only in a sentence like 'A length times a length "
                "specifies an area.'");
        return owner;
    }
}
```

This code is used in §56.

§**65.** This checking did not apply to the dimensionality rules case, which looks after itself.

⟨Check that the KOV is acceptable as the owner of a LP 65⟩ ≡

```
if (kov_is_built_in(kov)) {
    sentence_problem(_P_(C5LPBuiltInKOV),
        "you can only specify ways to write new kinds of value",
        "as created with sentences like 'A weight is a kind of value.', "
        "and not the built-in ones like 'number' or 'time'.");
    return owner;
}
if (kov_convert_to_unit(kov) == FALSE) {
    sentence_problem(_P_(C5LPEnumeration),
        "this is a kind of value which already has named values",
        "so it can't have a basically numerical form as well.");
    return owner;
}
```

This code is used in §56.

§**66.** Each word is either a whole token in itself, or a stream of tokens representing alphabetic vs numeric pieces of a word:

⟨Break down the specification text into tokens and elements 66⟩ ≡

```
int i, j, tc, ec;
for (i=0, tc=0, ec=0; spw1+i<=spw2; i++) {
    literal_pattern_token new_token;
    int digit_found = FALSE;
    char *text_of_word = lw_array[spw1+i].lw_rawtext;
    for (j=0; text_of_word[j]; j++) if (isdigit(text_of_word[j])) digit_found = TRUE;
    if (digit_found)
        ⟨Break up the word into at least one element token, and perhaps also character tokens 67⟩
    else {
        new_token = lpt_new(WORD_LPT, TRUE);
        new_token.token_wn = spw1+i;
        ⟨Add new token to LP 68⟩;
    }
}
lp->no_lp_tokens = tc;
lp->no_lp_elements = ec;
if (lp->no_lp_elements == 0) {
    sentence_problem(_P_(C5LPWithoutElement),
        "a way to specify a kind of value must involve numbers",
        "so '10kg specifies a weight' is allowed, but not 'tonne "
        "specifies a weight'.");
    return owner;
}
```

This code is used in §56.

§**67.** Bounds checking is easier here since we know that a LP specification will not ever need to create the maximum conceivable value which a C integer can hold – so we need not fool around with long long ints.

⟨Break up the word into at least one element token, and perhaps also character tokens 67⟩ ≡

```
int j, sgn = 1, next_token_begins_word = TRUE;
for (j=0; text_of_word[j]; j++) {
    int tot = 0, digit_found = FALSE;
    if ((text_of_word[j] == '-') && (isdigit(text_of_word[j+1])) && (ec == 0)) {
        sgn = -1; continue;
    }
    while (isdigit(text_of_word[j++])) {
        digit_found = TRUE;
        if (tot > 999999999) {
            sentence_problem(_P_(C5LPElementTooLarge),
                "that specification contains numbers that are too large",
                "and would construct values which could not sensibly "
                "be stored at run-time.");
            return owner;
        }
        tot = 10*tot + (text_of_word[j-1]-'0');
    }
    j--;
    if (digit_found) {
        literal_pattern_element new_element = lpe_new(ec, tot+1, sgn);
```

```
            if (ec >= MAX_ELEMENTS_PER_LITERAL) {
                sentence_problem(_P_(C5LPTooManyElements),
                    "that specification contains too many numerical elements",
                    "and is too complicated for Inform to handle.");
                return owner;
            }
            lp->lp_elements[ec++] = new_element;
            if (sgn == -1) lp->number_signed = TRUE;
            new_token = lpt_new(ELEMENT_LPT, next_token_begins_word);
            ⟨Add new token to LP 68⟩;
            j--;
        } else {
            new_token = lpt_new(CHARACTER_LPT, next_token_begins_word);
            new_token.token_char = text_of_word[j];
            ⟨Add new token to LP 68⟩;
        }
        sgn = 1; next_token_begins_word = FALSE;
    }
```

This code is used in §66.

§**68.** In fact counting tokens is not necessarily a good way to measure the complexity of an LP, since any long run of characters in a word which also contains a number will splurge the number of tokens. So `MAX_TOKENS_PER_LITERAL` is set to a high enough value that this will not really distort matters.

⟨Add new token to LP 68⟩ ≡

```
    if (tc >= MAX_TOKENS_PER_LITERAL) {
        sentence_problem(_P_(C5LPTooComplicated),
            "that specification is too complicated",
            "and will have to be shortened.");
        return owner;
    }
    lp->lp_tokens[tc++] = new_token;
```

This code is used in §66,67,66,67,66,67.

§**69.** The elements are created in parsing order, that is, left to right. But the multipliers can only be calculated by working from right to left, so this is deferred until all elements exist, at which point we –

⟨Calculate the multipliers for packing the elements into a single integer 69⟩ ≡

```
    int i, m = 1;
    for (i=lp->no_lp_elements-1; i>=0; i--) {
        literal_pattern_element *lpe = &(lp->lp_elements[i]);
        lpe->element_multiplier = m;
        m = m*(lpe->element_range);
    }
```

This code is used in §56.

§**70.** Today, we have naming of parts:

⟨Work through parts text to assign names to the individual elements 70⟩ ≡

```
int i, named_parts = 0;
for (i=0, named_parts = 0; (i<lp->no_lp_elements) && (parts_w1 <= parts_w2); i++) {
    literal_pattern_element *lpe = &(lp->lp_elements[i]);
    int x1, x2, opt1, opt2, j;
    lpe->word_ref1 = parts_w1;
    if (is_list_divided(parts_w1, parts_w2, LOOK_FOR_AND)) {
        lpe->word_ref2 = left_w2; parts_w1 = right_w1;
    } else {
        lpe->word_ref2 = parts_w2; parts_w1 = parts_w2+1;
    }
    if [[lpe == ... OPENBRACKET ... CLOSEBRACKET : j --> x1, x2 ... opt1, opt2]] {
        lpe->word_ref1 = x1; lpe->word_ref2 = x2;
        literal_specification_option(lpe, opt1, opt2);
    }
    named_parts++;
}
if ((named_parts > 0) && (named_parts != lp->no_lp_elements)) {
    sentence_problem(_P_(C5LPNotAllNamed),
        "you must supply names for all the parts",
        "if for any");
    return owner;
}
```

This code is used in §56.

§**71.** In fact, the test is a simple one: there can be only one element declared optional, and it must not be the first.

⟨Check that any notes to do with optional elements are mutually compatible 71⟩ ≡

```
int i, opt_count = 0;
for (i=0; i<lp->no_lp_elements; i++) if (lp->lp_elements[i].element_optional) {
    opt_count++;
    if (i == 0) {
        sentence_problem(_P_(C5LPFirstOptional),
            "the first part is not allowed to be optional",
            "since it is needed to identify the value.");
        return owner;
    }
}
if (opt_count >= 2) {
    sentence_problem(_P_(C5LPMultipleOptional),
        "only one part can be called optional",
        "since if any part is omitted then so are all subsequent parts.");
    return owner;
}
```

This code is used in §56.

§**72.** Finally, a small recursive routine looks at the bracketed notes attached to names of elements, and translates them directly to flags in the LPE structure:

```
void literal_specification_option(literal_pattern_element *lpe, int x1, int x2) {
    if (is_list_divided(x1, x2, LOOK_FOR_AND)) {
        x1 = right_w1; x2 = right_w2;
        literal_specification_option(lpe, left_w1, left_w2);
        literal_specification_option(lpe, x1, x2);
        return;
    }
    if [[x1, x2 == optional]] { lpe->element_optional = TRUE; return; }
    if [[x1, x2 == preamble optional]] {
        lpe->element_optional = TRUE; lpe->preamble_optional = TRUE; return;
    }
    if [[x1, x2 == without leading zeros]] { lpe->without_leading_zeros = TRUE; return; }
    quote_source(1, current_sentence);
    quote_words(2, x1, x2);
    handmade_problem(_P_(C5BadLPPartOption));
    issue_problem_segment(
        "In the specification %1, I was expecting that '%2' would be an optional "
        "note about one of the parts: it should have been one of 'optional', "
        "'preamble optional' or 'without leading zeros'.");
    issue_problem_end();
}
```

§**73. I7 phrases to print values in specified ways.** When an LP has a name, it's a notation which the source text can request to be used in saying a value.

```
void lp_set_notations(literal_pattern *lp, int w1, int w2) {
    if ((w1<0) || (w2<w1)) return;
    int lw1, lw2, rw1, rw2, j;
    if [[w1, w2 == ... COMMA ... : j --> lw1, lw2 ... rw1, rw2]] {
        lp_set_notations(lp, lw1, lw2);
        lp_set_notations(lp, rw1, rw2);
        return;
    }
    if [[w1, w2 == singular]] { lp->singular_form_only = TRUE; return; }
    if [[w1, w2 == plural]] { lp->plural_form_only = TRUE; return; }
    if [[w1, w2 == in ...]] {
        literal_pattern_name *lpn;
        LOOP_OVER(lpn, literal_pattern_name) {
            if (compare_word_range(lpn->word_ref1, lpn->word_ref2, w1, w2)) {
                literal_pattern_name *new = CREATE(literal_pattern_name);
                new->word_ref1 = -1; new->word_ref2 = -1;
                new->can_use_this_lp = lp;
                new->next = NULL;
                while ((lpn) && (lpn->next)) lpn = lpn->next;
                lpn->next = new;
                return;
            }
        }
        literal_pattern_name *new = CREATE(literal_pattern_name);
        new->word_ref1 = w1; new->word_ref2 = w2;
        new->can_use_this_lp = lp;
        new->next = NULL;
    } else {
        quote_source(1, current_sentence);
        quote_words(2, w1, w2);
        handmade_problem(_P_(C5BadLPNameOption));
        issue_problem_segment(
            "In the specification %1, I was expecting that '%2' would be an optional "
            "note about one of the notations: it should have been one of 'singular', "
            "'plural' or 'in ...'.");
        issue_problem_end();
    }
}
```

§**74.** And this is where the corresponding text substitutions are declared.

```
void define_named_printing_phrases(void) {
    literal_pattern_name *lpn;
    LOOP_OVER(lpn, literal_pattern_name)
        lpn->lpn_compiled_already = FALSE;
    LOOP_OVER(lpn, literal_pattern_name) {
        if (lpn->word_ref1 >= 0) {
            literal_pattern_name *lpn2;
            for (lpn2 = lpn; lpn2; lpn2 = lpn2->next)
                if (lpn2->lpn_compiled_already == FALSE)
                    ⟨Compile the printing phrase for this and perhaps subsequent LPs 75⟩;
        }
    }
    register_recently_lexed_phrases();
}
```

The function define_named_printing_phrases is invoked by a command in a `.i6t` template file.

§**75.** These text substitutions correspond exactly neither to the LPs nor to the names. For instance, "in tonnes" produces a text substitution which takes in both the LP for "1 tonne" and for "2 tonnes", deciding at run-time which to use. And on the other hand, "in metric units" may produce text substitutions for many different KOVs, distinguished by type-checking:

To say (val - mass) in metric units: ...

To say (val - length) in metric units: ...

The following creates one text substitution for each different KOV among the LPs under each named possibility.

⟨Compile the printing phrase for this and perhaps subsequent LPs 75⟩ ≡

```
    kind_of_value *kov = lpn2->can_use_this_lp->kov_specified;
    int ltw1, ltw2;
    int name1 = lpn->word_ref1, name2 = lpn->word_ref2;
    kov_get_name(kov, &ltw1, &ltw2, FALSE);
    int x1 = lexer_wordcount, x2;
    feed_into_lexer("To say ( val - ", FALSE, FALSE);
    splice_words(ltw1, ltw2);
    feed_into_lexer(" ) ", FALSE, FALSE);
    splice_words(name1, name2);
    x2 = lexer_wordcount-1;
    make_sentence_node(x1, x2, ':');
    x1 = lexer_wordcount;
    char print_rule_buff[1024];
    sprintf(print_rule_buff, " (- %s({val}, %d); -) ",
        kov_get_name_of_printing_rule(kov), lpn->allocation_id + 1);
    feed_into_lexer(print_rule_buff, FALSE, FALSE);
    x2 = lexer_wordcount-1;
    make_sentence_node(x1, x2, '.');
    literal_pattern_name *lpn3;
    for (lpn3 = lpn2; lpn3; lpn3 = lpn3->next)
        if (kov_compare(kov, lpn3->can_use_this_lp->kov_specified))
            lpn3->lpn_compiled_already = TRUE;
```

This code is used in §74.

§76. I7 phrases to pack and unpack the value. Creating a LP implicitly defines further I7 source text, as follows.

```
void define_packing_phrases(literal_pattern *lp, int ltw1, int ltw2) {
    ⟨Define phrases to convert from a packed value to individual parts 77⟩;
    ⟨Define a phrase to convert from numerical parts to a packed value 78⟩;
    register_recently_lexed_phrases();
}
```

§**77.** First, we automatically create n phrases to unpack the elements given the value. For instance, defining:

> $10.99 specifies a price with parts dollars and cents.

automatically generates:

> To define which number is dollars part of (full - price) :
> `(- ({full}/100) -).`
>
> To define which number is cents part of (full - price) :
> `(- ({full}%100) -).`

⟨Define phrases to convert from a packed value to individual parts 77⟩ ≡

```
    int i;
    for (i=0; i<lp->no_lp_elements; i++) {
        literal_pattern_element *lpe = &(lp->lp_elements[i]);
        int x1, x2;
        char print_rule_buff[1024];
        x1 = lexer_wordcount;
        feed_into_lexer("To decide which number is ", FALSE, FALSE);
        splice_words(lpe->word_ref1, lpe->word_ref2);
        feed_into_lexer(" part of ( full - ", FALSE, FALSE);
        splice_words(ltw1, ltw2);
        feed_into_lexer(" ) ", FALSE, FALSE);
        x2 = lexer_wordcount-1;
        make_sentence_node(x1, x2, ':');
        x1 = lexer_wordcount;
        if (i==0)
            sprintf(print_rule_buff, " (- ({full}/%d) -) ", lpe->element_multiplier);
        else if (lpe->element_multiplier > 1)
            sprintf(print_rule_buff, " (- (({full}/%d)%%%d) -) ",
                lpe->element_multiplier, lpe->element_range);
        else
            sprintf(print_rule_buff, " (- ({full}%%%d) -) ", lpe->element_range);
        feed_into_lexer(print_rule_buff, FALSE, FALSE);
        x2 = lexer_wordcount-1;
        make_sentence_node(x1, x2, '.');
    }
```

This code is used in §76.

§**78.** And similarly, a packing phrase to calculate the value given its elements. For instance, the dollars-and-cents example compiles:

```
To decide which price is price with dollars part ( part0 - a number ) cents part ( part1 - a number) :
(- ({part0}*100+{part1}) -).
```

⟨Define a phrase to convert from numerical parts to a packed value 78⟩ ≡

```
if (lp->no_lp_elements > 0) {
    int i, x1, x2;
    char print_rule_buff[1024];
    x1 = lexer_wordcount;
    if [[ltw1, ltw2 == a/an ... --> ltw1, ltw2]];
    feed_into_lexer("To decide which ", FALSE, FALSE);
    splice_words(ltw1, ltw2);
    feed_into_lexer(" is ", FALSE, FALSE);
    splice_words(ltw1, ltw2);
    feed_into_lexer(" with ", FALSE, FALSE);
    for (i=0; i<lp->no_lp_elements; i++) {
        literal_pattern_element *lpe = &(lp->lp_elements[i]);
        sprintf(print_rule_buff, " part%d ", i);
        splice_words(lpe->word_ref1, lpe->word_ref2);
        feed_into_lexer(" part ( ", FALSE, FALSE);
        feed_into_lexer(print_rule_buff, FALSE, FALSE);
        feed_into_lexer(" - a number ) ", FALSE, FALSE);
    }
    x2 = lexer_wordcount-1;
    make_sentence_node(x1, x2, ':');
    x1 = lexer_wordcount;
    sprintf(print_rule_buff, " (- (");
    for (i=0; i<lp->no_lp_elements; i++) {
        literal_pattern_element *lpe = &(lp->lp_elements[i]);
        if (i>0) sprintf(print_rule_buff+strlen(print_rule_buff), "+");
        if (lpe->element_multiplier != 1)
            sprintf(print_rule_buff+strlen(print_rule_buff), "%d*",
                lpe->element_multiplier);
        sprintf(print_rule_buff+strlen(print_rule_buff), "{part%d}", i);
    }
    sprintf(print_rule_buff+strlen(print_rule_buff), ") -) ");
    feed_into_lexer(print_rule_buff, FALSE, FALSE);
    x2 = lexer_wordcount-1;
    make_sentence_node(x1, x2, '.');
}
```

This code is used in §76.

Excerpt Meanings 5/em

Purpose

To register and deregister meanings for excerpts of text as nouns, adjectives, imperative phrases and other usages.

Definitions

¶1. Excerpt meanings. Most compilers keep a "symbols table" of identifier names and what meanings they have: for instance, when compiling Inform, GCC's symbols table records that `problem_count` is the name of an integer variable and `excerpt_meaning` of a defined type. This is usually stored so that a new name can rapidly be checked to see if it matches one that is currently known.

Inform has a similar need to remember meanings of excerpts. (Recall that an "excerpt" is a run of one or more adjacent words in the source text.) So far in Chapter 5 we have already seen how determiners, verbs and prepositions are parsed – by direct comparison against fixed wordings stored in specialised structures. The excerpt meanings, by contrast, form a single vocabulary bank for nouns, adjectives, imperative phrases, and phrases to make decisions – which is to say, almost everything else. Examples include:

> american dialect, say close bracket, player's command, open, Hall of Mirrors

Most compilers use a symbols table whose efficiency depends on the fact that symbol names are relatively long strings (say, 8 or more units) drawn from a small alphabet (say, the 37 letters, digits and the underscore). But Inform has short symbols (typically one to three units) drawn from a huge alphabet (say, all 5,000 different words found in the source text). And we also need to parse in ways which a conventional compiler does not. If C has registered the identifier `pink_martini` then it never needs to notice `pnk_martin` as being related to it. But when Inform registers "pink martini" as the name of a world object, it then has to spot that either "pink" or "martini" alone might also refer to the same object. So we are not going to use the conventional algorithms.

¶2. We now define the `excerpt_meaning` data structure, which holds a single entry in this super-dictionary. The text to be matched is specified as a sequence of at least one, and at most 32, tokens: these can either be pointers to specific vocabulary, or can be null, which implies that arbitrary non-empty text can appear in the given position. (It is forbidden for the token list to contain two nulls in a row.) For instance, the token list:

```
drink # milk #
```

matches "drink more milk today and every day", but not "drink milk". (The sharp symbol `#` is printed in place of a null token, both in this documentation and in the debugging log.)

Each excerpt meaning also comes with a hash code, which is automatically generated from its token list, and a pointer to some structure. For a world object, this will be a pointer to the appropriate `world_object` structure; for a quantity, a pointer to the `quantity`; and so on. Because the pointer can only be used if we know what category of meaning we are looking at, the excerpt meaning is also supplied with a pair of integers called the meaning code (MC) and secondary meaning code (SMC).

```
define MAX_TOKENS_PER_EXCERPT_MEANING 32

typedef struct excerpt_meaning {
    int meaning_code;                          what kind of meaning: a single MC, not a bitmap
    int secondary_code;                        relevant only if the MC is MISCELLANEOUS_MC
    struct general_pointer data;               data structure being referred to
```

```
    int no_em_tokens;                                                    length of token list
    struct vocabulary_entry *em_tokens[MAX_TOKENS_PER_EXCERPT_MEANING];  token list
    int excerpt_hash;                                                    hash code generated from the token list
    MEMORY_MANAGEMENT
} excerpt_meaning;
```

The structure excerpt_meaning is shared with 5/parse.

¶3. One meaning we find it convenient to have direct access to: the meaning of the noun "player".

```
excerpt_meaning *meaning_of_player = NULL;
```

¶4. Meaning codes. These assign a context to a meaning, and so decide how the `data` pointer for an excerpt meaning is to interpreted. For instance, "Persian carpet" might have a meaning with code `WORLD_OBJECT_MC`, since it refers to a world object (a thing or kind): the pointer will then be to a `world_object`.

Meaning codes are used in other contexts in Inform besides this one. There are exactly 31 of them and each is a distinct power of two; there is no significance to their ordering. The point is that a signed integer (which we know can hold values at least up to $2^{31} - 1$) can hold a bitmap representing any subset of these meaning codes; for instance, `QUANTITY_MC + TABLE_MC` would mean "either a quantity or a table".

¶5. Meaning codes are divided into two. The first set represent meanings of excerpts; the `meaning_code` field of an `excerpt_meaning` is always exactly one of the following. (It is never a bitmap combination.)

```
define MISCELLANEOUS_MC 0x00000001                    see the secondary meaning code below
define VOID_PHRASE_MC 0x00000002                      e.g., award # points
define VALUE_PHRASE_MC 0x00000004                     e.g., number of rows in #
define COND_PHRASE_MC 0x00000008                      e.g., # has not ended
define WORLD_OBJECT_MC 0x00000010                     e.g., upright chair
define PROPERTY_MC 0x00000020                         e.g., carrying capacity
define QUANTITY_MC 0x00000040                         e.g., left hand status line
define ADJECTIVE_MC 0x00000080                        e.g., invisible
define TABLE_MC 0x00000100                            e.g., table of rankings
define TABLE_COLUMN_MC 0x00000200                     e.g., rank
define DESIGNED_TYPE_MC 0x00000400                    e.g., weight
define RULE_MC 0x00000800                             e.g., update chronological records rule
define RULEBOOK_MC 0x00001000                         e.g., instead
define ACTIVITY_MC 0x00002000                         e.g., reading a command
define EQUATION_MC 0x00004000                         e.g., Newton's Second Law
an MC now dropped used to occupy this bit, which is now free
define NAMED_AP_MC 0x00010000                         e.g., bad behaviour
define END_PHRASE_MC 0x00020000                       e.g., end while
define SAY_PHRASE_MC 0x00040000                       e.g., say # in words
```

¶6. The second set of MCs signal the contexts in which particular words are found. They are used to mark `vocabulary_entry` structures of individual words, whose `flags` field is a bitmap combination of some or all of the following. For instance, the VE for "taking" is flagged with the `ING_MC` bit; "in" is flagged with both `INORON_MC` and `PREPOSITION_MC`.

```
define CVERB_MC 0x00080000                  a word which might introduce a verb usage
define ING_MC 0x00100000                    a word ending in -ing
define PREPOSITION_MC 0x00200000            a word which might introduce a relative clause
define NOT_MC 0x00400000                    not
define ORDINAL_MC 0x00800000                first, second, third, ..., twelfth
define ARTICLE_MC 0x01000000                a, an, the, some
define INORON_MC 0x02000000                 in, on
define BUILTINTYPE_MC 0x04000000            number, text, relation, rule, ...
define I6_MC 0x08000000                     piece of verbatim I6 code
define TEXTWITHSUBS_MC 0x10000000           double-quoted text literal with substitutions
define TEXT_MC 0x20000000                   double-quoted text literal without substitutions
define NUMBER_MC 0x40000000                 one, two, ..., twelve, 1, 2, ...
```

¶7. In fact, 31 meaning codes is not really enough, which is why we have the `MISCELLANEOUS_MC` value. This is a special case: it labels names which can relate to a variety of things, but which need not be parsed quickly.

A secondary meaning code, or SMC, then comes into play to explain what kind meaning we have. The SMC is irrelevant for any excerpt whose code is not `MISCELLANEOUS_MC`, and it is not a bitmap, but a simple enumeration. (Bitmaps of MCs are unable to distinguish the individual secondary meanings.)

In the examples below, the bracketed words are not stored in the token list, but are instead used when parsing to recognise that a secondary meaning may be needed. All of these cases have a "signature word" which alerts the S-parser that they may be being used: option, relation, Unicode, figure, outcome, action, sound, file. This is why we can afford to parse them a little more slowly than other excerpts – we are fairly unlikely to chase false leads.

```
define USE_OPTION_SMC 0x00000001            e.g., memory economy (option)
define RELATION_SMC 0x00000002              e.g., containment (relation)
define UNICODENAME_SMC 0x00000003           e.g., (Unicode) Greek letter alpha
define FIGURE_SMC 0x00000004                e.g., figure of daffodils
define RULE_OUTCOME_SMC 0x00000005          e.g., it is very unlikely (outcome)
define ACTION_NAME_SMC 0x00000006           e.g., going action
define SOUND_EFFECT_SMC 0x00000007          e.g., sound of creaking
define EXTERNAL_FILE_SMC 0x00000008         e.g., file of high scores
```

---

§**1. Creating EMs.** The following makes a skeletal EM structure, with no token list or hash code as yet.

```
excerpt_meaning *em_new(int mc, int smc, general_pointer data) {
    excerpt_meaning *em = CREATE(excerpt_meaning);
    em->meaning_code = mc;
    em->secondary_code = smc;
    em->data = data;
    em->no_em_tokens = 0;
    em->excerpt_hash = 0;
    return em;
}
```

The function em_new is called from 5/varc.

§**2.** Access routines:

```
int em_get_secondary_code(excerpt_meaning *em) {
    if (em->meaning_code != MISCELLANEOUS_MC)
        internal_error("looked up SMC for non-miscellaneous EM");
    return em->secondary_code;
}
general_pointer em_data(excerpt_meaning *em) {
    return em->data;
}
```

The function em_get_secondary_code is called from 5/unitr, 5/lit, 5/candd, 5/candp, 5/mlc, 10/fig, 10/sfx, 10/exf, 10/isin and 12/rb. The function em_data is called from 5/aph, 5/unitr, 5/lit, 5/candd, 5/varc, 5/mlc, 8/refpt, 9/qty, 9/wo, 9/prop, 10/tab, 10/isin, 11/ap, 11/nap, 11/av, 12/br and 12/rb.

§**3. Debugging log.** First to log a general bitmap made up from meaning codes:

```
void log_meaning_code(int mc) {
    int i, j, f = FALSE;
    for (i=0, j=1; i<31; i++, j*=2)
        if (mc & j) {
            if (f) LOG(" + "); f = TRUE;
            switch (j) {
                case ACTIVITY_MC: LOG("ACTIVITY_MC"); break;
                case ADJECTIVE_MC: LOG("ADJECTIVE_MC"); break;
                case ARTICLE_MC: LOG("ARTICLE_MC"); break;
                case BUILTINTYPE_MC: LOG("BUILTINTYPE_MC"); break;
                case COND_PHRASE_MC: LOG("COND_PHRASE_MC"); break;
                case CVERB_MC: LOG("CVERB_MC"); break;
                case DESIGNED_TYPE_MC: LOG("DESIGNED_TYPE_MC"); break;
                case END_PHRASE_MC: LOG("END_PHRASE_MC"); break;
                case EQUATION_MC: LOG("EQUATION_MC"); break;
                case I6_MC: LOG("I6_MC"); break;
                case ING_MC: LOG("ING_MC"); break;
                case INORON_MC: LOG("INORON_MC"); break;
                case MISCELLANEOUS_MC: LOG("MISCELLANEOUS_MC"); break;
                case NAMED_AP_MC: LOG("NAMED_AP_MC"); break;
                case NOT_MC: LOG("NOT_MC"); break;
                case NUMBER_MC: LOG("NUMBER_MC"); break;
                case ORDINAL_MC: LOG("ORDINAL_MC"); break;
                case PREPOSITION_MC: LOG("PREPOSITION_MC"); break;
                case PROPERTY_MC: LOG("PROPERTY_MC"); break;
                case QUANTITY_MC: LOG("QUANTITY_MC"); break;
                case RULE_MC: LOG("RULE_MC"); break;
                case RULEBOOK_MC: LOG("RULEBOOK_MC"); break;
                case SAY_PHRASE_MC: LOG("SAY_PHRASE_MC"); break;
                case TABLE_COLUMN_MC: LOG("TABLE_COLUMN_MC"); break;
                case TABLE_MC: LOG("TABLE_MC"); break;
                case TEXT_MC: LOG("TEXT_MC"); break;
                case TEXTWITHSUBS_MC: LOG("TEXTWITHSUBS_MC"); break;
                case VALUE_PHRASE_MC: LOG("VALUE_PHRASE_MC"); break;
                case VOID_PHRASE_MC: LOG("VOID_PHRASE_MC"); break;
                case WORLD_OBJECT_MC: LOG("WORLD_OBJECT_MC"); break;
                default: LOG("<unknown-mc>"); break;
```

```
            }
        }
}
```

The function log_meaning_code is called from 5/ml.

§4. And now for excerpt meanings:

```
void log_excerpt_meaning(excerpt_meaning *em) {
    int i;
    if (em == NULL) { LOG("<null-em>"); return; }
    LOG("{");
    for (i=0; i<em->no_em_tokens; i++) {
        if (i>0) LOG(" ");
        if (em->em_tokens[i] == NULL) { LOG("#"); continue; }
        if (vocab_get_exemplar(em->em_tokens[i], FALSE) == NULL)
            internal_error("Null token in EM");
        LOG("%s", vocab_get_exemplar(em->em_tokens[i], FALSE));
    }
    LOG(" = $p", em->meaning_code);
    if (em->secondary_code) LOG(" (secondary=%d)", em->secondary_code);
    LOG("}");
}
void log_all_meanings(void) {
    int i = 0;
    excerpt_meaning *em;
    LOOP_OVER(em, excerpt_meaning)
        LOG("%02d: %08x $M\n", i++, (pointer_sized_int) em, em);
}
```

The function log_excerpt_meaning is called from 2/dl.
The function log_all_meanings is called from 5/parse.

§5. **Hashing excerpts.** For excerpts `(w1, w2)`, we need a form of hash function which makes it easy to test whether the words in one excerpt can all be found in another, or to be more exact whether

$$\{I_j \mid w_1 \leq j \leq w_2\} \subseteq \{I_j \mid w_3 \leq j \leq w_4\}$$

where I_n is the identity of word n. As with all hash algorithms, we do not need to guarantee a positive match, only a negative, so we can throw away a lot of information. And we also want a hash function which makes it easy to test whether an excerpt contains any of the literals.

§**6.** There are two sources of text which we might want to hash in this way: first, actual excerpts found in the source text. These are not very expensive to calculate, but every ounce of speed helps here, so we cache the most recent.

The hash generated this way is an arbitrary bitmap of bits 1 to 30, with bits 31 and 32 left clear.

```
int cached_hash_w1 = -2, cached_hash_w2 = -2, cached_value;

int hash_code_from_excerpt(int w1, int w2) {
    int i, h = 0; vocabulary_entry *v;
    if ((w1 == cached_hash_w1) && (w2 == cached_hash_w2)) return cached_value;
    for (i=w1; i<=w2; i++) {
        v = lw_array[i].lw_identity;
        if (v) ⟨Allow this vocabulary entry to contribute to the excerpt's hash code 8⟩;
    }
    return h;
}
```

The function hash_code_from_excerpt is called from 5/parse and 12/phsf.

§**7.** Second, when a new excerpt meaning is to be registered, we want to hash code its token list. But only some of the tokens are vocabulary entries, while others instead represent gaps where arbitrary text can appear (referred to with a null pointer). Note that we simply ignore that gaps when hashing, that is, we produce the same hash as we would if the gaps were not there at all.

The hash generated this way is an arbitrary bitmap of bits 1 to 31, with bit 32 left clear. Bit 31 is set, as a special case, for excerpts in the context of text substitutions which begin with a word known to exist, and with differing meanings, in two differently cased forms: this is how "[the noun]" is distinguished from "[The noun]". (The lower 30 bits have the same meaning as in the first case above.)

```
define CAPITALISED_VARIANT_FORM (1 << 30)

void hash_code_from_token_list(excerpt_meaning *em) {
    int i, h = 0;
    if (em->no_em_tokens == 0) internal_error("Empty text when registering");
    if ((em->no_em_tokens >= 1) && (em->em_tokens[0])) {
        vocabulary_entry *lcf = vocab_get_lower_case_form(em->em_tokens[0]);
        if (lcf) {
            h = h | CAPITALISED_VARIANT_FORM;
            em->em_tokens[0] = lcf;
        }
    }
    for (i=0; i<em->no_em_tokens; i++) {
        vocabulary_entry *v = em->em_tokens[i];
        if (v) ⟨Allow this vocabulary entry to contribute to the excerpt's hash code 8⟩;
    }
    em->excerpt_hash = h;
}
```

§**8.** Now each vocabulary entry `v`, i.e., each distinct word identity, itself has a hash code to identify it. These are stored in `v->hash` and, except for literals, are more or less evenly distributed in about the range 0 to 1000.

The contribution made by a single word's individual hash to the bitmap hash for the whole excerpt is as follows.

⟨Allow this vocabulary entry to contribute to the excerpt's hash code 8⟩ ≡

```
if ((v->flags) & NUMBER_MC)    h = h | 1;
else if ((v->flags) & TEXT_MC) h = h | 2;
else if ((v->flags) & I6_MC)   h = h | 4;
else                           h = h | (8 << ((v->hash) % 27));
```

This code is used in §6,7,6,7,6,7.

§**9.** To sum up: the excerpt hash is a bitmap indicating what categories of words are present in the excerpt. It ignores "gaps" in token lists, and it ignores the order of the words and repetitions. The three least significant bits indicate whether numbers, text or I6 verbatims are present, and the next 27 bits indicate the presence of other words: e.g., bit 4 indicates that a word with hash code 0, 27, 54, ..., is present, and so on. Bit 31, which is used only for token lists of excerpt meanings, marks that an excerpt is a variant form whose first word must be capitalised in order for it to match. Bit 32 is always left blank (for superstitious reasons to do with the sign bit and differences between platforms in handling signed bit shifts).

The result is not a tremendously good hashing number, since it generally produces a sparse bitmap, so that the variety is not as great as might be thought. But it is optimised for the trickiest parsing cases where the rewards of saving unnecessary tests are greatest.

§**10. EM Listing.** We are clearly not going to store the excerpt meanings in a hash table keyed by the hash values of excerpts – with hash values as large as $2^{31} - 1$, that would be practically impossible.

Instead we key using the actual words. Each vocabulary entry has four linked lists of EMs: its subset list, its start list, its middle list, and its end list.

(a) If an EM needs to allow parsing as a subset, it must be placed in the subset list of every word. For instance, "buttress against cathedral wall" registered under the code `WORLD_OBJECT_MC` would be listed in the subset lists of "buttress", "against", "cathedral" and "wall".

(b) Otherwise it is placed in only one list:

(b1) If the token list consists only of a single gap `#`, we must be registering a "say" phrase to say a value. (There is one of these for each kind of value.) This meaning is listed under a special `blank_says_ml` list, which is not attached to any vocabulary entry.

(b2) Otherwise, if the first token is not a `#` gap, it goes into the start list for the first token's word: for instance, `award # points` joins the start list for "award".

(b3) Otherwise, if the last token is not a `#` gap, it goes into the end list for the last token's word: for instance, `# in # from now` joins the end list for "now".

(b4) Otherwise, it goes into the middle list of the word for the leftmost token which is not a `#`: for instance, `# plus #` joins the middle list for "plus".

Since no token lists of two or more consecutive `#`s cannot exist, this exhausts the possibilities.

Outside of subset mode, we will then test a given excerpt `(w1, w2)` in the source text against all possible meanings by checking the start list for `w1`, the end list for `w2` and the middle list for every one of `(w1+1, w2-1)`. Because of this:

(i) Performance suffers if lists for individual words become unbalanced in size. This is why we register Unicode translations as "white chess knight" rather than "Unicode white chess knight", and so on; the alternative would be a stupendously long start list for "unicode".

(ii) Middle lists are tested far more often than start or end lists, so we should keep them as small as possible. This is why (b4) above is our last resort; happily phrases both starting and ending with `#` are uncommon.

```
meaning_list *blank_says_ml = NULL;
void register_em(int meaning_code, excerpt_meaning *em) {
    warn_expression_cache();              the existence of new meanings jeopardises any cached parsing results
    ⟨Compute the new excerpt's hash code from its token list 11⟩;
    ⟨Watermark each word in the token list with the meaning code being applied 12⟩;
    LOGIF(EXCERPT_MEANINGS,
        "Logging meaning: $M with hash %08x, mc=%d, %d tokens\n",
        em, em->excerpt_hash, meaning_code, em->no_em_tokens);
    if (meaning_code & SUBSET_PARSING_BITMAP) {
        ⟨Place the new meaning under the subset list for each non-article word 13⟩;
    } else if ((em->no_em_tokens == 1) && (em->em_tokens[0] == NULL) &&
        (meaning_code == SAY_PHRASE_MC)) {
        ⟨Place the new meaning under the say-blank list 14⟩;
    } else if (em->em_tokens[0]) {
        ⟨Place the new meaning under the start list of the first word 15⟩;
    } else if (em->em_tokens[em->no_em_tokens-1]) {
        ⟨Place the new meaning under the end list of the last word 16⟩;
    } else {
        int i;
        for (i=1; i<em->no_em_tokens-1; i++)
            if (em->em_tokens[i]) { ⟨Place the new meaning under the middle list of word i 17⟩; break; }
        if (i >= em->no_em_tokens-1) internal_error("registered meaning of two or more #s");
    }
}
```

§**11.** See above.

⟨Compute the new excerpt's hash code from its token list 11⟩ ≡

```
    hash_code_from_token_list(em);
```

This code is used in §10.

§**12.** Another important optimisation is to flag each word in the meaning with the given meaning code – this is why vocabulary flags and excerpt meaning codes share the same numbering space. If we register "Table of Surgical Instruments" as a table name, the word "surgical", for instance, picks up the `TABLE_MC` bit in its `flags` bitmap.

The advantage of this is that if we want to see whether `(w1, w2)` might be a table name, we can take a bitwise AND of the flags for each word in the range; if the result doesn't have the `TABLE_MC` bit set, then at least one of the words never occurs in a table name, so the answer must be "no". This produces rapid, definite negatives with only a few false positives.

⟨Watermark each word in the token list with the meaning code being applied 12⟩ ≡

```
    int i;
    for (i=0; i<em->no_em_tokens; i++)
        if (em->em_tokens[i])
            ((em->em_tokens[i])->flags) |= meaning_code;
```

This code is used in §10.

§**13.** Note that articles (a, an, the, some) are excluded: this means we don't waste time trying to see if the excerpt "the" might be a reference to the object "Gregory the Great".

⟨Place the new meaning under the subset list for each non-article word 13⟩ ≡

```
int i;
for (i=0; i<em->no_em_tokens; i++) {
    vocabulary_entry *v = em->em_tokens[i];
    if (v == NULL) {
        LOG("Logging meaning: $M with hash %08x\n", em, em->excerpt_hash);
        internal_error("# in registration of subset meaning");
    }
    if ((v->flags) & ARTICLE_MC) continue;
    meaning_list *ml = new_permanent_ml(em->meaning_code);
    ml->em = em;
    ml->next_alternative = v->subset_list;
    v->subset_list = ml;
    v->subset_list_length++;
}
```

This code is used in §10.

§**14.** To register `#`, which is what "To say (N - a number)" and similar constructions translate to.

⟨Place the new meaning under the say-blank list 14⟩ ≡

```
meaning_list *ml = new_permanent_ml(em->meaning_code); ml->em = em;
if (blank_says_ml) {
    meaning_list *ml2 = blank_says_ml;
    while (ml2->next_alternative) ml2 = ml2->next_alternative;
    ml2->next_alternative = ml;
}
else blank_says_ml = ml;
LOGIF(EXCERPT_MEANINGS,
    "The blank list with $M is now:\n$m", em, blank_says_ml);
```

This code is used in §10.

§**15.**

⟨Place the new meaning under the start list of the first word 15⟩ ≡

```
meaning_list *ml = new_permanent_ml(em->meaning_code); ml->em = em;
ml->next_alternative = em->em_tokens[0]->start_list;
em->em_tokens[0]->start_list = ml;
```

This code is used in §10.

§**16.** ...and similarly...

⟨Place the new meaning under the end list of the last word 16⟩ ≡

```
meaning_list *ml = new_permanent_ml(em->meaning_code); ml->em = em;
ml->next_alternative = em->em_tokens[em->no_em_tokens-1]->end_list;
em->em_tokens[em->no_em_tokens-1]->end_list = ml;
```

This code is used in §10.

§**17.** ...and similarly again:

⟨Place the new meaning under the middle list of word i 17⟩ ≡

```
meaning_list *ml = new_permanent_ml(em->meaning_code); ml->em = em;
ml->next_alternative = em->em_tokens[i]->middle_list;
em->em_tokens[i]->middle_list = ml;
```

This code is used in §10.

§**18. Deregistration.** This is possible only for subset-parsed EMs, that is, world objects. We do it by striking the reference to the given data out of the relevant subset lists; a ticklish business but not needed often. (In fact, never, as of October 2008.)

```
void deregister_excerpt_meaning(int meaning_code, int w1, int w2,
    general_pointer data) {
    int k;
    meaning_list *ml, *back;
    LOGIF(EXCERPT_MEANINGS, "Unlogging meaning: <$W> data %08x\n", w1, w2,
        GENERAL_POINTER_AS_INT(data));
    for (k=w1; k<=w2; k++) {
        for (ml = lw_array[k].lw_identity->subset_list, back = NULL;
            ml; back = ml, ml = ml->next_alternative) {
            if ((COMPARE_GENERAL_POINTERS(ml_meaning(ml)->data, data))
                && (ml_meaning(ml)->meaning_code == meaning_code)) {
                if (back) back->next_alternative = ml->next_alternative;
                else lw_array[k].lw_identity->subset_list = ml->next_alternative;
                if (--(lw_array[k].lw_identity->subset_list_length) == 0)
                    lw_array[k].lw_identity->flags =
                        lw_array[k].lw_identity->flags - meaning_code;
                break;
            }
        }
    }
}
```

§**19. Registration.** The following is the main routine used throughout Inform to register new meanings.

```
excerpt_meaning *register_excerpt_meaning(int meaning_code, int secondary,
    int w1, int w2, general_pointer data) {
    excerpt_meaning *em = em_new(meaning_code, secondary, data);
    ⟨Unless this is a new phrase or text substitution, skip any initial article 20⟩;
    if (meaning_code == SAY_PHRASE_MC)
        ⟨Detect use of upper case on the first word of this new text substitution 21⟩;
    ⟨Build the token list for the new EM 22⟩;
    register_em(meaning_code, em);
    if ([[w1, w2 == player]] && (meaning_code & QUANTITY_MC)) meaning_of_player = em;
    return em;
}
```

The function register_excerpt_meaning is called from 5/aph, 5/unitr, 7/tids, 9/qty, 9/scene, 9/wo, 9/prop, 10/tab, 10/eqns, 10/fig, 10/sfx, 10/exf, 10/isin, 11/nap, 11/av, 12/phtd, 12/br and 12/rb.

§20. Articles are preserved at the front of phrase definitions, mainly because text substitutions need to distinguish (for instance) "say [the X]" from "say [an X]".

⟨Unless this is a new phrase or text substitution, skip any initial article 20⟩ ≡

```
if ((meaning_code & PARAMETRISED_PARSING_BITMAP) == 0) {
    if ((lw_array[w1].lw_identity->flags) & ARTICLE_MC) w1++;
    if (w1 > w2) internal_error("registered a meaning which was only an article");
}
```

This code is used in §19.

§21. Because an open bracket fails `isupper`, the following looks at the first letter of the first word only if it's not a blank. If it finds upper case, as it would when reading the "T" in:

To say The Portrait: ...

then it makes a new upper-case version of the word "the", i.e., "The", with a distinct lexical identity; and places this distinguished identity as the new first token. This ensures that we end up with a different token list from the one in:

To say the Portrait: ...

(These are the only circumstances in which phrase parsing has any case sensitivity.)

⟨Detect use of upper case on the first word of this new text substitution 21⟩ ≡

```
char *tx = lw_array[w1].lw_rawtext;
if ((tx[0]) && (isupper(tx[0]))) {
    vocabulary_entry *ucf = vocab_make_case_sensitive(lw_array[w1].lw_identity);
    lw_array[w1].lw_identity = ucf;
    LOGIF(EXCERPT_MEANINGS,
        "Allowing initial capitalised word %s: meaning_code = %08x\n",
            tx, meaning_code);
}
```

This code is used in §19.

§22. We read the text in something like:

award (P - a number) points

and transcribe it into the token list, collapsing bracketed parts into `#` tokens denoting gaps, to result in something like:

```
award # points
```

with a token count of 3.

⟨Build the token list for the new EM 22⟩ ≡

```
int i, tc;
for (i=0, tc=0; w1+i<=w2; i++) {
    if (tc >= MAX_TOKENS_PER_EXCERPT_MEANING) {
        ⟨Complain of excessive length of the new excerpt 25⟩;
        break;
    }
    if (compare_word(w1+i, OPENBRACKET_V)) {
        em->em_tokens[tc++] = NULL;
        ⟨Skip over bracketed token description 23⟩;
    } else em->em_tokens[tc++] = lw_array[w1+i].lw_identity;
}
em->no_em_tokens = tc;
```

This code is used in §19.

§**23.** This is all a little defensive, but syntax bugs higher up tend to find their way down to this plughole:

⟨Skip over bracketed token description 23⟩ ≡

```
while (compare_word(w1+i, CLOSEBRACKET_V) == FALSE) {
    i++;
    if (w1+i>w2)  {
        LOG("Bad meaning: <$W>\n", w1, w2);
        internal_error("Bracket mismatch when registering");
    }
}
if ((w1+i <= w2) && (compare_word(w1+i, OPENBRACKET_V))) {
    LOG("Bad meaning: <$W>\n", w1, w2);
    internal_error("Two consecutive bracketed tokens when registering");
}
```

This code is used in §22.

§**24. Reworded meanings.** In a few cases it is convenient to register a slightly altered wording. For instance, every table column – "quota", say – is registered not only under its own text ("quota") but under the same text with "column" added ("quota column"). The following routine allows up to two arbitrary words to be placed before `(w1, w2)`, and up to one after them.

But in other respects this is a simpler routine, because it's never needed for token lists with gaps in.

```
excerpt_meaning *register_reworded_meaning(int meaning_code, int secondary,
    vocabulary_entry *vb1, vocabulary_entry *vb2,
    int w1, int w2, vocabulary_entry *va, general_pointer data) {
    int i, tc = 0;
    excerpt_meaning *em = em_new(meaning_code, secondary, data);
    if ((w1 >= 0) &&
        ((meaning_code & PARAMETRISED_PARSING_BITMAP) == 0) &&
        ((lw_array[w1].lw_identity->flags) & ARTICLE_MC) && (vb1 == NULL)) w1++;
    if (vb1) em->em_tokens[tc++] = vb1;
    if (vb2) em->em_tokens[tc++] = vb2;
    if (w1 >= 0) {
        if (4 + w2 - w1 > MAX_TOKENS_PER_EXCERPT_MEANING) {
            ⟨Complain of excessive length of the new excerpt 25⟩;
            w2 = w1 + MAX_TOKENS_PER_EXCERPT_MEANING - 4;
        }
        for (i=0; w1+i<=w2; i++)
            em->em_tokens[tc++] = lw_array[w1+i].lw_identity;
    }
    if (va) em->em_tokens[tc++] = va;
    em->no_em_tokens = tc;
    register_em(meaning_code, em);
    return em;
}
```

The function register_reworded_meaning is called from 5/bp, 6/equal, 9/prop, 10/tab, 10/eqns, 11/act, 11/av, 12/phtd and 12/rb.

§**25.** In practice, nobody ever hits this message except deliberately. It has a tendency to fire twice or more on the same source text because of registering multiple inflected forms of the same text; but it's not worth going to any trouble to prevent this.

⟨Complain of excessive length of the new excerpt 25⟩ ≡

```
LOG("Tried to register <$W>\n", w1, w2);
sentence_problem(_P_(C5TooLongName),
    "that seems to involve far too long a name",
    "since in general names are limited to a maximum of 32 words.");
```

This code is used in §22,24,22,24,22,24.

Unicode Translations 5/unitr

Purpose

To manage the names assigned to Unicode character values.

---

§**1.** There are no data structures here; Unicode names are simply a category of excerpt meanings, so we read a "translates into Unicode as" sentence as a new name and its meaning to be.

```
sentence_handler TRANSLATESU_SH_handler =
    { SENTENCE_NT, TRANSLATESU_VB, 2, unicode_translates };

void unicode_translates(parse_node *pn) {
    int nn1, nn2, as1, as2, cc, occ;

    [[as1, as2 <-- pn->down->next->next]];
    [[as1, as2 == as ... --> as1, as2]];
    ⟨Check that we have a valid literal character code 2⟩;

    [[nn1, nn2 <-- pn->down->next]];
    occ = is_a_unicode_value(nn1, nn2);
    if (occ == cc) return;

    if (occ >= 0) {
        sentence_problem(_P_(C5UnicodeAlready),
            "this Unicode character name has already been translated",
            "so there must be some duplication somewhere.");
        return;
    }

    register_excerpt_meaning(
        MISCELLANEOUS_MC, UNICODENAME_SMC, nn1, nn2,
        STORE_POINTER_parse_node(new_nounphrase_raw(as1, as2)));
}
```

§**2.**

⟨Check that we have a valid literal character code 2⟩ ≡

```
    if ((as1 != as2) || (vocab_test_flags(as1, NUMBER_MC) == 0)) {
        sentence_problem(_P_(C5UnicodeNonLiteral),
            "a Unicode character name must be translated into a literal "
            "decimal number written out in digits",
            "which this seems not to be.");
        return;
    }
    cc = vocab_get_literal_number_value(lw_array[as1].lw_identity);
    if (Unicode_char_in_range(cc) == FALSE) return;
```

This code is used in §1.

§**3.** The following is called only on excerpts from the source where it is a fairly safe bet that a Unicode character is referred to. It returns either the character code, or −1 if the excerpt doesn't refer to such a character after all.

```
int is_a_unicode_value(int w1, int w2) {
    ⟨Match a literal Unicode character number in decimal 4⟩;
    ⟨Match a named Unicode character 5⟩;
    return -1;
}
```

The function is_a_unicode_value is called from 5/lit.

§**4.** The effect of making the following range check is that we can guarantee that any value of the kind "Unicode-character" in I7 is a valid Unicode:

⟨Match a literal Unicode character number in decimal 4⟩ ≡

```
    if ((w1 == w2) && (vocab_test_flags(w1, NUMBER_MC))) {
        int cc = vocab_get_literal_number_value(lw_array[w1].lw_identity);
        if (Unicode_char_in_range(cc)) return cc;
    }
```

This code is used in §3.

§**5.** We need not make the same check here because Inform only registers names for characters whose codes are known to be within range.

⟨Match a named Unicode character 5⟩ ≡

```
    meaning_list *ml = SP_excerpt(MISCELLANEOUS_MC, w1, w2);
    if ((ml) && (em_get_secondary_code(ml_meaning(ml)) == UNICODENAME_SMC)) {
        parse_node *p = RETRIEVE_POINTER_parse_node(em_data(ml_meaning(ml)));
        return vocab_get_literal_number_value(lw_array[p->word_ref1].lw_identity);
    }
```

This code is used in §3.

§**6.** And here is the range check used by both of the above routines:

```
int Unicode_char_in_range(int cc) {
    if ((cc < 0) || (cc >= 0x10000)) {
        sentence_problem(_P_(C5UnicodeOutOfRange),
            "Inform can only handle Unicode characters in the 16-bit range",
            "from 0 to 65535.");
        return FALSE;
    }
    return TRUE;
}
```

Meaning Lists 5/ml

Purpose

To build meaning lists, which despite the name are really tree structures, showing possible interpretations of a sequence of words.

Definitions

¶1. Meaning lists are an intermediate construction within the S-parser, used to hold the possible meanings of complex excerpts of text. The S-parser completes its work by turning any meaning list for a successfully parsed piece of text into a much more compact `specification` structure, perhaps with a proposition in predicate calculus attached. This is both smaller and much less ambiguous in meaning. It would remove a layer of code in Inform, and also one delicate interface between layers, if the S-parser could parse directly to specifications; and this is what the earliest builds did, in 2003 and 2004, but the result was that `specification` became a very complex structure, trying to perform two different tasks at once – being a sort of checklist of possibilities and then being a definite answer. Separating these two roles and inventing meaning lists was a very disruptive decision, but eventually resulted in cleaner and simpler code.

¶2. We have already seen meaning lists used to store lists of possible excerpt meanings attached to given words, but that's not their main function, and despite the name they are not necessarily simple lists. They are a sort of two-dimensional tree structure where every node P represents one possible meaning of a given excerpt of the original text. Because in general the meaning will be complicated, and not as simple as a single noun, P will also have children which are nodes representing meanings of subexcerpts.

A two-dimensional tree is hard to visualise, but in practice they are easy enough to read: they are basically standard parse-trees except that at certain points they fork off into different possibilities. For instance, if we try to parse the example phrase at the beginning of this chapter:

> if Mr Fitzwilliam Darcy was carrying at least three things which are in the box, increase the score by 7;

then the S-parser initially generates the following meaning list:

```
CMD_PRODUCTION / "if mr fitzwilliam darcy was carrying at least ..."
    PHRASE_PRODUCTION
        [1/2] (score 1) {# at # = VOID_PHRASE_MC} / "if mr fitzwilliam ..."
            UNPARSED_PRODUCTION / "if mr fitzwilliam darcy was carrying"
            UNPARSED_PRODUCTION / "least three things which are in the box..."
        [2/2] (score 1) {if # , # = VOID_PHRASE_MC} / "if mr fitzwilliam darcy..."
            UNPARSED_PRODUCTION / "mr fitzwilliam darcy was carrying at..."
            UNPARSED_PRODUCTION / "increase the score by 7"
```

The notation `[1/2]` means "possibility 1 of 2". This structure shows that the S-parser is certain that we have a command phrase, but that on textual grounds alone it could be one of two possibilities. In fact, `[2/2]` is the valid one, as will become clear when it returns to parse the arguments currently left as `UNPARSED_PRODUCTION` nodes.

¶3. When the S-parser gets to the condition (argument 1 of possibility `[2/2]` above), a more elaborate meaning list results, but which is unambiguous:

```
COND_PRODUCTION / "mr fitzwilliam darcy was carrying at least ... in the box"
    SV_PRODUCTION / "mr fitzwilliam darcy was carrying at least ... in the box"
        NP_PRODUCTION / "mr fitzwilliam darcy"
            VAL_PRODUCTION / "mr fitzwilliam darcy"
                DC_PRODUCTION / "mr fitzwilliam darcy"
                    DC_NOUN_PRODUCTION / "mr fitzwilliam darcy"
                        {mr fitzwilliam darcy = WORLD_OBJECT_MC}
        VP_PRODUCTION
            VERB_PRODUCTION => VU: was WAS_TENSE -> is
                PREP_PRODUCTION => PU: carrying -> carries
            NP_PRODUCTION / "at least three things which are in the box"
                VAL_PRODUCTION / "at least three things which are in the box"
                    DC_PRODUCTION / "at least three things which are in the box"
                        SN_PRODUCTION / "at least three things which are in the box"
                            DC_PRODUCTION / "at least three things"
                                DC_NOUN_PRODUCTION / "things"
                                    {things = WORLD_OBJECT_MC}
                                DETERMINER_PRODUCTION => Card>=3 / "at least three"
                            VP_PRODUCTION
                                VERB_PRODUCTION => VU: are IS_TENSE -> is
                                    PREP_PRODUCTION => PU: in -> is-in
                                DC_PRODUCTION / "box"
                                    DC_NOUN_PRODUCTION / "box"
                                        {mr bingham's box = WORLD_OBJECT_MC}
                        DETERMINER_PRODUCTION => Card>=3 / "at least three"
```

Note the three concrete noun phrases – Mr Darcy, the box, and "things". It's perhaps surprising that the determiner for "at least three" turns up twice in the tree, but this is because the sub-excerpt

> at least three things

is also validly subject to the determiner, so its subtree contains the appropriate node.

As this shows, the result of parsing can be an extravagantly big meaning list. When the S-parser finishes, it is translated into much more compact data: a single specification representing a condition,

```
(A)'mr fitzwilliam ... box'/CONDITION_FMY/TEST_PROPOSITION_SPC<0 times: WAS_TENSE>
```

and with the proposition

```
[ Card>=3 x: K2'thing'(x) ^ is(O104'mr bingham's box',ContainerOf(x)) =>
    is(O105'mr fitzwilliam darcy',CarrierOf(x)) ]
```

which can be paraphrased "at least three x which are things and such that their container is the box are also such that Mr Darcy is their carrier".

¶4. So, then, each meaning list node has children and siblings to place it into a parse tree, but also forking links to alternative meanings. The actual data at a node can be a value (the `type_spec`), an excerpt meaning, or in some cases a pointer to some other structure. For instance, a `VERB_PRODUCTION` node has a pointer to the relevant `verb_usage` attached.

```
typedef struct meaning_list {
    int expiry_time;                                  an integer measured in creation "cycles": see below
    int production;                                                                    a production code
    int word_pair;                                    word pair (w1,w2) packed into a single integer
    struct excerpt_meaning *em;                                                        what this seems to mean...
    struct specification *type_spec;                                                   evaluation used in compaction
    struct general_pointer data_attached;                                              certain productions have data attached
    struct meaning_list *next_alternative;                                             fork to alternative meaning
    int score;                                        a scoring system is used to choose most likely alternative
    struct meaning_list *sibling;                                                      tree of meanings of subordinate clauses
    struct meaning_list *child;
    MEMORY_MANAGEMENT
} meaning_list;

int no_permanent_MLs = 0, no_ephemeral_MLs = 0, GAP_movements = 0;
```

The structure meaning_list is shared with 5/em, 5/arch and 5/parse.

¶5. It will be noted that the meaning codes for excerpt meanings use the bottom 31 bits of a (presumably) 32-bit word to hold sets of contexts, but never have bit 32 set. Meaning codes with bit 32 set are *not* considered as sets but simply as magic values in themselves. When we parse a complicated piece of text, the result is a tree in which the leaves are excerpts with simple meanings, where the `production` field is a meaning code; but the higher nodes have `production` values from the following set.

(There is no significance to these numbers except that they must all be different, and must all have bit 32 set.)

```
define ABSENT_SUBJECT_PRODUCTION 0x80000010
define ACTION_PRODUCTION 0x80000020
define ADVERB_PRODUCTION 0x80000030
define AL_PRODUCTION 0x80000040
define AP_PRODUCTION 0x80000050
define CALLED_PRODUCTION            0x80000058
define CARRIED_PRODUCTION 0x80000060
define CASE_PRODUCTION 0x80000070
define OTHERWISE_PRODUCTION 0x80000080
define CMD_PRODUCTION 0x80000090
define COND_AND_PRODUCTION 0x800000a0
define COND_NOT_PRODUCTION 0x800000b0
define COND_OR_PRODUCTION 0x800000c0
define COND_PAST_PRODUCTION 0x800000d0
define COND_PHRASE_PRODUCTION 0x800000e0
define COND_PRODUCTION 0x800000f0
define DC_ADJS_PRODUCTION 0x80000100
define DC_ADJSNOUN_PRODUCTION 0x80000110
define DC_NOUN_PRODUCTION 0x80000130
define DC_PRODUCTION 0x80000140
define DETERMINER_PRODUCTION 0x80000180
define INSTEAD_PRODUCTION 0x80000190
define LITERAL_PRODUCTION 0x800001a0
```

```
define LOCAL_PRODUCTION 0x800001b0
define MEMBER_OF_PRODUCTION 0x800001c0
define ADJ_NOT_PRODUCTION 0x800001e0
define NP_PRODUCTION 0x800001f0
define OPTION_PRODUCTION 0x80000200
define PHR_OPT_PRODUCTION 0x80000210
define PHRASE_PRODUCTION 0x80000220
define PLAYER_PRODUCTION 0x80000230
define PREP_PRODUCTION 0x80000240
define SAY_PRODUCTION 0x80000250
define SN_PRODUCTION 0x80000270
define STV_PRODUCTION 0x80000280
define SV_PRODUCTION 0x80000290
define TE_CALLED_PRODUCTION 0x800002a0
define TE_EX_VAR_PRODUCTION 0x800002b0
define TE_GL_VAR_PRODUCTION 0x800002c0
define TE_NEW_VAR_PRODUCTION 0x800002e0
define TE_PRODUCTION 0x800002f0
define TE_VAR_PRODUCTION 0x80000300
define THERE_PRODUCTION             0x80000310
define TIME_PRODUCTION 0x80000320
define TR_CORR_PRODUCTION 0x80000330
define TR_ENTRY_PRODUCTION 0x80000340
define TR_IN_ROW_PRODUCTION 0x80000350
define TR_LISTED_IN_PRODUCTION 0x80000360
define TR_OF_IN_PRODUCTION 0x80000370
define TR_PRODUCTION 0x80000380
define TYPE_PRODUCTION 0x80000390
define UNPARSED_PRODUCTION 0x800003a0
define VAL_LIST_ENTRY_PRODUCTION 0x800003b0
define VAL_NOTHING_PRODUCTION 0x800003c0
define VAL_PRODUCTION 0x800003d0
define VAL_PROP_OF_PRODUCTION 0x800003e0
define VALUE_PHRASE_PRODUCTION 0x800003f0
define VERB_PRODUCTION 0x80000400
define VP_PRODUCTION 0x80000410
define EQUATION_INLINE_PRODUCTION 0x80000420
define EQUATION_WHERE_PRODUCTION 0x80000430
```

---

§1. Logging production values.

```
void log_production(int production) {
    if (production == 0) { LOG("<null-production>"); return; }
    if (production & 0x80000000) {
        switch (production) {
            case ABSENT_SUBJECT_PRODUCTION: LOG("ABSENT_SUBJECT_PRODUCTION"); break;
            case ACTION_PRODUCTION: LOG("ACTION_PRODUCTION"); break;
            case ADVERB_PRODUCTION: LOG("ADVERB_PRODUCTION"); break;
            case AL_PRODUCTION: LOG("AL_PRODUCTION"); break;
            case AP_PRODUCTION: LOG("AP_PRODUCTION"); break;
            case CALLED_PRODUCTION: LOG("CALLED_PRODUCTION"); break;
```

```
case CARRIED_PRODUCTION: LOG("CARRIED_PRODUCTION"); break;
case CASE_PRODUCTION: LOG("CASE_PRODUCTION"); break;
case OTHERWISE_PRODUCTION: LOG("OTHERWISE_PRODUCTION"); break;
case CMD_PRODUCTION: LOG("CMD_PRODUCTION"); break;
case COND_AND_PRODUCTION: LOG("COND_AND_PRODUCTION"); break;
case COND_NOT_PRODUCTION: LOG("COND_NOT_PRODUCTION"); break;
case COND_OR_PRODUCTION: LOG("COND_OR_PRODUCTION"); break;
case COND_PAST_PRODUCTION: LOG("COND_PAST_PRODUCTION"); break;
case COND_PHRASE_PRODUCTION: LOG("COND_PHRASE_PRODUCTION"); break;
case COND_PRODUCTION: LOG("COND_PRODUCTION"); break;
case DC_ADJS_PRODUCTION: LOG("DC_ADJS_PRODUCTION"); break;
case DC_ADJSNOUN_PRODUCTION: LOG("DC_ADJSNOUN_PRODUCTION"); break;
case DC_NOUN_PRODUCTION: LOG("DC_NOUN_PRODUCTION"); break;
case DC_PRODUCTION: LOG("DC_PRODUCTION"); break;
case DETERMINER_PRODUCTION: LOG("DETERMINER_PRODUCTION"); break;
case EQUATION_INLINE_PRODUCTION: LOG("EQUATION_INLINE_PRODUCTION"); break;
case EQUATION_WHERE_PRODUCTION: LOG("EQUATION_WHERE_PRODUCTION"); break;
case INSTEAD_PRODUCTION: LOG("INSTEAD_PRODUCTION"); break;
case LITERAL_PRODUCTION: LOG("LITERAL_PRODUCTION"); break;
case LOCAL_PRODUCTION: LOG("LOCAL_PRODUCTION"); break;
case MEMBER_OF_PRODUCTION: LOG("MEMBER_OF_PRODUCTION"); break;
case ADJ_NOT_PRODUCTION: LOG("ADJ_NOT_PRODUCTION"); break;
case NP_PRODUCTION: LOG("NP_PRODUCTION"); break;
case OPTION_PRODUCTION: LOG("OPTION_PRODUCTION"); break;
case PHR_OPT_PRODUCTION: LOG("PHR_OPT_PRODUCTION"); break;
case PHRASE_PRODUCTION: LOG("PHRASE_PRODUCTION"); break;
case PLAYER_PRODUCTION: LOG("PLAYER_PRODUCTION"); break;
case PREP_PRODUCTION: LOG("PREP_PRODUCTION"); break;
case SAY_PRODUCTION: LOG("SAY_PRODUCTION"); break;
case SN_PRODUCTION: LOG("SN_PRODUCTION"); break;
case STV_PRODUCTION: LOG("STV_PRODUCTION"); break;
case SV_PRODUCTION: LOG("SV_PRODUCTION"); break;
case TE_CALLED_PRODUCTION: LOG("TE_CALLED_PRODUCTION"); break;
case TE_EX_VAR_PRODUCTION: LOG("TE_EX_VAR_PRODUCTION"); break;
case TE_GL_VAR_PRODUCTION: LOG("TE_GL_VAR_PRODUCTION"); break;
case TE_NEW_VAR_PRODUCTION: LOG("TE_NEW_VAR_PRODUCTION"); break;
case TE_PRODUCTION: LOG("TE_PRODUCTION"); break;
case TE_VAR_PRODUCTION: LOG("TE_VAR_PRODUCTION"); break;
case THERE_PRODUCTION: LOG("THERE_PRODUCTION"); break;
case TIME_PRODUCTION: LOG("TIME_PRODUCTION"); break;
case TR_CORR_PRODUCTION: LOG("TR_CORR_PRODUCTION"); break;
case TR_ENTRY_PRODUCTION: LOG("TR_ENTRY_PRODUCTION"); break;
case TR_IN_ROW_PRODUCTION: LOG("TR_IN_ROW_PRODUCTION"); break;
case TR_LISTED_IN_PRODUCTION: LOG("TR_LISTED_IN_PRODUCTION"); break;
case TR_OF_IN_PRODUCTION: LOG("TR_OF_IN_PRODUCTION"); break;
case TR_PRODUCTION: LOG("TR_PRODUCTION"); break;
case TYPE_PRODUCTION: LOG("TYPE_PRODUCTION"); break;
case UNPARSED_PRODUCTION: LOG("UNPARSED_PRODUCTION"); break;
case VAL_LIST_ENTRY_PRODUCTION: LOG("VAL_LIST_ENTRY_PRODUCTION"); break;
case VAL_NOTHING_PRODUCTION: LOG("VAL_NOTHING_PRODUCTION"); break;
case VAL_PRODUCTION: LOG("VAL_PRODUCTION"); break;
case VAL_PROP_OF_PRODUCTION: LOG("VAL_PROP_OF_PRODUCTION"); break;
case VALUE_PHRASE_PRODUCTION: LOG("VALUE_PHRASE_PRODUCTION"); break;
```

```
            case VERB_PRODUCTION: LOG("VERB_PRODUCTION"); break;
            case VP_PRODUCTION: LOG("VP_PRODUCTION"); break;
            default: LOG("<unknown-production-%08x>", production); break;
        }
    } else log_meaning_code(production);
}
```

The function log_production is called from 2/dl.

§**2.** Logging a meaning list is more than a matter of displaying an indented tree, because of the ambiguities present. The log uses the notation `[1/3]` for "possibility 1 of 3".

```
void log_meaning_list(meaning_list *ml) {
    if (ml == NULL) { LOG("<null-meaning-list>\n"); return; }
    log_ml_recursively(ml, 0, 0);
}
void log_ml_recursively(meaning_list *ml, int num, int of) {
    int w1, w2;
    ⟨Calculate num and of such that this is [num/of] if they aren't already supplied 3⟩;
    if (ml == NULL) { LOG("NULL\n"); return; }
    if (of > 1) {
        LOG("[%d/%d] ", num, of);
        if (ml->score != 0) LOG("(score %d) ", ml->score);
    }
    if (ml->em) LOG("$M", ml->em);
    else log_production(ml->production);
    if (ml->type_spec) {
        LOG(" => ");
        if (ml->production == TIME_PRODUCTION) LOG("$t", spec_get_condition_tense(ml->type_spec));
        else LOG("$S", ml->type_spec);
    }
    ⟨Describe attached data for a few special cases with pointers attached 4⟩;
    ml_get_text(ml, &w1, &w2);
    if (w1 >= 0) LOG(" / \"$W\"", w1, w2);
    LOG("\n");
    if (ml->child) {
        STREAM_INDENT(dl); log_ml_recursively(ml->child, 0, 0); STREAM_OUTDENT(dl);
    }
    if (ml->next_alternative) log_ml_recursively(ml->next_alternative, num+1, of);
    if (ml->sibling) log_ml_recursively(ml->sibling, 0, 0);
}
```

The function log_meaning_list is called from 2/dl.

§**3.** When the first alternative is called, `log_ml_recursively` has arguments 0 and 0 for the possibility. The following code finds out the correct value for `of`, setting this possibility to be `[1/of]`. When we later iterate through other alternatives, we pass on correct values of `num` and `of`, so that this code won't be used again on the same horizontal list of possibilities.

⟨Calculate num and of such that this is [num/of] if they aren't already supplied 3⟩ ≡

```
if (num == 0) {
    meaning_list *ml2;
    for (ml2 = ml, of = 0; ml2; ml2 = ml2->next_alternative, of++) ;
    num = 1;
}
```

This code is used in §2.

§**4.** Most higher-up nodes in the list are described fully by production number alone – every `VAL_PRODUCTION` is like every other. But a few have data attached, a pointer to some other structure, to clarify them. Not every `PREP_PRODUCTION` is like every other; it depends what the preposition usage is.

⟨Describe attached data for a few special cases with pointers attached 4⟩ ≡

```
switch (ml->production) {
    case DETERMINER_PRODUCTION:
        LOG(" => ");
        log_quantifier(RETRIEVE_POINTER_quantifier(ml->data_attached), ml->score); break;
    case VERB_PRODUCTION:
        LOG(" => ");
        log_verb_usage(RETRIEVE_POINTER_verb_usage(ml->data_attached)); break;
    case PREP_PRODUCTION:
        LOG(" => ");
        log_preposition_usage(RETRIEVE_POINTER_preposition_usage(ml->data_attached)); break;
}
```

This code is used in §2.

§**5. Creation.** When we ask the memory manager to create a new structure, we increase the amount of memory claimed from the operating system, little by little. This memory will not be given back until Inform exits: structures, once created, are permanent. Normally this is what we want – to hold a phrase definition, for instance, which needs to be available for the rest of the run.

Meaning lists are the exception. In tests made in February 2009, compiling "Bronze" generated around 189,000 `meaning_list` structures, but only 1 in 50 were needed in permanent storage – to hold the lists of excerpt meanings attached to vocabulary words, which together make up Inform's equivalent of a symbols table. The other 49 in 50 `meaning_list` structures were ephemeral – an intermediate result of parsing text which could be thrown away once acted on. So that is what the following new system will do.

Each `meaning_list` is marked with an expiry date when created – most often the "current time", just as supermarket bread is tagged with a sell-by date which is the same as the day of baking. The rarer permanent MLs are marked instead with impossibly distant expiry dates, like Army field rations. The "current time", for this purpose, has no connection with the time of day. It begins at 0 and advances by 1 whenever Inform completes some parsing-heavy task: working through an assertion sentence, compiling a phrase, and so on. A meaning list structure whose expiry date is before the current time is said to have "expired".

```
define THE_INFINITE_FUTURE 2147483647
```

§**6.** When we need a new meaning list, we first look for an expired one to reuse – only if that fails do we ask the memory manager to create a new structure. A complete search of existing structures would produce the best-possible memory economy, but would also be slow. For speed reasons we therefore use the following pragmatic strategy:

(a) A new ephemeral ML reuses the first expired structure after the last-created permanent one, but
(b) A new permanent ML reuses the expired structure in memory.

We do this by keeping three markers: the earliest ephemeral ML occurring one or more places before some permanent ML, the "GAP"; the latest permanent one, the "LP"; and the start of the expired tail, "TAIL", characterised by the fact that it and all subsequent MLs have expired.

For instance, suppose the time is now 1021 and the list of MLs in memory shows expiry dates thus:

```
PERMANENT -> 1020 -> PERMANENT -> PERMANENT -> 1021 -> 1020 -> 976
             ^GAP                ^LP                 ^TAIL
```

A new ephemeral ML reuses the TAIL position, the second 1020, and TAIL moves forwards:

```
PERMANENT -> 1020 -> PERMANENT -> PERMANENT -> 1021 -> 1021 -> 976
             ^GAP                ^LP                          ^TAIL
```

Whereas a new permanent ML reuses the GAP position, filling in the gap, and GAP becomes `NULL`:

```
PERMANENT -> PERMANENT -> PERMANENT -> PERMANENT -> 1021 -> 1021 -> 976
                                       ^LP                          ^TAIL
```

Note that LP moves only forwards. GAP is `NULL` from time to time, but its non-`NULL` values always move forwards, too. The sequence is always GAP strictly behind LP strictly behind TAIL, when these are not `NULL`, and no two ever coincide.

Perfect efficiency is achieved when GAP is `NULL`, as here. In practice this doesn't always happen. But MLs do tend to concentrate early in memory; on a long run they end up about 90% contiguous, that is, if there are N permanent MLs then they tend to live in the first $1.1N$ positions. That will be good enough for us, and the important point about the above algorithm is that it allocates M objects in $O(M)$ time, not $O(M^2)$, which with $M \simeq 190,000$ would hurt.

```
meaning_list *LP_marker = NULL, *GAP_marker = NULL, *TAIL_marker = NULL;
meaning_list *earliest_ephemeral_ML_today = NULL;
int current_creation_time = 0, max_ML_creations_per_day = 0, no_ML_creations_today;
```

§**7.** First, the allocation.

```
meaning_list *get_available_ml(int seeking_permanent_slot) {
    meaning_list *new;
    if ((seeking_permanent_slot) && (GAP_marker)) {
        new = GAP_marker;
        LOGIF(ML_ALLOC, "Time %d: Using GAP position ML%d with expiry time %d\n",
            current_creation_time, new->allocation_id, new->expiry_time);
        here TAIL does not change
        ⟨Move the GAP marker forward to the next gap, if any 8⟩;
    } else if (TAIL_marker) {
        new = TAIL_marker;
        LOGIF(ML_ALLOC, "Time %d: Using TAIL position ML%d with expiry time %d\n",
            current_creation_time, new->allocation_id, new->expiry_time);
        here any GAP is unaltered
        TAIL_marker = NEXT_OBJECT(TAIL_marker, meaning_list);
        if (seeking_permanent_slot) LP_marker = new;
    } else {
        new = CREATE(meaning_list);
        LOGIF(ML_ALLOC, "Time %d: Using new position ML%d\n",
            current_creation_time, new->allocation_id);
        here any TAIL vanishes, but a GAP is unaltered
        TAIL_marker = NULL;
        if (seeking_permanent_slot) LP_marker = new;
    }
    if (seeking_permanent_slot) new->expiry_time = THE_INFINITE_FUTURE;
    else {
        new->expiry_time = current_creation_time;
        if (earliest_ephemeral_ML_today == NULL) earliest_ephemeral_ML_today = new;
    }
    no_ML_creations_today++;
    LOGIF(ML_ALLOC, "Time %d: GAP = ML%d, LP = ML%d, TAIL = ML%d\n",
        current_creation_time,
        (GAP_marker)?(GAP_marker->allocation_id):0,
        (LP_marker)?(LP_marker->allocation_id):0,
        (TAIL_marker)?(TAIL_marker->allocation_id):0);
    return new;
}
```

§**8.** Everything before the GAP is permanent, and the new item put there will also be permanent. We must move forward over unexpired items. There are then three possibilities: we run out (there are now no expired items in the list, so the new GAP is `NULL`); or we are at the TAIL position (so everything is expired from here on, and GAP must again be `NULL`); or we are at an expired item before the TAIL, which is therefore a new valid GAP item.

⟨Move the GAP marker forward to the next gap, if any 8⟩ ≡

```
    do {
        GAP_marker = NEXT_OBJECT(GAP_marker, meaning_list);
        GAP_movements++;
    } while ((GAP_marker) && (GAP_marker->expiry_time >= current_creation_time));
    if (GAP_marker == TAIL_marker) GAP_marker = NULL;
```

This code is used in §7.

§**9.** Second, moving time on to the next day. There is little prospect of reaching `THE_INFINITE_FUTURE`, but just in case we do, we stop time there; infinity is the day that never sees midnight.

```
void finish_this_session_of_parsing(void) {
    if (current_creation_time < THE_INFINITE_FUTURE) {
        if (dl_this(ML_ALLOC_DA)) {
            LOG("Time %d: %d items were created today ",
                current_creation_time, no_ML_creations_today);
            if (no_ML_creations_today > max_ML_creations_per_day) {
                LOG("- a new record!");
                max_ML_creations_per_day = no_ML_creations_today;
            }
            LOG("\n");
        }
        ⟨Adjust the markers at midnight 10⟩;
        current_creation_time++;
        earliest_ephemeral_ML_today = NULL;
        no_ML_creations_today = 0;
    }
}
```

The function finish_this_session_of_parsing is called from 8/tass, 10/str, 10/tab, 12/cs, 12/phsf, 12/cph and 13/tfg.

§**10.** To continue our example, at one minute to midnight on day 1021 we had:

```
PERMANENT -> PERMANENT -> PERMANENT -> PERMANENT -> 1021 -> 1021 -> 976
                                       ^LP                          ^TAIL
```

And at one minute past midnight on day 1022 we have:

```
PERMANENT -> PERMANENT -> PERMANENT -> PERMANENT -> 1021 -> 1021 -> 976
                                       ^LP          ^TAIL
```

LP doesn't move, since there is no change to the permanent items. Since every non-permanent item expired at midnight, the new TAIL always starts just after LP.

The tricky one is GAP. If it is non-`NULL` at midnight, it doesn't change, since it is still an expired item with everything before it permanent. But if it is `NULL`, we only know that any gaps in the permanent items are filled with day-1021 creations, like this:

```
PERMANENT -> PERMANENT -> PERMANENT -> 1021 -> 1021 -> PERMANENT -> 1021 -> 976
```

Consider the first day-1021 creation in the list. If it doesn't exist (i.e., there are no day-1021 items) there is still no GAP. Otherwise it exists, and is either before, at or after the new TAIL position. If before TAIL (or if there is no TAIL), it is the new GAP. If at or after TAIL then there are no permanent items after it, and once again there is still no GAP.

⟨Adjust the markers at midnight 10⟩ ≡

```
    TAIL_marker = NEXT_OBJECT(LP_marker, meaning_list);
    if (GAP_marker == NULL) {
        GAP_marker = earliest_ephemeral_ML_today;
        if ((TAIL_marker) && (GAP_marker) &&
            (GAP_marker->allocation_id >= TAIL_marker->allocation_id))
            GAP_marker = NULL;
    }
```

This code is used in §9.

§**11.** We can finally prove the running time of this algorithm over the entire run. Suppose M meaning lists are allocated during the run. The midnight operation runs in $O(1)$ but occurs on each of the D days; however, each day's activity requires the creation of at least one meaning list, so $D \leq M$ and midnight operations are worst $O(M)$. The allocation operation contains just one loop, when GAP moves forwards. Now GAP always advances until it becomes `NULL`, and since the list has length at most M, this is $\leq M$ iterations in total across Inform's run – *except* that the midnight operation sometimes puts it back after it has become `NULL`, forcing it to traverse the last few items again.

For each t, let $C(t)$ be the number of allocations made on day t. We prove by induction that the number of extra GAP movements due to being put back at the end of day t, which we call $X(t)$, satisfies $X(t) \leq C(t) + C(t+1)$.

Suppose, at midnight on day t, GAP is put back into the new list. How many extra steps forward will that cost us before it falls off again? The answer is that it is put back at the earliest ephemeral day t creation. This is at most $C(t)$ steps from the TAIL, which it will never advance past. However, the TAIL itself moves forwards during day $t+1$, by at most $C(t+1)$ steps. So the worst case is that the extra steps incurred due to day $t+1$ is actually $C(t) + C(t+1)$.

Therefore the total number of extra steps forward is

$$X = \sum_{t=0}^{D-1} X(t) \leq \sum_{t=0}^{D-1} C(t) + C(t+1) \leq 2 \sum_{t=0}^{D-1} C(t) = 2M.$$

It takes at most M regular steps forward, so, finally, the number of iterations of the loop is bounded across the whole run by $3M$, and hence our allocation algorithm runs in $O(M)$ time.

§**12.** Testing with "Bronze" in February 2009 in fact produced just 2438 steps where $M = 189165$, so in real-world usage it is very likely well under $3M$. At the end of the run there were 5411 MLs in memory, of which 4166 were permanent. Only 1245 ephemeral MLs existed, instead of 184,999, saving about 10.5MB of memory – a saving of 30% off the total memory bill.

Though it had been intended as a trade-off of speed for memory savings, it in fact shaved about 15% off Inform's total running time, because `get_available_ml` acts as a fast cache for spare memory in rapid-fire parsing.

(An interesting reflection on the usefulness of literate programming is that the author wrote the whole of this "Creation" section, all but this paragraph, before trying anything out. This took the better part of three hours, but the result compiled after two typos were fixed, and worked perfectly first time, completing the entire Inform test suite with no errors.)

§**13. Construction.** We can now forget how memory for MLs is found, and worry about how to make them and what to use them for. We request them either temporarily or permanently:

```
meaning_list *new_ml(int code_number) {
    meaning_list *ml = get_available_ml(FALSE);
    ⟨Initialise the rest of the ML structure 14⟩;
    no_ephemeral_MLs++;
    return ml;
}
meaning_list *new_permanent_ml(int code_number) {
    meaning_list *ml = get_available_ml(TRUE);
    ⟨Initialise the rest of the ML structure 14⟩;
    no_permanent_MLs++;
    return ml;
}
```

The function new_ml is called from 5/parse, 5/lit, 5/candd, 5/tandv, 5/varc, 5/candp and 5/mlc.
The function new_permanent_ml is called from 5/em.

§**14.** But in each case the result looks the same.

⟨Initialise the rest of the ML structure 14⟩ ≡

```
    ml->em = NULL;
    ml->next_alternative = NULL;
    ml->sibling = NULL;
    ml->child = NULL;
    ml->type_spec = NULL;
    ml->score = 0;
    ml->production = code_number;
    ml->word_pair = -1;
    ml->data_attached = NULL_GENERAL_POINTER;
```

This code is used in §13.

§**15.** The following constructor routines fill out the fields in useful ways. Here's one if a word range is to be attached:

```
meaning_list *new_ml_with_words(int code_number, int w1, int w2) {
    meaning_list *ml = new_ml(code_number);
    if ((w1 >= 0) && (w1<=w2)) ml->word_pair = w1 + ((w2-w1) << 20);
    else ml->word_pair = -1;
    return ml;
}
```

The function new_ml_with_words is called from 5/arch, 5/parse and 5/candd.

§**16.** And here is one deriving from a world object:

```
meaning_list *ml_from_wo(world_object *wo) {
    meaning_list *ml = new_ml(WORLD_OBJECT_MC);
    ml->em = wo_get_principal_meaning(wo);
    ml->score = 1;
    return ml;
}
```

The function ml_from_wo is called from 5/candd.

§**17.** TYPE_PRODUCTION nodes hold specifications. As we will see in Chapter 7, these can represent both actual and generic values (for instance, "7" and "a number"), but the ones attached to TYPE_PRODUCTION nodes are always generic – so they really do represent types. First, a version to create such a node from the type ID number:

```
meaning_list *make_type_ml_from_ID(int ID_number) {
    meaning_list *ml = new_ml(TYPE_PRODUCTION);
    ml->type_spec = new_generic_type_from_ID(ID_number);
    return ml;
}
```

The function make_type_ml_from_ID is called from 5/candd.

§**18.** Second, from a given kind of value:

```
meaning_list *make_type_ml_from_kov(kind_of_value *kov) {
    meaning_list *ml = new_ml(TYPE_PRODUCTION);
    ml->type_spec = new_generic_CONSTANT_type(kov);
    return ml;
}
```

The function make_type_ml_from_kov is called from 5/candd.

§**19.** DETERMINER_PRODUCTION nodes record a quantifier (the underlying meaning of a determiner, that is) together with its numeric parameter, in case relevant.

```
meaning_list *make_determiner_ml(quantifier *quant, int parameter, int w1, int w2) {
    meaning_list *ml = new_ml_with_words(DETERMINER_PRODUCTION, w1, w2);
    ml->data_attached = STORE_POINTER_quantifier(quant);
    ml->score = parameter;
    return ml;
}
```

The function make_determiner_ml is called from 5/candd.

§**20.** TIME_PRODUCTION nodes record a time period, a structure which despite its name can also simply be a tense indication. Slightly bogusly, the time period is attached by being part of a condition SP, though no specific condition is meant – it's just that conditions always have time periods attached to them, so this is the natural way to store them.

```
meaning_list *make_time_ml(time_period tp) {
    meaning_list *ml = new_ml(TIME_PRODUCTION);
    ml->type_spec = new_CONDITION_spec(store_tp(tp));
    return ml;
}
```

The function make_time_ml is called from 5/candp.

§**21.** An `OPTION_PRODUCTION` node holds a phrase option name. If negated with "not", it's the child of a `COND_NOT_PRODUCTION` production.

```
meaning_list *make_option_ml(int opt_num, int sense) {
    meaning_list *ml = new_ml(OPTION_PRODUCTION);
    ml->type_spec = new_TEST_PHRASE_OPTION_spec(opt_num);
    if (sense == FALSE) {
        meaning_list *ml2 = new_ml(COND_NOT_PRODUCTION);
        meaning_list *ml3 = new_ml(COND_PRODUCTION);
        ml2->child = ml3; ml3->child = ml;
        return ml2;
    }
    return ml;
}
```

The function make_option_ml is called from 5/candp.

§**22.** `ACTION_PRODUCTION` nodes hold action patterns – descriptions of a set of actions, such as "taking or dropping a container", in the present or past tense.

```
meaning_list *make_action_ml(action_pattern ap) {
    meaning_list *ml = new_ml(ACTION_PRODUCTION);
    ml->type_spec = new_TEST_ACTION_spec(store_ap(ap));
    return ml;
}
meaning_list *make_pastaction_ml(action_pattern ap) {
    meaning_list *ml = new_ml(ACTION_PRODUCTION);
    ml->type_spec = new_TEST_PAST_ACTION_spec(store_ap(ap));
    return ml;
}
```

The function make_action_ml is called from 5/candp.
The function make_pastaction_ml is called from 5/candp.

§**23. Copying MLs.** Note that this copies the contents of the ML, but not the expiry date, or indeed the memory manager's private fields.

```
void ml_copy(meaning_list *ml_to, meaning_list *ml_from) {
    ml_to->em = ml_from->em;
    ml_to->next_alternative = ml_from->next_alternative;
    ml_to->sibling = ml_from->sibling;
    ml_to->child = ml_from->child;
    ml_to->type_spec = ml_from->type_spec;
    ml_to->score = ml_from->score;
    ml_to->production = ml_from->production;
    ml_to->word_pair = ml_from->word_pair;
    ml_to->data_attached = ml_from->data_attached;
}
```

The function ml_copy is called from 5/parse.

§24. Access routines.

```
int ml_production(meaning_list *ml) {
    return ml->production;
}
void ml_set_production(meaning_list *ml, int pr) {
    ml->production = pr;
}
```

The function ml_production is called from 5/candd, 5/tandv, 5/varc, 5/mlc, 6/sconv, 12/phtd and 13/test.
The function ml_set_production is called from 5/varc and 5/mlc.

§25. Navigating the tree structure through subclauses:

```
meaning_list *ml_right(meaning_list *ml) {
    return ml->sibling;
}
void ml_set_right(meaning_list *ml, meaning_list *ml2) {
    ml->sibling = ml2;
}
meaning_list *ml_down(meaning_list *ml) {
    return ml->child;
}
void ml_set_down(meaning_list *ml, meaning_list *ml2) {
    ml->child = ml2;
}
```

The function ml_right is called from 5/candd, 5/varc, 5/candp, 5/mlc, 6/sconv and 12/phtd.
The function ml_set_right is called from 5/candd, 5/varc, 5/candp and 5/mlc.
The function ml_down is called from 5/tandv, 5/varc, 5/mlc, 6/sconv, 12/phtd and 13/test.
The function ml_set_down is called from 5/candd, 5/tandv, 5/varc, 5/candp and 5/mlc.

§26. Alternatives, where the meaning list forks:

```
meaning_list *ml_sideways(meaning_list *ml) {
    return ml->next_alternative;
}
int ml_match_score(meaning_list *ml) {
    return ml->score;
}
```

The function ml_sideways is called from 5/mlc and 9/wo.
The function ml_match_score is called from 5/candd, 5/mlc and 9/wo.

§**27.** The attached text:

```
void ml_set_text(meaning_list *ml, int w1, int w2) {
    if (w1<0) ml->word_pair = -1;
    else ml->word_pair = w1 + ((w2-w1) << 20);
}
void ml_get_text(meaning_list *ml, int *w1, int *w2) {
    int w = ml->word_pair;
    if (w == -1) {
        *w1 = -1; *w2 = -1;
    }
    else {
        *w1 = w & ((1<<20)-1);
        *w2 = (w >> 20) + (*w1);
    }
}
```

The function ml_set_text is called from 5/parse and 5/varc.
The function ml_get_text is called from 5/parse, 5/mlc, 6/sconv and 12/phtd.

§**28.** An excerpt meaning:

```
excerpt_meaning *ml_meaning(meaning_list *ml) {
    return ml->em;
}
void ml_set_meaning(meaning_list *ml, excerpt_meaning *em) {
    ml->em = em;
}
```

The function ml_meaning is called from 5/aph, 5/em, 5/unitr, 5/lit, 5/candd, 5/varc, 5/candp, 5/mlc, 8/refpt, 9/qty, 9/wo, 9/prop, 10/tab, 10/fig, 10/sfx, 10/exf, 10/isin, 11/ap, 11/nap, 11/av, 12/br and 12/rb.
The function ml_set_meaning is called from 5/varc.

§**29.** Or maybe a specification:

```
specification *ml_get_attached_spec(meaning_list *ml) {
    return ml->type_spec;
}
void ml_attach_spec(meaning_list *ml, specification *spec) {
    ml->type_spec = spec;
}
```

The function ml_get_attached_spec is called from 5/mlc.
The function ml_attach_spec is called from 5/lit and 5/tandv.

§**30.** And perhaps also an attached pointer to some other structure of unspecified type.

```
general_pointer ml_get_attached_data(meaning_list *ml) {
    return ml->data_attached;
}
void ml_attach_data(meaning_list *ml, general_pointer data) {
    ml->data_attached = data;
}
```

The function ml_get_attached_data is called from 5/mlc and 6/sconv.
The function ml_attach_data is called from 5/varc and 5/mlc.

Architecture of the S-Parser 5/arch

Purpose

To describe how the S-parser is organised, and to define a kit of C macros for writing its routines.

5/arch.§1 Plain Text; §2-6 Convenient macros for writing SP-routines

---

Definitions

¶1. The theory and practice of writing parsers, which turn text into explicit or implicit tree structures, was one of the first major fields of research in computer science. In the 1960s, work on systematising parsers, and on proving theorems about them, was perhaps the first independent line of thought in this new field – a field up to then divided between pragmatic engineering work and the numerical analysis of rounding errors. It was influenced by Chomsky, whose pioneering book on generative linguistics came at just the right moment for early compiler theorists such as the young Donald Knuth. Today this area of research is largely passé, and the problem is considered solved. University courses stress the importance of using tools like `yacc` ("yet another compiler compiler"), which are mathematically demonstrable as correct. In the sense that the output of `yacc` correctly implements the semantics implied by the grammar fed into it, `yacc` is indeed superb.

But Inform doesn't use it. I am a little stubborn, and prefer to write my own parsers; I also feel that automatically generated parsers tend to react badly to ungrammatical input. For Inform purposes, it's unacceptable to give generic or unspecific problem messages when something won't parse – the enormous cataracts of errors thrown by `gcc` when a single brace is missing, for instance. My own view is that incorrect input should be regarded as the main use case, not as some kind of aberration which can be given second-rate service. The user more often miscompiles than compiles, and quality lies in how good our problem messages are in reply. Besides this, the S-parser sometimes uses context in sneaky ways difficult to express in `yacc`. Natural language is, despite Chomsky, not all that easily reducible to generative syntax; and it tends not to have hierarchies of associative operators on different levels, which is the classical case for using `yacc`.

So we won't use `yacc`, `antlr`, etc. All the same, it would be foolish not to systematise the routines in the S-parser, if only because most of them take a standardised format – they take the same form of input, use the same variables in the same sort of way, return the same form of output.

¶2. To define terms: we will call a routine "an SP-routine" if it –

(i) takes as input an excerpt `(w1, w2)` and perhaps some other parameters;
(ii) returns `NULL` if the excerpt is empty, or if it makes no match;
(iii) returns a meaning list if it does make a match, and such that the code of the head of this list is predictable by the caller.

Many SP-routines automatically skip any initial article ("an", "the", etc.) at the start of the excerpt, but some do not.

To give an example of (iii), a call to `SP_value` returns either `NULL` or a list headed by `VAL_PRODUCTION`.

(We will sometimes loosely call these routines "productions", since they generally correspond to productions in the grammar being parsed, if we want to think about it formally.)

¶3. We will start at the bottom of the S-parser and work upwards, that is, we start at simple noun phrases and work up to complex sentence structure.

(a) `SP_plain_text` accepts any excerpt without parsing it, usually so that it can be looked at later rather than now;
(b) `SP_excerpt` is a Swiss Army knife able to handle excerpt meanings filed under many meaning codes – it parses names with a fixed wording, such as "the containment relation";
(c) `SP_literal` parses values expressed notationally rather than by name, such as "12" or "five PM".

For (a), see below. (b) and (c) take up a whole section of this chapter each: "Parse Excerpts.w" and "Parse Literals.w", respectively.

The remaining SP-routines build more complex structure, breaking down excerpts into composite meanings. Those are divided into two sets:

(d) "Early" SP-routines, gathered together in "Early Productions.w". These are the ones needed early in Inform's run, when the A-parser needs to ask the S-parser to look at excerpts occurring in assertion sentences. For instance, "The recycling target is a number that varies. The recycling target is 75." is parsed using `SP_type_expression` to read "number that varies", and `SP_constant_value` to read "75" (though of course it does this by making use of `SP_literal`).
(e) "Late" SP-routines, gathered together in "Late Productions.w". These are the additional ones needed later on in Inform's run, when compiling definitions of rules and phrases. We can now use values like "the score plus 5", or conditions like "the red door has been open", which would make no sense in assertions.

---

§**1. Plain Text.** The simplest example of an SP-routine is `SP_plain_text`, which accepts any excerpt and makes a leaf-node out of them.

Profiling suggests that about 5% of the compiler's CPU time is spent in this routine, or rather, in the memory allocation needed to make it work. (It is needed so often because arguments to phrases are left as unparsed nodes during one stage of parsing.)

```
meaning_list *SP_plain_text(int w1, int w2) {
    return new_ml_with_words(UNPARSED_PRODUCTION, w1, w2);
}
```

The function SP_plain_text is called from 5/parse and 5/tandv.

§**2. Convenient macros for writing SP-routines.** Since they share a specification and are all variations on a theme, we will write the more interesting SP-routines using a kit of C macros.

The routine should begin with one of the two macros:

```
    begin_SP_routine(w1, w2);
    begin_SP_routine_skipping_article(w1, w2);
```

These automatically calculate the conjunction and disjunction of the flags held by words in the range, putting the result in the variables `disjunction` and `conjunction`. As discussed above, these can be used to screen out many inapplicable, but expensive to check, possibilities.

```
define begin_SP_routine(w1, w2)
    BEGIN_PRODUCTION_PRIMITIVE(w1, w2, FALSE)
define begin_SP_routine_skipping_article(w1, w2)
    BEGIN_PRODUCTION_PRIMITIVE(w1, w2, TRUE)
define BEGIN_PRODUCTION_PRIMITIVE(w1, w2, skip_arts)
    int i, j, k, conjunction, disjunction;
    meaning_list *ml, *ml2, *ml3, *ml4, *ml5, *final_ml;
    if ((w1<0) || (w2<w1)) return NULL;                    reject an empty excerpt
```

```
    if ((skip_arts) && ((w2>w1) && ((lw_array[w1].lw_identity->flags) & ARTICLE_MC))) w1++;      skip
article word
    ml = NULL; ml2 = NULL; ml3 = NULL; ml4 = NULL; ml5 = NULL; final_ml = NULL;
    j = 0; k = 0;
    calculate conjunction and disjunction of the flags for words in the excerpt:
    conjunction = lw_array[w1].lw_identity->flags;
    disjunction = conjunction;
    for (i=w1+1; i<=w2; i++) {
        int f = lw_array[i].lw_identity->flags;
        disjunction = disjunction | f;
        conjunction = conjunction & f;
    }
```

§**3.** The routine should correspondingly end with the macro:

```
    on_success_invoke(P)
```

where `P` is the production number for the routine – for instance, the routine parsing values ends with this macro for `VAL_PRODUCTION`. If ordinary execution reaches this point, we have failed and return `NULL`. The idea is that code in the middle of the routine will have tried possibilities and performed a `goto` to the `Accept` label if any worked, with `ml` pointing to the children which the production node should have. So the macro defines this label, and what to do with those children.

```
define on_success_invoke(P)
    return NULL;
    Accept:
    final_ml = new_ml_with_words(P, w1, w2); final_ml->child = ml; return final_ml;
```

§**4.** But we don't want any `goto` in our own code, of course; the testing of possible syntaxes is also (mostly) done with macros. The most basic of these calls some other production routine, and accepts its findings if it has any:

```
define try_production(some_function_call) { if ((ml = some_function_call) != NULL) goto Accept; }
```

§**5.** And on the other hand, suppose we definitely want to accept the excerpt but as a subtree rather than a single node. We then use one of the `invoke_*` macros:

```
define invoke_0(x) { ml = new_ml_with_words(x, w1, w2); goto Accept; }
define invoke_1(x, y) { ml = new_ml_with_words(x, w1, w2); ml->child = y; goto Accept; }
define invoke_2(x, y, z) { ml = new_ml_with_words(x, w1, w2);
    ml->child = y; y->sibling = z; goto Accept; }
define invoke_3(x, y, z, w) {
    ml = new_ml_with_words(x, w1, w2); ml->child = y; y->sibling = z; z->sibling = w;
    goto Accept; }
define invoke_4(x, y, z, w, t) {
    ml = new_ml_with_words(x, w1, w2); ml->child = y; y->sibling = z; z->sibling = w;
    w->sibling = t;
    goto Accept; }
```

§**6.** For instance, the following bogus example tries to match a value followed by the literal word "alongside" and then another value. (It does this quite badly, but it's only an example.)

```
meaning_list *SP_hypothetical(int w1, int w2) {
    begin_SP_routine(w1, w2);
    if [[w1, w2 == ... alongside ... : i]]
        if ((ml2 = SP_value(w1, i-1)) && (ml3 = SP_value(i+1, w2)))
            invoke_2(ALONGSIDE_PRODUCTION, ml2, ml3);
    on_success_invoke(HYPOTHETICAL_PRODUCTION);
}
```

If it succeeded, the (top of the) resulting meaning list would be:

```
HYPOTHETICAL_PRODUCTION / "17 alongside the box"
    ALONGSIDE_PRODUCTION / "17 alongside the box"
        VAL_PRODUCTION / "17"
        VAL_PRODUCTION / "box"
```

Parse Excerpts 5/parse

Purpose

Given an excerpt `(w1,w2)`, to construct a meaning list of all registered excerpt meanings which it matches.

5/parse.§6-7 Exact parsing mode; §8-9 Maximal parsing mode; §10-11 Parametrised parsing mode; §12-13 Subset parsing mode; §14 Monitoring the efficiency of the parser

---

Template interpreter commands

```
14  {-callv:debug_parser_statistics}
```

Definitions

¶1. This is the lowest level of the S-parser, which looks only for atomic meanings ("red box"): routines higher up will look for more complex, compound meanings (such as "something in the red box"). The meaning list we return could be described as a list of alternative leaves, since each of the possible readings is a simple meaning which can't be broken down further (so, has no child nodes).

Any list of alternatives with two or more entries must be disambiguated much higher up in Inform. It is not at all easy to decide what "door" means, for instance: the class of doors, or a particular door, and if so then which one? We cannot answer that question here, and do not even try.

¶2. There are four basic parsing modes for excerpts.

(1) Exact parsing is what it sounds like: the texts have to match exactly, except that an initial article is skipped. Thus "the going action" exactly matches "going action", but "going" does not.
(2) In subset parsing, a match is achieved if the text parsed consists of words all of which are found in the meaning tested. Thus "red door" and "red" are each subset matches for "ornate red door with brass handle".
(3) In parametrised parsing, arbitrary (non-empty) text is allowed to match against `#` gaps in the token list. Thus "award 5 points" is a parametrised match for "award `#` points".
(4) In maximal parsing, we find the longest possible initial match, allowing it even if it does reach to the end of the excerpt, and we return a unique finding, not a list of possibilities.

```
define EXACT_PM 1
define SUBSET_PM 2
define PARAMETRISED_PM 4
define MAXIMAL_PM 8
```

¶3. We decide which of these to use with the following bitmaps.

```
define EXACT_PARSING_BITMAP
    (MISCELLANEOUS_MC + PROPERTY_MC + ADJECTIVE_MC +
    QUANTITY_MC + DESIGNED_TYPE_MC + RULE_MC + RULEBOOK_MC +
    TABLE_MC + TABLE_COLUMN_MC + NAMED_AP_MC + ACTIVITY_MC + EQUATION_MC)
define SUBSET_PARSING_BITMAP
    (WORLD_OBJECT_MC)
define PARAMETRISED_PARSING_BITMAP
    (VOID_PHRASE_MC + VALUE_PHRASE_MC +
    COND_PHRASE_MC + END_PHRASE_MC + SAY_PHRASE_MC)
define AL_BITMAP (NOT_MC + ARTICLE_MC + ADJECTIVE_MC)
```

¶4. To monitor the efficiency of the excerpt parser, we keep track of:

```
int no_calls_to_parse_excerpt = 0,
    no_meanings_tried = 0,
    no_meanings_tried_in_detail = 0,
    no_successful_calls_to_parse_excerpt = 0, no_matched_ems = 0;
```

---

§**1.** `SP_excerpt` is our first SP-routine, and it's the basic one from which most of the others are built.

As input, we give it not just the excerpt but also a context; or, to put it another way, a filter on which excerpt meanings to look at. This must be a bitmap made up from meaning codes, such as `TABLE_MC + TABLE_COLUMN_MC`, which would check for tables and table columns simultaneously.

However, there is one restriction on this. Recall that there are four parsing modes, and that different modes are used for different meaning codes. The `mc_bitmap` context is required not to mix MCs with different parsing modes.

```
meaning_list *SP_excerpt(int mc_bitmap, int w1, int w2) {
    int parsing_mode, h;
    meaning_list *results = NULL;
    no_calls_to_parse_excerpt++;
    if ((w1<0) || (w2<w1)) return NULL;                    in fact this case should never be needed
    h = 0;
    ⟨Choose which parsing mode we should use, given the MC bitmap 2⟩;
    ⟨Take note of casing on first word, in the few circumstances when we care 3⟩;
    ⟨Skip an initial article most of the time 4⟩;
    h = h | hash_code_from_excerpt(w1, w2);
    LOGIF(EXCERPT_PARSING,
        "Parsing excerpt <$W> hash %08x mc $p mode %d\n", w1, w2, h, mc_bitmap, parsing_mode);
    switch(parsing_mode) {
        case EXACT_PM: ⟨Enter exact parsing mode 6⟩; break;
        case MAXIMAL_PM: ⟨Enter maximal parsing mode 8⟩; break;
        case PARAMETRISED_PM: ⟨Enter parametrised parsing mode 10⟩; break;
        case SUBSET_PM: ⟨Enter subset parsing mode 12⟩; break;
        case 0: LOG("mc_bitmap: $p\n", mc_bitmap); internal_error("Unknown parsing mode");
        default: LOG("mc_bitmap: $p\n", mc_bitmap); internal_error("Mixed parsing modes");
    }
    LOGIF(EXCERPT_PARSING, "Completed:\n$m", results);
    meaning_list *loopy; for (loopy = results; loopy; loopy = loopy->next_alternative)
        no_matched_ems++;
    if (results) no_successful_calls_to_parse_excerpt++;
    return results;
}
```

The function SP_excerpt is called from 5/aph, 5/unitr, 5/lit, 5/candd, 5/tandv, 5/candp, 8/refpt, 8/mass, 9/qty, 9/wo, 9/prop, 10/tab, 10/fig, 10/sfx, 10/exf, 10/isin, 11/ap, 11/nap, 11/av, 12/br and 12/rb.

§**2.** Maximal parsing is something of a special case: it is used only for adjective lists, and we can only enter that mode by calling with exactly the correct bitmap for this. Otherwise, the parsing mode depends on which MC(s) are included in the bitmap.

⟨Choose which parsing mode we should use, given the MC bitmap 2⟩ ≡

```
parsing_mode = 0;
if (mc_bitmap & EXACT_PARSING_BITMAP) parsing_mode |= EXACT_PM;
if (mc_bitmap & SUBSET_PARSING_BITMAP) parsing_mode |= SUBSET_PM;
if (mc_bitmap & PARAMETRISED_PARSING_BITMAP) parsing_mode |= PARAMETRISED_PM;
if (mc_bitmap == AL_BITMAP) parsing_mode = MAXIMAL_PM;
```

This code is used in §1.

§**3.** Recall that excerpt parsing is case insensitive except for the first word of a text substitution, and then only when two definitions have been given, one capitalising the word and the other not. If we find the upper case form of such a text substitution, we set a special bit in the hash code. (The upper and lower case forms are both registered as excerpt meanings, with the same hash code except that one has this extra bit set and the other hasn't.)

⟨Take note of casing on first word, in the few circumstances when we care 3⟩ ≡

```
if (mc_bitmap & SAY_PHRASE_MC) {
    char *tx = lw_array[w1].lw_rawtext;
    if ((tx[0]) && (isupper(tx[0])) && (vocab_used_case_sensitively(lw_array[w1].lw_identity)))
        h = h | CAPITALISED_VARIANT_FORM;
}
```

This code is used in §1.

§**4.** An initial article is always skipped unless we are looking at a phrase; but then we are only allowed to skip an initial "the", and even then only if we aren't looking for text substitutions.

⟨Skip an initial article most of the time 4⟩ ≡

```
if (parsing_mode & PARAMETRISED_PM) {
    if (((mc_bitmap & SAY_PHRASE_MC) == 0) && [[w1, w2 == the ...]]) w1++;
} else {
    if ((w1<w2) && ((lw_array[w1].lw_identity->flags) & ARTICLE_MC)) w1++;
}
```

This code is used in §1.

§**5.** When checking cases below, we are always going to consider only those which have a meaning code among those we are looking for:

```
define EXCERPT_MEANING_RELEVANT(ml)
    (no_meanings_tried++, ((mc_bitmap & (ml->em->meaning_code))!=0))
define EXAMINE_EXCERPT_MEANING_IN_DETAIL
    LOGIF(EXCERPT_PARSING, "Trying $M (parsing mode %d)\n", ml->em, parsing_mode);
    no_meanings_tried_in_detail++;
```

§**6. Exact parsing mode.** Exact matching is just what it sounds like: the match must be word for word. Because of that, the excerpt meaning is guaranteed to be listed under the start list of the first word, if it matches (because there cannot be # tokens in the token list – if there were, we would be in parametrised parsing mode).

⟨Enter exact parsing mode 6⟩ ≡
```
meaning_list *ml;
vocabulary_entry *v = lw_array[w1].lw_identity;
if (v == NULL) internal_error("Unidentified word when parsing");
if ((v->flags) & mc_bitmap)
    for (ml = v->start_list; ml; ml = ml->next_alternative)
        ⟨Try to match excerpt in exact parsing mode 7⟩;
```
This code is used in §1.

§**7.** In exact parsing, the hash codes must agree perfectly:

⟨Try to match excerpt in exact parsing mode 7⟩ ≡
```
if (EXCERPT_MEANING_RELEVANT(ml) && (h == ml->em->excerpt_hash)) {
    EXAMINE_EXCERPT_MEANING_IN_DETAIL;
    if (ml->em->no_em_tokens == (w2-w1+1)) {
        int j, k, err;
        for (j=0, k=w1, err = FALSE; j<ml->em->no_em_tokens; j++, k++)
            if (ml->em->em_tokens[j] != lw_array[k].lw_identity) { err=TRUE; break; }
        if (err == FALSE) {
            meaning_list *further_result = new_ml(ml->em->meaning_code);
            further_result->em = ml->em;
            further_result->next_alternative = results;
            further_result->score = 1;
            results = further_result;
        }
    }
}
```
This code is used in §6.

§**8. Maximal parsing mode.**

⟨Enter maximal parsing mode 8⟩ ≡
```
vocabulary_entry *v = lw_array[w1].lw_identity;
if (v == NULL) internal_error("Unidentified word when parsing");
if ((v->flags) & mc_bitmap) {
    meaning_list *ml, *best_ml = NULL; int best_score = 0;
    for (ml = v->start_list; ml; ml = ml->next_alternative)
        ⟨Try to match excerpt in maximal parsing mode 9⟩;
    if (best_ml) {
        results = new_ml(best_ml->em->meaning_code);
        results->em = best_ml->em;
        results->score = best_score;
    }
}
```
This code is used in §1.

§**9.** In maximal matching, we keep only the longest exact match found, and if two have equal length then keep the first one found. (It should ideally never be the case that clashes occur.)

⟨Try to match excerpt in maximal parsing mode 9⟩ ≡
```
if (EXCERPT_MEANING_RELEVANT(ml) &&
    ((h & ml->em->excerpt_hash) == ml->em->excerpt_hash)) {
    EXAMINE_EXCERPT_MEANING_IN_DETAIL;
    if (ml->em->no_em_tokens <= w2-w1+1) {
        int j, k, err;
        for (err=FALSE, j=0, k=w1; j<ml->em->no_em_tokens; j++, k++)
            if (ml->em->em_tokens[j] != lw_array[k].lw_identity) { err = TRUE; break; }
        if ((err == FALSE) && (j>best_score)) {
            best_ml = ml; best_score = j;
        }
    }
}
```
This code is used in §8.

§**10. Parametrised parsing mode.** This is the only parsing mode which allows for arbitrary text to appear: i.e., where any text X can appear in "award X points", for example.

⟨Enter parametrised parsing mode 10⟩ ≡
```
int i;
vocabulary_entry *v = lw_array[w1].lw_identity;
if (v == NULL) internal_error("Unidentified word when parsing");
meaning_list *ml;
if (mc_bitmap & SAY_PHRASE_MC) {
    for (ml = blank_says_ml; ml; ml = ml->next_alternative) {
        int mw1, mw2;
        meaning_list *this_result = new_ml_with_words(SAY_PHRASE_MC, w1, w2);
        ml_get_text(this_result, &mw1, &mw2);
        ml_copy(this_result, ml);
        ml_set_text(this_result, mw1, mw2);
        this_result->child = SP_plain_text(w1, w2);
        this_result->next_alternative = results;
        results = this_result;
        no_meanings_tried++, no_meanings_tried_in_detail++;
    }
}
for (ml = v->start_list; ml; ml = ml->next_alternative)
    ⟨Try to match excerpt in parametrised parsing mode 11⟩;
if (w2>w1) {
    v = lw_array[w2].lw_identity;
    if (v == NULL) internal_error("Unidentified word when parsing");
    for (ml = v->end_list; ml; ml = ml->next_alternative)
        ⟨Try to match excerpt in parametrised parsing mode 11⟩;
}
for (i=w1+1; i<w2; i++) {
    v = lw_array[i].lw_identity;
    if (v == NULL) internal_error("Unidentified word when parsing");
    for (ml = v->middle_list; ml; ml = ml->next_alternative)
        ⟨Try to match excerpt in parametrised parsing mode 11⟩;
}
```

This code is used in §1.

§**11.** It is required here that the data supplied must be a pointer to a phrase, though it can be any type of phrase.

⟨Try to match excerpt in parametrised parsing mode 11⟩ ≡

```
    if (EXCERPT_MEANING_RELEVANT(ml) &&
        ((h & ml->em->excerpt_hash) == ml->em->excerpt_hash)) {
        int params_w1[32], params_w2[32];
        phrase *ph = RETRIEVE_POINTER_phrase(ml->em->data);
        int ph_opt_w1 = -1, ph_opt_w2 = -1, bl, j, k, t, err, saved_w1 = w1, saved_w2 = w2;
        EXAMINE_EXCERPT_MEANING_IN_DETAIL;
        if (ph_To_allows_options(ph)) {
            LOGIF(EXCERPT_PARSING, "Looking for phrase options\n");
            for (bl=0, k=w1+1; k<w2; k++) {
                if ((lw_array[k].lw_identity == COMMA_V) && (bl==0)) {
                    ph_opt_w1 = k+1; ph_opt_w2 = w2; w2 = k-1;
                    LOGIF(EXCERPT_PARSING, "Found phrase options <$W>\n",
                        ph_opt_w1, ph_opt_w2);
                    break;
                }
                if ((lw_array[k].lw_identity == OPENBRACKET_V) || (lw_array[k].lw_identity == OPENBRACE_V))
bl++;
                if ((lw_array[k].lw_identity == CLOSEBRACKET_V) || (lw_array[k].lw_identity == CLOSEBRACE_V))
bl--;
            }
        }
        for (err=FALSE, j=0, k=w1, t=0, bl=0; (j<ml->em->no_em_tokens) && (k<=w2); j++) {
            LOGIF(EXCERPT_PARSING, "j=%d, k=%d, t=%d\n", j, k, t);
            if (ml->em->em_tokens[j]) {
                if (ml->em->em_tokens[j] != lw_array[k].lw_identity) { err=TRUE; break; }
                k++;
            } else {
                int la;
                j++; params_w1[t]=k;
                if (j==ml->em->no_em_tokens) { params_w2[t++]=w2; k=w2+1; break; }
                for (la=j; la<ml->em->no_em_tokens; la++)
                    if (ml->em->em_tokens[la] == NULL) { la = -1; break; }
                if (la>0) {
                    params_w2[t++]=w2+j-la; k=w2+j-la+1; j--; continue;
                }
                if ((lw_array[k].lw_identity == OPENBRACKET_V) || (lw_array[k].lw_identity == OPENBRACE_V))
bl++;
                k++; if (k>w2) { err=TRUE; break; }
                while ((ml->em->em_tokens[j] != lw_array[k].lw_identity) || (bl>0)) {
                    if ((lw_array[k].lw_identity == OPENBRACKET_V) || (lw_array[k].lw_identity == OPENBRACE_V))
bl++;
                    if ((lw_array[k].lw_identity == CLOSEBRACKET_V) || (lw_array[k].lw_identity == CLOSEBRACE_V))
bl--;
                    k++; if (k>w2) { err=TRUE; break; }
                }
                j--;
                params_w2[t++] = k-1;
```

```
            }
        }
        LOGIF(EXCERPT_PARSING, "outcome has err=%d\n", err);
        if (j<ml->em->no_em_tokens) err = TRUE;
        if (k<=w2) err = TRUE;
        if (err == FALSE)
            for (k=0; k<t; k++)
                if ((params_w1[k]<0) || (params_w1[k]>params_w2[k]))
                    err = TRUE;
        if (err == FALSE) {
            meaning_list *last_param = NULL;
            meaning_list *this_result = new_ml_with_words(ml->em->meaning_code, w1, w2);
            this_result->em = ml->em;
            this_result->next_alternative = results;
            this_result->score = 1;
            if (ph_opt_w1 >= 0) {
                this_result->child = SP_plain_text(ph_opt_w1, ph_opt_w2);
                this_result->child->production = PHR_OPT_PRODUCTION;
                last_param = this_result->child;
            }
            for (k=0; k<t; k++) {
                meaning_list *ml2;
                ml2 = SP_plain_text(params_w1[k], params_w2[k]);
                if (last_param) last_param->sibling = ml2;
                else this_result->child = ml2;
                last_param = ml2;
            }
            results = this_result;
        }
        w1 = saved_w1; w2 = saved_w2;
    }
```

This code is used in §10.

§12. Subset parsing mode. In subset mode, each possible match is kept, and is assigned a numerical score based purely on the number of words in the full description which were missed out. This makes "door" a better match against "door" (0 words missed out) than against "green door" (1 word missed out).

Note that a single word which also has a meaning as a number is never matched. This is so that "11" (say) cannot be misinterpreted as an abbreviated form of an object name like "Chamber 11".

⟨Enter subset parsing mode 12⟩ ≡

```
    int i, j, k;
    if ((w1 == w2) && ((vocab_test_flags(w1, NUMBER_MC)) != 0)) goto SubsetFailed;
    for (i=w1, j=-1, k=-1; i<=w2; i++) {
        vocabulary_entry *v = lw_array[i].lw_identity;
        if (v == NULL) internal_error("Unidentified word when parsing");
        if ((v->flags) & ARTICLE_MC) continue;
        if (v->subset_list_length == 0) goto SubsetFailed;
        if (v->subset_list_length > j) { j = v->subset_list_length; k = i; }
    }
    if (k >= 0) {
        vocabulary_entry *v = lw_array[k].lw_identity;
        meaning_list *ml;
        for (ml = v->subset_list; ml; ml = ml->next_alternative)
```

```
        ⟨Try to match excerpt in subset parsing mode 13⟩;
    }
    SubsetFailed: ;
```

This code is used in §1.

§13.

⟨Try to match excerpt in subset parsing mode 13⟩ ≡

```
    if (EXCERPT_MEANING_RELEVANT(ml) && ((h & ml->em->excerpt_hash) == h)) {
        EXAMINE_EXCERPT_MEANING_IN_DETAIL;
        if (w2-w1 < ml->em->no_em_tokens) {
            int err = FALSE;
            if (ml->em->meaning_code == WORLD_OBJECT_MC) {
                world_object *wo = RETRIEVE_POINTER_world_object(ml->em->data);
                if ((wo) && (wo->kind_flag)) {
                    LOGIF(EXCERPT_PARSING, "Kind, so trying $M (exact)\n", ml->em);
                    err = TRUE;
                    if (ml->em->no_em_tokens == (w2-w1+1)) {
                        for (j=0, k=w1, err = FALSE; j<ml->em->no_em_tokens; j++, k++)
                            if (ml->em->em_tokens[j] != lw_array[k].lw_identity) {
                                err=TRUE; break;
                            }
                    }
                    goto SubsetMatchDecided;
                }
            }
            for (k=w1; k<=w2; k++) {
                err = TRUE;
                for (j=0; j<ml->em->no_em_tokens; j++)
                    if (ml->em->em_tokens[j] == lw_array[k].lw_identity) err=FALSE;
                if (err) break;
            }
            SubsetMatchDecided:
            if (err == FALSE) {
                meaning_list *this_result = new_ml(ml->em->meaning_code);
                this_result->em = ml->em;
                this_result->next_alternative = results;
                this_result->score = 100-((ml->em->no_em_tokens) - (w2-w1));
                results = this_result;
            }
        }
    }
```

This code is used in §12.

§14. Monitoring the efficiency of the parser. To give a rough idea, the medium-large *Bronze* runs to about 40,000 words of text including the extensions it uses, and around 4000 excerpt meanings are registered. In May 2008, `SP_excerpt` was being called 40,046 times (it seems a rough rule of thumb that it's called about once per word), so:

(a) a naive all-against-all comparison would mean 160,184,000 comparisons;
(b) considering only those excerpt meanings attached to relevant words reduces us to `no_meanings_tried` comparisons, which was 246,834;
(c) considering only those with a viable correct hash code reduces us further to `no_meanings_tried_in_detail` comparisons, which was 68,541.

Considering that `SP_excerpt` was successful 13,752 times, and that on many of those occasions it will have returned multiple results, we can be pretty confident that the tests in (b) and (c) – both very fast to perform – are quite good. In fact, further investigation showed that of the 68,541 comparisons we had to make, 63,762 were successful – so (b) and (c) were correct 93% of the time.

Moreover, even the expensive checking in detail was not too bad: in all but 21,774 of the 68,541 cases the token list to check was empty, because it was a test for "say [value]". Thus the mean number of direct word comparisons was only 0.5 per call, and of course even those only involve comparing pointers, not calling character-level routines like `strcmp`.

Profiling shows that time spent in the S-parser is dominated by memory allocation for the meaning lists it generates. Actual parsing is very rapid, and no further optimisation is worthwhile.

```
void debug_parser_statistics(void) {
    LOG("no_calls_to_parse_excerpt = %d\n", no_calls_to_parse_excerpt);
    LOG("no_meanings_tried = %d\n", no_meanings_tried);
    LOG("no_meanings_tried_in_detail = %d\n", no_meanings_tried_in_detail);
    LOG("no_successful_calls_to_parse_excerpt = %d\n", no_successful_calls_to_parse_excerpt);
    LOG("no_matched_ems = %d\n", no_matched_ems);
    LOG("number of excerpt meanings registered = %d\n", NUMBER_CREATED(excerpt_meaning));
    if (dl_this(EXCERPT_MEANINGS_DA)) log_all_meanings();
}
```

The function debug_parser_statistics is invoked by a command in a `.i6t` template file.

Parse Literals 5/lit

Purpose

To decide if the text in `(w1,w2)` is a value referred to notationally rather than by name.

Definitions

¶1. Literal values are explicitly written-out constants such as "false", "182" or "4:33 PM". In the event that the last literal value parsed was a number, a time, a truth state, a resource ID, or a quasi-numerical value written in a literal-pattern notation (see "Literal Patterns.w"), the actual value is conveniently stored in the global variable `last_literal_evaluated`. For other kinds of value, such as text, the variable is not used.

```
int last_literal_evaluated = 0;
```

---

§**1. The SP-routine for literals.** This amounts only to a wrapper for the general literal-parsing routine given below, which is also called by other parts of Inform directly to obtain constant values quickly.

```
meaning_list *SP_literal(int w1, int w2) {
    meaning_list *ml;
    specification *spec;
    kind_of_value *kov = is_a_literal(w1, w2);
    if (kov == NULL) return NULL;
    ml = new_ml(LITERAL_PRODUCTION);
    spec = new_actual_CONSTANT_spec(kov);
    spec->word_ref1 = w1; spec->word_ref2 = w2;
    ml_attach_spec(ml, spec);
    return ml;
}
```

The function SP_literal is called from 5/candd.

§**2.** It turns out to be convenient to provide routines which determine quickly whether a given excerpt is a literal number. Note that ordinals (`third`, `31st` and so on) are not normally legal as constants, and that `is_a_literal` below would return `NULL` on them.

```
int is_an_ordinal_or_cardinal(int w1, int w2) {
    if (is_a_literal_number(w1, w2, TRUE)) return TRUE;
    return FALSE;
}
int is_a_cardinal(int w1, int w2) {
    if (is_a_literal_number(w1, w2, FALSE)) return TRUE;
    return FALSE;
}
kind_of_value *is_a_literal_number(int w1, int w2, int allow_ordinals) {
    if ((w1<0) || (w1 != w2)) return NULL;
    ⟨Try a cardinal number as the literal 13⟩;
    if (allow_ordinals) ⟨Try an ordinal number as the literal 14⟩;
    return NULL;
}
```

The function is_an_ordinal_or_cardinal is called from 11/chron.
The function is_a_cardinal is called from 5/det.

§**3.** The following routine returns a KOV if `(w1, w2)` is a valid literal, storing the actual value in the global variable `last_literal_evaluated`: or else returns `NULL`, a value distinct from all valid KOV pointers, if not.

In the event that no match is made, so that the routine returns `NULL`, we set `last_literal_evaluated` to the quaint curiosity number. None of the rest of the code in Inform makes use of this fact, and it is done solely defensively, to protect us from bugs – more accurately, to protect us from writing incorrect code which accidentally works because a value left over from some previous call is still in the variable even though the most recent call failed.

```
define QUAINT_CURIOSITY_NUMBER 123456789
kind_of_value *is_a_literal(int w1, int w2) {
    last_literal_evaluated = QUAINT_CURIOSITY_NUMBER;
    if ((w1 < 0) || (w2<w1)) return NULL;
    if (w1 == w2) {
        ⟨Try a cardinal number as the literal 13⟩;
        ⟨Try double-quoted text as the literal 5⟩;
        ⟨Try true or false as the literal 6⟩;
    }
    ⟨Try a braced constant list as the literal 7⟩;
    ⟨Try a Unicode character code as the literal 8⟩;
    ⟨Try a figure, sound effect, or external file name as the literal 9⟩;
    ⟨Allow the word minus to precede a number or time 4⟩;
    ⟨Try any of the notations declared by literal patterns as the literal 10⟩;
    ⟨Try a length of time as the literal 11⟩;
    ⟨Try a time of day as the literal 12⟩;
    return NULL;
}
```

The function is_a_literal is called from 4/iext, 5/litp, 6/treec, 7/vasp, 8/sob, 8/mass, 9/wo, 9/madj, 10/tab, 10/bib, 10/fig, 10/sfx, 10/isin, 11/act, 12/phud, 12/phtd, 13/tfg and 13/gtok.

§**4.** "Minus" is only allowed for numbers and times, and the reason for this is that those are the numerical constants which are sometimes written out in words: thus "minus six" or "minus fifteen minutes". With units written in literal-pattern notations, numerical elements are written as digits, and a minus sign is used rather than the word "minus". So "minus 6 kg" (say) would not be legal, but "−6 kg" would.

⟨Allow the word minus to precede a number or time 4⟩ ≡

```
if [[w1, w2 == minus ...]] {
    kind_of_value *partial = is_a_literal(w1+1, w2);
    if ((is_kova(partial, NUMBER_TY)) || (is_kova(partial, TIME_TY))) {
        last_literal_evaluated = -last_literal_evaluated;
        return partial;
    }
}
```

This code is used in §3.

§**5. Text.** As far as we are concerned here, double-quoted strings of text are literals whose KOV is either `TEXT_ROUTINE_TY` (if they contain square-bracketed text substitutions) or `TEXT_TY` (if not). Higher up in Inform, they are sometimes reinterpreted as `QUOT_TY` (special quotation text used for box quotations overlaid on the main text window at run-time) or `UNDERSTANDING_TY` (grammar to match in the player's command), but that's invisible to us.

⟨Try double-quoted text as the literal 5⟩ ≡

```
if (vocab_test_flags(w1, TEXT_MC)) return kova(TEXT_TY);
if (vocab_test_flags(w1, TEXTWITHSUBS_MC)) return kova(TEXT_ROUTINE_TY);
```

This code is used in §3.

§**6. Truth states.**

⟨Try true or false as the literal 6⟩ ≡

```
if [[w1, w2 == false]] {
    last_literal_evaluated = 0;
    return kova(TRUTH_STATE_TY);
}
if [[w1, w2 == true]] {
    last_literal_evaluated = 1;
    return kova(TRUTH_STATE_TY);
}
```

This code is used in §3.

§**7. Constant lists.** Note that the empty list `{}` is a valid literal in Inform.

⟨Try a braced constant list as the literal 7⟩ ≡

```
if ([[w1, w2 == OPENBRACE CLOSEBRACE]] ||
    [[w1, w2 == OPENBRACE ... CLOSEBRACE]]) {
    kind_of_value *kov = parse_list_of_values(NULL, w1+1, w2-1, FALSE);
    if (kov) return kov;
}
```

This code is used in §3.

§**8. Unicode characters.** This includes named characters as well as decimal character codes: see "Unicode Translations.w".

⟨Try a Unicode character code as the literal 8⟩ ≡

```
if [[w1, w2 == unicode ...]] {
    int unicode_val = is_a_unicode_value(w1+1, w2);
    if (unicode_val >= 0) {
        last_literal_evaluated = unicode_val;
        return kova(UNICODECHAR_TY);
    }
}
```

This code is used in §3.

§**9. External resource names.** These three categories of name must all start with particular words, which enables us to avoid calling `SP_excerpt` almost all of the time – indeed this is why the names are set up that way, as it's essential for the current routine to produce negative results quickly.

⟨Try a figure, sound effect, or external file name as the literal 9⟩ ≡

```
if [[w1, w2 == figure/file/sound ...]] {
    meaning_list *ml = SP_excerpt(MISCELLANEOUS_MC, w1, w2);
    if (ml)
        switch(em_get_secondary_code(ml_meaning(ml))) {
            case FIGURE_SMC: {
                blorb_figure *bf = RETRIEVE_POINTER_blorb_figure(em_data(ml_meaning(ml)));
                last_literal_evaluated = fig_get_resource_ID(bf);
                return kova(FIGURENAME_TY);
            }
            case EXTERNAL_FILE_SMC: {
                external_file *exf =
                    RETRIEVE_POINTER_external_file(em_data(ml_meaning(ml)));
                last_literal_evaluated = exf->allocation_id;
                return kova(EXTERNALFILE_TY);
            }
            case SOUND_EFFECT_SMC: {
                blorb_sound *bs = RETRIEVE_POINTER_blorb_sound(em_data(ml_meaning(ml)));
                last_literal_evaluated = sfx_get_resource_ID(bs);
                return kova(SOUNDNAME_TY);
            }
        }
}
```

This code is used in §3.

§10. Literal patterns. This delegates the problem of matching user-defined notations for constants to "Literal Patterns.w".

⟨Try any of the notations declared by literal patterns as the literal 10⟩ ≡

```
literal_pattern *lp;
LOOP_OVER(lp, literal_pattern) {
    int val;
    kind_of_value *kov = match_literal_pattern(lp, w1, w2, &val);
    if (kov) {
        last_literal_evaluated = val;
        return kov;
    }
}
```

This code is used in §3.

§11. Time. An interesting, perhaps questionable, decision which Inform makes here is to assign durations – relative times – the same kind of value as absolute times. This is sensible in some ways, because arithmetic on times is then meaningful – 12:04 AM plus 40 minutes is 12:44 AM; 4:52 PM minus 3:31 PM is 1 hour 21 minutes, and so on. If we regarded relative and absolute times as being distinct KOVs, the arithmetic rules for them would be complicated to type-check. On the other hand, times really do not behave like integers, no matter how we might pretend. What is 4:52 PM plus 3:31 PM, in any very meaningful sense? (Inform adds these by treating them as durations since the previous midnight in each case, but it's hard to see why that makes much human sense.)

Recall that I7 stored time as integers, counting in minutes.

⟨Try a length of time as the literal 11⟩ ≡

```
int min1, min2, hr1, hr2, j;
if [[w1, w2 == ... hour/hours --> hr1, hr2]] {
    if (is_a_cardinal(hr1, hr2)) {
        last_literal_evaluated = last_literal_evaluated*60;
        return kova(TIME_TY);
    }
}
if [[w1, w2 == ... minute/minutes --> min1, min2]] {
    if ([[min1, min2 == ... hours ... : j --> hr1, hr2 ... min1, min2]] ||
        [[min1, min2 == ... hour ... : j --> hr1, hr2 ... min1, min2]]) {
        int time_minutes = 0, time_hours = 0, failed = FALSE;
        if (is_a_cardinal(hr1, hr2)) time_hours = last_literal_evaluated;
        else failed = TRUE;
        if (is_a_cardinal(min1, min2)) time_minutes = last_literal_evaluated;
        else failed = TRUE;
        if (failed) last_literal_evaluated = QUAINT_CURIOSITY_NUMBER;
        else {
            last_literal_evaluated = time_minutes + time_hours*60;
            return kova(TIME_TY);
        }
    } else {
        if (is_a_cardinal(min1, min2)) return kova(TIME_TY);
    }
}
```

This code is used in §3.

§**12.** An absolute time is stored as the number of minutes since midnight last, with midnight itself stored as 0.

Note that we allow "12:01 AM" (one minute past midnight) and "12:01 PM" (ditto noon), and also "0:01 AM" and "00:01 AM", but not "0:01 PM".

⟨Try a time of day as the literal 12⟩ ≡

```
int min1, min2, hr1, hr2, time_base = -1;
if [[w1, w2 == ... am --> w1, w2]] time_base = 0;
else if [[w1, w2 == ... pm --> w1, w2]] time_base = 12*60;
if (time_base >= 0) {
    int time_minutes = 0, time_hours = 0, j, failed = FALSE;
    if [[w1, w2 == ... COLON ... : j --> hr1, hr2 ... min1, min2]] {
        if (is_a_cardinal(hr1, hr2)) time_hours = last_literal_evaluated;
        else failed = TRUE;
        if (is_a_cardinal(min1, min2)) time_minutes = last_literal_evaluated;
        else failed = TRUE;
    } else {
        if (w1 != w2) failed = TRUE;
        else if (is_a_cardinal(w1, w1)) time_hours = last_literal_evaluated;
        else if (is_a_digital_clock_time(w1, &time_minutes, &time_hours)) ;
        else failed = TRUE;
    }
    if ((time_hours < 0) || (time_hours > 12)) failed = TRUE;
    if ((time_hours == 0) && (time_base > 0)) failed = TRUE;          reject for example "0:01 PM"
    if (time_hours == 12) time_hours = 0;                            allow for example "12:01 AM"
    if ((time_minutes < 0) || (time_minutes >= 60)) failed = TRUE;
    if (failed) last_literal_evaluated = QUAINT_CURIOSITY_NUMBER;
    else {
        last_literal_evaluated = time_base + time_hours*60 + time_minutes;
        return kova(TIME_TY);
    }
}
```

This code is used in §3.

§**13. Ordinals and cardinals.** A cardinal, that is, a number such as `five` or `351`:

⟨Try a cardinal number as the literal 13⟩ ≡

```
if (vocab_test_flags(w1, NUMBER_MC))
    ⟨Accept the evaluation of this literal number if it's in range for the VM 15⟩;
```

This code is used in §2,3,2,3,2,3.

§**14.** And an ordinal, that is, a number such as `fifth` or `351st`:

⟨Try an ordinal number as the literal 14⟩ ≡

```
if (vocab_test_flags(w1, ORDINAL_MC))
    ⟨Accept the evaluation of this literal number if it's in range for the VM 15⟩;
```

This code is used in §2.

§**15.** We are particularly sensitive to Z-machine overflows because this is a rare case where the choice of virtual machine affects the legal syntax for I7 source text – text originally written for use on Glulx, which allows for larger integers, might be moved over to a Z-machine project, with the user not realising the consequences.

⟨Accept the evaluation of this literal number if it's in range for the VM 15⟩ ≡

```
last_literal_evaluated = vocab_get_literal_number_value(lw_array[w1].lw_identity);
if (((last_literal_evaluated > 32767) || (last_literal_evaluated < -32768)) &&
    (target_VM_is_16_bit())) {
    sentence_problem(_P_(C5LiteralOverflow),
        "you use a number which is too large",
        "at least with the Settings for this project as they "
        "currently are. (Change to Glulx to be allowed to use "
        "much larger numbers.)");
    vocab_set_literal_number_value(lw_array[w1].lw_identity, 1);
    return NULL;
}
return kova(NUMBER_TY);
```

This code is used in §13,14,13,14,13,14.

Constants and Descriptions 5/candd

Purpose

To parse noun phrases in constant contexts, which specify values either explicitly or by describing them more or less vaguely.

5/candd.§1-3 Constant values; §4-7 Adjective lists; §8-19 Descriptions; §20 KOV names

---

§**1. Constant values.** The word "nothing" needs special treatment here. Sometimes it means the dummy value "not an object", and is genuinely a constant value; but at other times it behaves more like a determiner, as in "if nothing is on the table". To make this easier to resolve higher up, it gets its own production code `VAL_NOTHING_PRODUCTION`.

Constants are otherwise straightforward: literals and names of rules, rulebooks, quantities (but only if enumerated constant names), activities, tables, actions, use options, rulebook outcomes and relations.

```
meaning_list *SP_constant_value(int w1, int w2) {
    begin_SP_routine(w1, w2);
    if ((ml = SP_literal(w1, w2))) return ml;
    if [[w1, w2 == nothing]] return new_ml_with_words(VAL_NOTHING_PRODUCTION, w1, w2);
    [[w1, w2 == the ... --> w1, w2]];
    ⟨Try for an enumerated constant or the name of a rule, rulebook, activity or table 2⟩;
    ⟨Try for the name of an action, relation, rulebook outcome or use option 3⟩;
    return NULL;
}
```

The function SP_constant_value is called from 5/tandv.

§**2.** These nouns are all excerpt meanings registered under their own `*_MC` meaning codes, so we can combine their parsing in one call to `SP_excerpt`.

Note that quantity meanings can be either variable names or enumerated constant values: for instance, following

> Colour is a kind of value. Red, blue and green are colours. Hue is a colour that varies.

we want to recognise the quantity "blue" as a constant but not the quantity "hue".

```
define CONSTANT_VAL_BITMAP (RULE_MC + RULEBOOK_MC + QUANTITY_MC + ACTIVITY_MC + TABLE_MC + EQUATION_MC)
```

⟨Try for an enumerated constant or the name of a rule, rulebook, activity or table 2⟩ ≡

```
    if (disjunction & CONSTANT_VAL_BITMAP) {
        if ((ml = SP_excerpt(CONSTANT_VAL_BITMAP, w1, w2))) {
            if (ml_production(ml) == QUANTITY_MC) {
                quantity *q = RETRIEVE_POINTER_quantity(em_data(ml_meaning(ml)));
                if (qty_is_a_variable(q) == FALSE) return ml;
            } else return ml;
        }
    }
```

This code is used in §1.

§**3.** These nouns, on the other hand, are excerpt meanings registered under the "miscellaneous" category.

Note that `SP_excerpt` is called only if a signature word is present, which speeds matters up. Action names and relations are registered as meanings with the signature word included, rulebook outcomes and use options without.

⟨Try for the name of an action, relation, rulebook outcome or use option 3⟩ ≡

```
int x1 = -1, x2 = -1;
if [[w1, w2 == ... action/relation --> x1, x2]] x2++;
if [[w1, w2 == ... option/outcome --> x1, x2]] ;
if ((x1 >= 0) && ((ml = SP_excerpt(MISCELLANEOUS_MC, x1, x2)))) {
    switch (em_get_secondary_code(ml_meaning(ml))) {
        case ACTION_NAME_SMC: return ml;
        case RELATION_SMC: return ml;
        case RULE_OUTCOME_SMC: return ml;
        case USE_OPTION_SMC: return ml;
    }
}
```

This code is used in §1.

§**4. Adjective lists.** "I first tried to write a story when I was about seven. It was about a dragon. I remember nothing about it except a philological fact. My mother said nothing about the dragon, but pointed out that one could not say "a green great dragon", but had to say "a great green dragon". I wondered why, and still do" (Tolkien to Auden, 1955). We are going to allow lists of adjectives such as "green great" or "great green" in any order: although some have suggested conceptual hierarchies for adjectives (e.g., that size always precedes material) these are too tendentious to enforce.

The following tests whether an excerpt is a list of adjectives, mixing any of the various kinds of adjective which can occur. For example, the text

> exciting transparent green fixed in place

constructs the tree:

```
AL_PRODUCTION / exciting transparent green fixed in place
    {exciting = ADJECTIVE_MC}
    {transparent = ADJECTIVE_MC}
    {green = ADJECTIVE_MC}
    {fixed in place = ADJECTIVE_MC}
```

§**5.** Perhaps surprisingly, the word "not" is allowed in such lists, and for two independent reasons. Firstly, we want to be able to construct the negations of individual adjectives, so that for instance

> exciting not transparent fixed in place

constructs

```
AL_PRODUCTION / exciting not transparent fixed in place
    {exciting = ADJECTIVE_MC}
    ADJ_NOT_PRODUCTION
        {transparent = ADJECTIVE_MC}
    {fixed in place = ADJECTIVE_MC}
```

Though in this instance it would be equivalent to write "opaque" in place of "not transparent", some adjectives do not have named negations.

The other reason for negation is for a sentence such as:

> A hairless chimp is not a hairy animal.

(This was submitted as a bug report.) It would be grammatically sound to parse this as

(A hairless chimp) is not (a hairy animal).

but in fact we construe it as

(A hairless chimp) is (not a hairy animal).

This means that we do not need to reject the assertion as being negative, and can instead act on it, since "hairy" – in this example – is an either/or property; thus we can parse it as if it read

A hairless chimp is a not hairy animal.

```
meaning_list *parse_adjective_list(int w1, int w2, kind_of_value *domain_kov, int unrestricted) {
    int i, fw = w1, negate_all = FALSE;
    meaning_list *ml, *final_ml = NULL, *latest_ml = NULL;
    if ((w1<0) || (w2<w1)) return NULL;
    if [[fw, w2 == not a/an/some ... --> fw, w2]] negate_all = TRUE;
    for (i=fw; i<=w2; i++) if ((vocab_test_flags(i, AL_BITMAP)) == 0) return NULL;
    while (fw<=w2) {
        int negate = negate_all;
        if [[fw, w2 == not ... --> fw, w2]] negate = (negate)?FALSE:TRUE;
        ⟨Parse the longest possible adjective name from the front into ml 6⟩;
        if (negate) {
            meaning_list *ml2 = new_ml(ADJ_NOT_PRODUCTION);
            ml_set_down(ml2, ml); ml = ml2;
        }
        if (final_ml) ml_set_right(latest_ml, ml);
        else { final_ml = new_ml_with_words(AL_PRODUCTION, w1, w2); ml_set_down(final_ml, ml); }
        latest_ml = ml;
    }
    if (final_ml) {
        LOGIF(S_GRAMMAR, "parse_adjective_list on <$W> yields:\n$m\n", w1, w2, final_ml);
    }
    return final_ml;
}
```

§**6.** The constant `AL_BITMAP` has a pleasingly Arabic sound to it – a second-magnitude star, a corrupt arms deal – but is in fact a combination of the meaning codes found in an adjective list. The excerpt parser handles this request in a way specially arranged for us, finding the longest adjective matching at the start of the excerpt.

⟨Parse the longest possible adjective name from the front into ml 6⟩ ≡

```
int sc;
ml = SP_excerpt(AL_BITMAP, fw, w2);
if (ml == NULL) return NULL;
if (unrestricted == FALSE) {
    if (ml_production(ml) == ADJECTIVE_MC) {
        adjectival_phrase *aph = RETRIEVE_POINTER_adjectival_phrase(em_data(ml_meaning(ml)));
        if (aph_applicable_to(aph, domain_kov) == FALSE)
            return NULL;
    } else if (domain_kov) return NULL;
}
sc = ml_match_score(ml);
if (sc == 0) internal_error("Length-scored maximal parse with length 0");
fw += sc;
```

This code is used in §5.

§**7.** Wrapping that into an SP-routine gives:

```
meaning_list *SP_adjective_list(int w1, int w2) {
    begin_SP_routine_skipping_article(w1, w2);
    if ((ml2 = parse_adjective_list(w1, w2, NULL, TRUE))) invoke_1(DC_ADJS_PRODUCTION, ml2);
    on_success_invoke(DC_PRODUCTION);
}
```

The function SP_adjective_list is called from 5/tandv and 5/varc.

§**8. Descriptions.** In most programming languages, commands are like imperative verbs, but their noun phrases are specific. `print X`, say, prints an unambiguously determined quantity `X`. Sometimes wildcards are allowed, as when the Unix shell allows `ls *.txt` to list all text files in the current directory – just as a work of IF allows TAKE ALL. Inform goes further than this, and – in some circumstances at least – permits the `X` to be a more or less vague description of a collection of values, such that only at run-time can it be determined what values `X` refers to.

It might look as if a description is a generalisation of a value: that is, every value is a description (a completely specific one). In fact, that isn't true. "100", for instance, is a value but not a description. The only values allowed as descriptions are the constant world objects. This is done in order to make Inform a richer language for describing states of objects, just as we saw that the domain of an adjective can be a single object. In other words, "rusty iron door" is allowed as a description in order that "open rusty iron door" can also be a description, and therefore:

```
Instead of throwing something at the open rusty iron door: ...
```

can work as a rule. "Open rusty iron door" describes a set which contains either one object (the door, if it happens to be open) or none (if the door is currently closed). We don't allow the same thing with specific values for a variety of reasons – it would read badly, it would be much less efficient to parse quickly – and in any case it's not very useful. Adjectives applied to values tend not to be very dynamic, and source text like

```
if X is the even 3, ...
```

would make little sense.

§**9.** Whereas a value makes a simple meaning list, just a single leaf, a description is so much richer that it needs a larger tree to represent it. For example:

exactly one closed door (called the Portal)

results in the meaning list:

```
DC_PRODUCTION / "exactly one closed door ( called the portal )"
    DC_ADJSNOUN_PRODUCTION / "closed door"
        AL_PRODUCTION / "closed"
            {closed = ADJECTIVE_MC}
        {door = WORLD_OBJECT_MC}
    CALLED_PRODUCTION / "portal"
    DETERMINER_PRODUCTION => Card=1 / "exactly one"
```

§**10.** A description is a sequence of the following four elements, some of which are optional:

(a) *specifier*, which will be a determiner and/or an article (optional);

(b) *qualifier*, which for Inform means adjectives of the various kinds described above (optional);

(c) *noun*, which because these are physical descriptions will refer to a particular object or kind of object (compulsory unless the `allow_nounless` call parameter is set, which it only is to handle one peculiar exception – see "Late Parameters.w"); and

(d) *subordinate clause*, such as "in ..." or "which are on ..." (optional).

For the most part the sequence must be (a), (b), (c), (d), as in:

six of the / open / containers / in the Attic

but the composite words made up from quantifiers and kinds – something, anywhere, everybody, and such – force us to make an exception to this:

something / open / in the Attic

which takes the sequence (a) and (c), (b), (d).

Just occasionally both readings are possible, because words like "something" have a tendency to be used in the source text as part of proper nouns – if the author creates an item called "something your aunt gave you", or "everything you ever wanted". We allow the (a)-and-(c) words only if there is no more explicit determiner to play the role of (a), and even then if that doesn't work out then we backtrack and try again.

Note that we try not to reach (d). If we can parse the whole excerpt without need of relative clauses, we do so, thus preferring the simpler reading.

```
meaning_list *SP_description(int w1, int w2, int allow_nounless) {
    int cw1 = -1, cw2 = -1, original_w1 = w1, original_w2 = w2, rewind_point;
    quantifier *quantifier_used = NULL; int which_N = -1, determiner_w1 = -1, determiner_w2 = -1;
    world_object *some_kind = NULL;
    meaning_list *ml, *final_ml;
    if ((w1<0) || (w2<w1)) return NULL;
    if (word_range_ends_with_calling(w1, w2, &w1, &w2, &cw1, &cw2))
        LOGIF(S_GRAMMAR, "SP_description detects calling <$W>\n", cw1, cw2);
    ⟨Advance past any initial quantifier 11⟩;
    ⟨Advance past any single article 12⟩;
    rewind_point = w1;
    if (quantifier_used == NULL) ⟨Advance past any composite noun-and-quantifier 13⟩;
    ⟨Try to match the remaining text with qualifier and/or noun 14⟩;
    quantifier_used = NULL; cw1 = -1; cw2 = -1;
    if (some_kind) {
```

```
        w1 = rewind_point; some_kind = NULL;
        ⟨Try to match the remaining text with qualifier and/or noun 14⟩;
    }
    ⟨Try to match the original text as having a relative clause 18⟩;
    return NULL;
    Accept:
    final_ml = new_ml_with_words(DC_PRODUCTION, original_w1, original_w2);
    ⟨Post-process the meaning list tree for the physical description 19⟩;
    LOGIF(S_GRAMMAR, "SP_description on <$W>, %s nounless: $m\n",
        original_w1, original_w2, allow_nounless?"allowing":"forbidding", final_ml);
    return final_ml;
}
```

The function SP_description is called from 5/tandv and 5/varc.

§**11.** See "Determiners and Quantifiers.w" for how this is done. (We exclude "no one" because we want to handle that below as if it were "no-one"; we don't want to consider it as the determiner "no" plus the common noun "one", because we don't want to tell Inform that English sometimes considers "one" to be a person rather than a number.)

⟨Advance past any initial quantifier 11⟩ ≡

```
    if ((w1<w2) && (![[w1, w2 == no one ***]])) {
        int x1 = quant_parse_against_text(w1, w2, &which_N, &quantifier_used);
        if (x1 >= 0) {
            determiner_w1 = w1; determiner_w2 = x1-1; w1 = x1;
        } else quantifier_used = NULL;
    }
```

This code is used in §10.

§**12.** Note that an article can follow a determiner, as in "six of the people", where "six of" is a determiner. At this point we don't need to notice whether the article is definite or not, and we're similarly turning a blind eye to singular vs plural, so we just skip over it.

⟨Advance past any single article 12⟩ ≡

```
    if ((w1<w2) && (vocab_test_flags(w1, ARTICLE_MC))) w1++;
```

This code is used in §10.

§**13.** Words like "something" or "everywhere" combine a common noun – thing, and implicitly room – with a determiner – one thing, all rooms. When we detect them, we set both `quantifier_used` and `some_kind` appropriately. None can be recognised if the basic kinds are not created yet, which we check for by inspecting `kind_thing`. (Note that the S-parser may indeed be asked to parse "something" before this point, as when it scans the domains of adjective definitions, but that it's okay that it produces a null result.)

With the "some-" words, no quantifier is set because the meaning here is the `exists_quantifier`. Since this is the default behaviour for unquantified descriptions anyway – "a door is in the Great Hall" means that such a door exists – we needn't set the variable.

"Nothing" is conspicuously absent from the possibilities below. It gets special treatment elsewhere since it can also double as a value (the "not an object" pseudo-value).

⟨Advance past any composite noun-and-quantifier 13⟩ ≡

```
if (kind_thing) {
    determiner_w1 = w1; determiner_w2 = w1;
    if [[word w1 == something/anything]] {
        w1++; some_kind = kind_thing;
    } else if [[word w1 == somewhere/anywhere]] {
        w1++; some_kind = kind_room;
    } else if [[word w1 == someone/anyone/somebody/anybody]] {
        w1++; some_kind = kind_person;
    } else if [[word w1 == everything]] {
        w1++; quantifier_used = for_all_quantifier; some_kind = kind_thing;
    } else if [[word w1 == everywhere]] {
        w1++; quantifier_used = for_all_quantifier; some_kind = kind_room;
    } else if [[word w1 == everyone/everybody]] {
        w1++; quantifier_used = for_all_quantifier; some_kind = kind_person;
    } else if [[word w1 == nowhere]] {
        w1++; quantifier_used = not_exists_quantifier; some_kind = kind_room;
    } else if [[word w1 == nobody]] {
        w1++; quantifier_used = not_exists_quantifier; some_kind = kind_person;
    } else if [[word w1 == no_one]] {
        w1++; quantifier_used = not_exists_quantifier; some_kind = kind_person;
    } else if [[w1, w2 == no one ***]] {
        w1 += 2; determiner_w2++;
        quantifier_used = not_exists_quantifier; some_kind = kind_person;
    }
    if (some_kind) {
        if (w1 > w2) invoke_1(DC_NOUN_PRODUCTION, ml_from_wo(some_kind));
    } else {
        determiner_w1 = -1; determiner_w2 = -1;
    }
}
```

This code is used in §10.

§**14.** It is especially important not to spend time in this routine when what we have is not a description after all, so filtering with the following bitmaps makes a huge time saving:

```
define QUALIFIED_NOUN_BITMAP (WORLD_OBJECT_MC+DESIGNED_TYPE_MC+BUILTINTYPE_MC)
define DESCRIPTION_BITMAP ((NOT_MC+ARTICLE_MC+ADJECTIVE_MC+INORON_MC) | QUALIFIED_NOUN_BITMAP)
```

⟨Try to match the remaining text with qualifier and/or noun 14⟩ ≡

```
int i;
int only_qualifier_or_noun_words_left = TRUE;
for (i=w1; i<=w2; i++)
    if ((vocab_test_flags(i, DESCRIPTION_BITMAP)) == 0)
        only_qualifier_or_noun_words_left = FALSE;
if (only_qualifier_or_noun_words_left) {
    int first_possible_split, last_possible_split;
    int hopeless = FALSE;
    ⟨Find first and last possible splits between qualifier and noun part 15⟩;
    if (hopeless == FALSE)
        ⟨Try all possible splits within this range 16⟩;
}
```

This code is used in §10.

§**15.** The "split" is the word number at which the noun part begins. For instance, if "open transparent containers" is the text of words 10 to 12, then there are four possible split positions: 10 (entirely noun), 11 (open plus two words of noun), 12 (open transparent plus one word of noun) and 13 (entirely adjectives). But using that the adjective words have to match at least part of the `AL_BITMAP`, whereas the noun words have to have the `WORLD_OBJECT_MC` bit (indicating that they occur in excerpt meanings associated with objects or kinds), we can cut down the possible splits considerably – indeed usually to a single possibility, or even none at all, whereupon the `hopeless` flag is set. This is very well worthwhile, because the test at each split is fairly slow.

⟨Find first and last possible splits between qualifier and noun part 15⟩ ≡

```
int i, adjectival_start = -1, adjectival_end = -1, noun_start = -1, noun_end = -1;
for (i=w1; i<=w2; i++)
    if (vocab_test_flags(i, AL_BITMAP)) {
        if (adjectival_start<0) adjectival_start = i;
        adjectival_end = i;
    } else break;
if (some_kind == NULL)
    for (i=w2; i>=w1; i--) {
        if (vocab_test_flags(i, QUALIFIED_NOUN_BITMAP)) {
            if (noun_end<0) noun_end = i;
            noun_start = i;
        } else break;
    }
if (noun_start<0) {
    first_possible_split = w2+1;                          no noun words: purely adjectival
} else {
    first_possible_split = noun_start;                    as we can't split before the first noun
}
if (adjectival_start<0) {
    if (first_possible_split != w1) hopeless = TRUE;      begins with neither sort of word
    last_possible_split = w1;                             no adjectival words: purely noun
} else {
```

```
        if (adjectival_end+1 < first_possible_split) hopeless = TRUE;          gap in the middle
        last_possible_split = adjectival_end+1;                          after the adjectives, that is
    }
```

This code is used in §14.

§**16.** In cases of ambiguity, the earliest split wins: that is, the one maximising the length of the noun. This means that if the source text actually names a dark room, the text "dark room" will not be confused with "dark (i.e., the property) room (i.e, the kind)", since that splits later.

⟨Try all possible splits within this range 16⟩ ≡

```
    int split;
    for (split = first_possible_split; split <= last_possible_split; split++)
        ⟨Try splitting here 17⟩;
```

This code is used in §14.

§**17.** At each split, we test the noun first, because it is a quicker test and also the one less likely to succeed. (Since most adjectives are single-word, and the words up to the split all have `AL_BITMAP` bit(s) set, it's very likely that those words will successfully parse as an adjective list.)

⟨Try splitting here 17⟩ ≡

```
    meaning_list *ml2 = NULL, *ml3 = NULL;
    kind_of_value *restrict_to = NULL;
    if (some_kind) ml2 = ml_from_wo(some_kind);
    if (split <= w2) {
        ml2 = SP_excerpt(WORLD_OBJECT_MC, split, w2);
        if ((ml2 == NULL) && (![[split, w2 == object/objects]]))
            ml2 = SP_KOV_name(split, w2, QUALIFIED_NOUN_BITMAP, &restrict_to, TRUE);
        if (ml2 == NULL) continue;
    }
    if (split > w1) {
        ml3 = parse_adjective_list(w1, split-1, restrict_to, (ml2)?FALSE:TRUE);
        if (ml3 == NULL) continue;
    }
    if (ml2 == NULL) {
        if (allow_nounless == FALSE) continue;
        invoke_1(DC_ADJS_PRODUCTION, ml3);
    }
    if (ml3 == NULL) invoke_1(DC_NOUN_PRODUCTION, ml2);
    invoke_2(DC_ADJSNOUN_PRODUCTION, ml3, ml2);
```

This code is used in §16.

§**18.** To reach this possibility is nearly to admit defeat. `SP_description` has failed to match, and the only salvation would be if the excerpt contains a relative clause – "female people in the Atrium". We only bother to check this if one of the words in the excerpt is flagged as a possible verb or preposition; a verb is a possibility because of excerpts like "animals which carry things", where the word "carry" is the trigger.

But the actual parsing is handled by `SP_sentence` alone. Any work we have done up to now is thrown away, even if some part of it might have been successful. Should `SP_sentence` go well, of course, it's quite likely to call us again on sub-excerpts of the original.

⟨Try to match the original text as having a relative clause 18⟩ ≡

```
int i, might_contain_relative_clause = FALSE;
for (i=original_w1+1; i<original_w2; i++)
    if ((vocab_test_flags(i, (PREPOSITION_MC+CVERB_MC))) != 0)
        might_contain_relative_clause = TRUE;
if (might_contain_relative_clause)
    try_production(SP_sentence(original_w1, original_w2, SN_PRODUCTION));
```

This code is used in §10.

§**19.** If we reach this point, it's good news: success. We add extra nodes to the meaning list tree, though, to record any "calling" name and any determiner used.

⟨Post-process the meaning list tree for the physical description 19⟩ ≡

```
meaning_list *attach_point;
ml_set_down(final_ml, ml);
attach_point = ml;
if (cw1 >= 0) {
    if [[cw1, cw2 == a/an/the/some ...]] cw1++;
    ml_set_right(attach_point, new_ml_with_words(CALLED_PRODUCTION, cw1, cw2));
    attach_point = ml_right(attach_point);
}
if (quantifier_used)
    ml_set_right(attach_point,
        make_determiner_ml(quantifier_used, which_N, determiner_w1, determiner_w2));
```

This code is used in §10.

§**20. KOV names.** These are all converted into meaning lists consisting of a single leaf node, so for instance "object" becomes the meaning list:

```
TYPE_PRODUCTION => (G)VALUE_FMY/CONSTANT_SPC(<null>)-OBJECT_TY-KOV
```

The notation on the right denotes a generic constant value of KOV "object". Singular and plural are equivalent here, so "objects" would make an identical meaning list.

Names of KOVs come from three sources:

(a) The "built-in KOVs", that is, those defined in the "Types.i6t" template file, have single word names. They all represent base KOVs which can be indexed by an ID number, and for speed this ID number is attached to the vocabulary entry for the single word – thus avoiding the need to call the excerpt parser.

(b) The "designed KOVs", that is, those created by the source text. These have multi-word names and are registered with the excerpt parser but are still in the end base KOVs.

(c) Non-base KOVs made using KOV constructors, such as "list of numbers" – the "list of" constructor applied to the base KOV "numbers". These are parsed by `kov_parse_long_name` rather than here.

```
meaning_list *SP_KOV_name(int w1, int w2, int conjunction, kind_of_value **set_kov,
    int data_types_only) {
    if (conjunction & DESIGNED_TYPE_MC) {
        meaning_list *ml = SP_excerpt(DESIGNED_TYPE_MC, w1, w2);
        if (ml) {
            single_integer *sint = RETRIEVE_POINTER_single_integer(em_data(ml_meaning(ml)));
            int type_ID = sint_get(sint);
            if (type_ID > 1) {
                if (set_kov) *set_kov = kova(type_ID);
                return make_type_ml_from_ID(type_ID);
            }
        }
    }
    if (w1 == w2) {
        if (vocab_test_flags(w1, BUILTINTYPE_MC)) {
            int type_ID = vocab_get_literal_number_value(lw_array[w1].lw_identity);
            if ((type_ID > 1) &&
                ((data_types_only == FALSE) || (type_ID_is_data_type(type_ID)))) {
                if (set_kov) *set_kov = kova(type_ID);
                return make_type_ml_from_ID(type_ID);
            }
        }
    } else {
        kind_of_value *kov = kov_parse_long_name(w1, w2);
        if (kov) {
            if (set_kov) *set_kov = kov;
            return make_type_ml_from_kov(kov);
        }
        if (data_types_only == FALSE) {
            int x1 = w1, x2 = w2, c = parse_constructor_name(&x1, &x2);
            if (c != UNKNOWN) {
                if (SP_KOV_name(x1, x2,
                    DESIGNED_TYPE_MC+BUILTINTYPE_MC, &kov, data_types_only)) {
                    if (kov) {
                        if (set_kov) *set_kov = kovcon(c, kov);
                        return make_type_ml_from_kov(kovcon(c, kov));
                    }
```

```
                }
                world_object *k = parse_world_object(x1, x2, TRUE);
                if (k) {
                    if (set_kov) *set_kov = kovcon(c, kovko(k));
                    return make_type_ml_from_kov(kovcon(c, kovko(k)));
                }
            }
        }
    }
    return NULL;
}
```

The function SP_KOV_name is called from 5/tandv.

Type Expressions and Values 5/tandv

Purpose

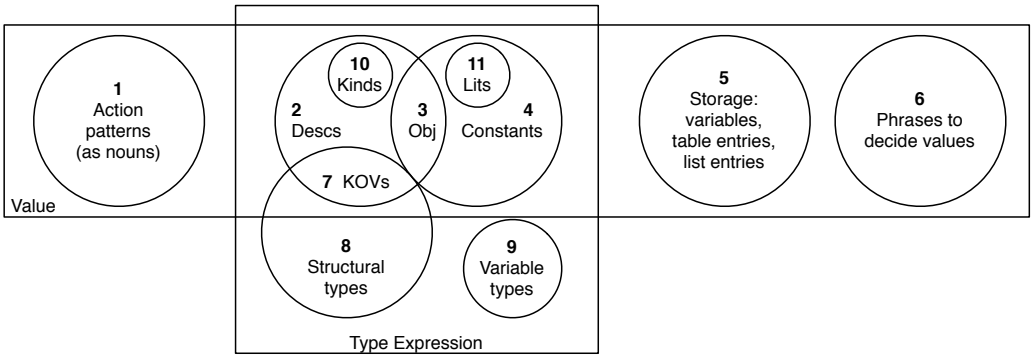
To parse two forms of noun: a noun phrase in a sentence, and a description of what text can be written in a given situation.

5/tandv.§1-2 Type expressions; §3-19 Values; §20-22 Delegated constructs; §23-28 Table references

---

Definitions

¶1. Inform recognises many noun-like constructions, some of which – out of a noun context – look like adjectives, actions or other excerpts which aren't at all evidently nouns. These many ways to describe nouns are gathered up into two central constructions. A "type expression" specifies what sort of excerpt should appear in a given place, whereas a "value" means anything which can be a noun phrase for a verb. There is considerable overlap between the two, but they are not the same, as the following Venn diagram shows:



The following example sentences have the relevant phrases in bold.

[1] if the idea of the gizmo is **taking the fish**, ...

[2] if there are **three women** in the Nunnery, ...

[3a: as description] Before taking **the harmonium**, ...

[3b: as constant] let X be **the harmonium**;

[4] now Y is **the can't reach inside rooms rule**;

[5] now Z is **the time of day**;

[6] let N be **the number of entries in L**;

[7] Understand "turn to [**number**]" as combination-setting.

[8] To repeat until (C - **condition**): ...

[9] The Zeppelin countdown is a **number that varies**.

[10] The little red car is a **vehicle**.

[11] The weight of the Space Shuttle is **68585 kg**.

¶2. A rough measure of complexity is how many times the S-parser has to try parsing a noun phrase, and we cut off (really as a defensive measure against bugs) if this exceeds a threshold:

```
define S_PARSER_COMPLEXITY_THRESHOLD 1000000

int no_calls_to_SP_noun_phrase = 0;
```

---

§**1. Type expressions.** This is a concept which does not exist for conventional programming languages, which would see it as a sort of half-way position between "value" and "type". A "type expression" is a possible specification for a parameter in a phrase definition, though it has other uses elsewhere. That certainly includes cases which traditional programming languages would call types, so

To adjust (X - closed door) by (N - number): ...

includes two type expressions, "closed door" and "number". Similarly "text", "list of rules" and so forth, and the special case of "value", the broadest possible type.

But a type expression can also be a constant, which C (for instance) would consider a value and not a type at all:

To adjust (X - closed door) by (N - 11): ...

gives a definition to be used only where the second parameter has the value 11. In this way any constant value is regarded as being a type – the narrow type representing only its own value.

Note that a list of adjectives with no noun does not always qualify as a type expression, according to the routine as written below. It looks as if it never should, on the face of it – "opaque" does not make clear what kind of object is to be opaque – but in practice English is a little fuzzy on the difference between nouns and adjectives. For instance, this:

To adjust (X - scenery): ...

is allowed even though "scenery" is an adjective in Inform. Bare adjective lists are in practice allowed in all cases except when defining kinds of variables: so "scenery variable" is impossible.

```
meaning_list *SP_type_expression(int w1, int w2, int allow_nounless) {
    begin_SP_routine_skipping_article(w1, w2);
    if (allow_nounless) try_production(SP_adjective_list(w1, w2));
    if (w2>w1) ⟨Try type expressions representing categories of variables 2⟩;
    try_production(SP_KOV_name(w1, w2, conjunction, NULL, FALSE));
    try_production(SP_description(w1, w2, FALSE));
    try_production(SP_constant_value(w1, w2));
    on_success_invoke(TE_PRODUCTION);
}
```

The function SP_type_expression is called from 7/ttts.

§**2.** Another form of type expression is exemplified by "number variable" (meaning: any name of a number variable can go here), and so on. Note that these forms recurse by calling `SP_type_expression` again, so that *syntactically* we allow "T that varies" for any type expression T. This would include contradictions in terms such as "15 that varies", but we want to allow the parse here so that a problem message can be issued higher up in Inform.

⟨Try type expressions representing categories of variables 2⟩ ≡

```
int v1 = -1, v2 = -1;
if ([[w1, w2 == ... variable/variables --> v1, v2]] ||
    [[w1, w2 == ... that/which vary/varies --> v1, v2]]) {
    if [[v1, v2 == existing]] invoke_0(TE_EX_VAR_PRODUCTION);
    if ([[v1, v2 == existing ...]] && (ml2 = SP_type_expression(v1+1, v2, FALSE)))
        invoke_1(TE_EX_VAR_PRODUCTION, ml2);
    if [[v1, v2 == nonexisting]] invoke_0(TE_NEW_VAR_PRODUCTION);
    if ([[v1, v2 == nonexisting ...]] && (ml2 = SP_type_expression(v1+1, v2, FALSE)))
        invoke_1(TE_NEW_VAR_PRODUCTION, ml2);
    if [[v1, v2 == global]] invoke_0(TE_GL_VAR_PRODUCTION);
    if ([[v1, v2 == global ...]] && (ml2 = SP_type_expression(v1+1, v2, FALSE)))
        invoke_1(TE_VAR_PRODUCTION, ml2);
    if ((ml2 = SP_type_expression(v1, v2, FALSE))) invoke_1(TE_VAR_PRODUCTION, ml2);
}
```

This code is used in §1.

§**3. Values.** As noted above, `SP_value` is used to parse a noun phrase arising in a typical phrase, or on either side of an assertion sentence. The boldface terms here are all parsed as values:

The cat is in **the bag**. The **time of day** is **11:10 AM**.

award **six** points;

if **more than three animals** are in **the kennel**, ...

```
kind_of_value *probable_noun_phrase_context = NULL;
meaning_list *SP_value(int w1, int w2) {
    begin_SP_routine(w1, w2);
    LOGIF(S_GRAMMAR, "PV on <$W> in context $u\n", w1, w2, probable_noun_phrase_context);
    if (no_calls_to_SP_noun_phrase >= S_PARSER_COMPLEXITY_THRESHOLD) return NULL;
    if (paired_brackets(w1, w2)) return SP_value(w1+1, w2-1);
    ⟨Value, priority 1: a non-global variable name 4⟩;
    ⟨Value, priority 2: a constant value 5⟩;
    ⟨Value, priority 3: a global variable name 8⟩;
    ⟨Value, priority 4: a property name used as noun 9⟩;
    ⟨Value, priority 5: a table column name, but only if we are expecting it 10⟩;
    ⟨Value, priority 6: an action pattern 11⟩;
    ⟨Value, priority 8: a value computed by a phrase at run-time 13⟩;
    ⟨Value, priority 7: a description which doesn't involve kinds of value 12⟩;
    ⟨Value, priority 9: a reference to entry or entries in a table 14⟩;
    ⟨Value, priority 10: "member of" used explicitly to make a description 15⟩;
    ⟨Value, priority 11: a value property 16⟩;
    ⟨Value, priority 12: a list entry 17⟩;
    ⟨Value, priority 13: a description which does involve kinds of value 18⟩;
    ⟨Value, priority 14: a table column name when we aren't expecting it 19⟩;
    LOGIF(S_GRAMMAR, "PV on <$W> failed\n", w1, w2);
    on_success_invoke(VAL_PRODUCTION);
}
```

The function SP_value is called from 5/varc, 5/candp and 7/ttts.

§**4.** Because such a wide range of text can pass `SP_value`, the order of matching is very important. Local and not-local-but-not-global-either variables come first, so that they can have names overlapping with globally-defined names of other constructs – this avoids namespace clashes, especially when using extensions.

⟨Value, priority 1: a non-global variable name 4⟩ ≡

```
    try_production(SP_local_variable(w1, w2));
    try_production(SP_stacked_variable(w1, w2));
```

This code is used in §3.

§**5.** Constants come next; in case of doubt, if text specified a constant then that ought to be the interpretation. Thus in the sentence:

let X be red;

where "red" is a value of "colour" which is a property of things, we read "red" as a noun – the actual hue – and as a result create "X" as a temporary colour variable; rather than reading "red" as an adjective – the behaviour of having the colour red – and making "X" a description. (We will sometimes be wrong about this: for instance, in "now the canvas is red", it is indeed the adjectival use that's meant. But this is dealt with when simplifying the resultant proposition, later on.)

⟨Value, priority 2: a constant value 5⟩ ≡

```
try_production(SP_constant_value(w1, w2));
⟨Value, priority 2a: an equation written with a "where" clause 6⟩;
⟨Value, priority 2b: an equation written inline 7⟩;
```

This code is used in §3.

§**6.** Equations are normally referred to by name, but they sometimes have "where..." caveats added:

⟨Value, priority 2a: an equation written with a "where" clause 6⟩ ≡

```
int i, eq1 = w1, eq2 = w2, wh1 = -1, wh2 = -1;
if ([[eq1, eq2 == ... where ... : i --> eq1, eq2 ... wh1, wh2]] &&
    ((ml2 = SP_value(eq1, eq2)) != NULL)) {
    switch (ml_production(ml_down(ml2))) {
        case EQUATION_INLINE_PRODUCTION:
        case EQUATION_MC:
            invoke_2(EQUATION_WHERE_PRODUCTION, ml_down(ml2), SP_plain_text(wh1, wh2));
            break;
    }
}
```

This code is used in §5.

§**7.** They can also be written "inline", that is, explicitly; we can tell this by the presence of an equals sign, and the run-up text "...be given by". (We have to be pretty cautious here.)

⟨Value, priority 2b: an equation written inline 7⟩ ≡

```
int pre1 = w1-3, pre2 = w1-1;
if ((pre1 >= 0) && [[pre1, pre2 == be given by]]) {
    int i, j;
    for (i=w1; i<=w2; i++) {
        char *p = lw_array[i].lw_rawtext;
        for (j=0; p[j]; j++)
            if (p[j] == '=')
                invoke_1(EQUATION_INLINE_PRODUCTION, SP_plain_text(w1, w2));
    }
}
```

This code is used in §5.

§**8.** Now global variables. In theory, a registered `QUANTITY_MC` excerpt might be the name of an enumerated constant or a stacked variable, but both of those have been parsed already, so if we reach this then it must be a global.

⟨Value, priority 3: a global variable name 8⟩ ≡
```
if (disjunction & QUANTITY_MC) {
    try_production(SP_excerpt(QUANTITY_MC, w1, w2));
    if [[w1, w2 == the ...]]
        try_production(SP_excerpt(QUANTITY_MC, w1+1, w2));
}
```

This code is used in §3.

§**9.** Property names are only rarely used as nouns, but here is an example:

if the bartered item provides colour, ...

where "colour" is a property name used as a noun. This example is easy to parse, but others are more ambiguous:

if the bartered item provides open, ...

Here "open" is a property name, but it could also be parsed (by `SP_description`) as a description of all items which are open. This is why property names have a higher priority here than descriptions do.

⟨Value, priority 4: a property name used as noun 9⟩ ≡
```
if (disjunction & PROPERTY_MC) {
    try_production(SP_excerpt(PROPERTY_MC, w1, w2));
    if [[w1, w2 == the ...]]
        try_production(SP_excerpt(PROPERTY_MC, w1+1, w2));
}
```

This code is used in §3.

§**10.** Because of Inform's feature allowing objects or enumerated kinds of value to be created by table, it's often the case that column 1 in a table has a name which clashes with the name of a kind of object or value. For instance, in the example "Tour des Maillots", we have:

A jersey is a kind of thing. Some jerseys are defined by the Table of Honorary Jerseys.

and column 1 in the Table is called "jersey". Now in fact column names are also registered with the word "column" suffixed, so that the writer only has to spell out "jersey column" to be unambiguous that he means the column, not the kind. But we don't want to insist on this in cases where the context clearly indicates that we're talking about the table, as here:

sort the Table of Honorary Jerseys in jersey order;

So in cases like this we allow "jersey" to be read as the column name, and in all other cases we fall through into lower-priority description parsing below, so that "jersey" is parsed as a description of the kind of object.

⟨Value, priority 5: a table column name, but only if we are expecting it 10⟩ ≡
```
if ((is_kova(probable_noun_phrase_context, TABLE_COLUMN_TY))
    && (disjunction & TABLE_COLUMN_MC)) {
    try_production(SP_excerpt(TABLE_COLUMN_MC, w1, w2));
    if [[w1, w2 == the ...]]
        try_production(SP_excerpt(TABLE_COLUMN_MC, w1+1, w2));
}
```

This code is used in §3.

§**11.** In practice an action pattern used as a noun will probably need to be specific ("taking the duck") rather than descriptive ("taking an animal") but we don't make syntactic distinctions and allow both here.

⟨Value, priority 6: an action pattern 11⟩ ≡
```
try_production(SP_action_pattern(w1, w2));
```
This code is used in §3.

§**12.** We finally allow descriptions to be matched, but only where they consist of lists of adjectives ("closed fixed in place"), objects ("open Weiss Safe") or kinds ("open container in the Dining Room").

⟨Value, priority 7: a description which doesn't involve kinds of value 12⟩ ≡
```
try_production(SP_adjective_list(w1, w2));
try_production(SP_purely_physical_description(w1, w2));
```
This code is used in §3.

§**13.** Now text like "number of rows in the Table of Ranks" which specifies a value that can only be determined during play, using a "To decide which..." phrase.

It's a little surprising, perhaps, that this comes only around half-way down the priority listing.

It isn't higher up the list because, if it were, a definition such as:

To decide which object is the open door: ...

would effectively redefine a description like "open door", undermining the language; similarly, we don't want a phrase to override a variable name, or worse still a literal –

To decide which number is 11: decide on 22.

But it isn't lower down the list because a phrase when invoked tends to contain secondary clauses, and it's important for those clauses to take priority over the other secondary clauses which appear lower down. For instance:

change the hunger of Ogg to **the hunger of Ogg plus 1**;

Because phrases have higher priority than property values, we read the new value as "(the hunger of (Ogg)) plus (1)" rather than "the hunger of (Ogg plus 1)".

⟨Value, priority 8: a value computed by a phrase at run-time 13⟩ ≡
```
if ((ml2 = SP_excerpt(VALUE_PHRASE_MC, w1, w2)) != NULL) {
    invoke_1(VALUE_PHRASE_PRODUCTION, ml2);
}
if [[w1, w2 == the ...]]
    if ((ml2 = SP_excerpt(VALUE_PHRASE_MC, w1+1, w2)) != NULL)
        invoke_1(VALUE_PHRASE_PRODUCTION, ml2);
```
This code is used in §3.

§**14.** Because `SP_table_reference` is a little slow, we try not to call it for excerpts which obviously can't pass.

⟨Value, priority 9: a reference to entry or entries in a table 14⟩ ≡
```
int try_tr = 0;
for (i=w1; i<=w2; i++) {
    if [[word i == table/entry/listed/row]] try_tr += 2;
    if [[word i == in/of]] try_tr += 1;
}
if (try_tr >= 2) try_production(SP_table_reference(w1, w2));
```
This code is used in §3.

§**15.** The "member of D" construction, where D is a description, can be useful to resolve ambiguities, but also allows us to use a description known only at run-time, since it matches some subset of objects using a routine whose address is stored in a local variable. This allows "To..." phrases to have descriptions as arguments.

⟨Value, priority 10: "member of" used explicitly to make a description 15⟩ ≡

```
int d1, d2;
if [[w1, w2 == member/members of ... --> d1, d2]] {
    if ((ml2 = SP_description(d1, d2, FALSE)) != NULL) {
        ml = new_ml(MEMBER_OF_PRODUCTION);
        ml_set_down(ml, ml2);
        LOGIF(S_GRAMMAR, "PV found member-of + description <$W>\n", d1, d2);
        goto Accept;
    }
    if ((ml2 = SP_local_variable(d1, d2)) != NULL) {
        ml = new_ml(MEMBER_OF_PRODUCTION);
        ml_set_down(ml, new_ml(VAL_PRODUCTION));
        ml_set_down(ml_down(ml), ml2);
        LOGIF(S_GRAMMAR, "PV found member-of + local <$W>\n", d1, d2);
        goto Accept;
    }
}
```

This code is used in §3.

§**16.** For instance, "the carrying capacity of the box", or – providing we are in a context where "it" meaningfully describes an object – "its colour".

Note that the property name is allowed to be unrecognised if we are parsing early in Inform's run. (It's by looking at clauses like this that Inform finds out what the property names are going to be.)

⟨Value, priority 11: a value property 16⟩ ≡

```
for (i=w1+1; i<=w2-1; i++)
    if [[word i == of]] {
        int j = i+1;
        [[j, w2 == the ... --> j, w2]];
        if ((ml3 = SP_value(j, w2)) != NULL) {
            ml2 = SP_excerpt(PROPERTY_MC, w1, i-1);
            if ((ml2 == NULL) && (model_world_constructed == FALSE))
                ml2 = SP_plain_text(w1, i-1);
            if (ml2) invoke_2(VAL_PROP_OF_PRODUCTION, ml2, ml3);
        }
    }
if ((enable_its >= 0) && [[w1, w2 == its/his/her/their ...]]) {
    ml2 = SP_excerpt(PROPERTY_MC, w1+1, w2);
    if (ml2) {
        specification *spec = new_LOCAL_VARIABLE_spec(-1, -1, enable_its);
        ml3 = new_ml(VAL_PRODUCTION);
        ml_set_down(ml3, new_ml(LOCAL_PRODUCTION));
        ml_attach_spec(ml_down(ml3), spec);
        invoke_2(VAL_PROP_OF_PRODUCTION, ml2, ml3);
    }
}
```

This code is used in §3.

§**17.** This is an explicit entry in a list – for instance, "entry 12 in L". Because of the tell-tale word "entry" it is much less tricky to handle.

⟨Value, priority 12: a list entry 17⟩ ≡
```
    if [[word w1 == entry]] {
        for (i=w1+1; i<=w2-1; i++)
            if [[word i == of/in/from]] {
                int j = i+1;
                [[j, w2 == the ... --> j, w2]];
                    if ((ml3 = SP_value(w1+1, i-1)) && (ml2 = SP_value(j, w2)))
                        invoke_2(VAL_LIST_ENTRY_PRODUCTION, ml2, ml3);
            }
    }
```
This code is used in §3.

§**18.** All other descriptions have gone by now; we are left with those, like "even number", which involve kinds of value rather than kinds of object.

⟨Value, priority 13: a description which does involve kinds of value 18⟩ ≡
```
    try_production(SP_description(w1, w2, FALSE));
```
This code is used in §3.

§**19.** And now that names of kinds and kinds of value have been taken care of, we can safely match anything registered as a table column name; it will be unambiguous at this point.

⟨Value, priority 14: a table column name when we aren't expecting it 19⟩ ≡
```
    if (disjunction & TABLE_COLUMN_MC) {
        try_production(SP_excerpt(TABLE_COLUMN_MC, w1, w2));
        if [[w1, w2 == the ...]]
            try_production(SP_excerpt(TABLE_COLUMN_MC, w1+1, w2));
    }
```
This code is used in §3.

§**20. Delegated constructs.** The S-parser doesn't parse every word of the text fed to it: some constructions are delegated to specific code in other chapters. To begin with, local variable names are not registered as excerpts, and are parsed from data on the current stack frame:

```
meaning_list *SP_local_variable(int w1, int w2) {
    int lvn = parse_name_local_to_current_stack_frame(w1, w2);
    if (lvn >= 0) {
        meaning_list *ml;
        specification *spec = new_LOCAL_VARIABLE_spec(-1, -1, lvn);
        ml = new_ml(LOCAL_PRODUCTION);
        ml_attach_spec(ml, spec);
        return ml;
    }
    return NULL;
}
```

§**21.** Stacked variables, which also lack global scope, are similarly context-dependent:

```
meaning_list *SP_stacked_variable(int w1, int w2) {
    ph_stack_frame *phsf = current_stack_frame();
    stacked_variable *stv;
    if (phsf == NULL) return NULL;
    stv = stvol_parse(phsf_get_stvol(phsf), w1, w2);
    if (stv) {
        meaning_list *ml = new_ml(STV_PRODUCTION);
        specification *spec = new_QUANTITY_spec(stv_get_quantity(stv));
        ml_attach_spec(ml, spec);
        return ml;
    }
    return NULL;
}
```

§**22.** And action patterns, such as "taking a container" or "opening a closed door", are parsed by code in the chapter on Actions:

```
meaning_list *SP_action_pattern(int w1, int w2) {
    action_pattern ap = parse_action_pattern(w1, w2, IS_TENSE);
    if (ap_is_valid(&ap)) {
        meaning_list *ml;
        specification *spec = new_TEST_ACTION_spec(store_ap(ap));
        ml = new_ml(AP_PRODUCTION);
        ml_attach_spec(ml, spec);
        return ml;
    }
    return NULL;
}
```

§**23. Table references.** The next parsing routine looks (and is) a little slow, but see `SP_value` above for why it is called only when success is at least plausible.

Table references come in five different forms:

```
meaning_list *SP_table_reference(int w1, int w2) {
    begin_SP_routine(w1, w2);
    LOGIF(S_GRAMMAR, "Possible table reference $W\n", w1, w2);
    ⟨Try an abbreviated table reference, naming only the column 24⟩;
    for (i=w1+1; i<=w2-2; i++) {
        ⟨Try an explicit identification of a cell in the table 25⟩;
        ⟨Try a conditional search of a column in the table 26⟩;
        ⟨Try an associative identification of a cell in the table 27⟩;
        ⟨Try an existential identification of a cell in the table 28⟩;
    }
    on_success_invoke(TR_PRODUCTION);
}
```

§**24.** For instance, "atomic number entry", meaning the entry in that column and implicitly in the table and row currently selected.

⟨Try an abbreviated table reference, naming only the column 24⟩ ≡
```
if ([[w1, w2 == ... entry]] &&
    ((ml2 = SP_excerpt(TABLE_COLUMN_MC, w1, w2-1)) != NULL))
    invoke_1(TR_ENTRY_PRODUCTION, ml2);
```

This code is used in §23.

§**25.** For instance, "atomic number in row 4 of the Table of Elements".

⟨Try an explicit identification of a cell in the table 25⟩ ≡
```
if ([[i, w2 == in row ...]] &&
    ((ml2 = SP_excerpt(TABLE_COLUMN_MC, w1, i-1)) != NULL))
    for (j=i+3; j<w2; j++)
        if ([[word j == of]] &&
            (ml3 = SP_value(i+2, j-1)) && (ml4 = SP_value(j+1, w2)))
            invoke_3(TR_IN_ROW_PRODUCTION, ml2, ml3, ml4);
```

This code is used in §23.

§**26.** For instance, "an atomic number listed in the Table of Elements" in the sentence "if 101 is an atomic number listed in the Table of Elements". (This is part of a condition, and can't evaluate.)

⟨Try a conditional search of a column in the table 26⟩ ≡
```
if ([[i, w2 == listed in ...]] &&
    ((ml2 = SP_excerpt(TABLE_COLUMN_MC, w1, i-1)) != NULL) &&
    ((ml3 = SP_value(i+2, w2)) != NULL))
    invoke_2(TR_LISTED_IN_PRODUCTION, ml2, ml3);
```

This code is used in §23.

§**27.** For instance, "atomic weight corresponding to an atomic number of 57 in the Table of Elements".

⟨Try an associative identification of a cell in the table 27⟩ ≡
```
if ([[i, w2 == corresponding to ...]] &&
    ((ml2 = SP_excerpt(TABLE_COLUMN_MC, w1, i-1)) != NULL))
    for (j=i+3; j<w2; j++)
        if ([[word j == of]] &&
            (ml3 = SP_excerpt(TABLE_COLUMN_MC, i+2, j-1)))
            for (k=j+2; k<w2; k++)
                if ([[word k == in]] &&
                    ((ml4 = SP_value(j+1, k-1)) != NULL) &&
                    ((ml5 = SP_value(k+1, w2)) != NULL))
                    invoke_4(TR_CORR_PRODUCTION, ml2, ml3, ml4, ml5);
```

This code is used in §23.

§**28.** For instance, “atomic weight of 20 in the Table of Elements” in the sentence “if there is an atomic weight of 20 in the Table of Elements”. Again, this is part of a condition, and can't evaluate.

⟨Try an existential identification of a cell in the table 28⟩ ≡

```
if ([[word i == of]] && ((ml2 = SP_excerpt(TABLE_COLUMN_MC, w1, i-1)) != NULL))
    for (j=i+2; j<w2; j++)
        if [[word j == in]] {
            ml3 = SP_value(i+1, j-1);
            ml4 = SP_value(j+1, w2);
            if (ml3 && ml4) invoke_3(TR_OF_IN_PRODUCTION, ml2, ml3, ml4);
        }
```

This code is used in §23.

Verbal and Relative Clauses 5/varc

Purpose

To break down an excerpt into NP and VP-like clauses, perhaps with a primary verb (to make a sentence), perhaps only a relative clause (to make a more complex NP).

---

§**1. The top level of the S-grammar.** English is an SVO language, where the main parts of the sentence occur in the order subject, verb, object. The following routine parses that crucial division, and note that it can be used either to form a complete sentence, where there is an active verb –

now the silver bars are in the Hall of Mists;

where the relevant excerpt parses to:

```
SV_PRODUCTION / "silver bars are in the hall of mists"
    NP_PRODUCTION / "silver bars"
        ...
    VP_PRODUCTION
        VERB_PRODUCTION => VU: are IS_TENSE -> is
            PREP_PRODUCTION => PU: in -> is-in
        NP_PRODUCTION / "the hall of mists"
            ...
```

or alternatively to make a more elaborate noun phrase, using a relative clause, where there is no active verb:

a woman who carries the silver bars;

```
SN_PRODUCTION / "woman who carries the silver bars"
    NP_PRODUCTION / "woman"
        ...
    VP_PRODUCTION
        VERB_PRODUCTION => VU: carries IS_TENSE -> carries
        NP_PRODUCTION / "the silver bars"
            ...
```

We sometimes also have to deal with English's use of "there" as a meaningless placeholder to stand for a missing noun phrase:

if there is an open door, ...

```
SV_PRODUCTION / "there is an open door"
    THERE_PRODUCTION
    VP_PRODUCTION
        VERB_PRODUCTION => VU: is IS_TENSE -> is
        NP_PRODUCTION / "an open door"
            ...
```

Note that we recognise "there" as a placeholder only in the subject position. As an object, it mainly functions in English as a pronoun referring to a specific place – "I go into the lobby. Julia is there." (This is a pronoun Inform doesn't try to understand. "There" as a subject can also function that way – "There is Julia." – but we won't try to disentangle such uses, since they don't arise in sentences with a conditional use: one would never write "if there is Julia" to mean "if Julia is in the lobby".)

§**2.** At any rate here is the SP-routine to parse such sentences. It needs to run quickly, but in principle also needs to try splitting the excerpt with the verb or preposition phrase at any intervening position. We optimise by trying to split only at words with a vocabulary bit identifying them as belonging to verb or preposition uses. Not every "of" is meant as a preposition, so there will still be plenty of false tries, but at least we can skip over "mists", "hall", and so on.

Note that we allow a preposition to be negated, as in this example:

a woman not in the Hall of Mists

(An author naming something which ends with "not" might have trouble here, but trouble is what such an author deserves.)

```
meaning_list *SP_sentence(int w1, int w2, int desired_outcome) {
    int bit_to_spot;
    begin_SP_routine(w1, w2);
    if (no_calls_to_SP_noun_phrase >= S_PARSER_COMPLEXITY_THRESHOLD) return NULL;
    LOGIF(S_GRAMMAR, "SP_sentence on <$W> seeking $p\n", w1, w2, desired_outcome);
    if ((desired_outcome != SN_PRODUCTION) && (desired_outcome != SV_PRODUCTION))
        internal_error("bad call to SP_sentence");
    if (desired_outcome == SN_PRODUCTION) bit_to_spot = PREPOSITION_MC;
    else bit_to_spot = CVERB_MC;
    for (i = w1; i<=w2; i++) if (vocab_test_flags(i, bit_to_spot)) {
        verb_usage *vu;
        int prep_only, w3 = i-1;
        if (desired_outcome == SN_PRODUCTION) prep_only = TRUE; else prep_only = FALSE;
        if [[i, w2 == which/who/that ... --> i, w2]] prep_only = FALSE;
        if (prep_only) {
            j = i; vu = regular_to_be;
            if ((i>w1) && [[word w3 == not]]) { vu = negated_to_be; w3--; }
            ⟨Attempt division into noun phrases for verb vu and position j 3⟩;
        } else {
            LOOP_OVER(vu, verb_usage) {
                j = vu_parse_text_against(i, w2, vu);
                if (j >= 0) ⟨Attempt division into noun phrases for verb vu and position j 3⟩;
            }
        }
    }
    return NULL;
    Accept: ml_set_text(ml, w1, w2);
    correct_S_subtree_for_adjectives(ml_down(ml));
    LOGIF(S_GRAMMAR, "SP_sentence on <$W> succeeds with:\n$m", w1, w2, ml);
    return ml;
}
```

The function SP_sentence is called from 5/candd and 5/candp.

§**3.** In English, the verb "to be" is considered "copular" because it acts to combine its subject and object: "X is 5", "Y is blue", and so on, refer to just one thing but make a statement about its nature or identity. Other verbs – "to carry", say – normally refer to two different things, at least in their most general forms: "X carries the briefcase".

The main complication here is "there". We must read

> there is a woman in the Dining Room

as if it read

> a woman is in the Dining Room

In other words, after "there is" or "there are", we should read a NP with a relative clause but break it up as a VP: "there is NP PU NP" must become "NP is PU NP", where PU is the prepositional usage. We do this only with the copular verb: although English occasionally uses "there" to fill the gap where a NP should be for other verbs ("there dwells in England a man who", and so on) the effect is archaic and not very useful. We don't allow "there exists" – its technical usage in mathematical logic is familiar to mathematical logicians, but the resulting text reads back strangely to anyone else, and "there is" means the same thing anyway.

In the case of "There is a button on the radio", or similar, a special placeholder leaf – `THERE_PRODUCTION` – is entered into the meaning list where the subject should be; the "object" of the sentence is required to be a description (as "a button on the radio" is).

⟨Attempt division into noun phrases for verb vu and position j 3⟩ ≡

```
int copular_flag = FALSE, there_flag = FALSE;
preposition_usage *preposition_used = NULL;
meaning_list *subject_part = NULL, *object_part = NULL;
if ([[w1, w3 == there]] &&
    (vu_get_meaning(vu) == a_is_b_predicate) &&
    (desired_outcome == SV_PRODUCTION)) there_flag = TRUE;
else subject_part = SP_noun_phrase(w1, w3, prep_only);
if (vu_get_meaning(vu) == a_is_b_predicate) {
    copular_flag = TRUE;
    ⟨Look for a preposition immediately after the copular verb 4⟩;
}
if ((prep_only == FALSE) || (preposition_used)) {
    if (object_part == NULL) object_part = SP_noun_phrase(j, w2, copular_flag);
    if (there_flag) {
        if ((object_part == NULL) ||
            (ml_production(ml_down(ml_down(object_part))) != DC_PRODUCTION))
            continue;
        subject_part = new_ml(THERE_PRODUCTION);
    }
    if ((object_part) && (subject_part))
        ⟨Construct the meaning list and invoke as a successful sentence 6⟩;
}
```

This code is used in §2.

§**4.** Some prepositions imply the player as object: "carried", in the sense of "to be carried", for instance – "The briefcase is carried". We fill the `object_part` subtree with a representation of the player-object for those.

⟨Look for a preposition immediately after the copular verb 4⟩ ≡

```
preposition_usage *pu;
LOOP_OVER(pu, preposition_usage) {
    k = pu_parse_text_against(j, w2, pu, TRUE);
    if (k >= 0) {
        if ((pu_implies_player(pu)) && (k > w2)) {
            object_part = new_ml(NP_PRODUCTION);
            ml_set_down(object_part, new_ml(VAL_PRODUCTION));
            ml_set_down(ml_down(object_part), new_ml(QUANTITY_MC));
            ml_set_meaning(ml_down(ml_down(object_part)), meaning_of_player);
        } else {
            object_part = SP_noun_phrase(k, w2, FALSE);
        }
        if (object_part) {
            preposition_used = pu;
            ⟨Allow a preposition to behave as a noun qualified by adjectives, in rare cases 5⟩;
            break;
        }
    }
}
```

This code is used in §3.

§**5.** The subject part is normally expected to include a noun: for instance,

> all of the opaque containers in the item

has as subject "all of the opaque containers" and object "the item", with "in" as the prepositional usage. But in rare cases the preposition is itself also noun-like in behaviour. Consider:

> all of the flame-retardant parts of the item

Here "parts of" is the preposition, but "parts" is also functioning as the noun governing the subject, or at least that is what the author intended. But Inform will have tried to parse "all of the flame-retardant" as subject, and returned `NULL` as a result. So we re-parse the subject, but allow it to contain a nounless adjective list this time.

This is surprisingly rarely necessary. In the whole test suite (as of July 2008) only a single sentence falls into this case.

⟨Allow a preposition to behave as a noun qualified by adjectives, in rare cases 5⟩ ≡

```
if ((pu_implicitly_behaves_as_noun(pu)) &&
    (subject_part == NULL) && (there_flag == FALSE))
    subject_part = SP_description(w1, w3, TRUE);
```

This code is used in §4.

§**6.** At this point there is no work remaining except to construct the tree: success is now certain.

⟨Construct the meaning list and invoke as a successful sentence 6⟩ ≡

```
meaning_list *VP_part = new_ml(VP_PRODUCTION);
ml_set_down(VP_part, new_ml(VERB_PRODUCTION));
ml_attach_data(ml_down(VP_part), STORE_POINTER_verb_usage(vu));
if (preposition_used) {
    ml_set_down(ml_down(VP_part), new_ml(PREP_PRODUCTION));
    ml_attach_data(ml_down(ml_down(VP_part)),
        STORE_POINTER_preposition_usage(preposition_used));
}
ml_set_right(ml_down(VP_part), object_part);
invoke_2(desired_outcome, subject_part, VP_part);
```

This code is used in §3.

§**7. Tidying up a sentence subtree.** This checks, in a paranoid sort of way, that a subtree is properly formed, and also makes one useful correction when it sees a wrong guess as to whether an adjective is meant as a noun.

```
void correct_S_subtree_for_adjectives(meaning_list *ml) {
    meaning_list *subject_phrase_subtree, *object_phrase_subtree, *verb_phrase_subtree;
    if (ml == NULL) internal_error("SV childless");
    if ((ml_right(ml) == NULL) && (ml_production(ml) != DC_PRODUCTION))
        internal_error("SV with only one child, which isn't a DC");
    subject_phrase_subtree = ml;
    verb_phrase_subtree = ml_right(ml);
    if (ml_down(verb_phrase_subtree) == NULL)
        internal_error("SV childless");
    if (ml_right(ml_down(verb_phrase_subtree)) == NULL)
        internal_error("SV only one child");
    object_phrase_subtree = ml_right(ml_down(verb_phrase_subtree));
    ⟨Modify the object from a noun to an adjective if the subject is also a noun 8⟩;
    if (ml_production(verb_phrase_subtree) != VP_PRODUCTION)
        internal_error("SV not a VP");
    if (ml_production(ml_down(verb_phrase_subtree)) != VERB_PRODUCTION)
        internal_error("child of SV not a verb");
}
```

The function correct_S_subtree_for_adjectives is called from 5/mlc.

§**8.** The following is used to correct the SV-subtree for something like "painting is orange" so that "orange" will be used not as a noun but as an adjective.

⟨Modify the object from a noun to an adjective if the subject is also a noun 8⟩ ≡

```
if ((ml_down(object_phrase_subtree)) && (ml_down(ml_down(object_phrase_subtree))) &&
    (ml_production(ml_down(ml_down(object_phrase_subtree))) == QUANTITY_MC) &&
    (ml_down(subject_phrase_subtree)) && (ml_down(ml_down(subject_phrase_subtree))) &&
    (ml_production(ml_down(ml_down(subject_phrase_subtree))) == DC_PRODUCTION)) {
    meaning_list *adjq = ml_down(ml_down(object_phrase_subtree));
    quantity *q = RETRIEVE_POINTER_quantity(em_data(ml_meaning(adjq)));
    if (qty_get_adjectival_phrase(q)) {
        ml_set_production(adjq, ADJECTIVE_MC);
        ml_set_meaning(adjq,
            em_new(ADJECTIVE_MC, 0,
                STORE_POINTER_adjectival_phrase(qty_get_adjectival_phrase(q))));
        ml_set_down(ml_down(object_phrase_subtree),
            new_ml(DC_PRODUCTION));
        ml_set_down(ml_down(ml_down(object_phrase_subtree)),
            new_ml(DC_ADJS_PRODUCTION));
        ml_set_down(ml_down(ml_down(ml_down(object_phrase_subtree))),
            new_ml(AL_PRODUCTION));
        ml_set_down(ml_down(ml_down(ml_down(ml_down(object_phrase_subtree)))), adjq);
    }
}
```

This code is used in §7.

§**9. Values as noun phrases.** It is very nearly true that the subject and object noun phrases are parsed by `SP_value`, which was given in "Type Expressions and Values". But there is a technicality: for reasons to do with ambiguities, `SP_value` needs to be able to try descriptions which involve only physical objects at one stage, and then later to try other descriptions.

Note that `SP_purely_physical_description` calls `SP_description` which in turn may, if there's a relative clause, call `SP_sentence` and thus `SP_noun_phrase`. Rather than passing endless copies of a flag down the call stack, we simply give `SP_noun_phrase` a global mode of operation.

```
int force_all_SP_noun_phrases_to_be_physical = FALSE;

meaning_list *SP_purely_physical_description(int w1, int w2) {
    meaning_list *ml;
    int s = force_all_SP_noun_phrases_to_be_physical;
    force_all_SP_noun_phrases_to_be_physical = TRUE;
    ml = SP_description(w1, w2, FALSE);
    force_all_SP_noun_phrases_to_be_physical = s;
    return ml;
}
```

The function SP_purely_physical_description is called from 5/tandv.

§**10.** The upshot of this is that `SP_noun_phrase` is only ever called in "purely physical mode" when it will later be called outside that mode in any event, and that therefore the set of excerpts matched by `SP_noun_phrase` genuinely is the same as that matched by `SP_value`.

```
meaning_list *SP_noun_phrase(int w1, int w2, int allow_nounless) {
    begin_SP_routine(w1, w2);
    LOGIF(S_GRAMMAR, "SP_noun_phrase on <$W>, %s nounless\n", w1, w2,
        allow_nounless?"allowing":"forbidding");
    if (no_calls_to_SP_noun_phrase++ >= S_PARSER_COMPLEXITY_THRESHOLD) return NULL;
    if (force_all_SP_noun_phrases_to_be_physical) {
        try_production(SP_description(w1, w2, allow_nounless));
    } else {
        try_production(SP_value(w1, w2));
    }
    on_success_invoke(NP_PRODUCTION);
}
```

§**11.** Finally, the following routine constructs an `SN_PRODUCTION` subtree for a noun-governed description or an adjective list. It is needed only for conditions ("if fixed in place scenery, ...") where the object referred to is understood from context.

```
meaning_list *SP_descriptive_NP(int w1, int w2) {
    begin_SP_routine(w1, w2);
    ml2 = SP_description(w1, w2, FALSE);
    if (ml2 == NULL) ml2 = SP_adjective_list(w1, w2);
    if (ml2) {
        if (ml_production(ml_down(ml2)) == SN_PRODUCTION) {
            ml = ml_down(ml2); goto Accept;
        }
        invoke_1(SN_PRODUCTION, ml2);
    }
    return NULL;
    Accept: ml_set_text(ml, w1, w2);
    return ml;
}
```

The function SP_descriptive_NP is called from 5/candp.

Conditions and Phrases 5/candp

Purpose

To parse the text of To... phrases, say phrases and conditions.

5/candp.§1-6 Conditions; §7-12 Command phrases; §13-18 Say phrases

---

Definitions

¶1. For the most part, the meaning-list scheme is modeless – no context is needed to determine what to do; all contextual work of interpretation is shifted later on, to the type-checking stage. But I have not quite been able to maintain this theoretically neat position. For one thing, "otherwise (P - phrase)" is handled as an independent phrase, and this perhaps questionable decision means that the meaning list converter needs to know whether there has just been an "if" or not – since "otherwise" can only be valid if there has. (To be fair, "else" is funny in all compiler grammars.)

```
int last_parsed_phrase_was_if = FALSE;
int last_parsed_phrase_opened_a_block = FALSE;
```

---

§**1. Conditions.** Inform's syntax would be purer if all conditions were the result of parsing `SP_sentence`. In general they are, but there two operations which enrich conditions, and three other sources of basic conditions: phrase options, phrases to decide, action patterns.

The reason a literal number is explicitly not allowed to be a condition is that if something is created called (say) "Room 62" then "62" might be read by `SP_description` as an abbreviated reference to that room.

```
meaning_list *SP_condition(int w1, int w2, int allow_bare_description) {
    begin_SP_routine(w1, w2);
    make_necessary_callings(w1, w2);
    if (paired_brackets(w1, w2)) return SP_condition(w1+1, w2-1, allow_bare_description);
    ⟨Allow two conditions to be joined with AND and OR 2⟩;
    ⟨Trim a stipulation on the chronology away from the condition 3⟩;
    if ((w1 == w2) && (vocab_test_flags(w1, NUMBER_MC))) return NULL;
    ⟨If inside a phrase, read the name of a phrase option as an implicit condition 4⟩;
    if (![[w1, w2 == there is/are ...]]) ⟨Try an invocation of a phrase-to-decide 5⟩;
    ⟨Read an action pattern as an implicit condition 6⟩;
    try_production(SP_sentence(w1, w2, SV_PRODUCTION));
    if (allow_bare_description) try_production(SP_descriptive_NP(w1, w2));
    if [[w1, w2 == there is/are ...]] ⟨Try an invocation of a phrase-to-decide 5⟩;
    on_success_invoke(COND_PRODUCTION);
}
```

The function SP_condition is called from 7/ttts and 13/test.

§**2.** Logical "and" and "or" are implemented directly right here, rather than being phrases defined in the Standard Rules.

⟨Allow two conditions to be joined with AND and OR 2⟩ ≡

```
int bl;
for (bl=0, i=w1; i<w2; i++) {
    int prd = 0, upto = i-1;
    if [[word i == OPENBRACKET]] bl++;
    if [[word i == CLOSEBRACKET]] bl--;
    if (bl > 0) continue;
    if [[word i == and]] prd = COND_AND_PRODUCTION;
    if [[word i == or]] prd = COND_OR_PRODUCTION;
    if ((prd != 0) && (upto>=w1)) {
        if ((upto>w1) && [[word upto == COMMA]]) upto--;
        if ((ml2 = SP_condition(w1, upto, FALSE)) &&
            (ml3 = SP_condition(i+1, w2, FALSE)))
            invoke_2(prd, ml2, ml3);
    }
}
```

This code is used in §1.

§**3.** This removes suffixes like "...for the third time" from a condition:

⟨Trim a stipulation on the chronology away from the condition 3⟩ ≡

```
time_period tp = parse_time_period(w1, w2);
int end_of_non_time_part = tp_is_valid(&tp);
if ((end_of_non_time_part >= w1) &&
    (ml2 = SP_condition(w1, end_of_non_time_part, allow_bare_description))) {
    ml3 = make_time_ml(tp);
    invoke_2(COND_PAST_PRODUCTION, ml2, ml3);
}
```

This code is used in §1.

§**4.** The option to use "not..." to test a phrase option negatively is parsed here only after the positive sense has been ruled out – people do sometimes create phrase options like "not printing anything" which begin with the word "not".

⟨If inside a phrase, read the name of a phrase option as an implicit condition 4⟩ ≡

```
if (phrase_being_compiled != NULL) {
    if ((i = ph_To_parse_phrase_option_used(phrase_being_compiled, w1, w2)) >= 0)
        try_production(make_option_ml(i, TRUE));
    if ([[w1, w2 == not ...]]
        && ((i = ph_To_parse_phrase_option_used(phrase_being_compiled, w1+1, w2)) >= 0))
        try_production(make_option_ml(i, FALSE));
}
```

This code is used in §1.

§**5.** As usual, we try to get better sensitivity to ambiguities by dividing the test for a phrase-to-decide into two, so that the following is used at a different point if the excerpt begins "there is" than if it doesn't. The point of this is that some phrases to decide have wording which coincides with a description, and in general the phrase should win, but in the case of "there is" we make the presumption that the author intends a sentence testing the existence of something.

⟨Try an invocation of a phrase-to-decide 5⟩ ≡

```
if ((ml2 = SP_excerpt(COND_PHRASE_MC, w1, w2)) != NULL)
    invoke_1(COND_PHRASE_PRODUCTION, ml2);
if ([[w1, w2 == not ...]] && (SP_excerpt(COND_PHRASE_MC, w1+1, w2))) {
    ml2 = SP_condition(w1+1, w2, FALSE);
    invoke_1(COND_NOT_PRODUCTION, ml2);
}
```

This code is used in §1.

§**6.** As a condition, an action pattern is implicitly considered as a test of what the current action is:

if examining an open door, ...

This wouldn't work so well for the past tense form:

if examined an open door, ...

because it seems too clunky as neither quite active nor passive. Who examined the open door? So Inform uses the following version instead:

if we have examined an open door, ...

thus adopting the "science we". Not very elegant, but the alternatives were difficult to parse. "We are" is allowed for consistency's sake, but does nothing, i.e., "we are taking" and "taking" are synonymous.

⟨Read an action pattern as an implicit condition 6⟩ ≡

```
action_pattern ap;
if [[w1, w2 == we have ...]] {
    ap = parse_action_pattern(w1+2, w2, HASBEEN_TENSE);
    if (ap_is_valid(&ap)) {
        if (ap_makes_callings(&ap)) {
            sentence_problem(_P_(C5PastActionCalled),
                "a description of an action cannot both refer to past "
                "history and also use '(called ...)'",
                "because that would require Inform in general to remember "
                "too much information about past events.");
        } else try_production(make_pastaction_ml(ap));
    }
}
if [[w1, w2 == we are ...]] {
    ap = parse_action_pattern(w1+2, w2, IS_TENSE);
    if (ap_is_valid(&ap)) try_production(make_action_ml(ap));
}
ap = parse_action_pattern(w1, w2, IS_TENSE);
if (ap_is_valid(&ap)) try_production(make_action_ml(ap));
```

This code is used in §1.

§**7. Command phrases.** The final clutch of SP-routines handle individual commands, written in their semicolon-divided list in the body of a rule or "To..." definition.

For instance, in the not very sensible rule:

```
Instead of jumping: now the score is 10; say "Greetings!" instead.
```

Inform will use `SP_command` to parse the text of the two commands in the rule body, with the results:

```
PHRASE_PRODUCTION
    {now # = VOID_PHRASE_MC} / "now the score is 10"
        UNPARSED_PRODUCTION / "the score is 10"
```

and, at greater length:

```
PHRASE_PRODUCTION
    INSTEAD_PRODUCTION
        SAY_PRODUCTION
            [1/22] {# = SAY_PHRASE_MC} / ""Greetings!""
                UNPARSED_PRODUCTION / ""Greetings!""
            ...
```

Note that `SP_command` passes the text with no attempt to judge whether the parameters of the phrase match; instead it records all possibilities at this point (case 1 of 22 is that a number is supplied, etc., etc.) for typechecking to resolve much later on.

```
meaning_list *SP_command(int w1, int w2) {
    begin_SP_routine(w1, w2);
    if (paired_brackets(w1, w2)) return SP_command(w1+1, w2-1);
    ⟨Police phrases beginning with else or otherwise 8⟩;
    ⟨Detect a switch case indication 9⟩;
    ⟨Detect the use of a To... phrase 10⟩;
    try_production(SP_rule_outcome(w1, w2));
    on_success_invoke(CMD_PRODUCTION);
}
```

The function SP_command is called from 7/ttts.

§**8.** "Else" or "otherwise" as a single word on its own is a divider in a long "if", and this has its own production (rather than being defined as a To... phrase in the Standard Rules). If a condition is added, though, we are in a more substantial construction which does indeed get defined in the Standard Rules. But we reject some obviously wrong cases now, to force a problem message to appear.

⟨Police phrases beginning with else or otherwise 8⟩ ≡

```
    if [[w1, w2 == else/otherwise ***]] {
        int c;
        if (w1 == w2) invoke_0(OTHERWISE_PRODUCTION);
        if ((last_parsed_phrase_was_if) &&
            (last_parsed_phrase_opened_a_block)) return NULL;                    to force error
        if ((last_parsed_phrase_was_if == FALSE) &&
            ((([[w1, w2 == else/otherwise if/unless ***]] == FALSE)
            || [[w1, w2 == else/otherwise if/unless ... begin]]
            || [[w1, w2 == ... COMMA ... : c]]))
                return NULL;                                                     to force error
        if [[w1, w2 == else/otherwise if/unless ... begin]]
            return NULL;                                                         to force error
    }
```

This code is used in §7.

§**9.** Switch cases are another form of divider, again with a production of their own rather than being defined as phrases in the SR.

⟨Detect a switch case indication 9⟩ ≡
```
    if [[w1, w2 == DASHDASH otherwise]]
        invoke_0(CASE_PRODUCTION);
    if ([[w1, w2 == DASHDASH ...]] && (ml2 = SP_value(w1+1, w2)))
        invoke_1(CASE_PRODUCTION, ml2);
```
This code is used in §7.

§**10.** This is the case we almost always fall into. Note that the phrase text can be prefixed or suffices with the "instead" keyword, but not both.

⟨Detect the use of a To... phrase 10⟩ ≡
```
    int v1 = -1, v2 = -1;
    if [[w1, w2 == instead ... --> v1, v2]] ;
    else if [[w1, w2 == ... instead --> v1, v2]] ;
    if (v1 >= 0) try_production(SP_command_phrase(v1, v2, TRUE));
    try_production(SP_command_phrase(w1, w2, FALSE));
```
This code is used in §7.

§**11.** Text substitutions ("say phrases") and void phrases have independent namespaces, as the following demonstrates.

```
meaning_list *SP_command_phrase(int w1, int w2, int instead_flag) {
    meaning_list *ml, *ml2, *ml4, *ml5;
    if ([[w1, w2 == say ...]] && (ml5 = SP_say_phrase(w1+1, w2))) {
        ml2 = new_ml(SAY_PRODUCTION); ml_set_down(ml2, ml5);
    } else {
        ml2 = SP_excerpt(VOID_PHRASE_MC + END_PHRASE_MC, w1, w2);
    }
    if (ml2) {
        if (instead_flag) {
            ml4 = ml2; ml2 = new_ml(INSTEAD_PRODUCTION); ml_set_down(ml2, ml4);
        }
        ml = new_ml(PHRASE_PRODUCTION); ml_set_down(ml, ml2);
        return ml;
    }
    return NULL;
}
```

§**12.** Rule outcomes are such a small linguistic category that they don't get a meaning code to themselves, so they share the miscellaneous category.

```
meaning_list *SP_rule_outcome(int w1, int w2) {
    meaning_list *ml = SP_excerpt(MISCELLANEOUS_MC, w1, w2);
    if ((ml) && (em_get_secondary_code(ml_meaning(ml)) == RULE_OUTCOME_SMC))
        return ml;
    return NULL;
}
```

§13. Say phrases. Uniquely, a say phrase takes the form of a comma-separated list, and is compiled by regarding each entry as an independent entry. Double-quoted text is literal if it contains no square brackets, but is expanded if it includes text substitutions in squares: thus the following routine ensures that

say "Look, [the noun] said."

is parsed as if it had read:

say "Look, ", the noun, " said."

The result is a flat list of `SAY_PHRASE_MC` leaves as siblings in the meaning list tree, one for each non-trivial part of the list. (We discard the empty text for efficiency's sake; the expansion of square bracket substitutions does sometimes produce these, so it is not as pointless an optimisation as it might look.)

```
meaning_list *SP_say_phrase(int w1, int w2) {
    int i;
    if ((w1<0) || (w2<w1)) return NULL;
    if ((i = is_word_intermediate_unbracketed(COMMA_V, w1, w2)) >= 0) {
        if ((compare_word_by_strcmp(w1, "\"\"")) && (i==w1+1))
            return SP_say_phrase(w1+2, w2);                          ignore empty initial string
        if ((compare_word_by_strcmp(w2, "\"\"")) && (i==w2-1))
            return SP_say_phrase(w1, w2-2);                          ignore empty final string
        ⟨Divide the say list at this first non-trivial comma 14⟩;
    }
    if ((w1 == w2) && (vocab_test_flags(w1, TEXTWITHSUBS_MC)))
        ⟨Unpack quoted text containing substitutions in square brackets and parse again 15⟩;
    return SP_excerpt(SAY_PHRASE_MC, w1, w2);
}
```

§14. Because we looked for commas left to right, the left-hand term is without commas, and will parse to a single leaf in the meaning list; the right-hand term may still be a comma-separated list.

⟨Divide the say list at this first non-trivial comma 14⟩ ≡

```
    meaning_list *ml = SP_say_phrase(w1, i-1);          otherwise take first term in comma list
    meaning_list *ml2 = SP_say_phrase(i+1, w2);                  and then recurse for the rest
    if (ml && ml2) {
        meaning_list *ml_last = ml;
        while (ml_right(ml_last)) ml_last = ml_right(ml_last);
        ml_set_right(ml_last, ml2);
        return ml;
    }
```

This code is used in §13.

§**15.** We ask the lexer to read the quoted text again, but this time in its mode which breaks up text at square brackets:

⟨Unpack quoted text containing substitutions in square brackets and parse again 15⟩ ≡

```
int lw1, lw2; char *p = lw_array[w1].lw_rawtext;
⟨Check that substitution does not contain suspicious punctuation 16⟩;
lw1 = lexer_wordcount;
feed_into_lexer(p, TRUE, FALSE);                    expand unquoted text substitution
lw2 = lexer_wordcount - 1;
feed_into_lexer(" . ", FALSE, FALSE);
for (i=lw1; i<=lw2; i++) if [[word i == FULLSTOP]]
    ⟨Issue problem message for unexpected punctuation in a substitution 17⟩;
return SP_say_phrase(lw1, lw2);                     ...and parse it
```

This code is used in §13.

§**16.** It would of course also be an error for the text to contain nested or mismatched square brackets, but if it did, then the text would not have been flagged as `TEXTWITHSUBS_MC` by the lexer.

⟨Check that substitution does not contain suspicious punctuation 16⟩ ≡

```
int k, sqb = 0;
for (k=0; p[k]; k++) {
    switch (p[k]) {
        case '[': sqb++; break;
        case ']': sqb--; break;
        case ':': if ((k>0) && (isdigit(p[k-1])) && (isdigit(p[k+1]))) break;
        case ';':
            if (sqb > 0) ⟨Issue problem message for unexpected punctuation in a substitution 17⟩;
            break;
        case ',':
            if (sqb > 0) ⟨Issue problem message for comma in a substitution 18⟩;
            break;
    }
}
```

This code is used in §15.

§**17.** So now just the problem messages:

⟨Issue problem message for unexpected punctuation in a substitution 17⟩ ≡

```
sentence_problem(_P_(C5TSWithPunctuation),
    "a substitution contains a '.', ':' or ';'",
    "which suggests that a close square bracket ']' may have gone astray.");
return NULL;
```

This code is used in §15,16,15,16,15,16.

§**18.** And the more specialised:

⟨Issue problem message for comma in a substitution 18⟩ ≡

```
sentence_problem(_P_(C5TSWithComma),
    "a substitution contains a comma ’,’",
    "which is against the rules, because ’say’ is a special phrase in "
    "which the comma divides items in a list of things to say, and so it "
    "loses its ordinary meanings. Because of this, no text substitution "
    "can contain a comma. "
    "(If you’re trying to use a value produced by a phrase with a phrase "
    "option - say ’the best route from A to B, using even locked doors’ - "
    "you’ll need to put this in a ’let’ variable first and then say that, "
    "or else define a better text substitution to do the job for you.)");
return NULL;
```

This code is used in §16.

Meaning List Conversions 5/mlc

Purpose

To convert meaning lists into specifications, for the use of the rest of NI.

Definitions

¶1. This final section of Chapter 5 is one of a pair of sections making up an interface layer between the S-parser (below) and the rest of Inform (above). The S-parser turned excerpts into a tree structure with both hierarchical depth and alternative readings, which was expansive and easy to generate but not easy to use. The rest of Inform wants to store essentially the same information in a single object (a "specification") with, in some cases, a proposition of predicate calculus attached. This section turns the meaning list into the specification part; the code which produces the proposition is the content of the "Sentence Conversions" section of Chapter 6.

¶2. Some notational conventions. Routines to convert ML subtrees to SPs are all given names in the form `*_to_spec` (and they are the only such routines in Inform). Only four of these are accessed from outside the interface layer, one for each of the four big concepts parsed by the S-parser:

(1) `TE_subtree_to_spec`, which converts a type expression subtree headed by `TE_PRODUCTION`.

(2) `VAL_subtree_to_spec`, which converts a value subtree headed by `VAL_PRODUCTION`.

(3) `CMD_subtree_to_spec`, which converts a command subtree headed by `CMD_PRODUCTION`.

(4) `COND_subtree_to_spec`, which converts a condition subtree headed by `COND_PRODUCTION`.

¶3. Given the nature of the code, it's prudent to make defensive assertions. (See also the meaning list conversion backtrace macro, in "Debugging Log", which is used to produce a meaningful backtrace if something goes wrong.) So:

```
define NOW_CONVERTING(pr)
    specification *spec = NULL;
    RECORD_MLC_BACKTRACE;
    if ((ml == NULL) || (ml_production(ml) != pr))
        internal_error("Non-" #pr " production");
define NOW_CONVERTING_SUBTREE(pr)
    NOW_CONVERTING(pr)
    if (ml_down(ml) == NULL)
        internal_error("Childless " #pr " production");
define NOW_CONVERTING_LEAF(pr)
    NOW_CONVERTING(pr)
    if (ml_down(ml))
        internal_error("Non-leaf " #pr " production");
define CHILD_UNCONVERTIBLE(pr)
    internal_error("Unknown child of " #pr " production");
    return new_UNKNOWN_spec();
```

---

§**1. Layout.** The code in this conversion layer is arranged in a top-down style as follows:

(I) The four externally used conversion routines described above.

(II) One routine for each possible node in the tree, responsible for converting that node and any subtree hanging from it, except for leaves corresponding to excerpt meanings.

(III) One routine for each possible leaf-node corresponding to an excerpt meaning.

(IV) More difficult code to handle sentences and phrase invocations, needed to support various node types in (II).

§**2. Level I: The four tops.** Command subtrees are all headed by `CMD_PRODUCTION`, which has four possible children, the interesting one being an excerpt-meaning leaf for a rulebook outcome.

Because we expect this to be called sequentially for each command in a rule body, we can preserve a little state from one call to the next: in particular, whether the previous call compiled an "if" or a block opening (detectable with an "else").

We have to be careful in case a phrase which doesn't open a block nevertheless causes another phrase, as here:

> while ... repeatedly if ...

where a "while" iterates a single phrase which happens to be an "if". So we change the sequential state only if we aren't being called in a nested way.

```
int PHRASE_subtree_to_type_nesting = 0;

specification *CMD_subtree_to_spec(meaning_list *ml) {
    if (ml == NULL) return new_UNKNOWN_spec();
    NOW_CONVERTING_SUBTREE(CMD_PRODUCTION);
    PHRASE_subtree_to_type_nesting++;
    if (PHRASE_subtree_to_type_nesting == 1) {
        int w1, w2;
        ml_get_text(ml, &w1, &w2);
        last_parsed_phrase_was_if = [[word w1 == if]];
        last_parsed_phrase_opened_a_block = [[word w2 == begin]];
    }
    ml = ml_down(ml);
    switch(ml_production(ml)) {
        case PHRASE_PRODUCTION: spec = PHRASE_subtree_to_spec(ml); break;
        case OTHERWISE_PRODUCTION: spec = OTHERWISE_subtree_to_spec(ml); break;
        case CASE_PRODUCTION: spec = CASE_subtree_to_spec(ml); break;
        case MISCELLANEOUS_MC:
            switch(em_get_secondary_code(ml_meaning(ml))) {
                case RULE_OUTCOME_SMC: spec = RULE_OUTCOME_SMC_to_spec(ml, FALSE); break;
                default: CHILD_UNCONVERTIBLE(VAL_PRODUCTION);
            }
            break;
        default: CHILD_UNCONVERTIBLE(CMD_PRODUCTION);
    }
    PHRASE_subtree_to_type_nesting--;
    return spec;
}
```

The function CMD_subtree_to_spec is called from 7/ttts.

§**3.** Condition subtrees.

```
specification *COND_subtree_to_spec(meaning_list *ml) {
    if (ml == NULL) return new_UNKNOWN_spec();
    NOW_CONVERTING_SUBTREE(COND_PRODUCTION);
    int mw1, mw2; ml_get_text(ml, &mw1, &mw2);
    ml = ml_down(ml);
    switch(ml_production(ml)) {
        case COND_NOT_PRODUCTION: spec = COND_NOT_subtree_to_spec(ml); break;
        case COND_AND_PRODUCTION: spec = COND_AND_subtree_to_spec(ml); break;
        case COND_OR_PRODUCTION: spec = COND_OR_subtree_to_spec(ml); break;
        case COND_PAST_PRODUCTION: spec = COND_PAST_subtree_to_spec(ml); break;
        case OPTION_PRODUCTION: spec = OPTION_subtree_to_spec(ml); break;
        case PHRASE_PRODUCTION: spec = PHRASE_subtree_to_spec(ml); break;
        case COND_PHRASE_PRODUCTION: spec = COND_PHRASE_subtree_to_spec(ml); break;
        case ACTION_PRODUCTION: spec = ACTION_subtree_to_spec(ml); break;
        case SN_PRODUCTION: spec = SN_subtree_to_spec(ml); break;
        case SV_PRODUCTION: spec = SV_subtree_to_spec(ml); break;
        default: CHILD_UNCONVERTIBLE(COND_PRODUCTION);
    }
    if ((mw1 >= 0) && (spec->word_ref1 < 0)) { spec->word_ref1 = mw1; spec->word_ref2 = mw2; }
    return spec;
}
```

The function COND_subtree_to_spec is called from 7/ttts.

§**4. Type expression subtrees.**

```
specification *TE_subtree_to_spec(meaning_list *ml) {
    if (ml == NULL) return new_UNKNOWN_spec();                    note that no problem message is issued
    NOW_CONVERTING_SUBTREE(TE_PRODUCTION);
    int mw1, mw2; ml_get_text(ml, &mw1, &mw2);
    ml = ml_down(ml);
    switch(ml_production(ml)) {
        case TE_CALLED_PRODUCTION:
            spec = TE_CALLED_subtree_to_spec(ml); break;
        case TE_EX_VAR_PRODUCTION:
            spec = TE_EX_VAR_subtree_to_spec(ml); break;
        case TE_NEW_VAR_PRODUCTION:
            spec = TE_NEW_VAR_subtree_to_spec(ml); break;
        case TE_GL_VAR_PRODUCTION:
            spec = TE_GL_VAR_subtree_to_spec(ml); break;
        case TE_VAR_PRODUCTION:
            spec = TE_VAR_subtree_to_spec(ml); break;
        case DC_PRODUCTION: spec = DC_subtree_to_spec(ml, FALSE); break;
        case VAL_NOTHING_PRODUCTION: spec = VAL_NOTHING_subtree_to_spec(ml); break;
        case LITERAL_PRODUCTION: spec = LITERAL_subtree_to_spec(ml); break;
        case TYPE_PRODUCTION: spec = TYPE_subtree_to_spec(ml); break;
        case QUANTITY_MC: spec = QUANTITY_MC_to_spec(ml); break;
        case RULE_MC: spec = RULE_MC_to_spec(ml); break;
        case RULEBOOK_MC: spec = RULEBOOK_MC_to_spec(ml); break;
        case ACTIVITY_MC: spec = ACTIVITY_MC_to_spec(ml); break;
```

```
        case EQUATION_MC: spec = EQUATION_MC_to_spec(ml); break;
        case TABLE_MC: spec = TABLE_MC_to_spec(ml); break;
        case MISCELLANEOUS_MC:
            switch(em_get_secondary_code(ml_meaning(ml))) {
                case ACTION_NAME_SMC: spec = ACTION_NAME_SMC_to_spec(ml); break;
                default: CHILD_UNCONVERTIBLE(VAL_PRODUCTION);
            }
            break;
        default: CHILD_UNCONVERTIBLE(TE_PRODUCTION);
    }
    if ((mw1 >= 0) && (spec->word_ref1 < 0)) { spec->word_ref1 = mw1; spec->word_ref2 = mw2; }
    return spec;
}
```

The function TE_subtree_to_spec is called from 7/ttts.

§5. Value subtrees.

```
specification *VAL_subtree_to_spec(meaning_list *ml) {
    if (ml == NULL) return new_UNKNOWN_spec();
    NOW_CONVERTING_SUBTREE(VAL_PRODUCTION);
    int mw1, mw2; ml_get_text(ml, &mw1, &mw2);
    ml = ml_down(ml);
    switch(ml_production(ml)) {
        case AP_PRODUCTION: spec = AP_subtree_to_spec(ml); break;
        case DC_PRODUCTION: spec = DC_subtree_to_spec(ml, TRUE); break;
        case LITERAL_PRODUCTION: spec = LITERAL_subtree_to_spec(ml); break;
        case LOCAL_PRODUCTION: spec = LOCAL_subtree_to_spec(ml); break;
        case MEMBER_OF_PRODUCTION: spec = MEMBER_OF_subtree_to_spec(ml); break;
        case STV_PRODUCTION: spec = STV_subtree_to_spec(ml); break;
        case TR_PRODUCTION: spec = TR_subtree_to_spec(ml); break;
        case VAL_LIST_ENTRY_PRODUCTION: spec = VAL_LIST_ENTRY_subtree_to_spec(ml); break;
        case VAL_NOTHING_PRODUCTION: spec = VAL_NOTHING_subtree_to_spec(ml); break;
        case VAL_PROP_OF_PRODUCTION: spec = VAL_PROP_OF_subtree_to_spec(ml); break;
        case VALUE_PHRASE_PRODUCTION: spec = VALUE_PHRASE_subtree_to_spec(ml); break;
        case EQUATION_INLINE_PRODUCTION: spec = EQUATION_MC_to_spec(ml_down(ml)); break;
        case EQUATION_WHERE_PRODUCTION: spec = EQUATION_MC_to_spec(ml_down(ml)); break;
        case ACTIVITY_MC: spec = ACTIVITY_MC_to_spec(ml); break;
        case EQUATION_MC: spec = EQUATION_MC_to_spec(ml); break;
        case PROPERTY_MC: spec = PROPERTY_MC_to_spec(ml); break;
        case QUANTITY_MC: spec = QUANTITY_MC_to_spec(ml); break;
        case RULE_MC: spec = RULE_MC_to_spec(ml); break;
        case RULEBOOK_MC: spec = RULEBOOK_MC_to_spec(ml); break;
        case TABLE_COLUMN_MC: spec = TABLE_COLUMN_MC_to_spec(ml); break;
        case TABLE_MC: spec = TABLE_MC_to_spec(ml); break;
        case WORLD_OBJECT_MC: spec = WORLD_OBJECT_MC_to_spec(ml); break;
        case MISCELLANEOUS_MC:
            switch(em_get_secondary_code(ml_meaning(ml))) {
                case ACTION_NAME_SMC: spec = ACTION_NAME_SMC_to_spec(ml); break;
                case RELATION_SMC: spec = RELATION_SMC_to_spec(ml); break;
                case RULE_OUTCOME_SMC: spec = RULE_OUTCOME_SMC_to_spec(ml, TRUE); break;
                case USE_OPTION_SMC: spec = USE_OPTION_SMC_to_spec(ml); break;
```

```
                default: CHILD_UNCONVERTIBLE(VAL_PRODUCTION);
            }
            break;
        default: CHILD_UNCONVERTIBLE(VAL_PRODUCTION);
    }
    if ((mw1 >= 0) && (spec->word_ref1 < 0)) { spec->word_ref1 = mw1; spec->word_ref2 = mw2; }
    return spec;
}
```

The function VAL_subtree_to_spec is called from 6/sconv and 7/ttts.

§6. Level II: Routines for each possible intermediate tree. These are given in alphabetical order, and most contain nothing worth comment.

§7. ACTION_PRODUCTION.

```
specification *ACTION_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(ACTION_PRODUCTION);
    spec = ml_get_attached_spec(ml);
    return spec;
}
```

§8. AP_PRODUCTION.

```
specification *AP_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(AP_PRODUCTION);
    spec = ml_get_attached_spec(ml);
    return spec;
}
```

§9. CASE_PRODUCTION.

```
specification *CASE_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING(CASE_PRODUCTION);
    ml = ml_down(ml);
    if (ml == NULL) spec = new_default_CASE_type();
    else switch(ml_production(ml)) {
        case VAL_PRODUCTION: spec = new_CASE_spec(VAL_subtree_to_spec(ml)); break;
        default: CHILD_UNCONVERTIBLE(CASE_PRODUCTION);
    }
    return spec;
}
```

§10. COND_AND_PRODUCTION.

```
specification *COND_AND_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(COND_AND_PRODUCTION);
    spec = new_LOGICAL_AND_spec(
        COND_subtree_to_spec(ml_down(ml)),
        COND_subtree_to_spec(ml_right(ml_down(ml))));
    return spec;
}
```

§11. COND_OR_PRODUCTION.

```
specification *COND_OR_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(COND_OR_PRODUCTION);
    spec = new_LOGICAL_OR_spec(
        COND_subtree_to_spec(ml_down(ml)),
        COND_subtree_to_spec(ml_right(ml_down(ml))));
    return spec;
}
```

§**12.** COND_NOT_PRODUCTION.

```
specification *COND_NOT_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(COND_NOT_PRODUCTION);
    spec = COND_subtree_to_spec(ml_down(ml));
    if (spec_test_flag(spec, CONDITION_NEGATED_SPFLAG))
        spec_clear_flag(spec, CONDITION_NEGATED_SPFLAG);
    else
        spec_set_flag(spec, CONDITION_NEGATED_SPFLAG);
    return spec;
}
```

§**13.** COND_PAST_PRODUCTION.

```
specification *COND_PAST_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(COND_PAST_PRODUCTION);
    spec = COND_subtree_to_spec(ml_down(ml));
    int tense_only = IS_TENSE;
    if (spec_get_condition_tense(spec)) tense_only = tp_get_tense(spec_get_condition_tense(spec));
    spec_set_condition_tense(spec,
        spec_get_condition_tense(ml_get_attached_spec(ml_right(ml_down(ml)))));
    tp_set_tense(spec_get_condition_tense(spec), tense_only);
    return spec;
}
```

§**14.** COND_PHRASE_PRODUCTION.

```
specification *COND_PHRASE_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING(COND_PHRASE_PRODUCTION);
    spec = general_phrase_to_spec(ml);
    return spec;
}
```

§**15.** DC_PRODUCTION, with its optional CALLED_PRODUCTION and DETERMINER_PRODUCTION annotations.

```
parse_node *C5SpecificCalling_last_issued = NULL;

specification *DC_subtree_to_spec(meaning_list *ml, int construed_as_noun) {
    NOW_CONVERTING_SUBTREE(DC_PRODUCTION);
    meaning_list *annotation;

    ml = ml_down(ml);
    switch(ml_production(ml)) {
        case SN_PRODUCTION: spec = SN_subtree_to_spec(ml); break;
        case DC_ADJS_PRODUCTION: spec = DC_ADJS_subtree_to_spec(ml, construed_as_noun); break;
        case DC_NOUN_PRODUCTION: spec = DC_NOUN_subtree_to_spec(ml); break;
        case DC_ADJSNOUN_PRODUCTION: spec = DC_ADJSNOUN_subtree_to_spec(ml); break;
        default: CHILD_UNCONVERTIBLE(DC_PRODUCTION);
    }

    annotation = ml_right(ml);

    while (annotation) {
        switch (ml_production(annotation)) {
            case CALLED_PRODUCTION:
                if (spec_is_actual_CONSTANT(spec)) {
                    if (C5SpecificCalling_last_issued != current_sentence) {
                        C5SpecificCalling_last_issued = current_sentence;
                        sentence_problem(_P_(C5SpecificCalling),
                            "a 'called' name can only be given to something "
                            "which is described vaguely",
                            "and can't be given to a definite object or value. "
                            "So 'if a thing (called the gadget) is carried' is "
                            "allowed, but 'if the X-Ray Zapper (called the gadget) "
                            "is carried' isn't allowed - if it's the X-Ray Zapper, "
                            "then call it that.");
                    }
                } else {
                    if (current_stack_frame()) {
                        int c1, c2;
                        ml_get_text(annotation, &c1, &c2);
                        spec_attach_calling(spec, c1, c2);
                        kind_of_value *kov = spec_get_described_kov(spec);
                        if (spec_is_generic_CONSTANT(spec)) kov = spec_get_kind_of_value(spec);
                        ensure_called_local(c1, c2, kov);
                    }
                }
                break;
            case DETERMINER_PRODUCTION:
                if (species_is(spec, DESCRIPTION_SPC))
                    spec_attach_quantifier(spec,
                        RETRIEVE_POINTER_quantifier(ml_get_attached_data(annotation)),
                        ml_match_score(annotation));
                else coerce_spec_to_UNKNOWN(spec);
                break;
            default: CHILD_UNCONVERTIBLE(DC_PRODUCTION);
        }
        annotation = ml_right(annotation);
    }
```

```
    return spec;
}
```

The function DC_subtree_to_spec is called from 6/sconv.

§16. DC_ADJS_PRODUCTION.

```
specification *DC_ADJS_subtree_to_spec(meaning_list *ml, int construed_as_noun) {
    NOW_CONVERTING_SUBTREE(DC_ADJS_PRODUCTION);
    if ((ml_right(ml_down(ml_down(ml))) == NULL) && (construed_as_noun)) {
        adjective_list_entry *tr = adj_to_adjective_list_entry(ml_down(ml_down(ml)));
        adjectival_phrase *aph = get_adjective_from_list_entry(tr);
        property_name *prn = NULL;
        if (aph_has_ENUMERATIVE_meaning(aph))
            prn = kov_get_coinciding_property(qty_kind_of_value(
                aph_has_ENUMERATIVE_meaning(aph)));
        else if (aph_has_EORP_meaning(aph))
            prn = aph_has_EORP_meaning(aph);
        if (prn){
            spec = property_name_to_PROPERTY_spec(prn);
            return spec;
        }
    }
    spec = qualified_noun_to_spec(ml_down(ml), NULL);
    return spec;
}
```

§17. DC_NOUN_PRODUCTION.

```
specification *DC_NOUN_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(DC_NOUN_PRODUCTION);
    spec = qualified_noun_to_spec(NULL, ml_down(ml));
    return spec;
}
```

§18. DC_ADJSNOUN_PRODUCTION.

```
specification *DC_ADJSNOUN_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(DC_ADJSNOUN_PRODUCTION);
    spec = qualified_noun_to_spec(ml_down(ml), ml_right(ml_down(ml)));
    return spec;
}
```

§19. LITERAL_PRODUCTION.

```
specification *LITERAL_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(LITERAL_PRODUCTION);
    spec = ml_get_attached_spec(ml);
    return spec;
}
```

§**20.** LOCAL_PRODUCTION.

```
specification *LOCAL_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(LOCAL_PRODUCTION);
    spec = ml_get_attached_spec(ml);
    return spec;
}
```

§**21.** MEMBER_OF_PRODUCTION.

```
specification *MEMBER_OF_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(MEMBER_OF_PRODUCTION);
    spec = NP_subtree_to_spec(ml_down(ml));
    if (spec_get_storage_form(spec) == LOCAL_VARIABLE_SPC) return spec;
    if (species_is(spec, DESCRIPTION_SPC) == FALSE) {
        LOG("Error on: $X", spec);
        internal_error("Unexpected member-of");
    }
    return spec;
}
```

§**22.** NP_PRODUCTION, though this routine is also used on subtrees which function as noun phrases but are not headed by NP_PRODUCTION nodes.

```
specification *NP_subtree_to_spec(meaning_list *ml) {
    if (ml == NULL) internal_error("missing NP");
    RECORD_MLC_BACKTRACE;
    specification *spec = NULL;
    if (ml_production(ml) == NP_PRODUCTION) ml = ml_down(ml);
    switch(ml_production(ml)) {
        case DC_PRODUCTION: spec = DC_subtree_to_spec(ml, FALSE); break;
        case PLAYER_PRODUCTION: spec = PLAYER_subtree_to_spec(ml); break;
        case VAL_PRODUCTION: spec = VAL_subtree_to_spec(ml); break;
        default: LOG("Offending subtree:\n$m", ml); internal_error("Non-NP production");
    }
    return spec;
}
```

§**23.** OPTION_PRODUCTION.

```
specification *OPTION_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(OPTION_PRODUCTION);
    spec = ml_get_attached_spec(ml);
    return spec;
}
```

§**24.** OTHERWISE_PRODUCTION.

```
specification *OTHERWISE_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(OTHERWISE_PRODUCTION);
    spec = new_OTHERWISE_spec();
    return spec;
}
```

§**25.** PHRASE_PRODUCTION.

```
specification *PHRASE_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING(PHRASE_PRODUCTION);
    spec = general_phrase_to_spec(ml);
    return spec;
}
```

§**26.** PLAYER_PRODUCTION.

```
specification *PLAYER_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(OPTION_PRODUCTION);
    spec = new_QUANTITY_spec(player_quantity);
    return spec;
}
```

§**27.** SN_PRODUCTION.

```
specification *SN_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(SN_PRODUCTION);
    spec = general_sentence_to_spec(ml);
    return spec;
}
```

§**28.** STV_PRODUCTION.

```
specification *STV_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(STV_PRODUCTION);
    spec = ml_get_attached_spec(ml);
    return spec;
}
```

§**29.** SV_PRODUCTION.

```
specification *SV_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(SV_PRODUCTION);
    spec = general_sentence_to_spec(ml);
    return spec;
}
```

§30. TE_CALLED_PRODUCTION.

```
specification *TE_CALLED_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(TE_CALLED_PRODUCTION);
    spec = TE_subtree_to_spec(ml_down(ml));
    if (spec_is_UNKNOWN(spec)) return spec;
    int c1, c2;
    ml_get_text(ml_right(ml_down(ml)), &c1, &c2);
    spec_attach_calling(spec, c1, c2);
    return spec;
}
```

§31. TE_EX_VAR_PRODUCTION.

```
specification *TE_EX_VAR_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING(TE_EX_VAR_PRODUCTION);
    kind_of_value *kov = police_te(ml_down(ml));
    spec = new_generic_LOCAL_VARIABLE_type(kov);
    return spec;
}
```

§32. TE_GL_VAR_PRODUCTION.

```
specification *TE_GL_VAR_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING(TE_GL_VAR_PRODUCTION);
    police_te(ml_down(ml));
    spec = new_generic_NONLOCAL_VARIABLE_type(NULL);
    return spec;
}
```

§33. TE_NEW_VAR_PRODUCTION.

```
specification *TE_NEW_VAR_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING(TE_NEW_VAR_PRODUCTION);
    kind_of_value *kov = police_te(ml_down(ml));
    spec = new_NEW_LOCAL_VARIABLE_NAME_spec(kov);
    return spec;
}
```

§34. TE_VAR_PRODUCTION.

```
specification *TE_VAR_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING(TE_VAR_PRODUCTION);
    kind_of_value *kov = police_te(ml_down(ml));
    spec = new_generic_NONLOCAL_VARIABLE_type(kov);
    return spec;
}
```

§**35.** TR_PRODUCTION.

```
specification *TR_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(TR_PRODUCTION);
    ml = ml_down(ml);
    switch(ml_production(ml)) {
        case TR_ENTRY_PRODUCTION: spec = TR_ENTRY_subtree_to_spec(ml); break;
        case TR_IN_ROW_PRODUCTION: spec = TR_IN_ROW_subtree_to_spec(ml); break;
        case TR_LISTED_IN_PRODUCTION: spec = TR_LISTED_IN_subtree_to_spec(ml); break;
        case TR_CORR_PRODUCTION: spec = TR_CORR_subtree_to_spec(ml); break;
        case TR_OF_IN_PRODUCTION: spec = TR_OF_IN_subtree_to_spec(ml); break;
        default: CHILD_UNCONVERTIBLE(VAL_PRODUCTION);
    }
    return spec;
}
```

§**36.** TR_CORR_PRODUCTION.

```
specification *TR_CORR_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(TR_CORR_PRODUCTION);
    spec = new_TABLE_ENTRY_spec();
    spec_set_argument(spec, 0, TABLE_COLUMN_MC_to_spec(ml_down(ml)));
    spec_set_argument(spec, 1, TABLE_COLUMN_MC_to_spec(ml_right(ml_down(ml))));
    spec_set_argument(spec, 2,
        VAL_subtree_to_spec(ml_right(ml_right(ml_down(ml)))));
    spec_set_argument(spec, 3,
        VAL_subtree_to_spec(ml_right(ml_right(ml_right(ml_down(ml))))));
    return spec;
}
```

§**37.** TR_ENTRY_PRODUCTION.

```
specification *TR_ENTRY_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(TR_ENTRY_PRODUCTION);
    spec = new_TABLE_ENTRY_spec();
    spec_set_argument(spec, 0, TABLE_COLUMN_MC_to_spec(ml_down(ml)));
    if ((do_we_need_ct_locals() == FALSE) &&
        (problem_count == 0)) {
        sentence_problem(_P_(C5NoRowSelected),
            "no row seems to have been chosen at this point",
            "so it doesn't make sense to talk about the entries "
            "within it. (By 'at this point', I mean the point "
            "when the table will have to be looked at. This "
            "might be at another time altogether if we are "
            "storing away instructions for later in a text "
            "substitution, e.g., writing 'now the description "
            "of the player is \"Thoroughly [vanity entry].\";' "
            "- remember that the substitution is acted on "
            "when the text is printed, which could be at any "
            "time, and no row will be chosen then.)");
    }
    return spec;
}
```

§**38.** TR_IN_ROW_PRODUCTION.

```
specification *TR_IN_ROW_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(TR_IN_ROW_PRODUCTION);
    spec = new_TABLE_ENTRY_spec();
    spec_set_argument(spec, 0, TABLE_COLUMN_MC_to_spec(ml_down(ml)));
    spec_set_argument(spec, 1, VAL_subtree_to_spec(ml_right(ml_down(ml))));
    spec_set_argument(spec, 2, VAL_subtree_to_spec(ml_right(ml_right(ml_down(ml)))));
    return spec;
}
```

§**39.** TR_LISTED_IN_PRODUCTION.

```
specification *TR_LISTED_IN_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(TR_LISTED_IN_PRODUCTION);
    spec = new_TABLE_ENTRY_spec();
    spec_set_argument(spec, 0, TABLE_COLUMN_MC_to_spec(ml_down(ml)));
    spec_set_argument(spec, 1, VAL_subtree_to_spec(ml_right(ml_down(ml))));
    return spec;
}
```

§**40.** TR_OF_IN_PRODUCTION.

```
specification *TR_OF_IN_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(TR_OF_IN_PRODUCTION);
    spec = new_TABLE_ENTRY_spec();
    spec_set_argument(spec, 0, TABLE_COLUMN_MC_to_spec(ml_down(ml)));
    spec_set_argument(spec, 1, TABLE_COLUMN_MC_to_spec(ml_down(ml)));
    spec_set_argument(spec, 2, VAL_subtree_to_spec(ml_right(ml_down(ml))));
    spec_set_argument(spec, 3, VAL_subtree_to_spec(ml_right(ml_right(ml_down(ml)))));
    return spec;
}
```

§**41.** TYPE_PRODUCTION.

```
specification *TYPE_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(TYPE_PRODUCTION);
    spec = ml_get_attached_spec(ml);
    return spec;
}
```

§**42.** VAL_LIST_ENTRY_PRODUCTION.

```
specification *VAL_LIST_ENTRY_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(VAL_LIST_ENTRY_PRODUCTION);
    spec = new_LIST_ENTRY_spec(
        VAL_subtree_to_spec(ml_down(ml)),
        VAL_subtree_to_spec(ml_right(ml_down(ml))));
    return spec;
}
```

§**43.** VAL_NOTHING_PRODUCTION.

```
specification *VAL_NOTHING_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(VAL_NOTHING_PRODUCTION);
    spec = new_nothing_object_constant();
    return spec;
}
```

§**44.** VAL_PROP_OF_PRODUCTION.

```
specification *VAL_PROP_OF_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING_SUBTREE(VAL_PROP_OF_PRODUCTION);
    if (ml_right(ml_down(ml))) {
        int v1, v2, p1, p2;
        specification *pts =
            PROPERTY_MC_to_spec(ml_down(ml));
        specification *vts =
            VAL_subtree_to_spec(ml_right(ml_down(ml)));
        ml_get_text(ml_down(ml), &p1, &p2);
        ml_get_text(ml_right(ml_down(ml)), &v1, &v2);
        spec = new_PROPERTY_VALUE_spec(pts, vts);
        if ((p1 >= 0) && (v1 >= 0)) {
            spec->word_ref1 = p1;
            if (spec->word_ref1 > v1) spec->word_ref1 = v1;
            spec->word_ref2 = p2;
            if (spec->word_ref2 < v2) spec->word_ref2 = v2;
        }
    } else {
        spec = PROPERTY_MC_to_spec(ml_down(ml));
    }
    return spec;
}
```

§**45.** VALUE_PHRASE_PRODUCTION.

```
specification *VALUE_PHRASE_subtree_to_spec(meaning_list *ml) {
    NOW_CONVERTING(VALUE_PHRASE_PRODUCTION);
    spec = general_phrase_to_spec(ml);
    return spec;
}
```

§**46. Level III: Routines for each possible excerpt-meaning leaf.** These routines are simple matters of copying a pointer from one husk of a data structure (a meaning-list node) to another (an actual constant type specification), so in most cases there's nothing to see. They are again in alphabetical order.

```
specification *ACTION_NAME_SMC_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(MISCELLANEOUS_MC);
    action_name *an = RETRIEVE_POINTER_action_name(em_data(ml_meaning(ml)));
    spec = action_name_to_ACTION_NAME_spec(an);
    return spec;
}
specification *ACTIVITY_MC_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(ACTIVITY_MC);
    activity *av = RETRIEVE_POINTER_activity(em_data(ml_meaning(ml)));
    spec = activity_to_ACTIVITY_spec(av);
    return spec;
}
specification *EQUATION_MC_to_spec(meaning_list *ml) {
    specification *spec;
    int w1, w2, named = TRUE;
    equation *eqn = NULL;
    if (ml_production(ml) == EQUATION_MC) {
        eqn = RETRIEVE_POINTER_equation(em_data(ml_meaning(ml)));
    } else {
        ml_get_text(ml, &w1, &w2);
        eqn = new_equation(w1, w2, TRUE);
        named = FALSE;
        LOG("Manufactured new equation from $W\n", w1, w2);
    }
    ml = ml_right(ml);
    if (ml) {
        ml_get_text(ml, &w1, &w2);
        if (named == FALSE) eqn_set_wherewithal(eqn, w1, w2);
        else eqn_set_usage_notes(eqn, w1, w2);
    }
    if (named == FALSE) {
        equation_declare_local_variables(eqn);
        examine_equation(eqn);
    }
    spec = equation_to_EQUATION_spec(eqn);
    return spec;
}
specification *PROPERTY_MC_to_spec(meaning_list *ml) {
    RECORD_MLC_BACKTRACE;
    if (ml_production(ml) == UNPARSED_PRODUCTION) {
        int w1, w2;
        specification *spec = new_UNKNOWN_spec();
        ml_get_text(ml, &w1, &w2);
        spec->word_ref1 = w1; spec->word_ref2 = w2;
        return spec;
    }
    property_name *prn = RETRIEVE_POINTER_property_name(em_data(ml_meaning(ml)));
    return property_name_to_PROPERTY_spec(prn);
}
```

```
specification *QUANTITY_MC_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(QUANTITY_MC);
    quantity *q = RETRIEVE_POINTER_quantity(em_data(ml_meaning(ml)));
    spec = new_QUANTITY_spec(q);
    return spec;
}
specification *RELATION_SMC_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(MISCELLANEOUS_MC);
    binary_predicate *bp = RETRIEVE_POINTER_binary_predicate(em_data(ml_meaning(ml)));
    spec = binary_predicate_to_RELATION_spec(bp);
    return spec;
}
specification *RULE_MC_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(RULE_MC);
    booked_rule *br = RETRIEVE_POINTER_booked_rule(em_data(ml_meaning(ml)));
    spec = booked_rule_to_RULE_spec(br);
    return spec;
}
specification *RULEBOOK_MC_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(RULEBOOK_MC);
    rulebook *rb = RETRIEVE_POINTER_rulebook(em_data(ml_meaning(ml)));
    spec = rulebook_to_RULEBOOK_spec(rb);
    return spec;
}
specification *RULE_OUTCOME_SMC_to_spec(meaning_list *ml, int as_value) {
    NOW_CONVERTING_LEAF(MISCELLANEOUS_MC);
    named_rulebook_outcome *nro =
        RETRIEVE_POINTER_named_rulebook_outcome(em_data(ml_meaning(ml)));
    if (as_value) spec = named_rulebook_outcome_to_RULEBOOK_OUTCOME_spec(nro);
    else spec = new_RULEBOOK_OUTCOME_COMMAND_spec(nro);
    return spec;
}
specification *TABLE_MC_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(TABLE_MC);
    table *t = RETRIEVE_POINTER_table(em_data(ml_meaning(ml)));
    spec = table_to_TABLE_spec(t);
    return spec;
}
specification *TABLE_COLUMN_MC_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(TABLE_COLUMN_MC);
    table_column *tc = RETRIEVE_POINTER_table_column(em_data(ml_meaning(ml)));
    spec = table_column_to_TABLE_COLUMN_spec(tc);
    return spec;
}
specification *USE_OPTION_SMC_to_spec(meaning_list *ml) {
    NOW_CONVERTING_LEAF(MISCELLANEOUS_MC);
    use_option *uo = RETRIEVE_POINTER_use_option(em_data(ml_meaning(ml)));
    spec = use_option_to_USEOPTION_spec(uo);
    return spec;
}
specification *WORLD_OBJECT_MC_to_spec(meaning_list *ml) {
```

```
    NOW_CONVERTING_LEAF(WORLD_OBJECT_MC);
    world_object *wo = disambiguate_world_object(ml, FALSE);
    spec = world_object_to_OBJECT_spec(wo);
    return spec;
}
```

§**47. Level IV: Support code for difficult cases.** Firstly, a number of productions above expect to see a type expression which describes a kind of value (KOV) which will be the contents of a variable.

```
kind_of_value *police_te(meaning_list *ml) {
    kind_of_value *kov = NULL;
    if (ml != NULL) {
        specification *spec = TE_subtree_to_spec(ml);
        if (spec_is_UNKNOWN(spec)) return NULL;
        if (species_is(spec, DESCRIPTION_SPC)) {
            if (spec_get_described_object(spec) != NULL)
                ⟨Give up with a problem message because what must vary is constant 48⟩;
            if ((number_of_adjectives_applied_to(spec) > 0) || (spec_get_described_kind(spec) == NULL))
                ⟨Give up with a problem message because what must vary is not maintainable 49⟩;
            kov = kovko(spec_get_described_kind(spec));
        } else {
            if (spec_is_actual(spec))
                ⟨Give up with a problem message because what must vary is constant 48⟩;
            if (spec_get_kind_of_value(spec) == NULL)
                ⟨Give up with a problem message because variables can't have this KOV 50⟩;
            kov = spec_get_kind_of_value(spec);
        }
    }
    return kov;
}
```

§**48.** For instance, "X is a 54 that varies."

⟨Give up with a problem message because what must vary is constant 48⟩ ≡

```
    quote_source(1, current_sentence);
    handmade_problem(_P_(C5TypeCantVary));
    issue_problem_segment(
        "In %1, '%9' is a contradiction in terms, as this is something that "
        "cannot ever vary.");
    issue_problem_end();
    return kova(OBJECT_TY);                                    to prevent further errors
```

This code is used in §47.

§**49.** For instance, "X is an open door that varies."

⟨Give up with a problem message because what must vary is not maintainable 49⟩ ≡

```
quote_source(1, current_sentence);
handmade_problem(_P_(C5TypeUnmaintainable));
issue_problem_segment(
    "In %1, '%9' is not a kind of value which a variable can safely have, "
    "as it cannot be guaranteed that the contents will always meet "
    "this criterion.");
issue_problem_end();
return kova(OBJECT_TY);                                    to prevent further errors
```

This code is used in §47.

§**50.** For instance, "X is a condition that varies."

⟨Give up with a problem message because variables can't have this KOV 50⟩ ≡

```
handmade_problem(_P_(C5BadVariableKind1));
quote_source(1, current_sentence);
issue_problem_segment(
    "In %1, '%9' is not a kind of value which a variable is allowed to "
    "have. (See the table at the end of the Kinds index for a list of "
    "the kinds allowed.)");
issue_problem_end();
return kova(NUMBER_TY);                                    to prevent further errors
```

This code is used in §47.

§**51.** Secondly, there are three basic kinds of phrase: those used as commands (i.e., void procedures in C terms), those used as values (returning values other than true/false) and those used as conditions (returning true or false). These are stored in a way making basically the same use of a specification's references, so all three are handled by the following code.

The usage of a phrase is called an "invocation" of it, and sometimes more than one invocation appears, for two reasons: a "say" phrase can contain a sequence of invocations to follow, one after another; and sometimes it will only be clear at run-time which of several possible definitions is to apply, so the possibilities will all be invoked.

```
specification *general_phrase_to_spec(meaning_list *ml) {
    int instead_flag = FALSE, group_count = 0, mw1, mw2;
    specification *spec;
    RECORD_MLC_BACKTRACE;
    ml_get_text(ml, &mw1, &mw2);
    switch(ml_production(ml)) {
        case PHRASE_PRODUCTION: spec = new_TO_PHRASE_spec(); break;
        case COND_PHRASE_PRODUCTION: spec = new_PHRASE_TO_DECIDE_IF_spec(); break;
        case VALUE_PHRASE_PRODUCTION: spec = new_PHRASE_TO_DECIDE_VALUE_spec(); break;
        default: CHILD_UNCONVERTIBLE(TE_PRODUCTION);
    }
    if (mw1 >= 0) { spec->word_ref1 = mw1; spec->word_ref2 = mw2; }
    ml = ml_down(ml);
    if (ml_production(ml) == INSTEAD_PRODUCTION) { instead_flag = TRUE; ml = ml_down(ml); }
    switch(ml_production(ml)) {
        case SAY_PRODUCTION: ⟨Build the invocation list for a say phrase 53⟩; break;
```

```
        case END_PHRASE_MC: ⟨Make an end phrase, with no invocation list 56⟩; break;
        default: ⟨Build the invocation list for a non-say phrase 52⟩; break;
    }
    int len = length_of_invocation_list(spec_invocation_list(spec));
    if (len >= MAX_INVOCATIONS_PER_PHRASE)
        ⟨Issue overcomplicated phrase problem message 58⟩;
    if (len > 0) {
        ⟨Set the instead flag on invocations in the final group 57⟩;
        sort_spec_invocation_list(spec_invocation_list(spec));
    }
    return spec;
}
```

§**52.** Here there is just one invocation group, but it may contain multiple invocations, each produced from another node in the meaning list tree as we run sideways through the alternative readings.

⟨Build the invocation list for a non-say phrase 52⟩ ≡

```
    for (; ml; ml = ml_sideways(ml)) {
        phrase *ph;
        if (ml_meaning(ml) == NULL) internal_error("Blank phrase meaning");
        ph = RETRIEVE_POINTER_phrase(em_data(ml_meaning(ml)));
        if (parse_against_PHTD(spec, ph, ml) == FALSE) {
            coerce_spec_to_UNKNOWN(spec);
            break;
        }
    }
    group_count = 1;                                    since just one group was added
```

This code is used in §51.

§**53.** The case of a say phrase is more interesting, because now there are multiple invocation groups, identified by numbers counting upwards from 0. Each group corresponds to a rightwards sibling in the meaning tree; each of the nodes can have sideways siblings too, though, which is why each group can have multiple invocations in it, one for each sideways alternative.

We convert each group in succession.

⟨Build the invocation list for a say phrase 53⟩ ≡

```
    for (ml=ml_down(ml); ml; group_count++, ml=ml_right(ml)) {
        specification* this_group = new_TO_PHRASE_spec();
        ⟨Fill references with the single invocation group at ML 54⟩;
        ⟨Copy the new group's references to the tail of SP, setting their group numbers 55⟩;
    }
```

This code is used in §51.

§**54.** Each single group is handled much as in the "non-say" case.

⟨Fill references with the single invocation group at ML 54⟩ ≡
```
meaning_list *ml2;
for (ml2 = ml; ml2; ml2 = ml_sideways(ml2)) {
    phrase *ph = RETRIEVE_POINTER_phrase(em_data(ml_meaning(ml2)));
    if (parse_against_PHTD(this_group, ph, ml2) == FALSE) {
        coerce_spec_to_UNKNOWN(this_group);
        break;
    }
}
```

This code is used in §53.

§**55.**

⟨Copy the new group's references to the tail of SP, setting their group numbers 55⟩ ≡
```
INVOCATION_VARIABLE(inv);
LOOP_THROUGH_INVOCATION_LIST(inv, spec_invocation_list(this_group))
    inv_set_group(inv, group_count);
append_invocation_list(spec_invocation_list(spec), spec_invocation_list(this_group));
```

This code is used in §53.

§**56.** An end phrase, like "end if", carries with it a reference to what is ending ("if", in this case).

⟨Make an end phrase, with no invocation list 56⟩ ≡
```
spec = new_END_BLOCK_spec(RETRIEVE_POINTER_phrase(em_data(ml_meaning(ml))));
```

This code is used in §51.

§**57.** It's important to set the instead flag only in the final group of invocations, or else (if set) it will prevent later groups from having any effect. Consider:

```
instead say "The score is ", score, "."
```

The effect of "instead" is to halt the current rule, so if the instead flag were set for every invocation then the rule would halt after only "The score is " has been printed (that being the effect of group 0: groups 1 and 2 would have no effect). Instead, in this example, only group 2 is flagged.

⟨Set the instead flag on invocations in the final group 57⟩ ≡
```
INVOCATION_VARIABLE(inv);
LOOP_THROUGH_INVOCATION_LIST(inv, spec_invocation_list(spec))
    if (inv_get_group(inv) == group_count-1)
        inv_set_instead_flag(inv, instead_flag);
```

This code is used in §51.

§**58.** This is in practice only ever experienced for long say phrases in a situation where many kinds of value have been created, so that "say V" for a value V is heavily ambiguous – pumping up the number of invocations generated. The limit is intentionally set high, but one day it should perhaps be removed altogether.

⟨Issue overcomplicated phrase problem message 58⟩ ≡

```
    log_invocation_list(spec_invocation_list(spec));
    coerce_spec_to_UNKNOWN(spec);
    spec->word_ref1 = current_sentence->word_ref1; spec->word_ref2 = current_sentence->word_ref2;
    quote_source(1, current_sentence);
    handmade_problem(_P_(C5OverlongSay));
    issue_problem_segment(
        "In %1, the phrase being constructed is just too "
        "long and complicated, and will need to be simplified. (This "
        "sometimes happens with a 'say', or a piece of text, containing "
        "many text substitutions in succession: if so, it may be worth "
        "defining some more powerful text substitutions - for instance "
        "writing 'To say super-duper: ...', giving the gory details, "
        "and then using the single substitution '[super-duper]' in the "
        "original phrase.");
    issue_problem_end();
    return spec;
```

This code is used in §51.

§**59.** Various forms of the DC conversion boil down to this: we have one or both of a noun (say "door") and an adjective list qualifying it (say "open fixed in place"). We need to combine these into a single specification.

```
specification *qualified_noun_to_spec(meaning_list *adjlist, meaning_list *noun) {
    specification *spec = NULL;
    if (noun) {
        switch (ml_production(noun)) {
            case WORLD_OBJECT_MC:
                ⟨Create type for a physical headword 60⟩;
                break;
            case TYPE_PRODUCTION:
                ⟨Create type for an abstract headword 61⟩;
                break;
            default: internal_error("Unknown noun production type");
        }
    } else spec = new_DESCRIPTION_spec();
    if (adjlist) ⟨Incorporate the adjectives from the list 62⟩;
    return spec;
}
```

§**60.** The noun is usually going to be a common noun (a kind or kind of value), but not always, because Inform allows proper nouns which represent objects to be qualified by adjectives ("the open Marble Door"). Note that we treat both of the possible noun productions differently from their treatment as values, though in the case of a specific object name unqualified by adjectives (just "Marble Door", say) the result does agree and is not a `DESCRIPTION` spec.

⟨Create type for a physical headword 60⟩ ≡

```
world_object *wo = disambiguate_world_object(noun, FALSE);
if (wo == NULL) internal_error("no WO follows disambiguation");
else if (wo->kind_flag) {
    spec = new_DESCRIPTION_spec();
    spec_set_described_kind(spec, wo);
} else if (adjlist) {
    spec = new_DESCRIPTION_spec();
    spec_set_described_object(spec, wo);
} else
    spec = world_object_to_OBJECT_spec(wo);
```

This code is used in §59.

§**61.** The abstract value case is simpler, since here only kinds of value are allowed (and there are no proper nouns).

⟨Create type for an abstract headword 61⟩ ≡

```
kind_of_value *kov = spec_get_kind_of_value(ml_get_attached_spec(noun));
spec = new_DESCRIPTION_spec();
spec_set_described_kov(spec, kov);
```

This code is used in §59.

§**62.**

⟨Incorporate the adjectives from the list 62⟩ ≡

```
meaning_list *ml = adjlist;
int mw1, mw2; ml_get_text(ml, &mw1, &mw2);
if (ml_production(ml) != AL_PRODUCTION) internal_error("Non-AL production");
ml = ml_down(ml);
if (ml == NULL) internal_error("Empty AL production");
while (ml) {
    adjective_list_entry *ale = adj_to_adjective_list_entry(ml);
    add_to_adjective_list(ale, spec);
    ml = ml_right(ml);
}
```

This code is used in §59.

§**63.** A utility to convert what is either an adjective node, or a negation marker with its adjective node as only child, into an entry suitable for a SP's list.

```
adjective_list_entry *adj_to_adjective_list_entry(meaning_list *ml) {
    adjective_list_entry *ale = NULL;
    int negate = FALSE;
    meaning_list *ml2 = ml;
    RECORD_MLC_BACKTRACE;
    RetrySwitch:
    switch(ml_production(ml2)) {
        case ADJ_NOT_PRODUCTION:
            ml2 = ml_down(ml); negate = TRUE;
            goto RetrySwitch;
        case ADJECTIVE_MC:
            ale = new_adjective_list_entry(
                RETRIEVE_POINTER_adjectival_phrase(em_data(ml_meaning(ml2))),
                    (negate)?FALSE:TRUE);
            break;
        default: internal_error("Unexpected AL production");
    }
    return ale;
}
```

§**64.** The last difficult case to handle is that of sentence-like constructions.

```
specification *general_sentence_to_spec(meaning_list *ml) {
    specification *spec;
    meaning_list *subject_noun_phrase = NULL, *verb_phrase = NULL;
    verb_usage *vu = NULL; preposition_usage *pu = NULL;
    int mw1, mw2; ml_get_text(ml, &mw1, &mw2);
    RECORD_MLC_BACKTRACE; s_subtree_error_set_position(ml);
    LOGIF(S_GRAMMAR, "general_sentence_to_spec applied to:\n$m", ml);
    ⟨Reconstruct a bare description as a sentence with an implied absent subject 65⟩;
    ⟨Check that the top structure of the tree is in order, and obtain the verb 66⟩;
    switch(ml_production(ml)) {
        case SN_PRODUCTION:
            spec = new_DESCRIPTION_spec();
            spec_set_proposition(spec, S_subtree_to_proposition(ml, NULL));
            s_subtree_accumulate_headword(spec, ml);
            break;
        case SV_PRODUCTION:
            spec = new_TEST_PROPOSITION_spec(S_subtree_to_proposition(ml, NULL));
            break;
        default: CHILD_UNCONVERTIBLE(VAL_PRODUCTION);
    }
    if (mw1 >= 0) { spec->word_ref1 = mw1; spec->word_ref2 = mw2; }
    if (vu_get_tense_used(vu) != IS_TENSE) ⟨Attach tense marker to the SP 67⟩;
    return spec;
}
```

§**65.** For instance, "if an open door" can contain a valid condition generating an SV-subtree: the implied subject is whatever is being discussed (the I6 `self` object, in practice) and the implied verb is "is", in the present tense.

⟨Reconstruct a bare description as a sentence with an implied absent subject 65⟩ ≡

```
if (ml_right(ml_down(ml)) == NULL) {
    switch(ml_production(ml_down(ml))) {
        case DC_PRODUCTION: {
            meaning_list *ml2 = ml_down(ml);
            ml_set_down(ml, new_ml(NP_PRODUCTION));
            ml_set_down(ml_down(ml), new_ml(ABSENT_SUBJECT_PRODUCTION));
            ml_set_right(ml_down(ml), new_ml(VP_PRODUCTION));
            verb_phrase = ml_right(ml_down(ml));
            ml_set_down(verb_phrase, new_ml(VERB_PRODUCTION));
            ml_set_right(ml_down(verb_phrase), ml2);
            ml_attach_data(ml_down(verb_phrase),
                STORE_POINTER_verb_usage(regular_to_be));
            ml_set_production(ml, SV_PRODUCTION);
            correct_S_subtree_for_adjectives(ml_down(ml));
            break;
        }
        default:
            s_subtree_error("only one child with no assumed subject");
    }
}
```

This code is used in §64.

§**66.** Having performed that manoeuvre, we can be certain that the top of the tree has the standard form, but we check it anyway.

⟨Check that the top structure of the tree is in order, and obtain the verb 66⟩ ≡

```
subject_noun_phrase = ml_down(ml);
verb_phrase = ml_right(ml_down(ml));
if (ml_down(verb_phrase) == NULL) s_subtree_error("VP childless");
if (ml_right(ml_down(verb_phrase)) == NULL)
    s_subtree_error("VP only one child");
if (ml_production(verb_phrase) != VP_PRODUCTION)
    s_subtree_error("VP not a VP");
if (ml_production(ml_down(verb_phrase)) != VERB_PRODUCTION)
    s_subtree_error("child of VP not a verb");
vu = RETRIEVE_POINTER_verb_usage(ml_get_attached_data(ml_down(verb_phrase)));
if (vu == NULL) s_subtree_error("verb null");
if (ml_down(ml_down(verb_phrase))) {
    if (ml_production(ml_down(ml_down(verb_phrase))) != PREP_PRODUCTION)
        s_subtree_error("child of VERB not a PREP");
    pu = RETRIEVE_POINTER_preposition_usage(
        ml_get_attached_data(ml_down(ml_down(verb_phrase))));
    if (pu == NULL) s_subtree_error("PREP null");
}
```

This code is used in §64.

§**67.** It's in order to detect past tense usage that we needed to extract the VU above.

⟨Attach tense marker to the SP 67⟩ ≡
```
    if (ml_production(ml) == SN_PRODUCTION) {
        sentence_problem(_P_(C5DescSubordinatePast),
            "subordinate clauses have to be in the present tense",
            "so 'the Black Door was open' is fine, but not 'something which "
            "was open'. Only the main verb can be in the past tense.");
    } else if (prop_detect_locals(spec_get_proposition(spec)) > 0) {
        sentence_problem(_P_(C5DescLocalPast),
            "conditions written in the past tense cannot refer to "
            "temporary values",
            "because they have no past. For instance, the name given in a "
            "'repeat...' can't be talked about as having existed before.");
    } else spec_attach_tense(spec, vu_get_tense_used(vu));
```
This code is used in §64.

§**68.** The idea of "accumulation" is to extract the maximum information we can about the headword noun to which a subtree refers – what its kind is, and what adjectives it satisfies. We will never get all of the information this way – "the open containers which have been in lighted rooms" cannot be summed up so simply, and has to be accumulated merely as "open containers". But we want to extract the maximum amount that we can, since this improves the fine sorting of rules by applicability.

```
void s_subtree_accumulate_headword(specification *spec,
    meaning_list *s_ml) {
    world_object *subtree_wo;
    specification *subtree_spec;
    LOGIF(S_GRAMMAR, "s_subtree_accumulate_headword applied to:\n$m", s_ml);
    if (ml_production(s_ml) == DC_PRODUCTION)
        subtree_spec = DC_subtree_to_spec(ml_down(s_ml), FALSE);
    else
        subtree_spec = NP_subtree_to_spec(ml_down(s_ml));
    subtree_wo = wo_of_CONSTANT_OBJECT_if_any(subtree_spec);
    if (subtree_wo) {
        restrict_DESCRIPTION_domain(spec, subtree_wo);
        ⟨Also accumulate descriptive matter from a relative clause applying to this 70⟩;
        return;
    }
    if (species_is(subtree_spec, DESCRIPTION_SPC)) {
        ⟨Copy all references over from the subtree SP to the headword SP 69⟩;
        if (spec_get_described_object(subtree_spec))
            restrict_DESCRIPTION_domain(spec, spec_get_described_object(subtree_spec));
        else if (spec_get_described_kind(subtree_spec))
            restrict_DESCRIPTION_domain(spec, spec_get_described_kind(subtree_spec));
        ⟨Also accumulate descriptive matter from a relative clause applying to this 70⟩;
        return;
    }
    if ((species_is(subtree_spec, CONSTANT_SPC)) ||
        (spec_get_storage_form(subtree_spec) == LOCAL_VARIABLE_SPC) ||
        (spec_get_storage_form(subtree_spec) == NONLOCAL_VARIABLE_SPC))
        return;
    coerce_spec_to_UNKNOWN(spec);
}
```

§**69.** These references all represent adjectives which the subtree SP is required to satisfy, so we copy them over to the SP being accumulated.

⟨Copy all references over from the subtree SP to the headword SP 69⟩ ≡
```
adjective_list_entry *tr_from;
LOOP_THROUGH_ADJECTIVE_LIST(tr_from, subtree_spec) {
    adjective_list_entry *tr_new = copy_adjective_list_entry(tr_from);
    add_to_adjective_list(tr_new, spec);
}
```

This code is used in §68.

§**70.** We use recursion to continue accumulating anything we can find in a relative clause, provided it is connected by the copular verb and in the present tense. The effect is that, e.g., "containers which are open" can be simplified into the simpler description "open containers": likewise "things which are people who are women" would collapse to "people who are women" and then "women", the noun at the head specialising from things to people to women.

⟨Also accumulate descriptive matter from a relative clause applying to this 70⟩ ≡
```
if (ml_production(s_ml) == SN_PRODUCTION) {
    verb_usage *vu;
    meaning_list *verb_phrase = ml_right(ml_down(s_ml));
    if (verb_phrase == NULL) s_subtree_error("no VP");
    if (ml_down(verb_phrase) == NULL) s_subtree_error("VP childless");
    if (ml_right(ml_down(verb_phrase)) == NULL)
        s_subtree_error("VP only one child");
    if (ml_production(verb_phrase) != VP_PRODUCTION)
        s_subtree_error("VP not a VP");
    if (ml_production(ml_down(verb_phrase)) != VERB_PRODUCTION)
        s_subtree_error("child of VP not a verb");
    vu = RETRIEVE_POINTER_verb_usage(ml_get_attached_data(ml_down(verb_phrase)));
    if (vu == NULL) s_subtree_error("verb null");
    if ((vu_get_meaning(vu) == a_is_b_predicate) &&
        (ml_down(ml_down(verb_phrase)) == NULL) &&
        (vu_get_tense_used(vu) == IS_TENSE)) {
        s_subtree_accumulate_headword(spec, ml_right(ml_down(verb_phrase)));
    }
}
```

This code is used in §68.

§**71.** That completes the interface layer handling meaning list conversions, except for the difficult part: the routine `S_subtree_to_proposition`, found in Chapter 6, which translates a sentence subtree to a proposition. It is time to leave the world of parsing behind for a while, and enter the less textual domain of predicate calculus.

6 Predicate Calculus

`6/iprc`: *Introduction to Predicate Calculus.w* An exposition of the form of predicate calculus used by Inform.

`6/term`: *Terms.w* Terms are the representations of values in predicate calculus: variables, constants or functions of other terms.

`6/aprop`: *Atomic Propositions.w* To build and modify atoms, the syntactic pieces from which propositions are built up.

`6/prop`: *Propositions.w* To build and modify structures representing propositions in predicate calculus.

`6/vars`: *Binding and Substitution.w* To substitute constants into propositions in place of variables, and to apply quantifiers to bind any unbound variables.

`6/pform`: *Propositional Form.w* The first of the three sources of propositions to conjure with: those which arise by conversion of existing data, in the form of type specifications.

`6/treec`: *Tree Conversions.w* The second of the three sources of propositions to conjure with: those which arise from subtrees constructed by the A-parser.

`6/sconv`: *Sentence Conversions.w* The third of the three sources of propositions to conjure with: those which arise by the parsing of complex sentence trees in the S-grammar.

`6/simp`: *Simplifications.w* A set of operations each of which takes a proposition and either leaves it unchanged or replaces it with a simpler one logically equivalent to the original.

`6/tcpr`: *Type Check Propositions.w* Predicate calculus is a largely symbolic exercise, and its rules of working tend to assume that all predicates are meaningful for all terms: this means, for instance, that "if blue is 14" is likely to make a well-formed sentence in predicate calculus. In this section we reject such propositions on the grounds that they violate type-checking requirements on relations – in this example, the equality relation.

`6/equal`: *The Equality Relation.w* To define that prince among predicates, the equality relation.

`6/asp`: *Assert Propositions.w* To declare that a given proposition is a true statement about the state of the world when play begins.

`6/sch`: *I6 Schemas.w* To create, and later expand upon, short prototypes of I6 syntax for such run-time tasks as the setting, unsetting or testing of a relation.

`6/atoms`: *Compile Atoms.w* In this section, given an atom of a proposition we compile I6 code as required for any of three possible outcomes: (i) to test whether it is true, (ii) to make it henceforth true, or (iii) to make it henceforth false.

`6/defer`: *Deciding to Defer.w* To decide whether a proposition can be compiled immediately, in the body of the current routine, or whether it must be deferred to a routine of its own, which is called from the current routine.

`6/cind`: *Cinders and Deferrals.w* To compile terms, having carefully preserved any constants which might have been lost in the process of deferring a proposition (such tricky constants being called "cinders").

`6/cdefp`: *Compile Deferred Propositions.w* To compile the I6 routines needed to perform the tests or tasks deferred as being too difficult in their original contexts.

Introduction to Predicate Calculus 6/iprc

Purpose

An exposition of the form of predicate calculus used by Inform.

---

§**1.** "Again and again Haddon thought he had found the key to the strange writings, but always he was disappointed. And then one day – he was an old man of seventy now – he fed a trial programme into his computer, and for the first time a translated sentence was delivered – his life-long task was rewarded. Yes, but for the fact that one man had been prepared to devote every spare hour of his life to solving the riddle, the amazing story of the Trigan Empire would never have been given to the world. WHAT FOLLOWS IS THAT STORY." (*The Rise and Fall of the Trigan Empire*, 1965)

§**2. Why predicate calculus.** Most attempts to codify the meaning of sentences in any systematic way involve predicate calculus, and most people generally seem to agree that linguistic concepts (like verbs, adjectives, and determiners) correspond uncannily well with logical ones (like binary predicates, unary predicates, and quantifiers). Since today's mathematical logic has its roots in work on the philosophy of language (chiefly by Frege and Wittgenstein), this is not altogether a coincidence. All the same, it is striking how good the fit is, considering that human language is so haphazard and logic so regular.

At any rate Inform goes along with this consensus, and converts the difficult passages in its source text into mathematical propositions – lines written in logical notation. This is useful partly as a well-defined way to store complicated meanings inside the program, but also because these propositions can then be simplified by logical rules. We can change their form so long as we do not change their meaning, in the hope of finding ways to carry out the same tasks but more efficiently than a simple-minded reading of the text would suggest.

There are four main tasks to perform:

(a) Building propositions from the other Inform data structures;

(b) Simplifying, rearranging and type-checking propositions;

(c) Asserting that certain propositions are true at the start of play;

(d) Compiling certain propositions to I6 code which can test them, make them true, or make them false.

In this chapter, we take these tasks in order. Because it contains all of Inform's predicate calculus lore in one place, it necessarily contains little pieces of algorithms from other chapters: a piece of the type-checker, a piece of the code for asserting the initial state of play, and so on. Well: but the overlap had to go somewhere.

§**3.** A glimpse of Inform's inner workings can be made by writing a test instruction like so:

```
Laboratory is a room. The box is a container.

Test sentence (internal) with a man can see the box in the Laboratory.
```

At the start of play, the compiled story file will print a textual form of the proposition in predicate calculus which that became:

```
1. a man can see the box in the Laboratory
[ Exists x : man(x) & thing('box') & is('laboratory', ContainerOf('box')) & can-see(x, 'box') ]
x - object.
```

(The `intest` test case `Calculus` runs about 200 sentences like this.) One can similarly "Test description (internal) with..." for any description, such as "animals which are in lighted rooms".

§**4. Formal description.** There are many flavours of predicate calculus, and though they behave in broadly similar ways, the details vary in practice. Inform's calculus is unusual in several respects, so here is its definition.

The terms "constant", "variable" and "function" below are used in their predicate-calculus senses, *not* their Inform ones. In the excerpt

> a container in the location of Nicole contains the second noun

the text "location of Nicole" is a phrase deciding an object – a function – and "the second noun" is an object that varies – a variable. But if we are looking at the sentence only for its logical meaning, we have to forget about the passage of time and think about what the sentence means at any single moment. "Location of Nicole" and "the second noun" are both fixed, in this instant; the only thing which varies is the identity of the possible container, because we might have to look at hundreds of different containers to see if the sentence is true. One logical translation might be

$$\exists x : \mathit{container}(x) \wedge \mathit{in}(x, C_1) \wedge \mathit{in}(C_2, x)$$

where C_1 and C_2 are constants ("location of Nicole" and "second noun" respectively), while the only variable is x, the mysterious container. (The symbol $\exists$ means "there exists".) Because objects move around in play, and C_1 and C_2 have different values at different moments, this sentence is sometimes true and sometimes false. But its *meaning* does not change.

§**5.** The propositions in our predicate calculus are those which can be made using the following ingredients and recipes.

1. There are 26 variables, which we print to the debugging log as `x`, `y`, `z`, `a`, `b`, `c`, ..., `w`.

2. The constants are specifications with family `VALUE` – that is, all literal constants, variables, list and table entries, or phrases which decide values.

3. A "predicate" P is a statement $P(a, b, c, ...)$ which is either true or false for any given combination $a, b, c, ...$ The "arity" of a predicate is the number of terms it looks at. There is speculative talk of allowing higher-order predicates in future (and Inform's data structures have been built with one eye on this), but for now we use only unary predicates $U(x)$ or binary predicates $B(x, y)$, of arity 1 and 2 respectively. The predicates in our calculus are as follows:

(a) The special binary predicate $\mathit{is}(x, y)$, true if and only if $x = y$.
(b) Every kind K (of value or of object) corresponds to a unary predicate $K(x)$.
(c) Every state of an either/or property corresponds to a unary predicate, e.g., $\mathit{open}(x)$.
(d) Every possible value of an enumerated kind of value which corresponds to a property similarly corresponds to a unary predicate: e.g., if we have defined "colour" as a kind of value and made it a property of things, then $\mathit{green}(x)$, $\mathit{red}(x)$, and $\mathit{blue}(x)$ might all be unary predicates.
(e) Every adjectival phrase, to which a definition has been supplied in the source, likewise produces a unary predicate: for example, $\mathit{visible}(x)$.
(f) An adjective given a definition which involves a threshold for a numeric property also produces a binary predicate for its comparative form: for instance, a definition for "tall" gives not only a unary predicate $\mathit{tall}(x)$ (as in (e) above) but also a binary predicate $\mathit{taller}(x, y)$.
(g) A special unary predicate $\mathit{everywhere}(x)$ which asserts that the backdrop x can be found in every room.
(h) Each table column name C gives rise to a binary predicate $\mathit{listed\text{-}in\text{-}C}(x, y)$, which tests whether value x is listed in the C column of table y. (This looks as if it should really be a single ternary predicate, but since we never need to quantify over choice of column, nothing would be gained by that.)
(i) Every value property P gives rise to a binary predicate $\mathit{same\text{-}P\text{-}value}(x, y)$, testing whether objects x and y have the same value of P. (Again, it would not be useful to quantify over P.)
(j) Every direction D gives rise to a binary predicated $\mathit{mapped\text{-}D}(x, y)$, testing whether there is a map connection from x to y in direction D.
(k) Each new relation defined in the source text is a binary predicate.

(l) The basic stock of spatial containment relations built into Inform – $in(x, y)$, $on(x, y)$, etc. – are similarly binary predicates.

(m) If P is a binary predicate present in Inform then so automatically is its "reversal" R, defined by $R(x, y)$ if and only if $P(y, x)$. For instance, the existence of *carries*(x, y) ensures that we also have *carried-by*(x, y), its reversal. The equality predicate $x = y$ is its own reversal, but all other binary predicates are formally different from their reversals, even if they always mean the same in practice. (The reversal of *same-carrying-capacity-as*(x, y) is true if and only if the original is true, but we regard them as different predicates just the same.)

4. If a binary predicate B has the property that for any x there is at most one y such that $B(x, y)$ (for instance, *carried-by* has this property) then we write f_B for the function which maps x to either the unique value y such that $B(x, y)$, or else to a zero value. (In the case where y is an object, we interpret this as "nothing", which for logical purposes is treated as if it were a valid object, so that f_B maps the set of objects to itself.) Another way of saying this is that f_B, if it exists, is defined by:

$$B(x, y) \Leftrightarrow y = f_B(x).$$

These are the only functions allowed in our predicate calculus, and they are always functions of just one variable.

5. A "quantifier" Qx is a logical expression for the range of values of a given variable x: for instance, $\forall x$ (read "for all x") implies that x can have any value, whereas $\exists x$ (read "there exists an x") means only that at least one value works for x. In our calculus, we allow not only these quantifiers but also the following generalised quantifiers, where n is a non-negative integer:

(a) The quantifier $V_{=n}x$ – meaning "for exactly n values of x".
(b) The quantifier $V_{\geq n}x$ – meaning "for at least n values of x".
(c) The quantifier $V_{\leq n}x$ – meaning "for at most n values of x".
(d) The quantifier $P_{\geq n}x$ – meaning "for at least a percentage of n values of x".
(e) The quantifier $P_{\leq n}x$ – meaning "for at most a percentage of n values of x".

Note that "for all x" corresponds to $P_{\geq 100}x$, and "there exists x" to $V_{\geq 1}x$, so the above scheme does indeed generalise the standard pair of quantifiers $\forall$ and $\exists$.

6. A "term" must be a constant, a variable or a function $f_B(t)$ of another term t. So x, "Miss Marple", $f_B(x)$ and $f_A(f_B(f_C(6)))$ are all examples of terms. We are only allowed to apply functions a finite number of times, so any term has the form:

$$f_{B_1}(f_{B_2}(...f_{B_n}(s)...))$$

for at most a finite number n of function usages (possibly $n = 0$), where at bottom s must be either a constant or a variable.

7. A proposition is defined by the following rules:

(a) The empty expression is a proposition. This is always true, so in these notes it will be written $\top$.
(b) For any unary predicate U and any term t, $U(t)$ is a proposition.
(c) For any binary predicate B and any terms s, t, $B(s, t)$ is a proposition.
(d) For any proposition ϕ, the negation $\neg(\phi)$ is a proposition. This is by definition true if and only if ϕ is false.
(e) For any propositions ϕ and ψ, the conjunction $\phi \wedge \psi$ – true if and only if both are true – is a proposition so long as it is well-formed (see below).
(f) For any variable v, the quantifier $\exists v$ is a proposition.
(g) For any variable v, any quantifier Q other than $\exists$, and any proposition ϕ in which v is a "free" variable (see below), $Qv \in \{v \mid \phi(v)\}$ is a proposition. The set denotes all possible values of v matching the condition $\phi(v)$, and this specifies the range of the quantifier.

§**6.** Note that there are two ways in which propositions can appear in brackets in a bigger proposition: negation (d), and specifying a quantification domain (g). We sometimes call the bracketed part a "subexpression" of the whole.

In particular, note that – unusually – we do not bracket quantification itself. Most definitions would say that given v and a proposition $\phi(v)$, we can form $\exists v : (\phi(v))$ – in other words that quantification is a way to modify an already created proposition, but that $\exists v$ is not a proposition in its own right, just as $\neg$ is not a proposition. Inform disagrees. Here $\exists v$ is a meaningful sentence: it means "an object exists". We can form "there is a door" by rule (e), conjoining $\exists x$ and $door(x)$ to form $\exists x : door(x)$. (As a nod to conventional mathematical notation, we write a colon after a quantifier instead of a conjunction sign $\wedge$. But Inform stores it as just another conjunction.)

We *do* bracket the domain of quantification. Most simple predicate calculuses (predicates calculus?) have no need, since their only quantifiers are $\forall$ and $\exists$, and there is a single universe set from which all values are drawn. But in Inform, some quantifiers range over doors, some over numbers, and so on. In most cases, a quantifier must specify its domain. For example,

$$\forall x \in \{x \mid number(x)\} : even(x)$$

("all numbers are even" – false, of course) specifies the domain of $\forall$ as the set of all x such that $number(x)$. $\exists$ is the one exception to this. The statement

$$\exists x \in \{x \mid number(x)\} : even(x)$$

("a number is even" – true this time) could equally be written

$$\exists x : number(x) \wedge even(x)$$

("there is an even number"). We take advantage of this, and Inform never specifies a domain for a $\exists$ quantifier.

§**7. Free and bound variables, well-formedness.** In any proposition ϕ, we say that a variable v is "bound" if it appears as the variable governed by a quantifier: it is "free" if it does appear somewhere in ϕ – either directly as a term or indirectly through a function application – and is not bound. For instance, in

$$\forall x : K(x) \wedge B(x, f_C(y))$$

the variable x is bound and the variable y is free. In most accounts of predicate calculus we say that a proposition is a "sentence" if all of its variables are bound, but Inform often needs to parse English text to a proposition with one free variable remaining in it, so we are not too picky about this.

A well-formed proposition is one in which a variable v is quantified at most once: and in which, if it is quantified, then it occurs only after (to the right of) its quantifier, and only within the subexpression containing its quantifier. Thus the following are not well-formed:

$$\exists v : open(v) \wedge \exists v : closed(v)$$

(v is quantified twice),

$$open(v) \wedge \exists v : closed(v)$$

(v occurs before its quantifier),

$$\neg(\exists v : closed(v)) \wedge openable(v)$$

(v occurs outside the subexpression containing the quantifier – in this case, outside the negation brackets).

§**8. The scope of quantifiers.** A quantifier introduces a variable into a proposition which would not otherwise be there, and it exists only for a limited range. For instance, in the proposition

$$open(x) \wedge \neg(\exists y : in(x, y)) \wedge container(x)$$

the variable y exists only within the negation brackets; it ceases to exist as soon as we move back out to the *container* atom. This range is called the "scope" of the quantifier. In general, scopes are always as large as possible in Inform: a variable lasts until the end of the subexpression in which it is created. If the quantifier is outside of any brackets, then the variable lasts until the end of the proposition.

§**9.** Earlier drafts of Inform played games with moving quantifiers around, in order to try to achieve more efficiently compiled propositions. The same thing is now done by building propositions in a way which places quantifiers as far forwards as possible, so we no longer actively move them once they are in place. But it seems still worth preserving the rule which says when this can be done:

Lemma. *Suppose that x is a variable; ϕ is a proposition in which x is unused; $\psi(x)$ is a proposition in which x is free; and that Q is a generalised quantifier. Then*

$$\phi \wedge Qx : \psi(x) \quad \Leftrightarrow \quad Qx : \phi \wedge \psi(x)$$

provided that Q requires at least one case in its range to be satisfied.

Proof. In any given evaluation, either ϕ is true, or it is false. Suppose it is true. Since $T \wedge \theta \Leftrightarrow \theta$, both sides reduce to the same expression, $Qx : \psi(x)$. On the other hand, suppose ϕ is false. Then $\phi \wedge Qx : \psi(x)$ is false, since $F \wedge \theta = F$ for any θ. But the other side is $Qx : F$. Since we know that Q can only be satisfied if at least one case of x works, and here every case of x results in falsity, $Qx : F$ is also false. So the two expressions have the same evaluation in this case, too.

§**10. What is not in our calculus.** The significant thing missing is disjunction. In general, the disjunction $\phi \vee \psi$ – "at least one of ϕ and ψ holds" – is not a proposition.

Natural language does not seem to share the general even-handedness of Boolean logic as between "and" and "or", perhaps because of the way that speech is essentially a one-dimensional stream of discourse. Talking makes it easier to reel off a linear list of requirements than to describe a deep tree structure.

Of course, the operations "not" and "and" are between them sufficient to express all other operations, and in particular we could imitate disjunction like so:

$$\neg(\neg(\phi) \wedge \neg(\psi))$$

("they are not both false" being equivalent to "at least one is true"), but Inform does not at present make use of this.

Terms 6/term

Purpose

Terms are the representations of values in predicate calculus: variables, constants or functions of other terms.

Definitions

¶1. Recall that a "term" can be anything which is a constant, can be a variable, or can be a function $f_B(t)$ of another term t. Our representation of this as a data structure therefore falls into three cases. At all times exactly one of the three relevant fields, `variable`, `constant` and `function` is used.

(a) Variables are represented by the numbers 0 to 25, and `-1` means "not a variable".

(b) Constants are pointers to `specification` structures of main type `VALUE`, and `NULL` means "not a constant".

(c) Functions are pointers to `pcalc_func` structures (see below), and `NULL` means "not a function".

Cinders are discussed in "Cinders and Deferrals", and can be ignored for now.

In order to verify that a proposition makes sense and does not mix up incompatible kinds of value, we will need to type-check it, and one part of that involves assigning a kind of value K to every term t occurring in the proposition. This calculation does involve some work, so we cache the result in the `term_checked_as_kov` field, in order that we only have to work it out once.

```
typedef struct pcalc_term {
    int variable;                                    0 to 25, or -1 for "not a variable"
    struct specification *constant;                  or NULL for "not a constant"
    struct pcalc_func *function;                     or NULL for "not a function of another term"
    int cinder;                                      complicated, this: used to worry about scope of I6 local variables
    struct kind_of_value *term_checked_as_kov;       or NULL if unchecked
} pcalc_term;
```

The structure pcalc_term is shared with 6/aprop, 6/vars, 6/sconv, 6/simp, 6/tcpr, 6/equal, 6/asp, 6/sch, 6/atoms, 6/cind, 6/cdefp and 9/provr.

¶2. The `pcalc_func` structure represents a usage of a function inside a term. Terms such as $f_A(f_B(f_C(x)))$ often occur, so a typical term might be stored as

(1) A `pcalc_term` structure which has a `function` field pointing to
(2) A `pcalc_func` structure whose `bp` field points to A, and whose `fn_of` field is
(3) A `pcalc_term` structure which has a `function` field pointing to
(4) A `pcalc_func` structure whose `bp` field points to B, and whose `fn_of` field is
(5) A `pcalc_term` structure which has a `function` field pointing to
(6) A `pcalc_func` structure whose `bp` field points to C, and whose `fn_of` field is
(7) A `pcalc_term` structure which has a `variable` field set to 0 (which is x).

```
typedef struct pcalc_func {
    struct binary_predicate *bp;                     the predicate B such that this is f_B(t)
    struct pcalc_term fn_of;                         the term t such that this is f_B(t)
    int from_term;                                   whether t is term 0 or 1 of B
} pcalc_func;
```

The structure pcalc_func is shared with 6/vars, 6/simp, 6/tcpr, 6/asp, 6/atoms, 6/cind and 6/cdefp.

§1. Creating new terms.

```
pcalc_term term_new_variable(int v) {
    pcalc_term pt; ⟨Make new blank term structure pt 2⟩;
    pt.variable = v;
    return pt;
}
pcalc_term term_new_constant(specification *c) {
    pcalc_term pt; ⟨Make new blank term structure pt 2⟩;
    pt.constant = c;
    return pt;
}
pcalc_term term_new_function(binary_predicate *bp, pcalc_term ptof, int t) {
    pcalc_term pt; ⟨Make new blank term structure pt 2⟩;
    pcalc_func *pf = CREATE(pcalc_func);
    pf->bp = bp; pf->fn_of = ptof; pf->from_term = t;
    pt.function = pf;
    return pt;
}
```

The function term_new_variable is called from 6/aprop, 6/prop, 6/pform, 6/treec, 6/sconv, 6/simp and 6/cdefp.
The function term_new_constant is called from 5/aph, 6/aprop, 6/vars, 6/treec, 6/sconv, 6/simp and 12/cinv.
The function term_new_function is called from 6/simp.

§2. Where, in all three cases:

⟨Make new blank term structure pt 2⟩ ≡

```
    pt.variable = -1;
    pt.constant = NULL;
    pt.function = NULL;
    pt.cinder = -1;                                   that is, no cinder
    pt.term_checked_as_kov = NULL;
```

This code is used in §1.

§3. Variable letters. The variables 0 to 25 are referred to by the letters $x, y, z, a, b, c, ..., w$: this convention is followed both in the debugging log and in functions compiled into I6 code.

The number 26 turns up quite often in this chapter, and while it's normally good style to define named constants, here we're not going to. 26 is a number which anyone other than a string theorist will immediately associate with the size of the alphabet. Moreover, we can't really raise the total, because there are only 26 single-character local variable names in I6, `a` to `z`. (Well, strictly speaking there is also `_`, but we won't go there.) To have a variable limit lower than 26 would be artificial, since there are no memory constraints arguing for it; and in any case a proposition with 27 or more variables would be so huge that it could not be evaluated at run-time in any remotely plausible length of time. So although the 26-variables-only limit is embedded in Inform, it really is not any restriction, and it greatly simplifies the code.

```
char *pcalc_vars = "xyzabcdefghijklmnopqrstuvw";
```

§**4. Underlying terms.** Routines to see if a term is a constant C, or if it is a chain of functions at the bottom of which is a constant C; and similarly for variables.

```
specification *term_constant_underlying(pcalc_term *t) {
    if (t->constant) return t->constant;
    if (t->function) return term_constant_underlying(&(t->function->fn_of));
    return NULL;
}
int term_variable_underlying(pcalc_term *t) {
    if (t->variable >= 0) return t->variable;
    if (t->function) return term_variable_underlying(&(t->function->fn_of));
    return -1;
}
```

The function term_constant_underlying is called from 6/simp, 6/tcpr and 6/atoms.
The function term_variable_underlying is called from 6/prop, 6/vars, 6/sconv and 6/simp.

§**5. Adjective-noun conversions.** As we shall see, a general unary predicate stores a type-reference pointer to an adjectival phrase – the adjective it tests. But sometimes the same word acts both as adjective and noun in English. In "the green door", clearly "green" is an adjective; in "the door is green", it is possibly a noun; in "the colour of the door is green", it must surely be a noun. Yet these are all really the same meaning. To cope with this ambiguity, we need a way to convert the adjectival form of such an adjective into its noun form, and back again.

```
pcalc_term prop_adj_to_noun_conversion(adjective_list_entry *tr) {
    adjectival_phrase *aph = get_adjective_from_list_entry(tr);
    quantity *q = aph_has_ENUMERATIVE_meaning(aph);
    if (q) return term_new_constant(new_QUANTITY_spec(q));
    property_name *prn = aph_has_EORP_meaning(aph);
    if (prn) return term_new_constant(property_name_to_PROPERTY_spec(prn));
    return term_new_variable(0);
}
```

The function prop_adj_to_noun_conversion is called from 6/prop.

§**6.** And conversely:

```
adjective_list_entry *prop_noun_to_adj_conversion(pcalc_term pt) {
    kind_of_value *kov;
    adjectival_phrase *aph;
    specification *spec = pt.constant;
    if (spec_is_actual_CONSTANT(spec) == FALSE) return NULL;
    kov = spec_get_kind_of_value(spec);
    if (kov_get_coinciding_property(kov) == NULL) return NULL;
    quantity *q = RETRIEVE_FROM_SPEC(spec, quantity);
    aph = qty_get_adjectival_phrase(q);
    return new_adjective_list_entry(aph, TRUE);
}
```

The function prop_noun_to_adj_conversion is called from 6/tcpr.

§**7. Debugging terms.** The art of this is to be unobtrusive; when a proposition is being logged, we don't much care about the constant terms, and want to display them concisely and without fuss.

```
void log_pcalc_term(pcalc_term *pt) {
    if (pt == NULL) {
        LOG("<null-term>");
    } else if (pt->constant) {
        specification *spec = pt->constant;
        if (pt->cinder >= 0) { LOG("const_%d", pt->cinder); return; }
        if (spec->word_ref1 >= 0) { LOG("'$W'", spec->word_ref1, spec->word_ref2); return; }
        if (species_is(spec, CONSTANT_SPC)) {
            world_object *wo = wo_of_CONSTANT_OBJECT_if_any(spec);
            if (wo) { LOG("$O", wo); return; }
        }
        if (spec->word_ref1 >= 0) { LOG("<$W>", spec->word_ref1, spec->word_ref2); return; }
        LOG("$S", spec);
    } else if (pt->function) {
        binary_predicate *bp = pt->function->bp;
        i6_schema *fn = bp_get_term_as_function_of_other(bp, 0);
        if (fn == NULL) fn = bp_get_term_as_function_of_other(bp, 1);
        if (fn == NULL) internal_error("Function of non-functional predicate");
        if (logging_to_I6_text) {
            log_i6_schema_applied(fn, &(pt->function->fn_of));
        } else {
            LOG("{$i:$0}", fn, &(pt->function->fn_of));
        }
    } else if (pt->variable >= 0) {
        int j = pt->variable;
        if (j<26) LOG("%c", pcalc_vars[j]); else LOG("<bad-var=%d>", j);
    } else {
        LOG("<bad-term>");
    }
}
```

The function log_pcalc_term is called from 2/dl, 6/aprop and 6/cind.

Atomic Propositions 6/aprop

Purpose

To build and modify atoms, the syntactic pieces from which propositions are built up.

Definitions

¶1. As the description in the Introduction showed, propositions are complicated data structures. Roughly speaking, they are made up of small independent pieces which can be combined in a variety of ways into larger assemblies. In this section, we look at the smallest pieces: some of these could be propositions in their own right, others are only structural items needed to form up the larger collections. But each individual piece, or "atom", is stored in a `pcalc_prop` structure.

The question of how these are joined together is left until the next section.

```
define MAX_ATOM_ARITY 2                                          for the moment, at any rate

typedef struct pcalc_prop {
    int element;                                          one of the constants below: always 1 or greater
    int arity;                                  1 for quantifiers and unary predicates; 2 for BPs; 0 otherwise
    struct general_pointer predicate;                   usually indicates which predicate structure is meant
    struct pcalc_term terms[MAX_ATOM_ARITY];                          terms to which the predicate applies
    struct kind_of_value *assert_kind_of_value;          KIND_ATOM: the kind of value of a variable
    int calling_data;                                    CALLED_ATOM: compacted form of a word range
    struct pcalc_prop *next;                                  next atom in the list for this proposition
} pcalc_prop;
```

The structure pcalc_prop is shared with 6/prop, 6/vars, 6/sconv, 6/simp, 6/tcpr, 6/asp, 6/atoms, 6/defer, 6/cind and 6/cdefp.

¶2. The Universe is filled with atoms, but they come in different kinds, called elements. For us, an "element" is the identifying number, stored in the `element` field, which tells Inform what kind of atom something is. The following is our Periodic Table of all possible elements:

```
define QUANTIFIER_ATOM 1                                                  any generalised quantifier

define PREDICATE_ATOM 10                  a property-based unary predicate, or any predicate of higher arity
define KIND_ATOM 11                                  a unary predicate K(x) associated with a kind K
define ISAKIND_ATOM 12                  a unary predicate asserting that x is the world-object for a kind
define ISAKOV_ATOM 13                    a unary predicate asserting that x is the generic SP for a KOV
define ISAVAR_ATOM 14                  a unary predicate asserting that x is the SP for a global variable
define EVERYWHERE_ATOM 15                                  a unary predicate asserting omnipresence
define HERE_ATOM 16                                      a unary predicate asserting presence "here"
define CALLED_ATOM 17                                 to keep track of "(called the intruder)"-style names

define NEGATION_OPEN_ATOM 20                          logical negation ¬ applied to contents of group
define NEGATION_CLOSE_ATOM 30                                                  end of logical negation ¬
define DOMAIN_OPEN_ATOM 21                            logical negation ¬ applied to contents of group
define DOMAIN_CLOSE_ATOM 31                                                    end of logical negation ¬
```

¶3. And as with columns in the Periodic Table, these elements come in what are called "groups", because it often happens that atoms of different elements behave similarly when the elements have something in common.

```
define STRUCTURAL_GROUP 10
define PREDICATES_GROUP 20
define OPEN_OPERATORS_GROUP 30
define CLOSE_OPERATORS_GROUP 40
```

---

§**1. The elements.** Given an element, the following returns the group to which it belongs.

```
int element_get_group(int element) {
    if (element <= 0) return 0;
    if (element < STRUCTURAL_GROUP) return STRUCTURAL_GROUP;
    if (element < PREDICATES_GROUP) return PREDICATES_GROUP;
    if (element < OPEN_OPERATORS_GROUP) return OPEN_OPERATORS_GROUP;
    if (element < CLOSE_OPERATORS_GROUP) return CLOSE_OPERATORS_GROUP;
    return 0;
}
```

The function element_get_group is called from 6/prop, 6/vars, 6/simp and 6/cdefp.

§**2.** Some atoms occur in pairs, which have to match like opening and closing parentheses. The following returns 0 for an element code which does not behave like this, or else returns the opposite number to any element code which does.

```
int element_get_match(int element) {
    switch (element) {
        case NEGATION_OPEN_ATOM: return NEGATION_CLOSE_ATOM;
        case NEGATION_CLOSE_ATOM: return NEGATION_OPEN_ATOM;
        case DOMAIN_OPEN_ATOM: return DOMAIN_CLOSE_ATOM;
        case DOMAIN_CLOSE_ATOM: return DOMAIN_OPEN_ATOM;
        default: return 0;
    }
}
```

The function element_get_match is called from 6/prop.

§**3. Creating atoms.** Every atom is created by the following routine:

```
pcalc_prop *atom_new(int element) {
    pcalc_prop *prop = CREATE(pcalc_prop);
    prop->next = NULL;
    prop->element = element;
    prop->assert_kind_of_value = NULL;
    prop->arity = 0;
    prop->predicate = NULL_GENERAL_POINTER;
    return prop;
}
```

The function atom_new is called from 6/prop, 6/pform, 6/treec, 6/sconv, 6/simp and 8/imp.

§**4. The STRUCTURAL group.** Some convenient routines to handle atoms of specific elements now follow: first, `QUANTIFIER` atoms. These have arity 1, and the single term must always be a variable, the one which is being bound. The parameter is a number needed for some `quantifier` types to identify the range: for instance, it would be 7 in the case of $V_{=7}$.

Tying specific variables to quantifiers seems to be out of fashion in modern computer science. Contemporary theorem-proving assistants mostly use de Bruijn's numbering scheme, in which numbers 1, 2, 3, ..., refer to variables being quantified in an indirect way. The advantage is that propositions are easier to construct, since the same numbers can be used in different subexpressions of the same proposition, and there's no worrying about clashes. But it all just moves the difficulty elsewhere, by making it less obvious how to pair up the numbers with variables at compilation time, and less obvious even how many variables are needed. So we stick to the old-fashioned way of imitating $\forall x : P(x)$ rather than $\forall 1.P$.

```
pcalc_prop *atom_QUANTIFIER_new(quantifier *quant, int v, int parameter) {
    pcalc_prop *prop = atom_new(QUANTIFIER_ATOM);
    prop->arity = 1;
    prop->terms[0] = term_new_variable(v);
    prop->predicate = STORE_POINTER_quantifier(quant);
    prop->calling_data = parameter;
    return prop;
}
```

The function atom_QUANTIFIER_new is called from 6/vars, 6/pform, 6/treec and 6/simp.

§**5.** Quantifier atoms can be detected as follows:

```
int atom_is_quantifier(pcalc_prop *prop) {
    if ((prop) && (prop->element == QUANTIFIER_ATOM)) return TRUE;
    return FALSE;
}
int atom_is_existence_quantifier(pcalc_prop *prop) {
    if ((prop) && (prop->element == QUANTIFIER_ATOM) &&
        (RETRIEVE_POINTER_quantifier(prop->predicate) == exists_quantifier))
        return TRUE;
    return FALSE;
}
int atom_is_nonexistence_quantifier(pcalc_prop *prop) {
    if ((prop) && (prop->element == QUANTIFIER_ATOM) &&
        (RETRIEVE_POINTER_quantifier(prop->predicate) == not_exists_quantifier))
        return TRUE;
    return FALSE;
}
int atom_is_forall_quantifier(pcalc_prop *prop) {
    if ((prop) && (prop->element == QUANTIFIER_ATOM) &&
        (RETRIEVE_POINTER_quantifier(prop->predicate) == for_all_quantifier))
        return TRUE;
    return FALSE;
}
int atom_is_notall_quantifier(pcalc_prop *prop) {
    if ((prop) && (prop->element == QUANTIFIER_ATOM) &&
        (RETRIEVE_POINTER_quantifier(prop->predicate) == not_for_all_quantifier))
        return TRUE;
    return FALSE;
}
```

```
int atom_is_for_all_x(pcalc_prop *prop) {
    if ((atom_is_forall_quantifier(prop)) && (prop->terms[0].variable == 0)) return TRUE;
    return FALSE;
}
```

The function atom_is_quantifier is called from 6/prop and 6/simp.
The function atom_is_existence_quantifier is called from 6/prop, 6/simp and 6/defer.
The function atom_is_nonexistence_quantifier is called from 6/simp.
The function atom_is_forall_quantifier is called from 6/simp.
The function atom_is_notall_quantifier is called from 6/simp.
The function atom_is_for_all_x is called from 6/prop.

§**6.** See "Determiners and Quantifiers" for what a now-assertable quantifier is:

```
int atom_is_now_assertable_quantifier(pcalc_prop *prop) {
    if (prop->element != QUANTIFIER_ATOM) return FALSE;
    return quant_is_now_assertable(RETRIEVE_POINTER_quantifier(prop->predicate));
}
```

The function atom_is_now_assertable_quantifier is called from 6/defer.

§**7. The PREDICATES group.** Next, unary predicates, beginning with the `EVERYWHERE` special case.

```
pcalc_prop *atom_EVERYWHERE_new(pcalc_term pt) {
    pcalc_prop *prop = atom_new(EVERYWHERE_ATOM);
    prop->arity = 1;
    prop->terms[0] = pt;
    return prop;
}
```

The function atom_EVERYWHERE_new is called from 6/treec and 6/simp.

§**8.** And `HERE`:

```
pcalc_prop *atom_HERE_new(pcalc_term pt) {
    pcalc_prop *prop = atom_new(HERE_ATOM);
    prop->arity = 1;
    prop->terms[0] = pt;
    return prop;
}
```

The function atom_HERE_new is called from 6/treec.

§**9.** And `ISAKIND`:

```
pcalc_prop *atom_ISAKIND_new(pcalc_term pt) {
    pcalc_prop *prop = atom_new(ISAKIND_ATOM);
    prop->arity = 1;
    prop->terms[0] = pt;
    return prop;
}
```

The function atom_ISAKIND_new is called from 6/treec.

§**10.** And `ISAKOV`:

```
pcalc_prop *atom_ISAKOV_new(pcalc_term pt) {
    pcalc_prop *prop = atom_new(ISAKOV_ATOM);
    prop->arity = 1;
    prop->terms[0] = pt;
    return prop;
}
```

The function atom_ISAKOV_new is called from 6/treec.

§**11.** And `ISAVAR`:

```
pcalc_prop *atom_ISAVAR_new(pcalc_term pt) {
    pcalc_prop *prop = atom_new(ISAVAR_ATOM);
    prop->arity = 1;
    prop->terms[0] = pt;
    return prop;
}
```

The function atom_ISAVAR_new is called from 6/treec.

§**12.** `CALLED` atoms are interesting because they exist only for their side-effects: they have no effect at all on the logical status of a proposition (well, except that they should not be applied to free variables referred to nowhere else). They can therefore be added or removed freely. In the phrase

> if a woman is in a lighted room (called the den), ...

we need to note that the value of the bound variable corresponding to the lighted room will need to be kept and to have a name ("the den"): this will probably mean the inclusion of a `CALLED=den(y)` atom.

The calling data for a `CALLED` atom is the textual name by which the variable will be called: a word range packed into a single `int` field.

```
pcalc_prop *atom_CALLED_new(int w1, int w2, pcalc_term pt, kind_of_value *kov) {
    pcalc_prop *prop = atom_new(CALLED_ATOM);
    prop->arity = 1;
    prop->terms[0] = pt;
    prop->calling_data = w1 + ((w2-w1) << 20);
    prop->assert_kind_of_value = kov;
    return prop;
}
void atom_CALLED_get_name(pcalc_prop *prop, int *w1, int *w2) {
    *w1 = prop->calling_data & ((1<<20)-1);
    *w2 = (prop->calling_data >> 20) + *w1;
}
```

The function atom_CALLED_new is called from 6/pform and 6/treec.
The function atom_CALLED_get_name is called from 6/asp, 6/atoms and 6/defer.

§**13.** Now for a KIND atom. At first sight, it looks odd that a unary predicate for a kind is represented differently from other predicates. Isn't it a unary predicate just like any other? Well: it is, but then again, we want to compile propositions to reasonably efficient I6 code which determines whether or not they are true. We particularly want to look out for patterns like

$$\forall x : ... \wedge container(x) \wedge ...$$

since they allow us to consider x ranging over a smaller, and therefore more efficiently searchable, domain: most objects aren't containers. So KIND_ATOM atoms are useful in ways which other unary predicate atoms are not.

Once again, this atom has arity 1, but the term no longer has to be a variable; when Inform reads a sentence like

> Viper Pit is a room.

the resulting proposition will include a KIND atom whose term is the constant value for the Viper Pit.

Any kind of value can be assigned, but the commonest case involves a kind of object, so a special routine exists just to create KIND atoms in that case.

```
pcalc_prop *atom_KIND_object_new(world_object *k, pcalc_term pt) {
    pcalc_prop *prop = atom_new(KIND_ATOM);
    prop->arity = 1;
    prop->assert_kind_of_value = kovko(k);
    prop->terms[0] = pt;
    return prop;
}
pcalc_prop *atom_KIND_value_new(kind_of_value *kov, pcalc_term pt) {
    pcalc_prop *prop = atom_new(KIND_ATOM);
    prop->arity = 1;
    prop->assert_kind_of_value = kov;
    prop->terms[0] = pt;
    return prop;
}
```

The function atom_KIND_object_new is called from 6/pform, 6/treec and 6/simp.
The function atom_KIND_value_new is called from 6/pform, 6/treec, 6/sconv and 6/simp.

§**14.** That just leaves the general sort of unary predicate. In principle we ought to be able to create $U(t)$ for any term t, but in practice we only ever need $t = x$, that is, variable 0.

```
pcalc_prop *atom_unary_PREDICATE_from_aph(adjectival_phrase *aph, int negated) {
    pcalc_prop *prop = atom_new(PREDICATE_ATOM);
    prop->arity = 1;
    prop->terms[0] = term_new_variable(0);
    prop->predicate = STORE_POINTER_adjective_list_entry(
        new_adjective_list_entry(aph, (negated)?FALSE:TRUE));
    return prop;
}
```

The function atom_unary_PREDICATE_from_aph is called from 6/pform, 6/treec, 6/sconv, 8/creat and 8/imp.

§**15.** And binary predicates are pretty well the same:

```
pcalc_prop *atom_binary_PREDICATE_new(binary_predicate *bp,
    pcalc_term pt1, pcalc_term pt2) {
    pcalc_prop *prop = atom_new(PREDICATE_ATOM);
    prop->arity = 2;
    prop->predicate = STORE_POINTER_binary_predicate(bp);
    prop->terms[0] = pt1; prop->terms[1] = pt2;
    return prop;
}
binary_predicate *atom_is_binary_predicate(pcalc_prop *prop) {
    if (prop == NULL) return NULL;
    if (prop->element != PREDICATE_ATOM) return NULL;
    if (prop->arity != 2) return NULL;
    return RETRIEVE_POINTER_binary_predicate(prop->predicate);
}
int atom_is_equality_predicate(pcalc_prop *prop) {
    binary_predicate *bp = atom_is_binary_predicate(prop);
    if (bp == a_is_b_predicate) return TRUE;
    return FALSE;
}
```

The function atom_binary_PREDICATE_new is called from 6/treec, 6/sconv and 6/simp.
The function atom_is_binary_predicate is called from 6/simp.
The function atom_is_equality_predicate is called from 6/simp.

§**16.** Given C, return the proposition $is(x, C)$:

```
pcalc_prop *prop_x_is_constant(specification *spec) {
    return atom_binary_PREDICATE_new(a_is_b_predicate,
        term_new_variable(0), term_new_constant(spec));
}
```

The function prop_x_is_constant is called from 6/pform.

§**17.** And conversely:

```
pcalc_term *atom_is_x_equals(pcalc_prop *prop) {
    if (atom_is_equality_predicate(prop) == FALSE) return NULL;
    if (prop->terms[0].variable != 0) return NULL;
    return &(prop->terms[1]);
}
```

The function atom_is_x_equals is called from 6/sconv.

§18. Validating atoms.

```
char *atom_validate(pcalc_prop *prop) {
    int group;
    if (prop == NULL) return NULL;
    group = element_get_group(prop->element);
    if (group == 0) return "atom of undiscovered element";
    if (prop->arity > MAX_ATOM_ARITY) return "atom with overly large arity";
    if (prop->arity < 0) return "atom with negative arity";
    if (prop->arity == 0) {
        if (group == PREDICATES_GROUP) return "predicate without terms";
        if (prop->element == QUANTIFIER_ATOM) return "quantifier without variable";
    } else {
        if ((prop->element != PREDICATE_ATOM) && (prop->arity != 1))
            return "unary atom with other than one term";
        if ((group == OPEN_OPERATORS_GROUP) || (group == CLOSE_OPERATORS_GROUP))
            return "parentheses with terms";
    }
    if ((prop->element == QUANTIFIER_ATOM) && (prop->terms[0].variable == -1))
        return "missing variable in quantification";
    return NULL;
}
```

The function atom_validate is called from 6/prop.

§19. Debugging log.

Logging atomic propositions divides into cases:

```
void log_pcalc_atom(pcalc_prop *prop) {
    if (prop == NULL) { LOG("<null-atom>"); return; }
    switch(prop->element) {
        case PREDICATE_ATOM:
            switch(prop->arity) {
                case 1: ⟨Log some suitable textual name for this unary predicate 20⟩; break;
                case 2: ⟨Log some suitable textual name for this binary predicate 21⟩; break;
                default: LOG("?exotic-predicate-arity=%d?", prop->arity); break;
            }
            break;
        case QUANTIFIER_ATOM: {
            quantifier *quant = RETRIEVE_POINTER_quantifier(prop->predicate);
            log_quantifier(quant, prop->calling_data);
            LOG(" "); ⟨Log a comma-separated list of terms for this atomic proposition 22⟩;
            return;
        }
        case CALLED_ATOM: {
            int w1, w2;
            atom_CALLED_get_name(prop, &w1, &w2);
            LOG("called='$W'", w1, w2);
            break;
        }
        case KIND_ATOM: {
            world_object *k = kovko_get_kind(prop->assert_kind_of_value);
            if (logging_to_I6_text == FALSE) LOG("kind=");
            if ((k) && (k->word_ref1 >= 0)) LOG("$W", k->word_ref1, k->word_ref2);
            else LOG("$u", prop->assert_kind_of_value); break;
```

```
        }
        case ISAKIND_ATOM: LOG("is-a-kind"); break;
        case ISAKOV_ATOM: LOG("is-a-KOV"); break;
        case ISAVAR_ATOM: LOG("is-a-var"); break;
        case EVERYWHERE_ATOM: LOG("everywhere"); break;
        case HERE_ATOM: LOG("here"); break;
        case NEGATION_OPEN_ATOM: LOG("NOT["); break;
        case NEGATION_CLOSE_ATOM: LOG("NOT]"); break;
        case DOMAIN_OPEN_ATOM: LOG("IN["); break;
        case DOMAIN_CLOSE_ATOM: LOG("IN]"); break;
        default: LOG("?bad-atom?"); break;
    }
    if (prop->arity > 0) {
        LOG("("); ⟨Log a comma-separated list of terms for this atomic proposition 22⟩; LOG(")");
    }
}
```

The function log_pcalc_atom is called from 2/dl and 6/prop.

§**20.**

⟨Log some suitable textual name for this unary predicate 20⟩ ≡

```
    adjective_list_entry *tr = RETRIEVE_POINTER_adjective_list_entry(prop->predicate);
    if (adjective_used_positively(tr) == FALSE) LOG("not-");
    log_adjectival_phrase(get_adjective_from_list_entry(tr));
```

This code is used in §19.

§**21.** And more easily:

⟨Log some suitable textual name for this binary predicate 21⟩ ≡

```
    binary_predicate *bp = RETRIEVE_POINTER_binary_predicate(prop->predicate);
    if (bp == NULL) LOG("?bad-bp?"); else LOG(bp_get_log_name(bp));
```

This code is used in §19.

§**22.** Just a diagnostic way of printing the terms in an atomic proposition, by their index numbers. (They are numbered from 0 to $A-1$, where A is the arity.)

⟨Log a comma-separated list of terms for this atomic proposition 22⟩ ≡

```
    int t;
    for (t=0; t<prop->arity; t++) {
        if (t>0) LOG(", ");
        log_pcalc_term(&(prop->terms[t]));
    }
```

This code is used in §19.

Propositions 6/prop

Purpose

To build and modify structures representing propositions in predicate calculus.

Definitions

¶1. We now begin on the data structures to hold propositions. Now a properly constructed proposition has a natural tree structure – one can regard quantification, negation, and conjunction as higher nodes, and predicates as leaves. So the idea of storing propositions as trees has a certain elegance. At first it seems an advantage that any such tree is necessarily a valid proposition. But in fact this is not so helpful, because we want to build propositions gradually, and in particular intermediate states need to exist which are not yet valid but will be. If we used a tree representation, we would also need some cursor-position-like marker for the region of current growth, and that could all become complicated. We will also find that the main operation we need to perform is a depth-first traverse of the tree, which is a little tiresome to do in a conventional loop (it lends itself to recursion, but that's inconvenient).

So we will instead store propositions in a linked list, imitating the notation used by mathematicians who write them along in a single line. Now there's a natural way to store incomplete propositions and a natural build-point (at the end), and depth-first traverses are easy – just work along from left to right. The disadvantage is that it's also easy to make malformed propositions, so we have to build carefully.

For instance, "Test sentence (internal) with no man can see the box." produces:

```
1. no man can see the box
[ DoesNotExist x IN[ man(x) IN] : can-see(x, 'box') ]
```

The proposition is stored as a linked list of atoms, of elements like so:

```
QUANTIFIER --> DOMAIN_OPEN --> PREDICATE --> DOMAIN_CLOSE --> PREDICATE
```

In short: a proposition is a linked list of `pcalc_prop` atoms, joined by their `next` fields. The present section contains routines to help build and edit such lists.

¶2. In particular:

(a) The empty list, a `NULL` pointer, represents the universally true proposition $\top$. Asserting it does nothing; testing it at run-time always evaluates to `true`.

(b) The conjunction $\phi \wedge \psi$ is just the concatenation of their linked lists.

(c) Negation $\neg(\phi)$ is the concatenation `NEGATION_OPEN --> P --> NEGATION_CLOSE`, where `P` is the linked list for ϕ.

(d) The quantifier $Qv \in \{v \mid \phi(v)\}$ is `QUANTIFIER --> DOMAIN_OPEN --> P --> DOMAIN_CLOSE`.

In this section, we'll call a segment of the list representing a pair of matched brackets, like `DOMAIN_OPEN --> P --> DOMAIN_CLOSE`, a "group".

¶3. We sometimes need to indicate a position within a proposition – a position not of an atom, but between atoms. Consider the possible places where letters could be inserted into the word "rap": before the "r" (trap), between "r" and "a" (reap), between "a" and "p" (ramp), after the "p" (rapt). Though "rap" is a three-letter word, there are four possible insertion points – so they can't exactly correspond to letters. The convention used with Inform propositions is that a position marker points to the `pcalc_prop` structure for the atom *before* the position meant: and a `NULL` pointer in this context means the front position, before the opening atom.

¶4. The code needed to perform a depth-first traverse of a proposition is abstracted by the following macros. Note that we often need to remember the atom before the current one, so we keep that in a spare variable during each traverse. (This saves us having to maintain the proposition data structure as a doubly linked list, which would be harder to edit.)

One macro declares the name of a marker variable to be used when traversing; the other is the necessary loop head. Note that we do not assume that `p` will still be non-`NULL` at the end of a loop iteration, just because it was at the beginning: local edits are sometimes performed in the traverse, and it can happen that an edit truncates the proposition so savagely that the loop finds its ground cut out from under it.

```
define TRAVERSE_VARIABLE(p)
    pcalc_prop *p = NULL, *p##_prev = NULL;
    int p##_repeat = FALSE;
define TRAVERSE_PROPOSITION(p, start)
    for (p=start, p##_prev=NULL, p##_repeat = FALSE;
        p;
        (p##_repeat == FALSE)?(p##_prev=p, p=(p)?(p->next):NULL):0, p##_repeat = FALSE)
```

¶5. An edit which happens during a traverse is permitted to make any change to the proposition at and beyond the marker position `p`, but not allowed to change what came before. Since such an edit might leave `p` pointing to an atom which has been cut, or moved later, we must perform the following macro after edits to restore `p`. We know that the atom which was before `p` at the start of the loop has not been changed – since edits aren't allowed there – so `p_prev` must be correct, and we therefore restore `p` to the next atom after `p_prev`.

There is a catch, however: if our edit consists only of deleting some atoms then using `PROPOSITION_EDITED` correctly resets `p` to the current atom at the marker position, and that will be the first atom after the ones deleted. If we then just go around the loop, we move on to the next atom; as a result, the first atom after the deleted ones is skipped over. We can avoid this by using `PROPOSITION_EDITED_REPEATING_CURRENT` instead.

Every routine which simplifies a proposition is expected to have an `int *` argument called `changed`: on exit, the `int` variable this points to should be set if and only if a change has been made to the proposition.

```
define PROPOSITION_EDITED(p, prop)
    if (p##_prev == NULL) p = prop; else p = p##_prev->next;
    *changed = TRUE;
define PROPOSITION_EDITED_REPEATING_CURRENT(p, prop)
    PROPOSITION_EDITED(p, prop)
    p##_repeat = TRUE;
```

---

§**1. Implied conjunction.** Conjunction (logical "and") occurs so densely in propositions arising from natural language that our data structures would grow large and unmanageable if we wrote all of them out. So we adopt a convention similar to the one in algebra, where the formula

$$xy + w(v - 1)$$

is understood to mean multiplication of x by y, and of w by $(v - 1)$. Note that if we were to write it out as a sequence of symbols

```
x y + w ( v - 1 )
```

then multiplication would only be understood at two positions, not between every pair of symbols. In the same way, the following routine looks at a pair of adjacent atoms and decides whether or not conjunction should be understood between them.

```
int implied_conjunction_between(pcalc_prop *p1, pcalc_prop *p2) {
    if ((p1 == NULL) || (p2 == NULL)) return FALSE;
    if (element_get_group(p1->element) == OPEN_OPERATORS_GROUP) return FALSE;
    if (element_get_group(p2->element) == CLOSE_OPERATORS_GROUP) return FALSE;
    if (p1->element == QUANTIFIER_ATOM) return FALSE;
    if (p1->element == DOMAIN_CLOSE_ATOM) return FALSE;
    return TRUE;
}
```

The function implied_conjunction_between is called from 6/defer.

§**2.** Purely decoratively, we print some punctuation when logging a proposition; this is chosen to look like standard mathematical notation.

```
char *debugging_log_text_between(pcalc_prop *p1, pcalc_prop *p2) {
    if ((p1 == NULL) || (p2 == NULL)) return "";
    if (p1->element == QUANTIFIER_ATOM) {
        if (p2->element == DOMAIN_OPEN_ATOM) return "";
        return ":";
    }
    if (p1->element == DOMAIN_CLOSE_ATOM) return ":";
    if (implied_conjunction_between(p1, p2)) {
        if (logging_to_I6_text) return("&");                    since ^ in I6 strings means newline
        return ("^");
    }
    return "";
}
```

§**3.** So we may as well complete the debugging log code now. Note that $\top$ is logged as just `[ ]`.

```
void log_pcalc_prop(pcalc_prop *prop) {
    TRAVERSE_VARIABLE(p);
    LOG("[ ");
    TRAVERSE_PROPOSITION(p, prop) {
        char *bridge = debugging_log_text_between(p_prev, p);
        if (bridge[0]) LOG("%s ", bridge);
        log_pcalc_atom(p);
        LOG(" ");
    }
    LOG("]");
}
```

The function log_pcalc_prop is called from 2/dl.

§**4. Validity.** Since the proposition data structure lets us build all kinds of nonsense, we'll be much safer if we can check our working – if we can verify that a proposition is valid. But what does that mean? We might mean:

(i) a proposition is good if its sequence of `next` pointers all correctly point to `pcalc_prop` structures, and don't loop around into a circle;
(ii) a proposition is good if (i) is true, and it is correctly punctuated;
(iii) a proposition is good if (ii) is true, and it never confuses together two different variables by giving both the same letter;
(iv) a proposition is good if (iii) is true, and all of its predicates can safely be applied to all of their terms, and we can identify what kind of value each variable ranges over.

These are steadily stronger conditions. The first is a basic invariant of our data structures: nothing failing (i) will ever be allowed to exist, provided the routines in this section are free of bugs. Condition (ii) is called *syntactic validity*; (iii) is *well-formedness*; (iv) is *type safety*. Correct source text eventually makes propositions which have all four properties, but intermediate half-built states often satisfy only (i).

§**5.** The following examples illustrate the differences. This one is not even syntactically valid:

```
DOMAIN_OPEN_ATOM --> NEGATION_CLOSE_ATOM --> NEGATION_CLOSE_ATOM
```

This one is syntactically valid, but not well-formed:

```
EVERYWHERE_ATOM(x) --> QUANTIFIER=for-all(x) --> PREDICATE=open(x)
```

(If `x` ranges over all objects at the middle of the proposition, it had better not already have a value, but if it doesn't, what can that first atom mean? It would be like writing the formula $n + \sum_{n=1}^{10} n^2$, where clearly two different things have been called n.)

And this proposition is well-formed but not type-safe:

```
QUANTIFIER=for-all(x) --> KIND=number(x) --> EVERYWHERE(x)
```

(Here `x` is supposed to be a number, and therefore has no location, but `EVERYWHERE` can validly be applied only to backdrop objects, so what could `EVERYWHERE(x)` possibly mean?)

§**6.** Well-formedness and type safety are left to later sections in this chapter, but we can at least test syntactic validity here.

define MAX_PROPOSITION_GROUP_NESTING 100 *vastly more than could realistically be used*

```
int prop_is_syntactically_valid(pcalc_prop *prop) {
    TRAVERSE_VARIABLE(p);
    int groups_stack[MAX_PROPOSITION_GROUP_NESTING], group_sp = 0;
    TRAVERSE_PROPOSITION(p, prop) {
        (1) each individual atom has to be properly built:
        char *err = atom_validate(p);
        if (err) { LOG("Atom error: %s: $o\n", err, p); return FALSE; }
        (2) every open bracket must be matched by a close bracket of the same kind:
        if (element_get_group(p->element) == OPEN_OPERATORS_GROUP) {
            if (group_sp >= MAX_PROPOSITION_GROUP_NESTING) {
                LOG("Group nesting too deep\n"); return FALSE;
            }
            groups_stack[group_sp++] = p->element;
        }
        if (element_get_group(p->element) == CLOSE_OPERATORS_GROUP) {
            if (group_sp <= 0) { LOG("Too many close groups\n"); return FALSE; }
            if (element_get_match(groups_stack[--group_sp]) != p->element) {
                LOG("Group open/close doesn't match\n"); return FALSE;
            }
        }
        (3) every quantifier except "exists" must be followed by domain brackets, which occur nowhere else:
        if ((atom_is_quantifier(p_prev)) && (atom_is_existence_quantifier(p_prev) == FALSE)) {
            if (p->element != DOMAIN_OPEN_ATOM) { LOG("Quant without domain\n"); return FALSE; }
        } else {
            if (p->element == DOMAIN_OPEN_ATOM) { LOG("Domain without quant\n"); return FALSE; }
        }
        if ((p->next == NULL) &&
            (atom_is_quantifier(p)) && (atom_is_existence_quantifier(p) == FALSE)) {
            LOG("Ends without domain of final quantifier\n"); return FALSE;
        }
    }
    (4) a proposition must end with all its brackets closed:
    if (group_sp != 0) { LOG("%d group(s) open\n", group_sp); return FALSE; }
    return TRUE;
}
```

The function prop_is_syntactically_valid is called from 6/vars.

§**7. Primitive operations on propositions.** First, copying, which means copying not just the current atom, but all subsequent ones.

```
pcalc_prop *prop_copy(pcalc_prop *original) {
    pcalc_prop *first = NULL, *last = NULL, *prop = original;
    while (prop) {
        pcalc_prop *copied_atom = atom_new(0);
        *copied_atom = *prop; copied_atom->next = NULL;
        if (first) last->next = copied_atom;
        else first = copied_atom;
        last = copied_atom;
        prop = prop->next;
    }
    return first;
}
```

The function prop_copy is called from 6/cdefp and 7/vasp.

§**8.** Now to concatenate propositions. If E and T are both syntactically valid, the result will be, too; but the same is not true of well-formedness, so we need to be careful in using this.

```
pcalc_prop *prop_concatenate(pcalc_prop *existing_body, pcalc_prop *tail) {
    pcalc_prop *end = existing_body;
    if (end == NULL) return tail;
    while (end && (end->next)) end = end->next;
    end->next = tail;
    return existing_body;
}
```

The function prop_concatenate is called from 6/pform, 6/treec, 6/sconv, 8/creat, 8/imp and 9/prop.

§**9.** And here is a version which protects us:

```
pcalc_prop *prop_conjoin(pcalc_prop *existing_body, pcalc_prop *tail) {
    vars_renumber_bound(tail, existing_body, -1);
    existing_body = prop_concatenate(existing_body, tail);
    return existing_body;
}
```

The function prop_conjoin is called from 6/treec.

§**10. Inserting and deleting atoms.** Here we insert an atom at a given position, or at the front if the position is NULL.

```
pcalc_prop *prop_insert_atom(pcalc_prop *prop, pcalc_prop *position,
    pcalc_prop *new_atom) {
    if (position == NULL) {
        new_atom->next = prop;
        return new_atom;
    } else {
        if (prop == NULL) internal_error("inserting atom nowhere");
        new_atom->next = position->next;
        position->next = new_atom;
        return prop;
    }
}
```

The function prop_insert_atom is called from 6/vars and 6/simp.

§**11.** And similarly, with the deleted atom the one after the position given:

```
pcalc_prop *prop_delete_atom(pcalc_prop *prop, pcalc_prop *position) {
    if (position == NULL) {
        if (prop == NULL) internal_error("deleting atom nowhere");
        return prop->next;
    } else {
        if (position->next == NULL) internal_error("deleting atom off end");
        position->next = position->next->next;
        return prop;
    }
}
```

The function prop_delete_atom is called from 6/simp and 6/cdefp.

§**12. Inspecting contents.** First, we count the number of atoms in a given proposition. This is used by other parts of Inform as a crude measure of how complicated it is; though in fact it is not all that crude so long as it is applied to a proposition which has been simplified.

```
int prop_length(pcalc_prop *prop) {
    int n = 0;
    TRAVERSE_VARIABLE(p);
    TRAVERSE_PROPOSITION(p, prop) n++;
    return n;
}
```

The function prop_length is called from 7/cosp.

§13. Matching sequences of atoms. The following sneakily variable-argument-length function can be used to detect subsequences within a proposition: say, the sequence

```
QUANTIFIER --> PREDICATE --> anything --> CALLED
```

starting at the current position, which could be tested with:

```
prop_match(p, 4, QUANTIFIER_ATOM, NULL, PREDICATE_ATOM, NULL,
 ANY_ATOM_HERE, NULL, CALLED_ATOM, &cp);
```

As can be seen, each atom is tested with an element number and an optional pointer; when a successful match is made, the optional pointer is set to the atom making the match. (So if the routine returns `TRUE` then we can be certain that `cp` points to the `CALLED_ATOM` at the end of the run of four.) There are two special pseudo-element-numbers:

```
define ANY_ATOM_HERE 0          match any atom, but don't match beyond the end of the proposition
define END_PROP_HERE -1         a sentinel meaning "the proposition must end at this point"
```

```
int prop_match(pcalc_prop *prop, int c, ...) {
    int i, outcome = TRUE;
    va_list ap;                                   the variable argument list signified by the dots
    va_start(ap, c);                              macro to begin variable argument processing
    for (i = 0; i < c; i++) {
        int a = va_arg(ap, int);
        pcalc_prop **atom_p = va_arg(ap, pcalc_prop **);
        if (atom_p != NULL) *atom_p = prop;
        switch (a) {
            case ANY_ATOM_HERE: if (prop == NULL) outcome = FALSE; break;
            case END_PROP_HERE: if (prop != NULL) outcome = FALSE; break;
            default: if (prop == NULL) outcome = FALSE;
                else if (prop->element != a) outcome = FALSE;
                break;
        }
        if (prop) prop = prop->next;
    }
    va_end(ap);                                   macro to end variable argument processing
    return outcome;
}
```

The function prop_match is called from 6/sconv and 6/simp.

§14. Seeking atoms. Here we run through the proposition looking for either a given element, or a given arity, or both:

```
pcalc_prop *prop_seek_atom(pcalc_prop *prop, int atom_req, int arity_req) {
    TRAVERSE_VARIABLE(p);
    TRAVERSE_PROPOSITION(p, prop)
        if (((atom_req < 0) || (p->element == atom_req)) &&
            ((arity_req < 0) || (p->arity == arity_req)))
                return p;
    return NULL;
}
```

§**15.** Seeking different kinds of atom is now easy:

```
int prop_contains_binary_predicate(pcalc_prop *prop) {
    if (prop_seek_atom(prop, PREDICATE_ATOM, 2)) return TRUE; return FALSE;
}
int prop_contains_quantifier(pcalc_prop *prop) {
    if (prop_seek_atom(prop, QUANTIFIER_ATOM, -1)) return TRUE; return FALSE;
}
int prop_contains_nonexistence_quantifier(pcalc_prop *prop) {
    while ((prop = prop_seek_atom(prop, QUANTIFIER_ATOM, 1)) != NULL) {
        quantifier *quant = RETRIEVE_POINTER_quantifier(prop->predicate);
        if (quant != exists_quantifier) return TRUE;
        prop = prop->next;
    }
    return FALSE;
}
int prop_contains_callings(pcalc_prop *prop) {
    if (prop_seek_atom(prop, CALLED_ATOM, -1)) return TRUE; return FALSE;
}
```

The function prop_contains_binary_predicate is called from 8/knowp.

The function prop_contains_quantifier is called from 6/defer and 10/tab.

The function prop_contains_nonexistence_quantifier is called from 6/asp.

The function prop_contains_callings is called from 6/defer and 11/chron.

§**16.** Here we try to find out the kind of value of variable 0 without the full expense of typechecking the proposition:

```
kind_of_value *prop_describes_KOV(pcalc_prop *prop) {
    pcalc_prop *p = prop;
    while ((p = prop_seek_atom(prop, ISAKOV_ATOM, 1)) != NULL) {
        if (term_variable_underlying(&(p->terms[0])) == 0) return p->assert_kind_of_value;
        p = p->next;
    }
    p = prop;
    while ((p = prop_seek_atom(p, KIND_ATOM, 1)) != NULL) {
        if (term_variable_underlying(&(p->terms[0])) == 0) return p->assert_kind_of_value;
        p = p->next;
    }
    return NULL;
}
```

The function prop_describes_KOV is called from 6/defer and 7/tc.

§**17.** Finding an adjective which cannot be construed as an either/or property name used adjectivally (as in "a closed door") is more involved:

```
int prop_contains_adjective(pcalc_prop *prop) {
    while ((prop = prop_seek_atom(prop, PREDICATE_ATOM, 1)) != NULL) {
        adjective_list_entry *tr = RETRIEVE_POINTER_adjective_list_entry(prop->predicate);
        adjectival_phrase *aph = get_adjective_from_list_entry(tr);
        if (aph_has_EORP_meaning(aph) == NULL) return TRUE;
        return TRUE;
        prop = prop->next;
    }
    return FALSE;
}
```

The function prop_contains_adjective is called from 8/knowp.

§**18.** The following searches not for an atom, but for the lexically earliest term in the proposition:

```
pcalc_term prop_get_first_cited_term(pcalc_prop *prop) {
    TRAVERSE_VARIABLE(p);
    TRAVERSE_PROPOSITION(p, prop)
        if (p->arity > 0)
            return p->terms[0];
    internal_error("prop_get_first_cited_term on termless proposition");
    return term_new_variable(0);                    never executed, but needed to prevent gcc warnings
}
```

The function prop_get_first_cited_term is called from 6/sconv.

§**19.** Here we attempt, if possible, to read a proposition as being either $adjective(v)$ or $\exists v : adjective(v)$, where the adjective can be also be read as a noun, and if so we return a constant term t for that noun; or if the proposition isn't in that form, we return $t = x$, that is, variable 0.

```
pcalc_term convert_adj_to_noun(pcalc_prop *prop) {
    pcalc_term pct = term_new_variable(0);
    if (prop == NULL) return pct;
    if (atom_is_existence_quantifier(prop)) prop = prop->next;
    if (prop == NULL) return pct;
    if (prop->next != NULL) return pct;
    if ((prop->element == PREDICATE_ATOM) && (prop->arity == 1)) {
        adjective_list_entry *tr = RETRIEVE_POINTER_adjective_list_entry(prop->predicate);
        return prop_adj_to_noun_conversion(tr);
    }
    return pct;
}
```

The function convert_adj_to_noun is called from 6/sconv.

§**20. Bracketed groups.** The following routine tests whether the entire proposition is a single bracketed group. For instance:

```
NEGATION_OPEN --> PREDICATE --> KIND --> NEGATION_CLOSE
```

would qualify. Note that detection succeeds only if the parentheses match, and that they may be nested.

```
int prop_is_a_group(pcalc_prop *prop, int governing) {
    int match = element_get_match(governing), level = 0;
    if (match == 0) internal_error("prop_is_a_group called on unmatchable");
    TRAVERSE_VARIABLE(p);
    if ((prop == NULL) || (prop->element != governing)) return FALSE;
    TRAVERSE_PROPOSITION(p, prop) {
        if (element_get_group(p->element) == OPEN_OPERATORS_GROUP) level++;
        if (element_get_group(p->element) == CLOSE_OPERATORS_GROUP) level--;
    }
    if ((p_prev->element == match) && (level == 0)) return TRUE;
    return FALSE;
}
```

The function prop_is_a_group is called from 6/defer.

§**21.** The following removes matched parentheses, leaving just the interior:

```
pcalc_prop *prop_remove_topmost_group(pcalc_prop *prop) {
    TRAVERSE_VARIABLE(p);
    if ((prop == NULL) || (prop_is_a_group(prop, prop->element) == FALSE))
        internal_error("tried to remove topmost group which wasn't there");
    LOGIF(WORKINGS, "ungrouping proposition: $D\n", prop);
    prop = prop->next;
    TRAVERSE_PROPOSITION(p, prop)
        if ((p->next) && (p->next->next == NULL)) { p->next = NULL; break; }
    LOGIF(WORKINGS, "to ungrouped result: $D\n", prop);
    return prop;
}
```

The function prop_remove_topmost_group is called from 6/defer.

§**22.** The main application of which is to remove negation:

```
pcalc_prop *prop_unnegate(pcalc_prop *prop) {
    if (prop_is_a_group(prop, NEGATION_OPEN_ATOM))
        return prop_remove_topmost_group(prop);
    return NULL;
}
```

The function prop_unnegate is called from 11/chron.

§**23.** More ambitiously, this removes matched parentheses found at any given point in a proposition (which can continue after the close bracket).

```
pcalc_prop *prop_ungroup_after(pcalc_prop *prop, pcalc_prop *position, pcalc_prop **last) {
    TRAVERSE_VARIABLE(p);
    pcalc_prop *from;
    int opener, closer, level;
    LOGIF(WORKINGS, "removing frontmost group from proposition: $D\n", prop);
    if (position == NULL) from = prop; else from = position->next;
    opener = from->element;
    closer = element_get_match(opener);
    if (closer == 0) internal_error("tried to remove frontmost group which doesn't open");
    from = from->next;
    prop = prop_delete_atom(prop, position);                                  remove opening atom
    if (from->element == closer) {                                  the special case of an empty group
        prop = prop_delete_atom(prop, position);                              remove opening atom
        goto Ungrouped;
    }
    level = 0;
    TRAVERSE_PROPOSITION(p, from) {
        if (p->element == opener) level++;
        if (p->element == closer) level--;
        if (level < 0) {
            if (last) *last = p_prev;
            prop = prop_delete_atom(prop, p_prev);                            remove closing atom
            goto Ungrouped;
        }
    }
    internal_error("tried to remove frontmost group which doesn't close");
    Ungrouped:
    LOGIF(WORKINGS, "to ungrouped result: $D\n", prop);
    return prop;
}
```

The function prop_ungroup_after is called from 6/simp.

§**24.** Occasionally we want to strip away a "for all", and since that is always followed by a domain specification, we must also ungroup this:

```
pcalc_prop *prop_trim_universal_quantifier(pcalc_prop *prop) {
    if ((atom_is_for_all_x(prop)) &&
        (prop_match(prop, 2, QUANTIFIER_ATOM, NULL, DOMAIN_OPEN_ATOM, NULL))) {
        prop = prop_ungroup_after(prop, prop, NULL);
        prop = prop_delete_atom(prop, NULL);
        LOGIF(WORKINGS, "prop_trim_universal_quantifier: $D\n", prop);
    }
    return prop;
}
```

The function prop_trim_universal_quantifier is called from 7/vasp.

§**25.** Less ambitiously:

```
pcalc_prop *prop_remove_final_close_domain(pcalc_prop *prop, int *move_domain) {
    *move_domain = FALSE;
        TRAVERSE_VARIABLE(p);
        TRAVERSE_PROPOSITION(p, prop)
            if ((p->next == NULL) && (p->element == DOMAIN_CLOSE_ATOM)) {
                *move_domain = TRUE;
                return prop_delete_atom(prop, p_prev);
            }
    return prop;
}
```

The function prop_remove_final_close_domain is called from 6/sconv.

Binding and Substitution 6/vars

Purpose

To substitute constants into propositions in place of variables, and to apply quantifiers to bind any unbound variables.

6/vars.§1-5 Well-formedness; §6-9 Renumbering; §10 Binding; §11-14 Substitution; §15-16 A footnote on variable 0

Definitions

¶1. In any given proposition:

(a) a variable is *unused* if it is never mentioned as, or in, any term, and is not the variable of any quantifier;

(b) a variable is *bound* if it appears as the variable of any `QUANTIFIER_ATOM`;

(c) a variable is *free* if it is used but not bound.

These are mutually exclusive (no two can be true at the same time), and in any given proposition, each of the 26 variables is always either unused, bound or free.

In this section we are concerned with three operations applied to propositions:

(a) *substitution* means replacing each mention of a given variable with a given constant: for instance, changing x to 3 throughout ("substituting $x = 3$"). This has no effect if x is unused, and is illegal if x is bound, since it could produce nonsense like "for all 3, 3 is odd".

(b) *binding* means adding a new quantifier to a proposition, ranging some variable v. If v were unused this would be unlikely to be sensible (it would just make an inefficient way to test the size of the domain set), whereas if v were already a bound variable then the result would be a proposition which is no longer well-formed. So binding can only be done to free variables.

(c) *renumbering* means replacing each mention of a given variable v with another variable w. Clearly w needs to be initially unused, or we could accidentally change "v is greater than w" into "w is greater than w". But provided w is unused, the proposition's truth or otherwise remains unchanged.

¶2. Propositions with free variables are vague, and we would like to get rid of them. It can be very difficult to guess their values, just as subtle human understanding seems to be needed to interpret pronouns like "it" (see the enormous literature on the donkey anaphora problem in linguistics). So we aim to translate excerpts of source text into just two kinds of proposition:

(a) an *S-proposition* which has no free variables – such as the result of translating "The tree is in the Courtyard" or "Every door is open";

(b) an *SN-proposition* in which only variable 0 (x) is free – such as the result of translating "open containers which are in lighted rooms", which comes out to a proposition $\phi(x)$ testing whether x is one.

Whole English sentences or conditions make S-propositions, but descriptions make SN-propositions. (By renumbering, any proposition with one free variable can be made into an SN-proposition.)

---

§**1. Well-formedness.** It might seem logical to have a routine which takes a proposition ϕ and a variable v and returns its status – unused, free or bound. But this would be inefficient, since we want to work with all 26 at once, so instead we take a pointer to an array of `int` which needs to have (at least, but probably exactly) 26 entries, and on exit each entry is set to one of the following. In the course of doing that, it's easy to test whether variables are used properly – a bound variable should occur for the first time in its quantification, and should not reoccur once the subexpression holding the quantifier has finished. We set the `valid` flag if all is well.

```
define UNUSED_VST 1
define FREE_VST 2
define BOUND_VST 3
```

```
void vars_determine_status(pcalc_prop *prop, int *var_states, int *valid) {
    TRAVERSE_VARIABLE(p);
    int j, unavailable[26], blevel = 0, dummy;
    if (valid == NULL) valid = &dummy;
    *valid = TRUE;
    for (j=0; j<26; j++) { var_states[j] = UNUSED_VST; unavailable[j] = 0; }
    TRAVERSE_PROPOSITION(p, prop) {
        if (element_get_group(p->element) == OPEN_OPERATORS_GROUP) blevel++;
        if (element_get_group(p->element) == CLOSE_OPERATORS_GROUP) {
            blevel--;
            for (j=0; j<26; j++) if (unavailable[j] > blevel) unavailable[j] = -1;
        }
        for (j=0; j<p->arity; j++) {
            int v = term_variable_underlying(&(p->terms[j]));
            if (v >= 0) {
                if (unavailable[v] == -1) {
                    *valid = FALSE;
                    LOG("$o invalid because of %c unavailable\n", p, pcalc_vars[v]);
                }
                if (p->element == QUANTIFIER_ATOM) {
                    if (var_states[v] != UNUSED_VST) {
                        *valid = FALSE;
                        LOG("$o invalid because of %c Q for F\n", p, pcalc_vars[v]);
                    }
                    var_states[v] = BOUND_VST; unavailable[v] = blevel;
                } else {
                    if (var_states[v] == UNUSED_VST) var_states[v] = FREE_VST;
                }
            }
        }
    }
}
```

The function vars_determine_status is called from 6/tcpr, 6/asp and 6/cdefp.

§**2.** With just a little wrapping, this gives us the test of well-formedness.

```
int prop_is_well_formed(pcalc_prop *prop) {
    int status, var_states[26];
    if (prop_is_syntactically_valid(prop) == FALSE) return FALSE;
    vars_determine_status(prop, var_states, &status);
    if (status == FALSE) { LOG("Variable usage malformed\n"); return FALSE; }
    return TRUE;
}
```

The function prop_is_well_formed is called from 6/sconv, 6/tcpr and 6/defer.

§**3.** Occasionally we really do care only about one of the 26 variables:

```
int status_of_var(pcalc_prop *prop, int v) {
    int var_states[26];
    if (v == -1) return UNUSED_VST;
    vars_determine_status(prop, var_states, NULL);
    return var_states[v];
}
```

The function status_of_var is called from 6/sconv.

§**4.** To distinguish sentences from descriptions, the following can be informative:

```
int vars_number_free(pcalc_prop *prop) {
    int var_states[26], j, c;
    vars_determine_status(prop, var_states, NULL);
    for (j=0, c=0; j<26; j++) if (var_states[j] == FREE_VST) c++;
    LOGIF(WORKINGS, "There %s %d free variable%s in $D\n",
        (c==1)?"is":"are", c, (c==1)?"":"s", prop);
    return c;
}
```

The function vars_number_free is called from 6/sconv, 6/defer, 7/vasp and 8/knowc.

§**5.** While this gives us a new variable which can safely be added to an existing proposition:

```
int vars_find_unused(pcalc_prop *prop) {
    int var_states[26], j;
    vars_determine_status(prop, var_states, NULL);
    for (j=0; j<26; j++) if (var_states[j] == UNUSED_VST) return j;
    return 25;                                          the best we can do: it avoids crashes, at least...
}
```

The function vars_find_unused is called from 6/sconv and 6/simp.

§**6. Renumbering.** Another "vector operation" on variables: to renumber them throughout a proposition according to a map array. If `renumber_map[j]` is -1, make no change; otherwise each instance of variable j should be changed to this new number.

Note that because `QUANTIFIER_ATOM`s store the variable being quantified as a term, the following changes quantification variables as well as predicate terms, which is as it should be.

```
void vars_map(pcalc_prop *prop, int *renumber_map) {
    TRAVERSE_VARIABLE(p);
    int j;
    TRAVERSE_PROPOSITION(p, prop)
        for (j=0; j<p->arity; j++) {
            pcalc_term *pt = &(p->terms[j]);
            while (pt->function) pt=&(pt->function->fn_of);
            if ((pt->variable >= 0) && (renumber_map[pt->variable] >= 0))
                pt->variable = renumber_map[pt->variable];
        }
}
```

§**7.** The following takes any proposition and edits it so that the variables used are the lowest-numbered ones; moreover, variables are introduced in numerical order – that is, the first mentioned will be x, then the next introduced will be y, and so on.

```
void vars_renumber(pcalc_prop *prop) {
    TRAVERSE_VARIABLE(p);
    int j, k, renumber_map[26];
    for (j=0; j<26; j++) renumber_map[j] = -1;
    k = 0;
    TRAVERSE_PROPOSITION(p, prop)
        for (j=0; j<p->arity; j++) {
            int v = term_variable_underlying(&(p->terms[j]));
            if ((v >= 0) && (renumber_map[v] == -1)) renumber_map[v] = k++;
        }
    vars_map(prop, renumber_map);
}
```

The function vars_renumber is called from 6/sconv and 6/simp.

§**8.** This more complicated routine renumbers bound variables in one proposition in order to guarantee that none of them coincides with a variable used in a second proposition. This is needed in order to take the conjunction of two propositions, because "for all x, x is a door" and "there exists x such that x is a container" mean different things by x; they can only be combined in a single proposition if one of the x variables is changed to, say, y.

The surprising thing here is the asymmetry. Why do we only renumber to avoid clashes with bound variables in `prop` – why not free ones as well? The answer is that we use a form of conjunction in Inform which assumes that a free variable in ϕ has the same meaning as it does in ψ; thus in conjoining "open" with "lockable" we assume that the same thing is meant to be both open and lockable. If we renumbered to avoid clashes in free variables, we would produce a proposition meaning that one unknown thing is open, and another one lockable: that would have two free variables and be much harder to interpret.

If we pass a `query` parameter which is a valid variable number, the routine returns its new identity when renumbered.

```
int vars_renumber_bound(pcalc_prop *prop, pcalc_prop *not_to_overlap, int query) {
    int prop_vstates[26], nto_vstates[26], renumber_map[26];
    int j, next_unused;
    vars_determine_status(prop, prop_vstates, NULL);
    vars_determine_status(not_to_overlap, nto_vstates, NULL);
    for (j=0, next_unused=0; j<26; j++)
        if ((prop_vstates[j] == BOUND_VST) && (nto_vstates[j] != UNUSED_VST)) {
            ⟨Advance to the next variable not used in either proposition 9⟩;
            renumber_map[j] = next_unused++;
        } else renumber_map[j] = -1;
    vars_map(prop, renumber_map);
    if (query == -1) return -1;
    if (renumber_map[query] == -1) return query;
    return renumber_map[query];
}
```

The function vars_renumber_bound is called from 6/prop and 6/sconv.

§**9.** Again, we fall back on variable 25 if we run out. (This can only happen if the conjunction of the two propositions had 26 variables.)

⟨Advance to the next variable not used in either proposition 9⟩ ≡

```
    int k;
    for (k=next_unused; (k<26) &&
        (!((prop_vstates[k] == UNUSED_VST) && (nto_vstates[k] == UNUSED_VST))); k++) ;
    if (k == 26) next_unused = 25; else next_unused = k;
```

This code is used in §8.

§**10. Binding.** In this routine, we look for free variables and preface the proposition with ∃ quantifiers to bind them. For instance, $open(x)$ becomes $\exists x : open(x)$.

We first renumber the proposition's variables from left to right, and then quantify in reverse order – thus starting with the innermost free variable and working outwards (i.e., towards the left). Since at each stage we are prefacing the proposition, though, the net effect is that in the final proposition the previously free variables are bound in increasing order. For instance:

$$in(x, y) \quad \rightarrow \quad \exists y : in(x, y) \quad \rightarrow \quad \exists x : \exists y : in(x, y)$$

```
pcalc_prop *vars_bind_existential(pcalc_prop *prop) {
    int var_states[26], j;
    vars_renumber(prop);
    vars_determine_status(prop, var_states, NULL);
    for (j=25; j>=0; j--)
        if (var_states[j] == FREE_VST)
            prop = prop_insert_atom(prop, NULL,
                atom_QUANTIFIER_new(exists_quantifier, j, 0));
    return prop;
}
```

The function vars_bind_existential is called from 6/sconv.

§**11. Substitution.** In the following, we substitute term T (a constant or function) in place of variable v in the given proposition. We begin with two utility routines to substitute into the variable "underneath" a given term.

```
int substitute_v_in_term(pcalc_term *pt, int v, pcalc_term *t) {
    if (pt->variable == v) { *pt = *t; return TRUE; }
    if (pt->function) return substitute_v_in_term(&(pt->function->fn_of), v, t);
    return FALSE;
}
void substitute_nothing_in_term(pcalc_term *pt, pcalc_term *t) {
    if ((pt->constant) && (spec_is_nothing_object_constant(pt->constant))) { *pt = *t; return; }
    if (pt->function) substitute_nothing_in_term(&(pt->function->fn_of), t);
}
void substitute_term_in_term(pcalc_term *pt, pcalc_term *t) {
    if (pt->constant) { *pt = *t; return; }
    if (pt->function) substitute_term_in_term(&(pt->function->fn_of), t);
}
```

The function substitute_nothing_in_term is called from 6/simp.
The function substitute_term_in_term is called from 6/simp.

§**12.** Now the main procedure. This is one of those deceptive problems where the actual algorithm is obvious, but the circumstances when it can validly be applied are less so.

The difficulty depends on the term T being substituted in for the variable v. In general every term is a chain of functions with, right at the end, either a constant or a variable. If a constant is underneath, there is no problem at all. But if there is a variable underneath T – a VUT, as we say below – then it's possible that the substitution introduces circularities which would make it invalid. If that happens, we run into this:

```
define DISALLOW(msg) {
    if (verify_only) { *allowed = FALSE; return prop; }
    internal_error(msg);
}
```

§**13.** So the routine is intended to be called twice: once to ask if the situation looks viable, and once to perform the substitution itself.

```
pcalc_prop *vars_substitute_term(pcalc_prop *prop, int v, pcalc_term t,
    int verify_only, int *allowed, int *changed) {
    TRAVERSE_VARIABLE(p);
    if (verify_only) *allowed = TRUE;
    if ((v<0) || (v>=26)) DISALLOW("variable substitution out of range");
    if (prop_is_well_formed(prop) == FALSE) DISALLOW("substituting into malformed prop");
    ⟨Make sure the substitution would not fail because of a circularity 14⟩;
    if (verify_only) return prop;
    LOGIF(WORKINGS, "Substituting %c = $0 in: $D\n", pcalc_vars[v], &t, prop);
    TRAVERSE_PROPOSITION(p, prop) {
        int i;
        for (i=0; i<p->arity; i++)
            if (substitute_v_in_term(&(p->terms[i]), v, &t))
                *changed = TRUE;
    }
    if (prop_is_well_formed(prop) == FALSE) internal_error("substitution made malformed prop");
    return prop;
}
```

The function vars_substitute_term is called from 6/simp.

§14. The problem we might find, then, is that setting $v = T$ will be circular because T itself depends on v. There are two ways this can happen: first, T might be directly a function of v itself, i.e., the VUT might be v; second, T might be a function of some variable w which, by being quantified after v, is allowed to depend on it, in some way that we can't determine. (For examples of this, see "Simplifications".)

The general rule, then, is that T can contain only constants or variables which are free *within and after the scope of* v. (If w is bound outside the scope of v but after it, this means w didn't exist at the time that v did, and the attempted substitution would produce a proposition which isn't well-formed – w would occur before its quantifier.) We can check this condition pretty easily, it turns out:

⟨Make sure the substitution would not fail because of a circularity 14⟩ ≡

```
    if ((verify_only == FALSE) && (status_of_var(prop, v) == BOUND_VST))
        DISALLOW("substituting bound variable");
    int vut = term_variable_underlying(&t);
    if (vut >= 0) {
        int v_has_been_seen = FALSE;
        if (v == vut) DISALLOW("resubstituting same variable");
        TRAVERSE_PROPOSITION(p, prop) {
            if (v_has_been_seen == FALSE) {
                int i;
                for (i=0; i<p->arity; i++)
                    if (term_variable_underlying(&(p->terms[i])) == v)
                        v_has_been_seen = TRUE;
            }
            if ((p->element == QUANTIFIER_ATOM) && (p->terms[0].variable == vut) &&
                (v_has_been_seen))
                    DISALLOW("substituted value may be circular");
        }
    }
```

This code is used in §13.

§15. A footnote on variable 0. Because of the special status of x (variable 0) – the one allowed to be free in SN-propositions – we sometimes need to know about it. The range of a bound variable can be found by looking at its quantifier, but a free variable can remain ambiguous. The presence of a `KIND` atom will explicitly solve the problem for us; if we don't find one, though, we will simply have to assume that the set of objects is the domain of x. (We return `NULL` here, but that's the assumption which the caller will have to make.)

```
world_object *vars_kind_of_variable_0(pcalc_prop *prop) {
    TRAVERSE_VARIABLE(p);
    TRAVERSE_PROPOSITION(p, prop)
        if ((p->element == KIND_ATOM) && (p->terms[0].variable == 0)) {
            world_object *wo = kovko_get_kind(p->assert_kind_of_value);
            if (wo) return wo;
        }
    return NULL;
}
```

The function vars_kind_of_variable_0 is called from 8/refpt.

§**16.** And a quick way to substitute it:

```
pcalc_prop *substitute_var_0_in(pcalc_prop *prop, specification *spec) {
    int bogus;
    return vars_substitute_term(prop, 0, term_new_constant(spec), FALSE, NULL, &bogus);
}
```

The function substitute_var_0_in is called from 6/defer.

Propositional Form 6/pform

Purpose

The first of the three sources of propositions to conjure with: those which arise by conversion of existing data, in the form of type specifications.

Definitions

¶1. It is of course a sin for the same concept to have two different data representations in one computer program. But natural language subtly blurs the distinction between conditions and nouns in a way which lures us into this sin. Consider the word "closed". It's the name of a property, and as such is a noun; but it's also an adjective, and in a sentence like "if the marble tomb is closed" it clearly represents a condition.

In any case, we have two different needs. Every description must eventually be stored as a proposition, so that we can test or assert it. But we also need to be able to look at the structure of the original form of the description – was it one adjective, or two? With a common noun like "number", or a proper one like "Queen Mary"? And the propositional form disguises that, because of simplification, rearrangement and so on.

We therefore store simpler descriptions, such as "closed lockable door", by attaching a sort of transcription to their specifications; we write "closed, lockable" and "door" into relevant fields of an attached structure called the "description docket" of the SP. This is how such SPs are stored when parsed, and in the early stages of work on them.

More complicated descriptions with relative clauses, such as "women who like Mr Darcy", are stored in propositional form as soon as parsed.

Eventually we need to test or assert that a description applies to something. If the SP is still stored in docket form, we have to perform a conversion of that data into a (fairly simple) proposition before we can proceed. This is the main use for the single routine in this section.

---

§**1.** The following routine takes a SP and returns the best proposition it can, with a single unbound variable, to represent SP. In general, if V is a value, then this will be the proposition $is(x, V)$, but if it is possible to construe V as a list of one or more unary predicates (or `KIND` atoms, which are really unary predicates in disguise) then we return a logical conjunction of these predicates applied to the unbound x. For instance, the SP for the name of the property "closed" would become the proposition $closed(x)$.

```
pcalc_prop *prop_from_spec(specification *spec, int quantify) {
    pcalc_prop *prop = NULL;
    if (spec == NULL) return NULL;                                    the null description is universally true
    if (spec_get_proposition(spec)) return spec_get_proposition(spec);    a propositional form is already made
    int c1, c2;
    spec_get_calling(spec, &c1, &c2);
    if (c1 >= 0) prop = prop_concatenate(prop,
        atom_CALLED_new(c1, c2, term_new_variable(0), spec_get_described_kov(spec)));
    ⟨If this is a named constant for a KOV, but can be used adjectivally, convert it as such 2⟩;
    ⟨If it's an either-or property name, it must be being used adjectivally 3⟩;
    ⟨If it isn't a description at this point, it must be an ordinary noun 4⟩;
    ⟨Begin the proposition for a description-like SP with an atom for the head noun 5⟩;
    ⟨Concatenate atoms for any adjectives in the description 6⟩;
    if (quantify) ⟨Bind with a quantifier if the SP marks for one 7⟩;
    ⟨Typecheck the propositional form, and return 8⟩;
}
```

The function prop_from_spec is called from 6/sconv, 6/defer, 7/cosp, 8/knowc and 11/ap.

§**2.** For example, if we have written:

Colour is a kind of value. The colours are pink, green and black. A thing has a colour.

then "pink" is both a noun and an adjective. If SP is its representation as a noun, we return the proposition testing it adjectivally: *pink*(x).

```
⟨If this is a named constant for a KOV, but can be used adjectivally, convert it as such 2⟩ ≡
    quantity *q = spec_get_constant_quantity_if_any(spec);
    if (q) {
        property_name *pname = kov_get_coinciding_property(qty_kind_of_value(q));
        if (pname) {
            prop = prop_concatenate(prop,
                    atom_unary_PREDICATE_from_aph(qty_get_adjectival_phrase(q), FALSE));
            ⟨Typecheck the propositional form, and return 8⟩;
        }
    }
```

This code is used in §1.

§**3.** For example, if the SP is "scenery", we return the proposition *scenery*(x).

```
⟨If it's an either-or property name, it must be being used adjectivally 3⟩ ≡
    if (spec_is_CONSTANT_of_kova(spec, PROPERTY_TY)) {
        property_name *prn = PROPERTY_spec_to_property_name(spec);
        if (prn_is_either_or(prn)) {
            prop = prop_concatenate(prop, atom_unary_PREDICATE_from_aph(
                    prn_either_or_get_aph(prn), FALSE));
            ⟨Typecheck the propositional form, and return 8⟩;
        }
    }
```

This code is used in §1.

§**4.** For example, if the SP is the number 17, we return the proposition *is*$(x, 17)$.

```
⟨If it isn't a description at this point, it must be an ordinary noun 4⟩ ≡
    if ((species_is(spec, DESCRIPTION_SPC) == FALSE) &&
        (spec_is_CONSTANT_of_kova(spec, OBJECT_DESCRIPTION_TY) == FALSE) &&
        (spec_is_CONSTANT_of_kova(spec, VALUE_DESCRIPTION_TY) == FALSE)) {
        prop = prop_concatenate(prop, prop_x_is_constant(spec_copy(spec)));
        ⟨Typecheck the propositional form, and return 8⟩;
    }
```

This code is used in §1.

§**5.** A description need not have any heading noun – "open door" does ("door"), but "lockable" does not. If there is such a noun, it will either be a common noun – the name of a kind or KOV – or a proper noun – the name of a specific object. Common nouns translate directly into `KIND_ATOM`s, while proper nouns can be treated exactly as we would treat a noun like "54".

⟨Begin the proposition for a description-like SP with an atom for the head noun 5⟩ ≡

```
pcalc_prop *head = NULL;
if (spec_get_described_kind(spec))
    head = atom_KIND_object_new(spec_get_described_kind(spec), term_new_variable(0));
else if (spec_get_described_kov(spec))
    head = atom_KIND_value_new(spec_get_described_kov(spec), term_new_variable(0));
else if (spec_get_described_object(spec)) {
    specification *the_object = world_object_to_OBJECT_spec(spec_get_described_object(spec));
    head = prop_x_is_constant(the_object);
}
if (head) prop = prop_concatenate(prop, head);
```

This code is used in §1.

§**6.** So now we extract the linked list of adjectives attached to the SP (if any) and convert them into atoms:

⟨Concatenate atoms for any adjectives in the description 6⟩ ≡

```
adjective_list_entry *ale;
LOOP_THROUGH_ADJECTIVE_LIST(ale, spec) {
    adjectival_phrase *aph = get_adjective_from_list_entry(ale);
    int negated = FALSE;
    if (adjective_used_positively(ale) == FALSE) negated = TRUE;
    prop = prop_concatenate(prop, atom_unary_PREDICATE_from_aph(aph, negated));
}
```

This code is used in §1.

§**7.** This is something we don't always want:

⟨Bind with a quantifier if the SP marks for one 7⟩ ≡

```
if (spec_get_quantifier(spec)) {
    if (spec_get_quantifier(spec) != exists_quantifier)
        prop = prop_concatenate(atom_new(DOMAIN_OPEN_ATOM), prop);
    prop = prop_concatenate(atom_QUANTIFIER_new(spec_get_quantifier(spec), 0,
        spec_get_quantification_parameter(spec)), prop);
    if (spec_get_quantifier(spec) != exists_quantifier)
        prop = prop_concatenate(prop, atom_new(DOMAIN_CLOSE_ATOM));
}
```

This code is used in §1.

§**8.** In all cases, we finish by doing the following. In the one-atom noun cases it's a formality, but we want to enforce the rule that all propositions created in Inform go through type-checking, so:

⟨Typecheck the propositional form, and return 8⟩ ≡

```
prop_type_check(prop, tc_no_problem_reporting());
return prop;
```

This code is used in §1,2,3,4,1,2,3,4,1,2,3,4.

Tree Conversions 6/treec

Purpose

The second of the three sources of propositions to conjure with: those which arise from subtrees constructed by the A-parser.

§**1. Elementary propositions.** The S-parser attacks whole sentences, but the A-parser more cautiously takes on only fragments, and so it makes much shorter propositions – usually with a single free variable representing an object or value to which something must be done.

We start with some elementary propositions which create things, by asserting their existence.

```
pcalc_prop *prop_to_make_a_kind(void) {
    return atom_ISAKIND_new(term_new_variable(0));
}
pcalc_prop *prop_to_make_a_kov(void) {
    return atom_ISAKOV_new(term_new_variable(0));
}
pcalc_prop *prop_to_make_a_var(void) {
    return atom_ISAVAR_new(term_new_variable(0));
}
pcalc_prop *prop_to_create_something(kind_of_value *kov, int w1, int w2) {
    pcalc_prop *prop = atom_QUANTIFIER_new(exists_quantifier, 0, 0);
    if (kovko_get_kind(kov))
        prop = prop_concatenate(prop,
            atom_KIND_object_new(kovko_get_kind(kov), term_new_variable(0)));
    else if ((kov) && (is_kova(kov, OBJECT_TY) == FALSE))
        prop = prop_concatenate(prop,
            atom_KIND_value_new(kov, term_new_variable(0)));
    if (w1 >= 0)
        prop = prop_concatenate(prop,
            atom_CALLED_new(w1, w2, term_new_variable(0), kov));
    return prop;
}
```

The function prop_to_make_a_kind is called from 8/creat.

The function prop_to_make_a_kov is called from 8/creat and 9/prop.

The function prop_to_make_a_var is called from 8/creat.

The function prop_to_create_something is called from 8/creat, 9/prop, 9/pp and 10/tab.

§**2.** A proposition to assert that an object has a given kind:

```
pcalc_prop *prop_to_set_kind(world_object *wo, world_object *k) {
    return atom_KIND_value_new(kovko(k), term_new_variable(0));
}
```

The function prop_to_set_kind is called from 8/knowc.

§**3.** Now propositions to assert that relations hold:

```
pcalc_prop *prop_to_set_mapping_relation(binary_predicate *bp, world_object *wo) {
    specification *spec;
    if (wo) spec = world_object_to_OBJECT_spec(wo);
    else spec = new_nothing_object_constant();
    return atom_binary_PREDICATE_new(bp,
        term_new_variable(0), term_new_constant(spec));
}
pcalc_prop *prop_to_set_relation(binary_predicate *bp,
    world_object *wo0, specification *spec0, world_object *wo1, specification *spec1) {
    pcalc_term pt0, pt1;
    pcalc_prop *prop;
    if (wo0) pt0 = term_new_constant(world_object_to_OBJECT_spec(wo0));
    else pt0 = term_new_constant(spec0);
    if (wo1) pt1 = term_new_constant(world_object_to_OBJECT_spec(wo1));
    else pt1 = term_new_constant(spec1);
    prop = atom_binary_PREDICATE_new(bp, pt0, pt1);
    int dummy;
    prop = simp_make_kinds_of_value_explicit(prop, &dummy);
    return prop;
}
```

The function prop_to_set_mapping_relation is called from 8/relv.
The function prop_to_set_relation is called from 8/knowp and 8/relv.

§**4.** Property provision is itself a relation:

```
pcalc_prop *prop_to_provide_property(property_name *prn) {
    return atom_binary_PREDICATE_new(a_provides_b_predicate,
        term_new_variable(0), term_new_constant(property_name_to_PROPERTY_spec(prn)));
}
pcalc_prop *prop_to_set_property(property_name *prn, specification *val) {
    if (val == NULL) return NULL;
    return atom_binary_PREDICATE_new(prn_get_setting_bp(prn),
        term_new_variable(0), term_new_constant(val));
}
```

The function prop_to_provide_property is called from 5/rel, 9/pp and 10/tab.
The function prop_to_set_property is called from 8/knowp.

§**5.** "Everywhere" and "here":

```
pcalc_prop *prop_to_put_everywhere(void) {
    return atom_EVERYWHERE_new(term_new_variable(0));
}
pcalc_prop *prop_to_put_here(void) {
    return atom_HERE_new(term_new_variable(0));
}
```

The function prop_to_put_everywhere is called from 8/relv.
The function prop_to_put_here is called from 8/relv.

§**6. Property setting.** Sometimes the A-parser wants to assert that a given property has the value whose text can be found in a node `py`...

```
pcalc_prop *prop_from_property_subtree(property_name *prn, parse_node *py) {
    return prop_to_set_property(prn, property_value_from_property_subtree(prn, py));
}
```

The function prop_from_property_subtree is called from 8/knowp.

§**7.** ...and sometimes it wants to assert a more elaborate list, such as "carrying capacity 10 and weight 4kg" or "lockable unlocked". This is a syntax allowed by the A-parser but not the S-parser, and here is where we deal with it.

```
pcalc_prop *prop_from_property_list(parse_node *p, kind_of_value *kov) {
    pcalc_prop *prop = NULL;
    if (p) {
        switch(pn_get_node_type(p)) {
            case AND_NT:
                for (p = p->down; p; p = p->next)
                    prop = prop_conjoin(prop, prop_from_property_list(p, NULL));
                break;
            case PROPERTYLIST_NT:
                ⟨Conjoin atoms to assert from a property list 8⟩;
                break;
            case ADJECTIVE_NT:
                ⟨Conjoin atoms to assert from an adjective node 11⟩;
                break;
            default: internal_error_on_node_type(p);
        }
    }
    if ((prop) && (kov))
        prop = prop_conjoin(atom_KIND_value_new(kov, term_new_variable(0)), prop);
    return prop;
}
```

The function prop_from_property_list is called from 8/knowp.

§**8.** Recall that a PROPERTYLIST_NT node is unannotated, as yet, and we have to parse the text to find what which property is referred to.

⟨Conjoin atoms to assert from a property list 8⟩ ≡

```
property_name *prn = NULL;
int pw1 = -1, pw2 = 0, vw1 = -1, vw2 = 0;
⟨Divide the property list entry into property name and value text 9⟩;
if ((pw1 >= 0) && (pw1 <= pw2)) prn = parse_property_name(pw1, pw2);
if (prn == NULL) ⟨Issue a problem message for no-such-property 10⟩;
if ((vw1 >= 0) && (vw1 <= vw2)) {                                   a value is supplied...
    if (prn_is_either_or(prn)) {
        quote_source(1, current_sentence);
        quote_words(2, p->word_ref1, p->word_ref2);
        quote_property(3, prn);
        quote_words(4, vw1, vw2);
        handmade_problem(_P_(C6WithEitherOrValue));
        issue_problem_segment(
            "The sentence '%1' seems to be trying to create something which "
            "has '%2', where the %3 property is being set equal to %4. But "
            "this made no sense to me, because %3 is an either/or property "
            "and cannot have a value.");
        issue_problem_end();
        return NULL;
    }
    parse_node *pn = new_nounphrase_worldly(vw1, vw2, FALSE);
    resolve_references(pn);
    prop = prop_conjoin(prop, prop_from_property_subtree(prn, pn));
} else {                                                            no value is supplied...
    if (prn_is_either_or(prn) == FALSE) {
        quote_source(1, current_sentence);
        quote_words(2, p->word_ref1, p->word_ref2);
        quote_property(3, prn);
        quote_kov(4, prn_get_kind_of_value(prn));
        handmade_problem(_P_(C6WithValuelessValue));
        issue_problem_segment(
            "The sentence '%1' seems to be trying to create something which "
            "has '%2', where the %3 property is being set in some way. But "
            "this made no sense to me, because %3 is a value property (it "
            "needs to be %4) and no value for it was given here.");
        issue_problem_end();
        return NULL;
    }
    prop = prop_conjoin(prop, prop_from_property_subtree(prn, NULL));
}
```

This code is used in §7.

§**9.** The node has text in the form "property name property value", with no obvious division of punctuation between the two. What makes matters worse is that we do not yet know all the property names, nor do we have the ability to discern values. So we seek the division by

(i) trying to find the longest known property name at the start of the text; if there is no known name,
(ii) we see if the final word of the text is a literal, such as a number or a quoted text, and if so we assume this is the entire property value and that the rest is property name; and otherwise
(iii) we assume the property name is one word only.

⟨Divide the property list entry into property name and value text 9⟩ ≡

```
if ((p->word_ref1 < 0) || (p->word_ref2 < p->word_ref1)) {
    assertion_problem(_P_(BelievedImpossible),
        "this looked to me as if it might be trying to create something "
        "which has certain properties",
        "and that made no sense on investigation. This sometimes happens "
        "if a sentence uses 'to have' oddly?");
    return NULL;
}
int name_length = match_longest_pname(p->word_ref1, p->word_ref2);
if (name_length < 0) {
    name_length = 1;
    if (is_a_literal(p->word_ref2, p->word_ref2))
        name_length = p->word_ref2 - p->word_ref1;
}
pw1 = p->word_ref1; pw2 = p->word_ref1+name_length-1;
vw1 = pw2+1; vw2 = p->word_ref2;
```

This code is used in §8.

§**10.**

⟨Issue a problem message for no-such-property 10⟩ ≡

```
LOG("Failed property list: pname = <$W>; pval = <$W>\n", pw1, pw2, vw1, vw2);
assertion_problem(_P_(C6BadPropertyList),
    "this looked to me as if it might be trying to create something "
    "which has certain properties",
    "and that made no sense on investigation. This sometimes happens "
    "if a sentence uses 'with' a little too liberally, or to specify "
    "a never-declared property. For instance, 'An antique is a kind of "
    "thing with an age.' would not be the right way to declare the "
    "property 'age' (because it does not tell Inform what kind of "
    "value this would be). Instead, try 'An antique is a kind of "
    "thing. An antique has a number called age.' It would then be all "
    "right to say 'The Louis Quinze chair is an antique with age 241.'");
return NULL;
```

This code is used in §8.

§**11.** An `ADJECTIVE_NT` node, on the other hand, is annotated with a valid property name `property` already, and may also have a value ready to put into that property, stored in `evaluation`. Nodes like this have been created from descriptions like "open openable door in the kitchen", and it's important not to lose the location information ("in the kitchen"): so the full evaluation of the original phrase is preserved in the `full_phrase_evaluation` field.

⟨Conjoin atoms to assert from an adjective node 11⟩ ≡

```
int negate_me = FALSE;
if (pn_int_annotation(p, negated_boolean_ANNOT)) negate_me = TRUE;
if (pn_get_aph(p) == NULL)
    internal_error("Bogus adjective at assert_property_list");
if ((pn_get_full_phrase_evaluation(p) != NULL) &&
    (spec_get_proposition(pn_get_full_phrase_evaluation(p)))) {
    prop = prop_conjoin(prop, spec_get_proposition(pn_get_full_phrase_evaluation(p)));
    pn_set_full_phrase_evaluation(p, NULL);
}
if (negate_me) prop = prop_conjoin(prop, atom_new(NEGATION_OPEN_ATOM));
prop = prop_conjoin(prop, atom_unary_PREDICATE_from_aph(pn_get_aph(p), FALSE));
if (negate_me) prop = prop_conjoin(prop, atom_new(NEGATION_CLOSE_ATOM));
```

This code is used in §7.

Sentence Conversions 6/sconv

Purpose

The third of the three sources of propositions to conjure with: those which arise by the parsing of complex sentence trees in the S-grammar.

6/sconv.§1-15 The meaning of a sentence; §16 Simplification; §17-26 The meaning of a noun phrase

---

§**1. The meaning of a sentence.** This section provides a single, but crucial, function to the rest of Inform: it takes a sentence subtree output by the S-parser and turns it into a proposition.

The sentence subtree can be headed by either a `SV_PRODUCTION`, representing a whole sentence, which we turn into a proposition with no free variables; or by an `SN_PRODUCTION`, representing a description which includes a relative clause, such as "an animal which can see the player". In this section we will loosely refer to the text parsed by either sort of subtree as the "sentence".

The basic idea is simple. The sentence will have a verb phrase (VP), together with two noun phrases: a subject phrase (SP) and object phrase (OP). English is an SVO language, so phrases (usually) occur in the sequence SP-VP-OP. In "Katy examines the painting", "Katy" is the SP and "painting" is the OP. (Although the subject is sometimes the more active participant, that isn't always the case: in "the painting is examined by Katy", "the painting" is now the SP. The subject is what the sentence talks about.) At this point in the program, the S-parser has turned the sentence into a neat tree structure identifying the SP, VP and OP. We need to find meanings for the SP, VP and OP independently, and then combine these into a single proposition representing the meaning of the whole sentence.

```
int conv_log_depth = 0;                                   recursion depth: used only to clarify the debugging log
pcalc_prop *S_subtree_to_proposition(meaning_list *ml, pcalc_term *subject_of_sentence) {
    int w1, w2, SV_not_SN = FALSE;
    meaning_list *subject_phrase_subtree = NULL, *object_phrase_subtree = NULL;
    pcalc_prop *subject_phrase_prop, *object_phrase_prop;
    pcalc_term subject_phrase_term, object_phrase_term;
    binary_predicate *verb_phrase_relation = NULL;
    int verb_phrase_negated = FALSE;
    pcalc_prop *sentence_prop = NULL;
    ⟨Check the tree position makes sense, and tell the debugging log 2⟩;
    if (ml_production(ml_down(ml)) == THERE_PRODUCTION) {
        ⟨Handle a THERE subtree, used for "there is/are NP" 4⟩;
    } else {
        ⟨Find meaning of the VP as a relation and a parity 5⟩;
        ⟨Find meanings of the SP and OP as propositions and terms 10⟩;
        ⟨Bind up any free variable in the OP and sometimes the SP, too 11⟩;
        ⟨Combine the SP, VP and OP meanings into a single proposition for the sentence 15⟩;
    }
    ⟨Simplify the resultant proposition 16⟩;
    ⟨Tell the debugging log what the outcome of the sentence was 3⟩;
    if (subject_of_sentence) *subject_of_sentence = subject_phrase_term;
    return sentence_prop;
}
```

The function S_subtree_to_proposition is called from 5/mlc and 13/test.

§**2.** Like all good computer-science algorithms, this is essentially divide and conquer: we handle a multi-clause sentence by recursive use of two routines, `S_subtree_to_proposition` at each SV- or SN-node, and `NP_subtree_to_proposition` at each noun phrase. These are rather like coroutines: the recursion is always S calls NP calls S calls NP... and so on, the entry point always being S, and the bottom leaves of the recursion always being NPs. For example:

```
S_subtree_to_proposition "a man is in a room which is adjacent to the Garden"
    NP_subtree_to_proposition "a man"
    NP_subtree_to_proposition "a room which is adjacent to the Garden"
        S_subtree_to_proposition "a room which is adjacent to the Garden"
            NP_subtree_to_proposition "a room"
            NP_subtree_to_proposition "the Garden"
```

⟨Check the tree position makes sense, and tell the debugging log 2⟩ ≡

```
if (ml == NULL) internal_error("S_subtree_to_proposition on null S-subtree");
switch(ml_production(ml)) {
    case SN_PRODUCTION: SV_not_SN = FALSE; break;
    case SV_PRODUCTION: SV_not_SN = TRUE; break;
    default: internal_error("S_subtree_to_proposition on non-S-subtree");
}
s_subtree_error_set_position(ml);
if ((conv_log_depth > 0) && (SV_not_SN))
    internal_error("S_subtree_to_proposition on S-subtree with an SV- as lower node");
ml_get_text(ml, &w1, &w2);
if (conv_log_depth == 0) LOGIF(CALCULUS, "----------\n");
conv_log_depth++;
LOGIF(CALCULUS, "[%d] Starting S_subtree_to_proposition on: <$W>\n", conv_log_depth, w1, w2);
```

This code is used in §1.

§**3.** And similarly, on the way out:

⟨Tell the debugging log what the outcome of the sentence was 3⟩ ≡

```
LOGIF(CALCULUS, "[%d] S_subtree_to_proposition: $W --> $D\n",
    conv_log_depth, w1, w2, sentence_prop);
conv_log_depth--;
```

This code is used in §1.

§**4.** The English verb "to be" has the syntactic quirk that it likes to have both SP and OP, even when only when thing is being discussed. We say "it is raining" and "there are seven continents", inserting "it" and "there" even though they refer to nothing at all, because we don't like to say "raining is" or "seven continents are". (There are rare exceptions, but usually sportive ones. In Stoppard's play *Jumpers*, the theologian opens his lecture by asking: "Is God?", then wonders if it should be "Are God?" – but he is using the wonkiness of this grammar to make points.)

At any rate Inform parses a sentence in the form "There is X" or "There are Y" into a simpler form of tree with just one noun phrase, and no verb phrase at all. We convert the noun phrase to a proposition ϕ in which x is free, then bind it with $\exists x$ to form $\exists x : \phi(x)$, making an S-proposition as required.

⟨Handle a THERE subtree, used for "there is/are NP" 4⟩ ≡

```
if (SV_not_SN == FALSE) internal_error("THERE subtree misplaced");
specification *spec = DC_subtree_to_spec(
    ml_down(ml_down(ml_right(ml_down(ml_right(ml_down(ml)))))), FALSE);
sentence_prop = prop_from_spec(spec, TRUE);
sentence_prop = vars_bind_existential(sentence_prop);
```

This code is used in §1.

§**5.** Here we only locate the subject and object subtrees – their meanings we leave for later – but we do find the content of the verb phrase. Given the combination of verb and preposition usage (the latter optional), we extract a binary predicate B plus a parity flag indicating if it is meant positively or negatively.

Of course a VU also records the tense of the verb, but we ignore that here. It has no effect on the proposition, only on the moment in history to which it can be applied.

⟨Find meaning of the VP as a relation and a parity 5⟩ ≡

```
meaning_list *verb_phrase_subtree;
verb_usage *vu; preposition_usage *pu = NULL;
subject_phrase_subtree = ml_down(ml);
if (subject_phrase_subtree == NULL) s_subtree_error("SP subtree null");
verb_phrase_subtree = ml_right(subject_phrase_subtree);
if (verb_phrase_subtree == NULL) s_subtree_error("VP subtree null");
if (ml_down(verb_phrase_subtree) == NULL) s_subtree_error("VP subtree broken");
object_phrase_subtree = ml_right(ml_down(verb_phrase_subtree));
if (object_phrase_subtree == NULL) s_subtree_error("OP subtree null");
vu = RETRIEVE_POINTER_verb_usage(ml_get_attached_data(ml_down(verb_phrase_subtree)));
if (vu == NULL) s_subtree_error("verb null");
if ((SV_not_SN == FALSE) && (vu_get_tense_used(vu) != IS_TENSE))
    ⟨Disallow the past tenses in relative clauses 6⟩;
if (ml_down(ml_down(verb_phrase_subtree))) {
    if (ml_production(ml_down(ml_down(verb_phrase_subtree))) != PREP_PRODUCTION)
        s_subtree_error("child of VERB not a PREP");
    pu = RETRIEVE_POINTER_preposition_usage(
        ml_get_attached_data(ml_down(ml_down(verb_phrase_subtree))));
    if (pu == NULL) s_subtree_error("PREP null");
}
if (pu) verb_phrase_relation = pu_get_meaning(pu);
else verb_phrase_relation = vu_get_meaning(vu);
verb_phrase_negated = (vu_is_used_negatively(vu))?TRUE:FALSE;
if ((pu) && (pu_implicitly_negates(pu)))
    verb_phrase_negated = (verb_phrase_negated)?FALSE:TRUE;
```

This code is used in §1.

§**6.** A sad necessity:

⟨Disallow the past tenses in relative clauses 6⟩ ≡

```
sentence_problem(_P_(C6PastSubordinate),
    "subordinate clauses have to be in the present tense",
    "so 'the Black Door was open' is fine, but not 'if the Black Door is "
    "something locked which had been open'. Only the main verb can be in "
    "the past tense.");
```

This code is used in §5.

§**7. First Rule.** The "meaning" of a noun phrase is a pair (ϕ, t), where ϕ is a proposition and t is a term. We read this as "t such that ϕ is true". Exactly one of the following will always be true:

(a) If the NP talks about a single definite thing C, then $t = C$ and $\phi = T$, the empty proposition.

(b) If the NP talks about a single definite thing C but imposes conditions about its current situation, then $t = v$ for some variable v and $\phi = \exists v : is(v, C) \wedge \psi$ where ψ is a description of the conditions with no free variables and which does not use v.

(c) If the NP talks about a single but vague thing, identifying it only by its current situation, then $t = v$ for some variable v and ϕ is a proposition having v as its unique free variable.

(d) If the NP talks about a range, number or proportion of things, then $t = v$ for some variable v and $\phi = Qv : v \in \{v \mid \psi(v)\}$, where v is the unique free variable of ψ, and Q is a generalised quantifier which is not $\exists$.

§**8.** As examples of all four cases:

(a) "Reverend Green" returns $t =$ `Green`, $\phi = T$ – a single definite thing.

(b) "Colonel Mustard in the Library" returns $t = x$ such that $\phi = \exists x : is(x, \texttt{Mustard}) \wedge in(\texttt{Mustard}, \texttt{Library})$ – a single definite thing but subject to conditions.

(c) "A suspect carrying the lead piping" returns $t = x$ and $\phi = suspect(x) \wedge carries(x, \texttt{piping})$ – a single but vague thing.

(d) "All the weapons in the Billiard Room" returns $t = x$ and $\phi = \forall x : x \in \{x \mid weapon(x) \wedge in(x, \texttt{Billiard})\}$ – a range of things.

§**9.** Thus ϕ can contain at most 1 free variable, and then only in case (c). But why does it do so at all? Why do we return "an open door" as $open(x) \wedge door(x)$? It would be more consistent with the way we handle "two open doors" to return it as $\exists x : open(x) \wedge door(x)$. The answer is that if we were only parsing whole sentences (SV-trees) then it would make no difference, because x ends up bound by $\exists x$ anyway when the final sentence is being put together. But we also want to parse descriptions. Consider:

(1) let L be the list of open doors in the Dining Room;

(2) let L be the list of two open doors in the Dining Room;

Here (1) is legal in Inform, (2) is not, because it implies a requirement about the list which will probably not be satisfied. (Maybe there are three open doors there, maybe none.) In case (1), `NPstp` applied to "open doors" will return $open(x) \wedge door(x)$, whose free variable x can become any single object we might want to test for open-door-ness. But in case (2), `NPstp` applied to "two open doors" will return $V_{=2}x : open(x) \wedge door(x)$, and here x is bound, and can't be set equal to some object being tested.

Or to put this more informally: it's possible for a single item to be an "open door", but it's not possible for a single item to be (say) "more than three open doors". So ϕ contains a free variable if and only if the NP describes a single but vague thing.

§**10.** The First Rule is implemented by `NP_subtree_to_proposition` below, and we apply it independently to the SP and OP:

⟨Find meanings of the SP and OP as propositions and terms 10⟩ ≡

```
subject_phrase_prop =
    NP_subtree_to_proposition(&subject_phrase_term, subject_phrase_subtree,
        bp_term_kind_of_value(verb_phrase_relation, 0));
object_phrase_prop =
    NP_subtree_to_proposition(&object_phrase_term, object_phrase_subtree,
        bp_term_kind_of_value(verb_phrase_relation, 1));
LOGIF(CALCULUS, "[%d] subject NP: $0 such that: $D\n",
    conv_log_depth, &subject_phrase_term, subject_phrase_prop);
LOGIF(CALCULUS, "[%d] object NP: $0 such that: $D\n",
    conv_log_depth, &object_phrase_term, object_phrase_prop);
```

This code is used in §1.

§**11.** The First Rule tells us that SP and OP are now each represented by propositions with either no free variables, or just one, and then only if the phrase refers to a single but vague thing.

So far we have treated the subject and object exactly alike, running the same computation on meaning lists generated by the same method. This is the first point at which the placement as subject rather than object will start to make a difference:

(i) we always bind a free variable in the object, but
(ii) we only bind a free variable in the subject if we are looking at the topmost verb in a whole sentence (i.e., for an SV rather than SN subtree).

The SP is called the "subject phrase" because it contributes the subject of a sentence: what it is a sentence about. For instance, for an SV-subtree for "a woman is carrying an animal", we produce $\phi_S = \exists x : \mathit{woman}(x)$ and $\phi_O = \exists x : \mathit{animal}(x)$. But for an SN-subtree for "a woman carrying an animal" – which vaguely describes something, in a way that can be tested for any given candidate x – we produce $\phi_S = \mathit{woman}(x)$ with x remaining free.

⟨Bind up any free variable in the OP and sometimes the SP, too 11⟩ ≡

```
if (SV_not_SN) subject_phrase_prop = vars_bind_existential(subject_phrase_prop);
object_phrase_prop = vars_bind_existential(object_phrase_prop);
```

This code is used in §1.

§**12.** Of all the thousands of paragraphs of code in Inform, this is the one which most sums up "how it works". We started with a sentence in the source text, and have now extracted the following components of its meaning: the subject phrase (SP) has become the term t_S subject to the proposition ϕ_S being true; the object phrase (OP) is similarly now a pair t_O such that ϕ_O. From the verb phrase (VP), we have found a binary relation B, meant either in a positive sense (B does hold) or a negative one (it doesn't). And now:

Second Rule. The combined "meaning" Σ is as follows:

(1) if we are parsing a whole sentence (i.e., an SV-subtree), or ϕ_S is not in the form $Qx \in \{x \mid \theta(x)\}$, then:

$$\Sigma = \begin{cases} \phi_S \wedge \phi'_O \wedge B(t_S, t'_O) & \text{if sense positive} \\ \phi_S \wedge \neg(\phi'_O \wedge B(t_S, t'_O)) & \text{if sense negative} \end{cases}$$

(2) if we are parsing a relative clause (i.e., an SN-subtree), and ϕ_S is of the form $Qx \in \{x \mid \theta(x)\}$, then:

$$\Sigma = \begin{cases} Qx \in \{x \mid \theta(x) \wedge \phi'_O \wedge B(t_S, t'_O)\} & \text{if sense positive} \\ Qx \in \{x \mid \theta(x) \wedge \neg(\phi'_O \wedge B(t_S, t'_O))\} & \text{if sense negative} \end{cases}$$

Here ϕ'_O and t'_O are ϕ_O and t_O modified to relabel its variables so that there are no accidental clashes with variables named in ϕ_S.

§**13.** That simple rule took the author a long, long time to work out, so it may be worth a little discussion:

(a) The Second Rule is a generalisation of the way comparison operators like `==` or `>=` work in conventional programming languages. For if t_S and t_O are both constants, and ϕ_S and ϕ_O both empty, we obtain just $B(t_S, t_O)$ and $\neg(B(t_S, t_O))$. For instance, "score is 10" becomes just $is(\texttt{score}, 10)$, which compiles just to `(score == 10)`.

(b) In general, though, the meaning of an English sentence is not just that the verb is true, but also that the subject and object make sense. For "a woman is carrying an animal" to be true, there has to be such a woman, and such an animal. This is the content of ϕ_S and ϕ_O. So the formula above can be read as "the subject makes sense, and the object makes sense, and they relate to each other in the way that the verb claims".

(c) In the case of negation, it's important that we produce $\phi_S \wedge \neg(\phi'_O \wedge B(t_S, t_O))$ rather than $\phi_S \wedge \phi'_O \wedge \neg(B(t_S, t_O))$. To see the difference, consider the sentence "The box does not contain three coins". The first formula, which is correct, means roughly "it's not true that there are three coins x such that x is in the box", whereas the second, wrong, means "three coins x exist such that x is not in the box".

(d) The difference between cases (1) and (2) is actually very slight. Case (2) arises only when a relative clause is qualifying the range of a collection of things: for instance, in "every man who is in the Garden", we have $\phi_S = \forall x \in \{x \mid man(x)\}$ and then need to apply the relation $in(x, \texttt{Garden})$. If we used formula (1) we would then have

$$\Sigma = \forall x \in \{x \mid man(x)\} : in(x, \texttt{Garden})$$

which means "every man is in the Garden" – making a statement about everything covered by ϕ_S, not restricting the coverage of ϕ_S, as a relative clause should. Using formula (2), however, we get:

$$\Sigma = \forall x \in \{x \mid man(x) \wedge in(x, \texttt{Garden})\}$$

Note that these formulae are identical except for what we might call punctuation.

(e) The modification needed to make ϕ'_O out of ϕ_O is pretty well inconsequential. It makes no difference to the meaning of ϕ_O. Consider the example "a woman is carrying an animal" once again. $t_S = x$ and $\phi_S = woman(x)$, which use x; and on the other hand $t_O = x$ and $\phi_O = animal(x)$, which also use x. Clearly we don't mean the same x on both sides, so we relabel the OP to get y such that $animal(y)$. There is not really any asymmetry between the SP and OP here, because it would have been just as good to relabel the SP.

§**14.** **Lemma.** The result Σ of the Second Rule is a proposition containing either 0 or 1 free variables; Σ has 1 free variable if and only if we are converting an SN-subtree, and the subject phrase of the sentence describes a single thing vaguely.

Proof. ϕ_O contains no free variables, since we bound it up above, and the same must be true of its relabelled version ϕ'_O. If we have an SV-subtree then ϕ_S similarly contains no free variables; we only leave it unbound for an SN-subtree. In that case, the First Rule tells us that it has a free variable if and only if the SP describes a single thing vaguely. The only other content of Σ is the predicate $B(t_S, t'_O)$, so extra free variables can only appear if either t_S or t'_O contains a variable not already seen in ϕ_S and ϕ'_O. But cases (b), (c) and (d) of the First Rule make clear that in any pair (t, ϕ) arising from a noun phrase, either t is a constant or else it is a variable appearing in ϕ. So the terms of the final B predicate in Σ cannot add new free variables, and the lemma is proved.

By similar argument, if ϕ_S and ϕ_O are well-formed propositions (syntactically valid and using variables either freely or within the scope of quantification) then so is Σ.

§**15.** Now to implement the Second Rule:

⟨Combine the SP, VP and OP meanings into a single proposition for the sentence 15⟩ ≡

```
int use_case_2 = FALSE;
if (SV_not_SN == FALSE)
    subject_phrase_prop = prop_remove_final_close_domain(subject_phrase_prop, &use_case_2);
object_phrase_term.variable =
    vars_renumber_bound(object_phrase_prop, subject_phrase_prop, object_phrase_term.variable);
sentence_prop = subject_phrase_prop;
if (verb_phrase_negated)
    sentence_prop = prop_concatenate(sentence_prop, atom_new(NEGATION_OPEN_ATOM));
sentence_prop = prop_concatenate(sentence_prop, object_phrase_prop);
sentence_prop = prop_concatenate(sentence_prop,
    atom_binary_PREDICATE_new(verb_phrase_relation, subject_phrase_term, object_phrase_term));
if (verb_phrase_negated)
    sentence_prop = prop_concatenate(sentence_prop, atom_new(NEGATION_CLOSE_ATOM));
if (use_case_2)
    sentence_prop = prop_concatenate(sentence_prop, atom_new(DOMAIN_CLOSE_ATOM));
LOGIF(CALCULUS, "[%d] Initial meaning: $D\n", conv_log_depth, sentence_prop);
```

This code is used in §1.

§**16. Simplification.** Every proposition generated by `S_subtree_to_proposition`, whether it arises as "there is/are" plus a noun phrase or by the Second Rule, is simplified before being returned. Because of the way the recursion is set up, this means that intermediate propositions for relative clauses within a sentence are always simplified before being used to build the whole sentence.

What happens here is that we try a sequence of tactical moves to change the proposition for the better – which usually means eliminating bound variables, where we can: they are a bad thing because they compile to loops which may be slow and awkward to construct.

Simplifications are allowed to change Σ – indeed that's the whole idea – but not t_S, the term representing what the sentence talks about. (Indeed, they aren't even shown what it is.) Moreover, a simplification can only turn Σ to Σ' if:

(i) Σ' remains a syntactically correct proposition with well-formed quantifiers,
(ii) Σ' has the same number of free variables as Σ, and
(iii) in all situations and for all possible values of any free variables, Σ' is true if and only if Σ is.

Rules (i) and (ii) are checked as we go, with internal errors thrown if ever they should fail; the checking takes only a trivial amount of time, and I generally agree with Tony Hoare's maxim that removing checks like this in the program as shipped to users is like wearing a life-jacket while learning to sail on dry land, and then taking it off when going to sea. Still, rule (iii) can only be ensured by writing the routines carefully.

The simplification routines can all be found in "Simplifications".

```
define APPLY_SIMPLIFICATION(proposition, simp) {
    int changed = FALSE, NF = vars_number_free(proposition);
    if (proposition) proposition = simp(proposition, &changed);
    if (changed) LOGIF(CALCULUS, "[%d] %s: $D\n", conv_log_depth, #simp, proposition);
    if ((prop_is_well_formed(proposition) == FALSE) ||
        (NF != vars_number_free(proposition))) {
        LOG("Failed after applying %s: $D", #simp, proposition);
        internal_error(#simp " simplified proposition into one which is not well-formed");
    }
}
```

⟨Simplify the resultant proposition 16⟩ ≡

```
    if (prop_is_well_formed(sentence_prop) == FALSE) {
        LOG("Failed before simplification: $D", sentence_prop);
        internal_error("tried to simplify proposition which is not well-formed");
    }
    APPLY_SIMPLIFICATION(sentence_prop, simp_nothing_constant);
    APPLY_SIMPLIFICATION(sentence_prop, simp_use_listed_in);
    APPLY_SIMPLIFICATION(sentence_prop, simp_negated_determiners_nonex);
    APPLY_SIMPLIFICATION(sentence_prop, simp_negated_satisfiable);
    APPLY_SIMPLIFICATION(sentence_prop, simp_make_kinds_of_value_explicit);
    APPLY_SIMPLIFICATION(sentence_prop, simp_redundant_kinds);
    APPLY_SIMPLIFICATION(sentence_prop, simp_turn_right_way_round);
    APPLY_SIMPLIFICATION(sentence_prop, simp_region_containment);
    APPLY_SIMPLIFICATION(sentence_prop, simp_reduce_predicates);
    APPLY_SIMPLIFICATION(sentence_prop, simp_eliminate_redundant_variables);
    APPLY_SIMPLIFICATION(sentence_prop, simp_not_related_to_something);
    APPLY_SIMPLIFICATION(sentence_prop, simp_convert_gerunds);
    APPLY_SIMPLIFICATION(sentence_prop, simp_eliminate_to_have);
    APPLY_SIMPLIFICATION(sentence_prop, simp_is_all_rooms);
    APPLY_SIMPLIFICATION(sentence_prop, simp_redundant_kinds);
    vars_renumber(sentence_prop);                    just for the sake of tidiness
```

This code is used in §1.

§**17. The meaning of a noun phrase.** The First Rule tells us to translate a noun phrase (NP) into a pair of a term t and a proposition ϕ. We read this as "t such that ϕ is true".

For reasons explained below, a small amount of context is supplied: if the term will need to have a particular kind of value, then that KOV is given to us. (But if it only needs to be an object, or if we don't know anything about its kind, the `kov` argument will be `NULL`.)

As can be seen, an NP subtree consists *either* of an SN subtree representing two further NPs joined by a verb to make a relative clause, *or* one of three basic noun phrases: a value, a description, or a marker for an implied but missing noun.

```
pcalc_prop *NP_subtree_to_proposition(pcalc_term *subject_of_NP, meaning_list *ml,
    kind_of_value *kov) {
    pcalc_prop *NP_prop = NULL; int w1, w2;
    ⟨Tell the debugging log about the NP-subtree 18⟩;
    ⟨Clamber up to the interesting part of the NP subtree 20⟩;
    switch(ml_production(ml)) {
        case SN_PRODUCTION: NP_prop = S_subtree_to_proposition(ml, subject_of_NP); break;
        case VAL_PRODUCTION: ⟨This NP was parsed as a value 21⟩; break;
        case DC_PRODUCTION: ⟨This NP was parsed as a description 22⟩; break;
        case ABSENT_SUBJECT_PRODUCTION: ⟨This NP is only a ghostly presence 23⟩; break;
        default: LOG("Under NP subtree:\n$m", ml); internal_error("Malformed NP subtree");
    }
    ⟨If we have a single adjective which could also be a noun, and a value is required, convert it to a noun 24⟩;
    ⟨If we have a constant qualified by a substantive proposition, rewrite in terms of variable 25⟩;
    ⟨Close any open domain group 26⟩;
    ⟨Verify that the output satisfies the First Rule, throwing internal errors if not 19⟩;
    return NP_prop;
}
```

§**18.** Just as for SV-subtrees, we tell the debugging log at the start...

⟨Tell the debugging log about the NP-subtree 18⟩ ≡

```
    ml_get_text(ml, &w1, &w2);
    conv_log_depth++;
    LOGIF(CALCULUS, "[%d] Starting NP_subtree_to_proposition on: <$W>\n",
        conv_log_depth, w1, w2);
```

This code is used in §17.

§**19.** ...and also at the end.

⟨Verify that the output satisfies the First Rule, throwing internal errors if not 19⟩ ≡

```
if (prop_is_well_formed(NP_prop) == FALSE) internal_error("malformed NP proposition");
int NF = vars_number_free(NP_prop);
if (NF >= 2) internal_error("two or more free variables from NP");
if (subject_of_NP->constant) {
    if (NP_prop) internal_error("constant plus substantive prop from NP");
} else if (NF == 1) {
    int v = term_variable_underlying(subject_of_NP);
    if (status_of_var(NP_prop, v) != FREE_VST)
        internal_error("free variable from NP but not the preferred term");
}
LOGIF(CALCULUS, "[%d] NP_subtree_to_proposition: $W --> t = $0, phi = $D\n",
    conv_log_depth, w1, w2, subject_of_NP, NP_prop);
conv_log_depth--;
```

This code is used in §17.

§**20.** The recursive method of the S-parser means that it sometimes begins SV- and SN-subtrees with a tendril of not very meaningful nodes. They cause no trouble, though, and this is where we skip down them to where the action is.

⟨Clamber up to the interesting part of the NP subtree 20⟩ ≡

```
if (ml == NULL) internal_error("Null ML");
if (ml_production(ml) == NP_PRODUCTION) ml = ml_down(ml);
if ((ml_production(ml) == VAL_PRODUCTION) && (ml_down(ml)) &&
    (ml_production(ml_down(ml)) == DC_PRODUCTION)) ml = ml_down(ml);
if ((ml_production(ml) == DC_PRODUCTION) && (ml_down(ml)) &&
    (ml_production(ml_down(ml)) == SN_PRODUCTION)) ml = ml_down(ml);
```

This code is used in §17.

§**21.** Here we find a constant C and return $t = C$ with a null ϕ, except in one case: where C is the name of an either/or property, such as "closed". In the context of a value, this is a noun – it identifies which property we are talking about – and this is why `VAL_subtree_to_spec` returns it as a constant. But inside a sentence, it has to be considered an adjective, so rather than returning $t =$ `closed`, $\phi = T$, we return $t = x$ and $\phi = \mathit{closed}(x)$. If we didn't do this, text like "the trapdoor is closed" would translate to the proposition $\mathit{is}($`trapdoor`, `closed`$)$, which would then fail in type-checking.

(Note that this is a different sort of noun/adjective ambiguity than the one arising below, which is to do with enumerated value properties.)

⟨This NP was parsed as a value 21⟩ ≡

```
specification *spec = VAL_subtree_to_spec(ml);
*subject_of_NP = term_new_constant(spec);
if (spec_is_CONSTANT_of_kova(spec, PROPERTY_TY)) {
    property_name *prn = PROPERTY_spec_to_property_name(spec);
    if (prn_is_either_or(prn)) {
        *subject_of_NP = term_new_variable(0);
        NP_prop = atom_unary_PREDICATE_from_aph(prn_either_or_get_aph(prn), FALSE);
    } else if (prn_coincides_with_kind_of_value(prn)) {
        *subject_of_NP = term_new_variable(0);
        kind_of_value *kov = prn_get_kind_of_value(prn);
        NP_prop = atom_KIND_value_new(kov, term_new_variable(0));
    }
}
```

This code is used in §17.

§**22.** If `prop_from_spec` is given a constant value C then it returns the proposition $is(x, C)$: we look out for this and translate it to $t = C, \phi = T$. Otherwise, ϕ can be exactly the proposition returned, and the first term occurring in it will be chosen as the subject t. (In particular, if ϕ opens with a quantifier then t will be the variable it binds.)

⟨This NP was parsed as a description 22⟩ ≡

```
specification *spec = DC_subtree_to_spec(ml, FALSE);
NP_prop = prop_from_spec(spec, TRUE);
if (prop_match(NP_prop, 2, PREDICATE_ATOM, NULL, END_PROP_HERE, NULL)) {
    pcalc_term *pt = atom_is_x_equals(NP_prop);
    if (pt) { *subject_of_NP = *pt; NP_prop = NULL; }
}
if (NP_prop) *subject_of_NP = prop_get_first_cited_term(NP_prop);
```

This code is used in §17.

§**23.** When Inform reads a condition so abbreviated that both the subject and the verb have been left out, it assumes the verb is "to be" and that the subject will be whatever is being worked on. For instance,

if an unlocked container, ...

is read as the verb phrase "is" with `ABSENT_SUBJECT_PRODUCTION` as SP and "an unlocked container" as OP.

`ABSENT_SUBJECT_PRODUCTION` nodes are easy to deal with since they translate to the I6 variable `self` in the final compiled code; the `new_self_object_constant` routine returns a specification which refers to this. From a predicate calculus point of view, this is just another constant.

⟨This NP is only a ghostly presence 23⟩ ≡

```
*subject_of_NP = term_new_constant(new_self_object_constant());
```

This code is used in §17.

§**24.** Suppose we have a situation like this:

Texture is a kind of value. Rough, smooth and jagged are textures. A thing has a texture.

Feeling relates various rooms to one texture. The verb to feel (he feels) implies the feeling relation.

and consider the sentences:

[1] the broken bottle is jagged [2] the Spiky Cavern feels jagged

Now suppose we are working on the NP "jagged". In (1), it's an adjective: we are talking about a quality of the bottle. But in (2), it's a noun: we are establishing a relation between two values, the Cavern and the jagged texture.

Up to this point, "jagged" will have produced $t = x$, $\phi = jagged(x)$ in both cases – the adjectival reading of the word. The way we can tell if we are in case (2) is if a value of a specific kind (and not an object) is expected as the outcome. In the case of the equality relation used in (1), "is", the terms can be anything; but in the case of the feeling relation used in (2), the second term, corresponding to the noun phrase "jagged" in this sentence, has to have the kind of value "texture". So we convert it into noun form, and return $t =$ `texture`, $\phi = T$.

(Note that this is a different sort of noun/adjective ambiguity than the one arising above, which is to do with either/or properties.)

Another case which can occur is:

the bottle provides the property closed

where the presence of the words "the property" needs to alert us that "closed" is a noun referring to the property itself, not to a nameless object possessing that property.

⟨If we have a single adjective which could also be a noun, and a value is required, convert it to a noun 24⟩ ≡
```
if ((([[w1, w2 == property ...]] || [[w1, w2 == the property ...]]) || (kov)) {
    pcalc_term pct = convert_adj_to_noun(NP_prop);
    if (pct.constant) { *subject_of_NP = pct; NP_prop = NULL; }
}
```

This code is used in §17.

§25. If we have so far produced a constant term $t = C$ and a non-null proposition ϕ, then we convert t to a new free variable, say $t = y$, we then bind any free variable in the old ϕ and then change to $\exists y : is(y, C) \wedge \phi$. For instance, if we are working on the OP "the box in a room" from this:

> a thing in the box in a room

then the constant is $C =$ `box`, and Sstp returned $\phi = \exists x : room(x) \wedge is(x, \texttt{ContainerOf(box)})$.

⟨If we have a constant qualified by a substantive proposition, rewrite in terms of variable 25⟩ ≡
```
if ((subject_of_NP->constant) && (NP_prop)) {
    int y = vars_find_unused(NP_prop);
    NP_prop = prop_concatenate(
        atom_binary_PREDICATE_new(a_is_b_predicate, *subject_of_NP, term_new_variable(y)),
        NP_prop);
    NP_prop = vars_bind_existential(NP_prop);
    *subject_of_NP = term_new_variable(y);
}
```

This code is used in §17.

§26. If the NP was something like "at least four open doors", we will so far have built `QUANTIFIER --> DOMAIN_OPEN --> KIND --> PREDICATE`, and now that we have reached the end of the noun phrase we need to add a `DOMAIN_CLOSE` atom. The following is written in a way that guarantees all such open groups are closed, but in fact there should only ever be one open, so `nq` should always evaluate to 0 or 1.

⟨Close any open domain group 26⟩ ≡
```
int i, nq = 0;
TRAVERSE_VARIABLE(p);
TRAVERSE_PROPOSITION(p, NP_prop)
    switch (p->element) {
        case DOMAIN_OPEN_ATOM: nq++; break;
        case DOMAIN_CLOSE_ATOM: nq--; break;
    }
if (nq < 0) internal_error("malformed proposition with too many domain ends");
for (i=1; i<=nq; i++)
    NP_prop = prop_concatenate(NP_prop, atom_new(DOMAIN_CLOSE_ATOM));
```

This code is used in §17.

Simplifications 6/simp

Purpose

A set of operations each of which takes a proposition and either leaves it unchanged or replaces it with a simpler one logically equivalent to the original.

6/simp.§1-3 Simplify the nothing constant (fudge); §4 Use listed-in predicates (deduction); §5-6 Simplify negated determiners (deduction); §7 Simplify negated satisfiability (deduction); §8 Make kind requirements explicit (deduction); §9-16 Remove redundant kind predicates (deduction); §17 Turn binary predicates the right way round (deduction); §18 Simplify region containment (fudge); §19 Reduce binary predicates (deduction); §20-22 Eliminating determined variables (deduction); §23 Simplify non-relation (deduction); §24 Convert gerunds to nouns (deduction); §25-27 Eliminate to have meaning property ownership (fudge); §28 Turn all rooms to everywhere (fudge)

Definitions

¶1. Recall the three rules for simplification routines: they take a proposition Σ (which they are allowed to destroy or modify) and return Σ', but such that:

(i) Σ' remains a syntactically correct proposition with well-formed quantifiers,
(ii) Σ' has the same number of free variables as Σ, and
(iii) in all situations and for all possible values of any free variables, Σ' is true if and only if Σ is.

Rules (i) and (ii) are always strictly obeyed. A simplification which obeys (iii) in its purely logical sense is called a "deduction"; one which bends (iii) to change the proposition from what the user wrote, to what he meant to write, is called a "fudge". Fudges are needed because English is quirky, and does not correspond perfectly to predicate logic.

What we mean by simpler is not shorter, or with less exotic contents: often the simplified form of a proposition is longer and uses relations or determiners less obviously derived from the original text. Instead, simpler means simpler to test and assert. Our ideal is a conjunction of a sequence of predicates, using the smallest possible number of variables (and hence quantifiers).

---

§**1. Simplify the nothing constant (fudge).** The word "nothing" is sometimes a noun ("the holder of the oak tree is nothing"), sometimes a determiner ("nothing is in the box"). This doesn't arise with the other no- words, nowhere and nobody, since those are not allowed as nouns in Inform.

Here we look for the noun form as one term in a binary predicate, and convert it to the determiner form unless the predicate is equality. Thus "X is nothing" is allowed to use the noun form, "X contains nothing" has to use the determiner form. (In particular "nothing is nothing" compares two identical nouns and is always true. I thought this was a sentence nobody would write, but Google finds 223,000 hits for it.)

```
pcalc_prop *simp_nothing_constant(pcalc_prop *prop, int *changed) {
    TRAVERSE_VARIABLE(pl);
    *changed = FALSE;
    TRAVERSE_PROPOSITION(pl, prop)
        if ((atom_is_binary_predicate(pl)) && (atom_is_equality_predicate(pl) == FALSE)) {
            binary_predicate *bp = RETRIEVE_POINTER_binary_predicate(pl->predicate);
            int i;
            for (i=0; i<2; i++) {
                if (spec_is_nothing_object_constant(term_constant_underlying(&(pl->terms[i])))) {
                    ⟨Substitute for the term and quantify with does-not-exist 3⟩;
                    PROPOSITION_EDITED(pl, prop);
                    break;
                }
            }
        }
    return prop;
}
```

The function simp_nothing_constant is called from 6/sconv.

§**2.** Formally, if B is a predicate other than equality,

$$\Sigma = \cdots B(t, f_X(f_Y(\cdots(\texttt{nothing})))) \cdots \quad \longrightarrow \quad \Sigma' = \cdots \nexists v \in \{v \mid K_1(v)\} : B(t, f_X(f_Y(\cdots(v)))) \cdots$$

where v is unused in Σ, and – note the difference in placing –

$$\Sigma = \cdots B(f_X(f_Y(\cdots(\texttt{nothing}))), t) \cdots \quad \longrightarrow \quad \Sigma' = \nexists v \in \{v \mid K_2(v)\} : \cdots B(f_X(f_Y(\cdots(v))), t) \cdots$$

where K_1 and K_2 are the kinds of terms 1 and 2 in the predicate B.

The difference in where we place the quantifier is important in double-negative sentences. Consider these:

[1] the box does not contain nothing

which produces $\Sigma = \neg(\mathit{contains}(\texttt{box}, \texttt{nothing}))$. Here, `nothing` is part of the object phrase, not the subject phrase, and we need to quantify it within the OP – which means, within the negation, because our recipe for negated sentences was (roughly) SP $\wedge\neg($ OP $\wedge$ VP). We thus make $\Sigma' = \neg(\nexists x : \mathit{contains}(\texttt{box}, x))$, although later simplification converts that to $\exists x : \mathit{contains}(\texttt{box}, x)$, just as if the original sentence had been "the box contains something". On the other hand,

[2] nothing does not annoy Peter

produces $\Sigma = \neg(\mathit{annoys}(\texttt{nothing}, \texttt{Peter}))$, and now we have to quantify as $\Sigma' = \nexists x : \neg(\mathit{annoys}(x, \texttt{Peter}))$.

Double-negatives are a little odd. If natural language were really the same as predicate logic with some grunting sounds for decoration, then a double negative would always be a positive. But in 18th-century English, that wasn't true: it was a way of emphasising the negation, just as characters in Aaron Sorkin's *The West Wing* scripts are always saying "not for nothing, but..." when their meaning is equivalent to "this is nothing, but...". Still, Inform takes the view that a double negative is a positive.

§**3.** The code is simpler than the explanation:

⟨Substitute for the term and quantify with does-not-exist 3⟩ ≡

```
int nv = vars_find_unused(prop);
pcalc_term new_var = term_new_variable(nv);
substitute_nothing_in_term(&(pl->terms[i]), &new_var);
pcalc_prop *position = NULL;                                   at the front if nothing is the SP...
if (i == 1) position = pl_prev;                                ...but up close to the predicate if it's the OP
insert four atoms, in reverse order, at this position:
prop = prop_insert_atom(prop, position, atom_new(DOMAIN_CLOSE_ATOM));
if (bp_term_kind(bp, i))
    prop = prop_insert_atom(prop, position,
        atom_KIND_object_new(bp_term_kind(bp, i), new_var));
else if (bp_term_kind_of_value(bp, i))
    prop = prop_insert_atom(prop, position,
        atom_KIND_value_new(bp_term_kind_of_value(bp, i), new_var));
prop = prop_insert_atom(prop, position, atom_new(DOMAIN_OPEN_ATOM));
prop = prop_insert_atom(prop, position, atom_QUANTIFIER_new(not_exists_quantifier, nv, 0));
```

This code is used in §1.

§4. Use listed-in predicates (deduction). Specifications for entries in tables have a variety of forms (well, four of them), and one is by implication a condition: this is the 2-reference form of `TABLE_ENTRY_SPC` specification. Suppose the source text reads

> if X is a density listed in the Table of Solid Stuff, ...

Inform parses "a density listed in the Table of Solid Stuff" as a single specification, with references to the density column and the Solid Stuff table. In logical terms, this is just another constant, which we'll call L. At this point the sentence looks like:

$$is(X, L)$$

The trouble is that L is really a predicate, acting on some free variable v, because it stands for any value v found in the density column. We could try to interpret *is* so as to accommodate this, but it is much better to use Inform's regular predicate apparatus.

So we must make its nature explicit. Every table column has an associated binary predicate, and we rewrite:

$$\exists v : number(v) \wedge listed\text{-}in\text{-}density\text{-}column(v, T) \wedge is(v, X)$$

This looks extravagant, but later simplification will reduce it to

$$listed\text{-}in\text{-}density\text{-}column(X, T).$$

More formally suppose we write $L(C, T)$ for the constant representing "a C listed in T", and suppose we use the notation $P(\dots t \dots)$ for any predicate containing a term underlying which is t. Let v be a variable unused in Σ, and let K be the kind of value of entries in the C column. Then:

$$\Sigma = \cdots P(\dots L(C, T) \dots) \cdots \quad \longrightarrow \quad \Sigma' = \cdots \exists v : K(v) \wedge listed\text{-}in\text{-}C\text{-}column(v, T) \wedge P(\dots v \dots) \cdots$$

where the variable v has replaced the constant $L(C, T)$ underlying one of the terms of P.

Note that this can act twice on the same predicate, if such terms occur twice. For example,

> if a crispiness listed in the Table of Salad Stuff is a density listed in the Table of Solid Stuff, ...

generates $is(L(C_1, T_1), L(C_2, T_2))$. As it happens, Inform can't compile this efficiently and will produce a problem message to say so, but it's important that our code here should generate the correct proposition,

$$\exists x : number(x) \wedge listed\text{-}in\text{-}C1\text{-}column(x, T_1) \wedge listed\text{-}in\text{-}C2\text{-}column(x, T_2).$$

```
pcalc_prop *simp_use_listed_in(pcalc_prop *prop, int *changed) {
    TRAVERSE_VARIABLE(pl);
    *changed = FALSE;
    TRAVERSE_PROPOSITION(pl, prop) {
        int j;
        for (j=0; j<pl->arity; j++) {
            specification *spec = term_constant_underlying(&(pl->terms[j]));
            if ((spec) && (spec_get_storage_form(spec) == TABLE_ENTRY_SPC) &&
                (spec_get_argc(spec) == 2)) {
                specification *col = spec_get_argument(spec, 0);
                specification *tab = spec_get_argument(spec, 1);
                table_column *tc = TABLE_COLUMN_spec_to_table_column(col);
                kind_of_value *kov = get_data_type_stored(tc);
                if (is_kova(kov, UNDERSTANDING_TY)) kov = kova(SNIPPET_TY);
                int nv = vars_find_unused(prop);
                pcalc_term nv_term = term_new_variable(nv);
                prop = prop_insert_atom(prop, pl_prev,
```

```
                    atom_binary_PREDICATE_new(tc_get_listed_in_predicate(tc),
                        nv_term, term_new_constant(tab)));
                pcalc_prop *new_KIND = atom_KIND_value_new(kov, nv_term);
                new_KIND->predicate =
                    STORE_POINTER_binary_predicate(tc_get_listed_in_predicate(tc));
                prop = prop_insert_atom(prop, pl_prev, new_KIND);
                prop = prop_insert_atom(prop, pl_prev,
                    atom_QUANTIFIER_new(exists_quantifier, nv, 0));
                substitute_term_in_term(&(pl->terms[j]), &nv_term);
                we_need_ct_locals();
                PROPOSITION_EDITED(pl, prop);
            }
        }
    }
    return prop;
}
```

The function simp_use_listed_in is called from 6/sconv.

§**5. Simplify negated determiners (deduction).** The negation atom is worth removing wherever possible, since we want to keep propositions in a flat conjunction form if we can, and the negation of a string of atoms is therefore a bad thing. We therefore change thus:

$$\Sigma = \cdots \neg(Qv \in \{v \mid \phi(v)\} : \psi) \cdots \quad \longrightarrow \quad \Sigma' = \cdots \overline{Q}v \in \{v \mid \phi(v)\} : \psi \cdots$$

where $\overline{Q}$ is the negation of the generalised quantifier Q: for instance, $V_{<5}y$ becomes $V_{\geq 5}y$.

A curiosity here is that when simplifying during sentence conversion, we choose not to apply this deduction in the case of $Q = \exists$. This saves us having to make difficult decisions about the domain set of Q (see below), and also preserves $\exists v$ atoms in Σ, which are useful since they make some of our later simplifications more applicable.

```
pcalc_prop *simp_negated_determiners_nonex(pcalc_prop *prop, int *changed) {
    return simp_negated_determiners(prop, changed, FALSE);
}
pcalc_prop *simp_negated_determiners(pcalc_prop *prop, int *changed,
    int negate_existence_too) {
    TRAVERSE_VARIABLE(pl);
    *changed = FALSE;
    TRAVERSE_PROPOSITION(pl, prop) {
        pcalc_prop *quant_atom;
        if (prop_match(pl, 2,
            NEGATION_OPEN_ATOM, NULL, QUANTIFIER_ATOM, &quant_atom)) {
            if ((negate_existence_too) ||
                ((atom_is_existence_quantifier(quant_atom) == FALSE))) {
                prop = prop_ungroup_and_negate_determiner(prop, pl_prev, TRUE);
                PROPOSITION_EDITED(pl, prop);
            }
        }
    }
    return prop;
}
```

The function simp_negated_determiners_nonex is called from 6/sconv.
The function simp_negated_determiners is called from 6/cdefp.

§**6.** And the following useful routine actually performs the change. The only tricky point here is that we store $\exists v$ without domain brackets; so if Q happens to be $\exists v$ then we have to turn $\neg(\exists v : \cdots)$ into $\nexists v \in \{v \mid \cdots\} \cdots$, and it's not obvious where to place the }. While there's no logical difference – the proposition means the same wherever we put it – the assert-propositions code is better at handling $\exists v \in X : \phi(v)$ than $\exists v \in Y$. So we want the braces to enclose fixed, unassertable matter – v being a container, say – and the ϕ outside the braces should then contain predicates which can be asserted.

In practice that's way too hard for this routine to handle. If `add_domain_brackets` is set, then it converts

$$\neg(\exists v : \cdots) \quad \longrightarrow \quad \nexists v \in \{v \mid \cdots\}$$

– that is, it will make the entire negated subproposition the domain of the quantifier. If `add_domain_brackets` is clear, the routine will return a syntactically incorrect proposition lacking the domain brackets, and it's the caller's responsibility to put that right.

```
pcalc_prop *prop_ungroup_and_negate_determiner(pcalc_prop *prop, pcalc_prop *after,
    int add_domain_brackets) {
    pcalc_prop *quant_atom, *last;
    int fnd;
    if (after == NULL)
        fnd = prop_match(prop, 2,
            NEGATION_OPEN_ATOM, NULL, QUANTIFIER_ATOM, &quant_atom);
    else
        fnd = prop_match(after, 3, ANY_ATOM_HERE, NULL,
            NEGATION_OPEN_ATOM, NULL, QUANTIFIER_ATOM, &quant_atom);
    if (fnd) {
        quantifier *quant = RETRIEVE_POINTER_quantifier(quant_atom->predicate);
        quantifier *antiquant = quant_get_negation(quant);
        quant_atom->predicate = STORE_POINTER_quantifier(antiquant);
        prop = prop_ungroup_after(prop, after, &last);                    remove negation group brackets
        if ((quant == exists_quantifier) && (add_domain_brackets)) {
            prop = prop_insert_atom(prop, quant_atom, atom_new(DOMAIN_OPEN_ATOM));
            prop = prop_insert_atom(prop, last, atom_new(DOMAIN_CLOSE_ATOM));
        }
        if (antiquant == exists_quantifier) {
            prop = prop_ungroup_after(prop, quant_atom, NULL);            remove domain group brackets
        }
        LOGIF(WORKINGS, "prop_ungroup_and_negate_determiner: $D\n", prop);
    } else internal_error("not a negate-group-determiner");
    return prop;
}
```

§**7. Simplify negated satisfiability (deduction).** When simplifying converted sentences, we chose not to use the `simp_negated_determiners` tactic on existence quantifiers $\exists v$, partly because it's tricky to establish their domain in a way helpful to the rest of Inform.

Here we handle a simple case which occurs frequently and where we can indeed identify the domain well:

$$\Sigma = \neg(\exists v : K(v) \wedge P) \quad \longrightarrow \quad \Sigma' = \nexists v \in \{v \mid K(v)\} : P$$

where K is a kind, and P is any single predicate other than equality. (In the case of equality, we'd rather leave matters as they stand, because substitution will later eliminate all of this anyway.)

```
pcalc_prop *simp_negated_satisfiable(pcalc_prop *prop, int *changed) {
    pcalc_prop *quant_atom, *predicate_atom, *kind_atom;
    *changed = FALSE;
    if ((prop_match(prop, 6,
        NEGATION_OPEN_ATOM, NULL,
        QUANTIFIER_ATOM, &quant_atom,
        KIND_ATOM, &kind_atom,
        PREDICATE_ATOM, &predicate_atom,
        NEGATION_CLOSE_ATOM, NULL,
        END_PROP_HERE, NULL)) &&
        (atom_is_existence_quantifier(quant_atom)) &&
        (atom_is_equality_predicate(predicate_atom) == FALSE) &&
        (kind_atom->terms[0].variable == quant_atom->terms[0].variable)) {
        prop = prop_ungroup_and_negate_determiner(prop, NULL, FALSE);
        prop = prop_insert_atom(prop, quant_atom, atom_new(DOMAIN_OPEN_ATOM));
        prop = prop_insert_atom(prop, kind_atom, atom_new(DOMAIN_CLOSE_ATOM));
        *changed = TRUE;
    }
    return prop;
}
```

The function simp_negated_satisfiable is called from 6/sconv.

§8. Make kind requirements explicit (deduction). Many predicates contain implicit requirements about the kinds of their terms. For instance, if R relates a door to a number, then $R(x, y)$ can only be true if x is a door and y is a number. We insert these requirements explicitly in order to defend the code testing R; it ensures we never have to test bogus values. We need do this only for variables, as with more constants and functions type-checking will certainly be able to test their kind in any event (whereas a free variable is anonymous enough that we can't necessarily know by other means).

Formally, let K_1 and K_2 be the kinds of value of terms 1 and 2 of the binary predicate R. Let v be a variable. Then:

$$\Sigma = \cdots R(v, t) \cdots \quad \longrightarrow \quad \Sigma' = \cdots K_1(v) \wedge R(v, t)$$
$$\Sigma = \cdots R(t, v) \cdots \quad \longrightarrow \quad \Sigma' = \cdots K_2(v) \wedge R(t, v)$$

and therefore, if both cases occur,

$$\Sigma = \cdots R(v, w) \cdots \quad \longrightarrow \quad \Sigma' = \cdots K_1(v) \wedge K_2(w) \wedge R(v, w)$$

Some of these new kind atoms are unnecessary, but `simp_redundant_kinds` will detect and remove those.

Why do we do this for binary predicates, but not unary predicates? The answer is that there's no need, and it's impracticable anyway, because adjectives are allowed to have multiple definitions for different kinds of value, and because the code testing them is written to cope properly with bogus values.

```
pcalc_prop *simp_make_kinds_of_value_explicit(pcalc_prop *prop, int *changed) {
    TRAVERSE_VARIABLE(pl);
    TRAVERSE_PROPOSITION(pl, prop)
        if (atom_is_binary_predicate(pl)) {
            int i;
            binary_predicate *bp = RETRIEVE_POINTER_binary_predicate(pl->predicate);
            for (i=1; i>=0; i--) {
                int v = pl->terms[i].variable;
                if (v >= 0) {
                    kind_of_value *kov = bp_term_kind_of_value(bp, i);
                    if ((kov == NULL) && (bp_term_kind(bp, i)))
                        kov = kovko(bp_term_kind(bp, i));
                    if (kov) {
                        pcalc_prop *new_KIND = atom_KIND_value_new(kov, term_new_variable(v));
                        new_KIND->predicate = STORE_POINTER_binary_predicate(bp);
                        prop = prop_insert_atom(prop, pl_prev, new_KIND);
                    }
                    *changed = TRUE;
                }
            }
        }
    return prop;
}
```

The function simp_make_kinds_of_value_explicit is called from 6/treec and 6/sconv.

§**9. Remove redundant kind predicates (deduction).** Propositions often contain more `KIND` atoms than they need, not least as a result of `simp_make_kinds_of_value_explicit`. Here we remove (some of) those, and move the survivors to what we consider the best positions within the line. For reasons to be revealed below, we run this process twice over:

```
pcalc_prop *simp_redundant_kinds(pcalc_prop *prop, int *changed) {
    int c1 = FALSE, c2 = FALSE;
    prop = simp_redundant_kinds_dash(prop, prop, 1, &c1);
    prop = simp_redundant_kinds_dash(prop, prop, 1, &c2);
    if ((c1) || (c2)) *changed = TRUE; else *changed = FALSE;
    return prop;
}
```

The function simp_redundant_kinds is called from 6/sconv.

§**10.** This routine works recursively on subexpressions within the main proposition. These all begin and end with matched `*_OPEN_ATOM` and `*_CLOSED_ATOM` brackets, with one exception: the main proposition itself. `start_group` represents the first atom of the expression (the open bracket, in most cases) and `start_level` the level of bracket nesting at that point. This is 1 for the main proposition, but 0 for subexpressions, so that inside the brackets the main content will be at level 1.

This would all go wrong if the proposition were not well-formed, but we know that it is – an internal error would have been thrown if not.

```
pcalc_prop *simp_redundant_kinds_dash(pcalc_prop *prop, pcalc_prop *start_group,
    int start_level, int *changed) {
    pcalc_prop *optimal_kind_placings_domain[26], *optimal_kind_placings_statement[26];
    ⟨Recursively simplify all subexpressions first 11⟩;
    ⟨Find optimal positions for kind predicates 12⟩;
    ⟨Strike out redundant kind predicates applied to variables 13⟩;
    ⟨Strike out tautological kind predicates applied to constants 16⟩;
    return prop;
}
```

§**11.** For all of the atoms in the body of the group we're working on, the bracket level will be 1. When it raises to 2, then, we begin a subexpression, and we recurse into it. (We don't recurse at levels 3 and up because the level 2 call will already have taken care of those sub-sub-expressions.)

⟨Recursively simplify all subexpressions first 11⟩ ≡

```
    TRAVERSE_VARIABLE(pl);
    int blevel = start_level;
    TRAVERSE_PROPOSITION(pl, start_group) {
        if (element_get_group(pl->element) == OPEN_OPERATORS_GROUP) {
            blevel++;
            if (blevel == 2) prop = simp_redundant_kinds_dash(prop, pl, 0, changed);
        }
        if (element_get_group(pl->element) == CLOSE_OPERATORS_GROUP) blevel--;
        if (blevel == 0) break;
    }
```

This code is used in §10.

§**12.** Suppose we have a kind predicate $K(v)$ applied to a variable. What would be the best place to put this? Generally speaking, we want it as early as possible, because tests of K are cheap and this will keep running time low in compiled code. On the other hand we must not move it outside the current subexpression. The doctrine is:

(i) if v is unbound *within the subexpression* then $K(v)$ should move right to the front;
(ii) if v is bound by $\exists$, then $K(v)$ should move immediately after $\exists v$;
(iii) if v is bound by $Qv \in \{v \mid \phi(v)\}$ then any $K(v)$ occurring in the domain $\phi(v)$ should move to the front of v, whereas any later $K(v)$ should move after the domain closing, except
(iv) where v is bound by $\nexists v \in \{v \mid \phi(v)\}$, when $K(v)$ should move into the domain set, even if it occurs in the statement.

Rule (iv) there looks a little surprising. For instance, it causes

$$\Sigma = \nexists x \in \{x \mid \mathit{thing}(x) \wedge \mathit{contains}(\texttt{Ballroom}, x)\} : \mathit{container}(x) \wedge \mathit{open}(x) \quad \longrightarrow$$
$$\Sigma' = \nexists x \in \{x \mid \mathit{container}(x) \wedge \mathit{thing}(x) \wedge \mathit{contains}(\texttt{Ballroom}, x)\} : \mathit{open}(x).$$

These are logically equivalent because $\nexists$ behaves that way – they wouldn't be equivalent for other quantifiers. Rule (iii) would have said no movement was necessary; the reason we made the move is that it makes Σ' possible to assert with "now", as in the phrase "now nothing in the Ballroom is an open container".

The following calculates two arrays: `optimal_kind_placings_domain` marks the start of ϕ for each variable v, while `optimal_kind_placings_statement` marks the start of the statement following the quantifier.

⟨Find optimal positions for kind predicates 12⟩ ≡

```
TRAVERSE_VARIABLE(pl);
int bvsp = 0, bound_vars_stack[26];
int blevel = start_level, j;
for (j=0; j<26; j++) {
    optimal_kind_placings_domain[j] = (start_level == 1)?NULL:start_group;
    optimal_kind_placings_statement[j] = optimal_kind_placings_domain[j];
}
TRAVERSE_PROPOSITION(pl, start_group) {
    if (element_get_group(pl->element) == OPEN_OPERATORS_GROUP) blevel++;
    if (element_get_group(pl->element) == CLOSE_OPERATORS_GROUP) blevel--;
    if (blevel == 0) break;
    pcalc_prop *dom;
    if (prop_match(pl, 2,
            QUANTIFIER_ATOM, NULL, DOMAIN_OPEN_ATOM, &dom)) {
        if ((atom_is_existence_quantifier(pl)) ||
            (atom_is_nonexistence_quantifier(pl)))
            bound_vars_stack[bvsp++] = -1;
        else
            bound_vars_stack[bvsp++] = pl->terms[0].variable;
        optimal_kind_placings_domain[pl->terms[0].variable] = dom;
        optimal_kind_placings_statement[pl->terms[0].variable] = dom;
    } else if (atom_is_existence_quantifier(pl)) {
        optimal_kind_placings_domain[pl->terms[0].variable] = pl;
        optimal_kind_placings_statement[pl->terms[0].variable] = pl;
    }
    if (pl->element == DOMAIN_CLOSE_ATOM) {
        int v = bound_vars_stack[--bvsp];
        if (v >= 0) optimal_kind_placings_statement[v] = pl;
    }
}
```

This code is used in §10.

§**13.** The following looks at the predicates $K(v)$ applied to variables which are in the subexpression at the top level. It then does two things:

Suppose K and L are kinds of value such that $L \subseteq K$, and let ψ be a well-formed proposition. Then

$$\Sigma = \psi \wedge L(v) \cdots K(v) \cdots \quad \longrightarrow \quad \Sigma' = \psi \wedge L(v) \cdots$$

(that is, $K(v)$ is eliminated). This is clearly valid since $L(v) \Rightarrow K(v)$ and $L(v)$ is valid throughout the subexpression after its appearance.

Secondly, and it's not worth finding a logical notation for this, the kind is moved back to its optimal position, as calculated above.

At first sight, this process only removes redundancies when the stronger kind appears before the weaker one. What if they occur the other way around? This is why the simplification is run twice, and why it's important that the process of moving predicates back to their optimal position reverses their order. Suppose we start with $person(x) \wedge vehicle(y) \wedge woman(x)$.

(1a) On pass 1, *person* occurs before *woman*, but it is weaker – every woman is a person, but not necessarily vice versa – so neither is deleted.
(1b) But pass 1 also moves the kinds back, and this produces $woman(x) \wedge vehicle(y) \wedge person(x)$.
(2a) On pass 2, the stronger *woman* now occurs before *person*, so we eliminate to get $woman(x) \wedge vehicle(y)$.
(2b) And pass 2 again moves kinds back, producing $vehicle(y) \wedge woman(x)$.

(Because the order is reversed twice, any surviving kind predicates continue to appear in the same order as they did in the original proposition. This doesn't matter, but it's tidy.)

⟨Strike out redundant kind predicates applied to variables 13⟩ ≡

```
TRAVERSE_VARIABLE(pl);
int domain_passed[26], j;
int blevel = start_level;
for (j=0; j<26; j++) domain_passed[j] = FALSE;
TRAVERSE_PROPOSITION(pl, start_group) {
    for (j=0; j<26; j++)
        if (pl == optimal_kind_placings_statement[j])
            domain_passed[j] = TRUE;
    if (element_get_group(pl->element) == OPEN_OPERATORS_GROUP) blevel++;
    if (element_get_group(pl->element) == CLOSE_OPERATORS_GROUP) blevel--;
    if (blevel == 1) {
        if (pl->element == KIND_ATOM) {
            kind_of_value *early_kov = pl->assert_kind_of_value;
            int v = pl->terms[0].variable;
            if ((v >= 0) && (early_kov)) {
                ⟨Strike out any subsequent but weaker kind predicate on the same variable 14⟩;
                ⟨Move this predicate backwards to its optimal position 15⟩;
            }
        }
    }
    if (blevel == 0) break;
}
```

This code is used in §10.

§**14.** The noteworthy thing here is that we continue through the subexpression, deleting any weaker form of $K(v)$ that we find, but also allow ourselves to continue *beyond* the subexpression in one case. Suppose we have

$$Qv \in \{v \mid K(v) \wedge ...\} : L(v)$$

and we are working on the $K(v)$ term. If we continue only to the end of the current subexpression, that runs out at the }, the end of the domain specification. So in that one case alone we allow ourselves to sidestep the `DOMAIN_CLOSE_ATOM` and continue looking for $L(v)$ in the outer subexpression – the one which is governed by the quantifier.

⟨Strike out any subsequent but weaker kind predicate on the same variable 14⟩ ≡

```
TRAVERSE_VARIABLE(gpl);
int glevel = 1;
TRAVERSE_PROPOSITION(gpl, pl) {
    if (element_get_group(gpl->element) == OPEN_OPERATORS_GROUP) glevel++;
    if (gpl->element == DOMAIN_CLOSE_ATOM) {
        if (glevel > 1) glevel--;
    } else if (element_get_group(gpl->element) == CLOSE_OPERATORS_GROUP) glevel--;
    if (glevel == 0) break;
    if ((gpl != pl) && (gpl->element == KIND_ATOM) && (v == gpl->terms[0].variable)) {
        i.e., gpl now points to a different kind atom on the same variable
        kind_of_value *later_kov = gpl->assert_kind_of_value;
        if ((later_kov) && (can_we_cast_kovs(early_kov, later_kov) == ALWAYS_MATCH)) {
            prop = prop_delete_atom(prop, gpl_prev);
            PROPOSITION_EDITED_REPEATING_CURRENT(gpl, prop);
        }
    }
}
```

This code is used in §13.

§**15.**

⟨Move this predicate backwards to its optimal position 15⟩ ≡

```
pcalc_prop *best_place = optimal_kind_placings_domain[v];
if (domain_passed[v]) best_place = optimal_kind_placings_statement[v];
if (pl_prev != best_place) {
    prop = prop_delete_atom(prop, pl_prev);                    that is, delete the current K(v)
    prop = prop_insert_atom(prop, best_place,
        atom_KIND_value_new(early_kov, term_new_variable(v)));          insert a new one
    PROPOSITION_EDITED_REPEATING_CURRENT(pl, prop);
}
```

This code is used in §13.

§**16.** Suppose we find a term $K(C)$, where C is a constant in the sense of predicate calculus – that is, a `specification`. There is no need to perform such a test at run-time because we can determine the kind of C and compare it against K right now. For instance, *number*(`score`) is necessarily true at all times.

Formally, suppose C is a constant which, when evaluated, has kind of value L. Suppose that $L \subseteq K$ and that K is not a kind of object. Then

$$\Sigma = \cdots K(C) \cdots \quad \longrightarrow \quad \Sigma' = \cdots\cdots$$

(That is, we eliminate the $K(C)$ term.)

We could clearly go further than this:

(a) Why don't we eliminate $K(C)$ when K is an object, too? Logically this would be fine, but we choose not to, for two reasons: people sometimes write phrases in I6 which claim to return a room, say, but sometimes return `nothing`. Technically this is a violation of type safety. If t is a term representing a call to this function, then *room*(t) ought to be redundant. But in practice it will protect against the `nothing` value. The other reason is to ensure that text like "Peter is a man" is not simplified all the way down to the null proposition (as it clearly can be, if Peter is indeed a man). That might seem harmless, but means that "now Peter is a man" doesn't produce the problem message saying that kinds can't be asserted – a common mistake made by beginners. It's better consistently to reject all such attempts than to be clever and allow the ones which are logically redundant.

(b) Why don't we reduce $K(C)$ to falsity when C is a constant clearly not of the kind K, such as *text*(4)? Again, it would make it harder to issue a good problem message later, in type-checking; and besides our calculus lacks a "falsity" atom, so there's no way to store the universally false proposition which would result if we eliminated every atom this way. (It also doesn't matter what the running time of compiled code will be if the proposition is going to fail type-checking anyway.)

⟨Strike out tautological kind predicates applied to constants 16⟩ ≡

```
TRAVERSE_VARIABLE(pl);
int blevel = start_level;
TRAVERSE_PROPOSITION(pl, start_group) {
    if (element_get_group(pl->element) == OPEN_OPERATORS_GROUP) blevel++;
    if (element_get_group(pl->element) == CLOSE_OPERATORS_GROUP) blevel--;
    if (blevel == 1) {
        if (pl->element == KIND_ATOM) {
            kind_of_value *early_kov = pl->assert_kind_of_value;
            specification *spec = pl->terms[0].constant;
            if (family_is(spec, VALUE_FMY)) {
                kind_of_value *kov = kov_when_VALUE_is_evaluated(spec);
                if ((kov) && (kovko_get_kind(early_kov) == NULL) &&
                    (can_we_cast_kovs(early_kov, kov) == ALWAYS_MATCH)) {
                    prop = prop_delete_atom(prop, pl_prev);
                    PROPOSITION_EDITED_REPEATING_CURRENT(pl, prop);
                }
            }
        }
    }
}
```

This code is used in §10.

§**17. Turn binary predicates the right way round (deduction).** Recall that BPs are manufactured in pairs, each being the reversal of the other, in the sense of transposing their terms. Of each pair, one is considered the canonical way to represent the relation, and is "the right way round". This routine turns all BPs in the proposition the right way round, if they aren't already.

Suppose B is a binary predicate which is marked as the wrong way round, and R is its reversal. Then we change:

$$\Sigma = \cdots B(t_1, t_2) \cdots \quad \longrightarrow \quad \Sigma' = \cdots R(t_2, t_1) \cdots$$

(Note that the equality predicate *is* only has one way round, and it's the right one – this is the only exception to the rule that BPs come in pairs – so equality predicates won't be turned around here, not that it would matter if they were.)

```
pcalc_prop *simp_turn_right_way_round(pcalc_prop *prop, int *changed) {
    TRAVERSE_VARIABLE(pl);
    *changed = FALSE;
    TRAVERSE_PROPOSITION(pl, prop) {
        binary_predicate *bp = atom_is_binary_predicate(pl);
        if ((bp) && (bp_is_the_wrong_way_round(bp))) {
            pcalc_term pt = pl->terms[0];
            pl->terms[0] = pl->terms[1];
            pl->terms[1] = pt;
            pl->predicate = STORE_POINTER_binary_predicate(bp_get_reversal(bp));
            PROPOSITION_EDITED(pl, prop);
        }
    }
    return prop;
}
```

The function simp_turn_right_way_round is called from 6/sconv.

§18. Simplify region containment (fudge). Most of Inform's prepositions are unambiguous, but "in" can mean two quite different relations. Usually it means (direct) containment, but there is an alternative interpretation as regional containment. "The diamond is in the teddy bear" is direct containment, but "The diamond is in Northumberland" is regional containment. We need to separate out these ideas into two different binary predicates because direct containment has a function f_D allowing simplification of many common sentences, but regional containment allows no such simplification. Basically: you can be directly contained by only one thing at a time, but might be in many regions at once.

So far we assume every "in" means the `a_contains_b_predicate`. This is the point where we choose to divert some uses to `a_region_contains_b_predicate`. If R is a constant region name, and C_D, C_R are the predicates for direct and region containment, then

$$\Sigma = \cdots C_D(t, R) \cdots \quad \longrightarrow \quad \Sigma' = \cdots C_R(t, R) \cdots$$

$$\Sigma = \cdots C_D(R, t) \cdots \quad \longrightarrow \quad \Sigma' = \cdots C_R(R, t) \cdots$$

(Note that a region cannot directly contain any object.)

```
pcalc_prop *simp_region_containment(pcalc_prop *prop, int *changed) {
    TRAVERSE_VARIABLE(pl);
    *changed = FALSE;
    TRAVERSE_PROPOSITION(pl, prop) {
        binary_predicate *bp = atom_is_binary_predicate(pl);
        if (bp == a_contains_b_predicate) {
            int j;
            for (j=0; j<2; j++) {
                world_object *the_region =
                    spec_object_exactly_described_if_any(pl->terms[j].constant);
                if ((the_region) && (wo_of_kind(the_region, kind_region))) {
                    pl->predicate = STORE_POINTER_binary_predicate(a_region_contains_b_predicate);
                    PROPOSITION_EDITED(pl, prop);
                }
            }
        }
    }
    return prop;
}
```

The function simp_region_containment is called from 6/sconv.

§**19. Reduce binary predicates (deduction).** If we are able to reduce a binary to a unary predicate, we will probably gain considerably by being able to eliminate a variable altogether. For instance, suppose we have "Mme Cholet is in a burrow". This will initially come out as

$$\exists x : burrow(x) \wedge in(\texttt{Cholet}, x)$$

To test that proposition requires trying all possible burrows x. But exploiting the fact that Mme Cholet can only be in one place at a time, we can reduce the binary predicate to equality, thus:

$$\exists x : burrow(x) \wedge is(\texttt{ContainerOf}(\texttt{Cholet}), x)$$

A later simplification can then observe that this tells us what x must be, and eliminate both quantifier and variable.

Formally, suppose B is a predicate with a function f_B such that $B(x, y)$ is true if and only $y = f_B(x)$. Then:

$$\Sigma = \cdots B(t_1, t_2) \cdots \quad \longrightarrow \quad \Sigma' = \cdots is(f_B(t_1), t_2) \cdots$$

Similarly, if there is a function g_B such that $B(x, y)$ if and only if $x = g_B(y)$ then

$$\Sigma = \cdots B(t_1, t_2) \cdots \quad \longrightarrow \quad \Sigma' = \cdots is(t_1, g_B(t_2)) \cdots$$

See "Binary Predicates" in Chapter 5 for how such functions are constructed. Not all BPs have them: the reason for our fudge on regional containment (above) is that direct containment does, but region containment doesn't, and this is why it was necessary to separate the two out.

```
pcalc_prop *simp_reduce_predicates(pcalc_prop *prop, int *changed) {
    TRAVERSE_VARIABLE(pl);
    *changed = FALSE;
    TRAVERSE_PROPOSITION(pl, prop) {
        binary_predicate *bp = atom_is_binary_predicate(pl);
        if (bp) {
            int j;
            for (j=0; j<2; j++)
                if ((bp_get_term_as_function_of_other(bp, j)) &&
                    (bp_allows_function_simplification(bp))) {
                    pl->terms[1-j] = term_new_function(bp, pl->terms[1-j], 1-j);
                    pl->predicate = STORE_POINTER_binary_predicate(a_is_b_predicate);
                    PROPOSITION_EDITED(pl, prop);
                }
        }
    }
    return prop;
}
```

The function simp_reduce_predicates is called from 6/sconv.

§20. Eliminating determined variables (deduction). The above operations will try to get as many variables as possible into a form which makes their values explicit with a predicate $is(v, t)$. We detect such equations and use them to eliminate the variable concerned, where this is safe.

```
define NOT_BOUND_AT_ALL 1
define BOUND_BY_EXISTS 2
define BOUND_BY_SOMETHING_ELSE 3

pcalc_prop *simp_eliminate_redundant_variables(pcalc_prop *prop, int *changed) {
    TRAVERSE_VARIABLE(pl);
    int level, binding_status[26], binding_level[26], binding_sequence[26];
    pcalc_prop *position_of_binding[26];
    *changed = FALSE;
    ⟨Find out where and how variables are bound 21⟩;
    EliminateVariables:
    level = 0;
    TRAVERSE_PROPOSITION(pl, prop) {
        if (element_get_group(pl->element) == OPEN_OPERATORS_GROUP) level++;
        if (element_get_group(pl->element) == CLOSE_OPERATORS_GROUP) level--;
        if (atom_is_equality_predicate(pl)) {
            int j;
            for (j=1; j>=0; j--) {
                int var_to_sub = pl->terms[j].variable;
                int var_in_other_term = term_variable_underlying(&(pl->terms[1-j]));
                int var_is_redundant = FALSE, value_can_be_subbed = FALSE;
                ⟨Decide if the variable is redundant, and if its value can safely be subbed 22⟩;
                if ((var_is_redundant) && (value_can_be_subbed)) {
                    int permitted;
                    vars_substitute_term(prop, var_to_sub, pl->terms[1-j],
                        TRUE, &permitted, changed);
                    if (permitted) {
                        LOGIF(WORKINGS, "Substituting %c <-- $0\n",
                            pcalc_vars[var_to_sub], &(pl->terms[1-j]));
                        first delete the is(v,t) predicate
                        prop = prop_delete_atom(prop, pl_prev);
                        then unbind the variable, by deleting its ∃v quantifier
                        prop = prop_delete_atom(prop, position_of_binding[var_to_sub]);
                        binding_status[var_to_sub] = NOT_BOUND_AT_ALL;
                        then substitute for all other occurrences of v
                        prop = vars_substitute_term(prop, var_to_sub, pl->terms[1-j],
                            FALSE, NULL, changed);
                        *changed = TRUE;
                        since the proposition is now shorter by 2 atoms, this loop terminates
                        goto EliminateVariables;
                    }
                }
            }
        }
    }
    vars_renumber(prop);                                   for the sake of tidiness
    return prop;
}
```

The function simp_eliminate_redundant_variables is called from 6/sconv.

§**21.** The information-gathering stage:

⟨Find out where and how variables are bound 21⟩ ≡

```
int j, c = 0, level = 0;
for (j=0; j<26; j++) {
    binding_status[j] = NOT_BOUND_AT_ALL;
    binding_level[j] = 0; binding_sequence[j] = 0; position_of_binding[j] = NULL;
}
TRAVERSE_PROPOSITION(pl, prop) {
    if (element_get_group(pl->element) == OPEN_OPERATORS_GROUP) level++;
    if (element_get_group(pl->element) == CLOSE_OPERATORS_GROUP) level--;
    if (atom_is_quantifier(pl)) {
        int v = pl->terms[0].variable;
        if (atom_is_existence_quantifier(pl)) binding_status[v] = BOUND_BY_EXISTS;
        else binding_status[v] = BOUND_BY_SOMETHING_ELSE;
        binding_level[v] = level;
        binding_sequence[v] = c;
        position_of_binding[v] = pl_prev;
    }
    c++;
}
```

This code is used in §20.

§**22.** At this point we have a predicate $is(t, f_A(f_B(\cdots s)))$. Should the term t be a variable v, which is bound by an $\exists v$ atom at the same level in its subexpression, then we can consider eliminating v by substituting $v = f_A(f_B(\cdots s))$.

But only if the term s underneath those functions does not make the equation $is(v, f_A(f_B(\cdots s)))$ implicit. Suppose s depends on a variable w which is bound and occurs *after* the binding of v. The value of such a variable w can depend on the value of v. Saying that $v = s$ may therefore not determine a unique value of v at all: it may be a subtle condition passed by a whole class of possible values, or none.

The simplest example of such circularity is $is(v, v)$, true for all v. More problematic is $is(v, f_C(v))$, "v is the container of v", which is never true. Still worse is

$$\exists v : V_{=2} w : is(v, w)$$

which literally says there is a value of v equal to two different things – certainly false. But if we were allowed to eliminate v, we would get just

$$V_{=2} w$$

which asserts "there are exactly two objects" – which is certainly not a valid deduction, and might even be true.

Here `var_to_sub` is v and `var_in_other_term` is w, or else they are -1 if no variables are present in their respective terms.

⟨Decide if the variable is redundant, and if its value can safely be subbed 22⟩ ≡

```
if ((var_to_sub >= 0)
    && (binding_status[var_to_sub] == BOUND_BY_EXISTS)
    && (binding_level[var_to_sub] == level))
        var_is_redundant = TRUE;
if ((var_in_other_term < 0)
    || (binding_status[var_in_other_term] == NOT_BOUND_AT_ALL)
    || (binding_sequence[var_in_other_term] < binding_sequence[var_to_sub]))
        value_can_be_subbed = TRUE;
```

This code is used in §20.

§23. Simplify non-relation (deduction). As a result of the previous simplifications, it fairly often happens that we find a term like

$$\neg(\mathit{thing}(t.\texttt{component\_parent}))$$

in the proposition. This comes out of text such as "... not part of something", asserting first that there is no y such that t is a part of y, and then simplifying to remove the y variable. A term like the one above is then left behind. But the negation is cumbersome, and makes the proposition harder to assert or test. Exploiting the fact that `component_parent` is a property which is either the part-parent or else `nothing`, we can simplify to:

$$\mathit{is}(t.\texttt{component\_parent}, \texttt{nothing})$$

And similar tricks can be pulled for other various-to-one-object predicates.

Formally, let B be a binary predicate supporting either a function f_B such that $B(x, y)$ iff $f_B(x) = y$, or else such that $B(x, y)$ iff $f_B(y) = x$; and such that the values of f_B are objects. Let K be a kind of object. Then:

$$\Sigma = \cdots \neg(K(f_B(t))) \cdots \quad \longrightarrow \quad \Sigma' = \cdots \mathit{is}(f_B(t), \texttt{nothing}) \cdots$$

A similar trick for kinds of value is not possible, because – unlike objects – they have no "not a valid case" value analogous to the non-object `nothing`.

```
pcalc_prop *simp_not_related_to_something(pcalc_prop *prop, int *changed) {
    TRAVERSE_VARIABLE(pl);
    *changed = FALSE;
    TRAVERSE_PROPOSITION(pl, prop) {
        pcalc_prop *kind_atom;
        if (prop_match(pl, 3,
            NEGATION_OPEN_ATOM, NULL,
            KIND_ATOM, &kind_atom,
            NEGATION_CLOSE_ATOM, NULL)) {
            world_object *k = kovko_get_kind(kind_atom->assert_kind_of_value);
            if (k) {
                binary_predicate *bp = NULL;
                pcalc_term KIND_term = kind_atom->terms[0];
                if (KIND_term.function) bp = KIND_term.function->bp;
                if ((bp) && (bp_term_kind(bp, 1) == k)) {
                    specification *new_nothing =
                        new_QUANTITY_spec(i6_nothing_quantity);
                    prop = prop_ungroup_after(prop, pl_prev, NULL);          remove negation grouping
                    prop = prop_delete_atom(prop, pl_prev);                  remove KIND_ATOM
                    now insert equality predicate:
                    prop = prop_insert_atom(prop, pl_prev,
                        atom_binary_PREDICATE_new(a_is_b_predicate,
                            KIND_term, term_new_constant(new_nothing)));
                    PROPOSITION_EDITED(pl, prop);
                }
            }
        }
    }
    return prop;
}
```

The function simp_not_related_to_something is called from 6/sconv.

§24. Convert gerunds to nouns (deduction). Suppose we write:

The funky thing to do is a stored action that varies.

and subsequently:

the funky thing to do is waiting

Here "waiting" is a gerund, and although it describes an action it is a noun (thus a value) rather than a condition. We coerce its constant value accordingly.

```
pcalc_prop *simp_convert_gerunds(pcalc_prop *prop, int *changed) {
    TRAVERSE_VARIABLE(pl);
    *changed = FALSE;
    TRAVERSE_PROPOSITION(pl, prop) {
        if (atom_is_equality_predicate(pl)) {
            int i;
            for (i=0; i<2; i++)
                if (species_is(pl->terms[i].constant, TEST_ACTION_SPC))
                    coerce_TEST_ACTION_to_DESCRIPTION_OF_ACTION(pl->terms[i].constant);
        }
    }
    return prop;
}
```

The function simp_convert_gerunds is called from 6/sconv.

§25. Eliminate to have meaning property ownership (fudge). The verb "to have" normally means ownership of a physical thing, but it can also arise from text such as

the balloon has weight at most 1

where it's the numerical "weight" property which is owned by the balloon. (The language of abstract "property" always echoes that of real physical things – consider how the iTunes Music Store invites you to "buy" what is at best a lease of temporary, partial and revocable rights to make use of something with no physical essence. This isn't a con trick, or not altogether so. We like the word "buy"; we immediately understand it.) At this stage of simplification, the above has produced

$$\mathit{at\text{-}most}(\texttt{weight}, 1) \wedge \mathit{is}(\texttt{balloon}, f_H(\texttt{weight}))$$

where H is the predicate `a_has_b_predicate`. As it stands, this proposition will fail type-checking, because it contains an implicit free variable – the object which owns the weight. We make this explicit by removing $\mathit{is}(\texttt{balloon}, f_H(\texttt{weight}))$ and replacing all other references to `weight` with "the weight of `balloon`".

This is a fudge because it assumes – possibly wrongly – that all references to the weight are to the weight of the same thing. In sufficiently contrived sentences, this wouldn't be true.

```
pcalc_prop *simp_eliminate_to_have(pcalc_prop *prop, int *changed) {
    TRAVERSE_VARIABLE(pl);
    *changed = FALSE;
    TRAVERSE_PROPOSITION(pl, prop) {
        if (atom_is_equality_predicate(pl)) {
            int i;
            for (i=0; i<2; i++)
                if ((pl->terms[i].function) &&
                    (pl->terms[i].function->bp == a_has_b_predicate) &&
                    (pl->terms[i].function->fn_of.constant) && (pl->terms[1-i].constant)) {
                    specification *spec = pl->terms[i].function->fn_of.constant;
                    if (spec_is_actual_CONSTANT_of_kova(spec, PROPERTY_TY))
                        ⟨Found an indication of who owns a property 26⟩;
                }
        }
    }
    return prop;
}
```

The function simp_eliminate_to_have is called from 6/sconv.

§26. So the current atom is $\mathit{is}(f_H(P), C)$ or $\mathit{is}(C, f_H(P))$ (according to whether i is 0 or 1), for a property P and a constant term C.

⟨Found an indication of who owns a property 26⟩ ≡

```
    property_name *prn = PROPERTY_spec_to_property_name(spec);
    specification *po_spec =
        new_PROPERTY_VALUE_spec(spec, pl->terms[1-i].constant);
    po_spec->word_ref1 = prn->word_ref1; po_spec->word_ref2 = prn->word_ref2;
    int no_substitutions_made;
    prop = prop_substitute_prop_cons(prop, prn, po_spec, &no_substitutions_made, pl);
    if (no_substitutions_made > 0) {
        prop = prop_delete_atom(prop, pl_prev);
        PROPOSITION_EDITED_REPEATING_CURRENT(pl, prop);
    }
```

This code is used in §25.

§**27.** Here we make the necessary substitution of "P" with "the P of C", where P is a property and C the constant value of its owner. We make this to every occurrence throughout the proposition, except for the one in the original $is(f_H(P), C)$ atom, and we count the number of changes made.

```
pcalc_prop *prop_substitute_prop_cons(pcalc_prop *prop, property_name *prn,
    specification *po_spec, int *count, pcalc_prop *not_this) {
    TRAVERSE_VARIABLE(pl);
    int j, c = 0;
    TRAVERSE_PROPOSITION(pl, prop)
        if (pl != not_this)
            for (j=0; j<pl->arity; j++) {
                pcalc_term *pt = &(pl->terms[j]);
                while (pt->function) pt = &(pt->function->fn_of);
                if (pt->constant == NULL) continue;
                if (spec_is_actual_CONSTANT_of_kova(pt->constant, PROPERTY_TY)) {
                    property_name *tprn;
                    tprn = PROPERTY_spec_to_property_name(pt->constant);
                    if (tprn == prn) {
                        pt->constant = po_spec;
                        c++;
                    }
                }
            }
    *count = c;
    return prop;
}
```

§28. Turn all rooms to everywhere (fudge). This rather special rule handles the consequences of the English word "everywhere". Inform reads that as "all rooms", literally "every where", which is logical but loses the connotation of place – by "everywhere", we usually mean "in all rooms", so that the sentence

The sky is everywhere.

means the sky is in every room, not that the sky is equal to every room. Since the literal reading would make no useful sense, Inform fudges the proposition to change it to the idiomatic one.

$$\Sigma = \cdots \forall v \in \{v \mid room(v)\} : is(v, t) \quad \longrightarrow \quad \Sigma' = \cdots everywhere(t)$$
$$\Sigma = \cdots \forall v \in \{v \mid room(v)\} : is(t, v) \quad \longrightarrow \quad \Sigma' = \cdots everywhere(t)$$
$$\Sigma = \cdots \not\forall v \in \{v \mid room(v)\} : is(v, t) \quad \longrightarrow \quad \Sigma' = \cdots \neg(everywhere(t))$$
$$\Sigma = \cdots \not\forall v \in \{v \mid room(v)\} : is(t, v) \quad \longrightarrow \quad \Sigma' = \cdots \neg(everywhere(t))$$

Note that we match this only at the end of a proposition, where v can have no other consequence.

```
pcalc_prop *simp_is_all_rooms(pcalc_prop *prop, int *changed) {
    TRAVERSE_VARIABLE(pl);
    *changed = FALSE;
    TRAVERSE_PROPOSITION(pl, prop) {
        pcalc_prop *q_atom, *k_atom, *bp_atom;
        if ((prop_match(pl, 6,
            QUANTIFIER_ATOM, &q_atom,
            DOMAIN_OPEN_ATOM, NULL,
            KIND_ATOM, &k_atom,
            DOMAIN_CLOSE_ATOM, NULL,
            PREDICATE_ATOM, &bp_atom,
            END_PROP_HERE, NULL)) &&
            ((atom_is_forall_quantifier(q_atom)) || (atom_is_notall_quantifier(q_atom))) &&
            (kovko_get_kind(k_atom->assert_kind_of_value) == kind_room) &&
            (bp_atom->arity == 2) &&
            (RETRIEVE_POINTER_binary_predicate(bp_atom->predicate) == a_is_b_predicate)) {
            int j, v = k_atom->terms[0].variable;
            for (j=0; j<2; j++) {
                if ((bp_atom->terms[1-j].variable == v) && (v >= 0)) {
                    prop = prop_delete_atom(prop, pl_prev);                  remove QUANTIFIER_ATOM
                    prop = prop_delete_atom(prop, pl_prev);                  remove DOMAIN_OPEN_ATOM
                    prop = prop_delete_atom(prop, pl_prev);                  remove KIND_ATOM
                    prop = prop_delete_atom(prop, pl_prev);                  remove DOMAIN_CLOSE_ATOM
                    prop = prop_delete_atom(prop, pl_prev);                  remove PREDICATE_ATOM
                    if (atom_is_notall_quantifier(q_atom))
                        prop = prop_insert_atom(prop, pl_prev, atom_new(NEGATION_CLOSE_ATOM));
                    prop = prop_insert_atom(prop, pl_prev,
                        atom_EVERYWHERE_new(bp_atom->terms[j]));
                    if (atom_is_notall_quantifier(q_atom))
                        prop = prop_insert_atom(prop, pl_prev, atom_new(NEGATION_OPEN_ATOM));
                    PROPOSITION_EDITED(pl, prop);
                    break;
                }
            }
        }
    }
    return prop;
}
```

The function simp_is_all_rooms is called from 6/sconv.

Type Check Propositions 6/tcpr

Purpose

Predicate calculus is a largely symbolic exercise, and its rules of working tend to assume that all predicates are meaningful for all terms: this means, for instance, that "if blue is 14" is likely to make a well-formed sentence in predicate calculus. In this section we reject such propositions on the grounds that they violate type-checking requirements on relations – in this example, the equality relation.

Definitions

¶1. We can unambiguously find the kind of value of any constant C, so if a proposition's terms are all constant then type-checking is easy. $is(4, \texttt{score})$ good, $is(4, \texttt{"fish"})$ bad. The subtlety comes in interpreting $is(4, x)$, where x is a variable. Our calculus allows variables to range over many domains – numbers, texts, scenes, objects, and so on. So it turns out to be convenient to have a structure holding a choice of kinds of value for the contents of the variables; such structures exist temporarily on the stack while we work.

```
typedef struct variable_type_assignment {
    struct kind_of_value *assigned_KOVs[26];                    one for each of the 26 variables
} variable_type_assignment;
```

The structure variable_type_assignment is private to this section.

¶2. Another convenience is a sort of kit for preparing problem messages:

```
typedef struct tc_problem_kit {
    int issue_error;
    int ew1, ew2;
    char *intention;
    int log_to_I6_text;
    int flag_problem;
} tc_problem_kit;
```

The structure tc_problem_kit is shared with 2/prob3, 6/equal, 9/cmpbp and 10/qnbp.

---

§**1. Problem reporting kit.** The caller to `prop_type_check` has to fill this form out first. Paperwork, what can you do, eh?

```
tc_problem_kit tc_no_problem_reporting(void) {
    tc_problem_kit tck;
    tck.issue_error = FALSE; tck.ew1 = -1; tck.ew2 = -1; tck.intention = "be silent checking";
    tck.log_to_I6_text = FALSE; tck.flag_problem = FALSE; return tck;
}
tc_problem_kit tc_problem_reporting(int w1, int w2, char *intent) {
    tc_problem_kit tck = tc_no_problem_reporting();
    tck.issue_error = TRUE; tck.ew1 = w1; tck.ew2 = w2; tck.intention = intent;
    return tck;
}
```

The function tc_no_problem_reporting is called from 6/pform, 7/tc, 11/ap, 13/nft and 13/test.
The function tc_problem_reporting is called from 6/asp, 6/defer and 7/tc.

§**2.** A version used only for the internal testing mode, when we print the outcome into the debugging log, but diverted to an I6 string in the compiled code.

```
tc_problem_kit tc_problem_logging(void) {
    tc_problem_kit tck = tc_no_problem_reporting();
    tck.intention = "be internal testing"; tck.log_to_I6_text = TRUE; return tck;
}
```

The function tc_problem_logging is called from 13/test.

§3. Type-checking whole propositions. This section provides a single routine to the rest of Inform: `prop_type_check`. We determine the KOVs for all variables, then work through the proposition, ensuring that every predicate-like atom has terms which match at least one possible reading of the meaning of the atom.

As usual in Inform, type-checking is not a passive process. If it can make sense of the proposition by changing it, it will do so.

```
int prop_type_check(pcalc_prop *prop, tc_problem_kit tck_s) {
    TRAVERSE_VARIABLE(pl);
    variable_type_assignment vta;
    tc_problem_kit *tck = &tck_s;
    int j;
    if (prop == NULL) return ALWAYS_MATCH;
    if (prop_is_well_formed(prop) == FALSE) internal_error("type-checking malformed proposition");
    ⟨First make sure any constants in the proposition have themselves been typechecked 4⟩;
    for (j=0; j<26; j++) vta.assigned_KOVs[j] = NULL;
    ⟨Look at KIND atoms to see what kinds of value are asserted for the variables 5⟩;
    ⟨Assume any still-unfathomable variables represent objects 6⟩;
    TRAVERSE_PROPOSITION(pl, prop) {
        for (j=0; j<pl->arity; j++) kov_of_term(&(pl->terms[j]), &vta, tck);
        if (tck->flag_problem) return NEVER_MATCH;
        if (pl->element == KIND_ATOM)
            ⟨A KIND atom is not allowed if it can be proved to be false 8⟩;
        if ((pl->element == PREDICATE_ATOM) && (pl->arity == 2))
            ⟨A binary predicate is required to apply to terms of the right kinds 10⟩;
        if ((pl->element == PREDICATE_ATOM) && (pl->arity == 1))
            ⟨A unary predicate is required to have an interpretation matching the kind of its term 9⟩;
        if (pl->element == EVERYWHERE_ATOM)
            ⟨An EVERYWHERE atom needs its term to be an object 13⟩;
        if (pl->element == ISAKIND_ATOM)
            ⟨An ISAKIND atom needs its term to be an object 12⟩;
        if (pl->element == HERE_ATOM)
            ⟨A HERE atom needs its term to be an object 14⟩;
    }
    if (tck->log_to_I6_text) ⟨Show the variable assignment in the debugging log 15⟩;
    return ALWAYS_MATCH;
}
```

The function prop_type_check is called from 6/pform, 6/asp, 6/defer, 7/tc, 11/ap, 13/nft and 13/test.

§**4. Finding KOVs for variables.** Every specification compiled by Inform has to pass through type-checking. That includes the ones which occur as constants inside propositions, and this is where.

The presence of an `UNKNOWN` constant indicates something which failed to be recognised by the S-parser. That shouldn't happen, but we allow for it so that we can recover from an already reported problem.

After this point, we can be sure that every constant is an actual rather than generic value, which makes sense within itself. For instance, if it is a phrase to decide a value, that its parameters themselves satisfy type-checking: "the location of 134" would be rejected.

⟨First make sure any constants in the proposition have themselves been typechecked 4⟩ ≡

```
TRAVERSE_PROPOSITION(pl, prop) {
    int j;
    for (j=0; j<pl->arity; j++) {
        specification *spec = term_constant_underlying(&(pl->terms[j]));
        if (spec) {
            if ((spec_is_UNKNOWN(spec)) ||
                (spec_is_generic(spec)) ||
                (typecheck_without_expectations(spec) == NEVER_MATCH))
                ⟨Recover from problem in S-parser by not issuing problem 7⟩;
        }
    }
}
```

This code is used in §3.

§**5.** If the proposition contains contradictory `KIND` atoms, it automatically fails type-checking, even if there is no implication that both apply at once. This throws out, for instance:

```
1. a scene which is not a number
[ scene(x) & NOT[ number(x) NOT] ]
Failed: proposition would not type-check
x is both scene and number
```

It could be argued that all scenes ought to pass this proposition, but we will treat it as a piece of nonsense, like "if Wednesday is not custard".

⟨Look at KIND atoms to see what kinds of value are asserted for the variables 5⟩ ≡

```
TRAVERSE_PROPOSITION(pl, prop)
    if (pl->element == KIND_ATOM) {
        int v = pl->terms[0].variable;
        if (v >= 0) {
            kind_of_value *new_kov = pl->assert_kind_of_value;
            if (is_kova(new_kov, OBJECT_TY)) new_kov = kova(OBJECT_TY);
            kind_of_value *old_kov = vta.assigned_KOVs[v];
            if (old_kov) {
                if (can_we_cast_kovs(old_kov, new_kov) == NEVER_MATCH) {
                    if (tck->log_to_I6_text)
                        LOG("%c is both $u and $u\n", pcalc_vars[v], old_kov, new_kov);
                    issue_kind_typecheck_error(old_kov, new_kov, tck, pl);
                    return NEVER_MATCH;
                }
            }
            vta.assigned_KOVs[v] = new_kov;
        }
    }
```

This code is used in §3.

§**6.** It's possible for a proposition to specify nothing about the kind of a variable (usually a free one). If so, it's assumed to be an object. For instance, if we define

Definition: a container is empty if the number of things in it is 0.

then we find that, say:

```
1. empty which is empty
[ 'empty'(x) & 'empty'(x) ]
x (free) - object.
```

though in fact it would also have been viable for x to be a rulebook, a list, or various other kinds of value.

⟨Assume any still-unfathomable variables represent objects 6⟩ ≡

```
int j;
for (j=0; j<26; j++)
    if (vta.assigned_KOVs[j] == NULL)
        vta.assigned_KOVs[j] = kova(OBJECT_TY);
```

This code is used in §3.

§**7.** The following is really rather paranoid; it ought to be certain that a problem message has already been issued, but just in case not...

⟨Recover from problem in S-parser by not issuing problem 7⟩ ≡

```
if (problem_count == 0) {
    if (tck->log_to_I6_text) LOG("Atom $o contains failed constant\n", pl);
    if (tck->issue_error == FALSE) return NEVER_MATCH;
    sentence_problem(_P_(BelievedImpossible),                    well, except via internal tests
        "this sentence seems to contain a value which makes no sense",
        "and I'm unfortunately not able to say why not. I can only "
        "suggest staring at it for a while.");
    LOG("Stare at this, then: $S\n", spec);
}
return NEVER_MATCH;
```

This code is used in §4.

§**8. Type-checking enforced on predicate-like atoms.** As with contradictory `KIND` atoms applied to variables, we will reject any `KIND` atom applied to a constant if it necessarily fails – even when the sense of the proposition is arguably correct. For example:

```
1. 100 is not a text
[ NOT[ text('100') NOT] ]
Failed: proposition would not type-check
Term '100' is number not text
```

"100 is not a number" would pass, on the other hand. It is obviously false, but not meaningless.

⟨A KIND atom is not allowed if it can be proved to be false 8⟩ ≡

```
kind_of_value *need_to_find = pl->assert_kind_of_value;
if (is_kova(need_to_find, OBJECT_TY)) need_to_find = kova(OBJECT_TY);
kind_of_value *actually_find = kov_of_term(&(pl->terms[0]), &vta, tck);
if (can_we_cast_kovs(actually_find, need_to_find) == NEVER_MATCH) {
    if (tck->log_to_I6_text)
        LOG("Term $0 is $u not $u\n", &(pl->terms[0]), actually_find, need_to_find);
    issue_kind_typecheck_error(actually_find, need_to_find, tck, pl);
    return NEVER_MATCH;
}
```

This code is used in §3.

§**9.** See below, of course.

⟨A unary predicate is required to have an interpretation matching the kind of its term 9⟩ ≡
```
if (type_check_unary_predicate(pl, &vta, tck) == NEVER_MATCH) {
    if (tck->log_to_I6_text) LOG("Adjective $o cannot be applied\n", pl);
    return NEVER_MATCH;
}
```
This code is used in §3.

§**10.** The BP case is interesting because it forgives a failure in one case: of $is(t, C)$, where C is a constant representing a value of an enumerated KOV. Sentence conversion is actually quite good at distinguishing these cases and can see the difference between "the bus is red" and "the fashionable hue is red", but it is defeated by cases where adjectives representing values are used about other values – "the Communist Rally is red", where "Communist Rally" is a scene rather than an object, for instance. We first try requiring `Rally` to be a colour: when that fails, we see if the atom *red*(`Rally`) would work instead. If it would, we make the change within the proposition.

⟨A binary predicate is required to apply to terms of the right kinds 10⟩ ≡
```
binary_predicate *bp = RETRIEVE_POINTER_binary_predicate(pl->predicate);
if (bp_is_the_wrong_way_round(bp)) internal_error("BP wrong way round");
if (type_check_binary_predicate(pl, &vta, tck) == NEVER_MATCH) {
    if (bp == a_is_b_predicate) {
        adjective_list_entry *alt = prop_noun_to_adj_conversion(pl->terms[1]);
        if (alt) {
            pcalc_prop test_unary = *pl;
            test_unary.arity = 1;
            test_unary.predicate = STORE_POINTER_adjective_list_entry(alt);
            if (type_check_unary_predicate(&test_unary, &vta, tck) == NEVER_MATCH)
                ⟨The BP fails type-checking 11⟩;
            pl->arity = 1;
            pl->predicate = STORE_POINTER_adjective_list_entry(alt);
        } else ⟨The BP fails type-checking 11⟩;
    } else ⟨The BP fails type-checking 11⟩;
}
```
This code is used in §3.

§**11.**

⟨The BP fails type-checking 11⟩ ≡
```
if (tck->log_to_I6_text) LOG("BP $o cannot be applied\n", pl);
return NEVER_MATCH;
```
This code is used in §10.

§**12.**

⟨An ISAKIND atom needs its term to be an object 12⟩ ≡
```
kind_of_value *actually_find = kov_of_term(&(pl->terms[0]), &vta, tck);
if (can_we_cast_kovs(actually_find, kova(OBJECT_TY)) == NEVER_MATCH)
    internal_error("ISAKIND atom misapplied");
```
This code is used in §3.

§**13.**

⟨An EVERYWHERE atom needs its term to be an object 13⟩ ≡

```
kind_of_value *actually_find = kov_of_term(&(pl->terms[0]), &vta, tck);
if (can_we_cast_kovs(actually_find, kova(OBJECT_TY)) == NEVER_MATCH) {
    if (tck->log_to_I6_text)
        LOG("Term $0 is $u not an object\n", &(pl->terms[0]), actually_find);
    quote_kov(4, actually_find);
    tcp_problem(_P_(C6EverywhereMisapplied), tck,
        "that seems to say that a value - specifically, %4 - is everywhere. "
        "To Inform, everywhere means 'in every room', and only objects "
        "can be everywhere - in fact not even all of those, as it's a "
        "privilege reserved for backdrops. (For instance, 'The sky is a "
        "backdrop. The sky is everywhere.' is allowed.)");
    return NEVER_MATCH;
}
```

This code is used in §3.

§**14.** It seems to be true that the A-parser never generates propositions which apply `HERE` incorrectly, but just in case:

⟨A HERE atom needs its term to be an object 14⟩ ≡

```
kind_of_value *actually_find = kov_of_term(&(pl->terms[0]), &vta, tck);
if (can_we_cast_kovs(actually_find, kova(OBJECT_TY)) == NEVER_MATCH) {
    if (tck->log_to_I6_text)
        LOG("Term $0 is $u not an object\n", &(pl->terms[0]), actually_find);
    quote_kov(4, actually_find);
    tcp_problem(_P_(BelievedImpossible), tck,
        "that seems to say that a value - specifically, %4 - is here. "
        "To Inform, here means 'in the room we're talking about', so only "
        "objects can be 'here'.");
    return NEVER_MATCH;
}
```

This code is used in §3.

§**15.** Not so much for the debugging log as for the internal test, in fact, which prints the log to an I6 string. This is the type-checking report in the case of success.

⟨Show the variable assignment in the debugging log 15⟩ ≡

```
int j, var_states[26], c=0;
vars_determine_status(prop, var_states, NULL);
for (j=0; j<26; j++)
    if (var_states[j] != UNUSED_VST) {
        LOG("%c%s - $u. ", pcalc_vars[j],
            (var_states[j] == FREE_VST)?" (free)":"",
            vta.assigned_KOVs[j]); c++;
    }
if (c>0) LOG("\n");
```

This code is used in §3.

§**16. The KOV of a term.** The following routine works out the kind of value stored in a term, something which requires contextual information: unless we know the kind of value stored in each variable, we cannot know the kind of value a general term represents, which is why the routine is here and not in the Terms section.

```
kind_of_value *kov_of_term(pcalc_term *pt, variable_type_assignment *vta,
    tc_problem_kit *tck) {
    kind_of_value *kov = kov_of_term_inner(pt, vta, tck);
    pt->term_checked_as_kov = kov;
    if (kov == NULL) tck->flag_problem = TRUE;
    return kov;
}
```

§**17.** The case needing attention is a term in the form $t = f_B(s)$. By recursion we can know the KOV of s, but we must check that f_B can validly be applied to a value of that kind. If B is a binary predicate with domains R and S (i.e., a subset of $R \times S$) then we will either have $f_B : R \to S$ or vice versa; so we have to check that s lies in R (or S, respectively).

```
kind_of_value *kov_of_term_inner(pcalc_term *pt, variable_type_assignment *vta,
    tc_problem_kit *tck) {
    if (pt->constant) return spec_evaluates_to(pt->constant);
    if (pt->variable >= 0) return vta->assigned_KOVs[pt->variable];
    if (pt->function) {
        binary_predicate *bp = pt->function->bp;
        kind_of_value *kov_found = kov_of_term(&(pt->function->fn_of), vta, tck);
        kind_of_value *kov_from = approximate_argument_kov(bp, pt->function->from_term);
        kind_of_value *kov_to = approximate_argument_kov(bp, 1 - pt->function->from_term);
        if ((kov_from) && (can_we_cast_kovs(kov_found, kov_from) == NEVER_MATCH)) {
            if (tck->log_to_I6_text)
                LOG("Term $0 applies function to $u not $u\n", pt, kov_found, kov_from);
            issue_bp_typecheck_error(bp, kov_found, kov_to, tck);
            kov_found = kov_from;                                    the better to recover
        }
        if (kov_to) return kov_to;
        return kov_found;
    }
    return NULL;
}
```

§**18.** Some relations specify a KOV or kind of object for their terms, others do not. When they do specify a kind of object, we usually want to be forgiving about nuances of the kind of object – for our purposes here, any object will do. (Run-time type checking takes care of those nuances better.)

The following gives either the KOV or, for any kind of object at all, `OBJECT_TY`, for a given term. It should be used only where the BP is one constraining its terms, such as when it provides a f_B function.

```
kind_of_value *approximate_argument_kov(binary_predicate *bp, int i) {
    kind_of_value *kov = bp_term_kind_of_value(bp, i);
    if (kov) return kov;
    return kova(OBJECT_TY);
}
```

§**19. Type-checking predicates.** We take unary predicates first, then binary. Unary predicates are just adjectives, and all of the work for that has already been done in Chapter 5, so we need only produce a problem message when the worst happens.

```
int type_check_unary_predicate(pcalc_prop *pl, variable_type_assignment *vta,
    tc_problem_kit *tck) {
    adjective_list_entry *tr = RETRIEVE_POINTER_adjective_list_entry(pl->predicate);
    adjectival_phrase *aph = get_adjective_from_list_entry(tr);
    kind_of_value *kov = kov_of_term(&(pl->terms[0]), vta, tck);
    if (aph_applicable_to(aph, kov) == FALSE) {
        if (tck->log_to_I6_text)
            LOG("Adjective '$W' undefined on $u\n", aph->word_ref1, aph->word_ref2, kov);
        quote_words(4, aph->word_ref1, aph->word_ref2);
        quote_kov(5, kov);
        tcp_problem(_P_(C6AdjectiveMisapplied), tck,
            "that seems to involve applying the adjective '%4' to %5 - and I "
            "have no definition of it which would apply in that situation. "
            "(Try looking it up in the Lexicon part of the Phrasebook index "
            "to see what definition(s) '%4' has.)");
        return NEVER_MATCH;
    }
    return ALWAYS_MATCH;
}
```

§**20.** Binary predicates (BPs) are both easier and harder. Easier because they have only one definition at a time (unlike, say, the adjective "empty"), harder because the work hasn't already been done and because some BPs – like "is" – are polymorphic. Here goes:

```
int type_check_binary_predicate(pcalc_prop *pl, variable_type_assignment *vta,
    tc_problem_kit *tck) {
    binary_predicate *bp = RETRIEVE_POINTER_binary_predicate(pl->predicate);
    kind_of_value *kovs_of_terms[2], *kovs_required[2];
    ⟨Work out what KOVs we find, and what we should have found 21⟩;
    int result = bp_typecheck(bp, kovs_of_terms, kovs_required, tck);
    if (result != DECLINE_TO_MATCH) {
        if (result == NEVER_MATCH_SAYING_WHY_NOT) {
            if (tck->issue_error == FALSE) return NEVER_MATCH;
            if (pl->terms[0].function)
                issue_bp_typecheck_error(pl->terms[0].function->bp,
                    kov_of_term(&(pl->terms[0].function->fn_of), vta, tck),
                    kovs_of_terms[1], tck);
            else if (pl->terms[1].function)
                issue_bp_typecheck_error(pl->terms[1].function->bp,
                    kovs_of_terms[0],
                    kov_of_term(&(pl->terms[1].function->fn_of), vta, tck),
                    tck);
            else {
                LOG("($u, $u) failed in $2\n", kovs_of_terms[0], kovs_of_terms[1], bp);
                quote_kov(4, kovs_of_terms[0]);
                quote_kov(5, kovs_of_terms[1]);
                tcp_problem(_P_(C6ComparisonFailed), tck,
                    "that would mean comparing two kinds of value which cannot mix - "
```

```
                "%4 and %5 - so this must be incorrect.");
        }
        return NEVER_MATCH;
    }
    return result;
}
⟨Apply default rule applying to most binary predicates 22⟩;
return ALWAYS_MATCH;
}
```

§**21.** Once again we treat any kind of object as just `OBJECT_TY`, but we do take note that some BPs – like *is* – specify no kinds at all, and so produce a `kov_required` which is `NULL`.

⟨Work out what KOVs we find, and what we should have found 21⟩ ≡

```
int i;
for (i=0; i<2; i++) {
    if (bp_term_kind_of_value(bp, i)) kovs_required[i] = bp_term_kind_of_value(bp, i);
    else if (bp_term_kind(bp, i)) kovs_required[i] = kova(OBJECT_TY);
    else kovs_required[i] = NULL;
    kovs_of_terms[i] = kov_of_term(&(pl->terms[i]), vta, tck);
}
```

This code is used in §20.

§**22.** The default rule is straightforward: the KOVs found have to match the KOVs required.

⟨Apply default rule applying to most binary predicates 22⟩ ≡

```
int i;
for (i=0; i<2; i++)
    if (kovs_required[i])
        if (can_we_cast_kovs(kovs_of_terms[i], kovs_required[i]) == NEVER_MATCH) {
            if (tck->log_to_I6_text)
                LOG("Term %d is $u not $u\n",
                    i, kovs_of_terms[i], kovs_required[i]);
            issue_bp_typecheck_error(bp, kovs_of_terms[0], kovs_of_terms[1], tck);
            return NEVER_MATCH;
        }
```

This code is used in §20.

§23. Two problem messages needed more than once.

```
void issue_bp_typecheck_error(binary_predicate *bp,
    kind_of_value *t0, kind_of_value *t1, tc_problem_kit *tck) {
    quote_kov(4, t0);
    quote_kov(5, t1);
    quote_relation(6, bp);
    tcp_problem(_P_(C6TypeCheckBP2), tck,
        "that would mean applying %6 to kinds of value which do not "
        "fit - %4 and %5 - so this must be incorrect.");
}

void issue_kind_typecheck_error(kind_of_value *actually_find,
    kind_of_value *need_to_find, tc_problem_kit *tck, pcalc_prop *ka) {
    binary_predicate *bp = NULL;
    if ((ka) && (GENERAL_POINTER_IS_NULL(ka->predicate) == FALSE))
        bp = RETRIEVE_POINTER_binary_predicate(ka->predicate);
    quote_kov(4, actually_find);
    quote_kov(5, need_to_find);
    if (bp) {
        quote_relation(6, bp);
        tcp_problem(_P_(C6TypeCheckBP2a), tck,
            "that doesn't work because you use %6 with %4 instead of %5.");
    } else {
        tcp_problem(_P_(C6TypeCheckKind), tck,
            "%4 cannot be %5, so this must be incorrect.");
    }
}
```

The function issue_bp_typecheck_error is called from 9/mapbp and 9/cmpbp.

The Equality Relation 6/equal

Purpose

To define that prince among predicates, the equality relation.

Definitions

¶1. This predicate expresses the meaning of $a = b$, and plays a very special role in our calculus.

```
binary_predicate *a_is_b_predicate = NULL;
```

---

§**1. Initial stock.** This relation is hard-wired in, and it is made in a slightly special way since (alone among binary predicates) it has no distinct reversal.

```
void EQUALITY_KBP_create_initial_stock(void) {
    a_is_b_predicate = make_equality();
    register_reworded_meaning(MISCELLANEOUS_MC, RELATION_SMC,
        equality_V, relation_V, -1, -1, 0,
        STORE_POINTER_binary_predicate(a_is_b_predicate));
}
```

The function EQUALITY_KBP_create_initial_stock is called from 5/bp.

§**2. Second stock.** There is none, of course.

```
void EQUALITY_KBP_create_second_stock(void) {
}
```

The function EQUALITY_KBP_create_second_stock is called from 5/bp.

§**3. Typechecking.** This is a very polymorphic relation, in that it can accept terms of almost any kind.

```
int EQUALITY_KBP_typecheck(binary_predicate *bp,
        kind_of_value **kovs_of_terms, kind_of_value **kovs_required, tc_problem_kit *tck) {
    if ((is_kova(kovs_of_terms[0], OBJECT_TY)) &&
        (kov_name_can_coincide_with_property(kovs_of_terms[1])))
        ⟨Apply rule for "is" applied to an object and a value 4⟩
    else if ((is_kova(kovs_of_terms[0], STORED_ACTION_TY)) &&
        (is_kova(kovs_of_terms[1], DESCRIPTION_OF_ACTION_TY)))
        ⟨Apply rule for "is" applied to a stored action and a gerund 5⟩
    else
        ⟨Allow comparison only where left domain and right domain are not disjoint 6⟩;
    return ALWAYS_MATCH;
}
```

The function EQUALITY_KBP_typecheck is called from 5/bp.

§**4.** This case is only separated in order to provide a better problem message for a fairly common mistake:

⟨Apply rule for "is" applied to an object and a value 4⟩ ≡

```
property_name *prn = kov_get_coinciding_property(kovs_of_terms[1]);
if (prn == NULL) {
    if (tck->log_to_I6_text)
        LOG("Comparison of object with $u value\n", kovs_of_terms[1]);
    quote_kov(4, kovs_of_terms[1]);
    tcp_problem(_P_(C6NonPropertyCompared), tck,
        "taken literally that says that an object is the same as a "
        "value. Maybe you intended to say that the object "
        "has a property - but right now %4 is not yet a property; if you "
        "want to use it as one, you'll need to say so. (You can turn a "
        "kind of value - say, 'colour' - into a property by writing - "
        "say - 'A thing has a colour.')");
    return NEVER_MATCH;
}
```

This code is used in §3.

§**5.** This rule is the easiest of all – we always allow it. For instance,

if the current action is taking something...

⟨Apply rule for "is" applied to a stored action and a gerund 5⟩ ≡

```
;
```

This code is used in §3.

§**6.** With comparisons there is no restriction on the two KOVs except that they must match each other; $is(t, s)$ is allowed if $K(t) \subseteq K(s)$ or vice versa. So rules and rulebooks are comparable, for instance, but numbers and scenes are not.

⟨Allow comparison only where left domain and right domain are not disjoint 6⟩ ≡

```
if ((can_we_cast_kovs(kovs_of_terms[0], kovs_of_terms[1]) == NEVER_MATCH) &&
    (can_we_cast_kovs(kovs_of_terms[1], kovs_of_terms[0]) == NEVER_MATCH)) {
    if (tck->log_to_I6_text)
        LOG("Unable to compare $u with $u\n", kovs_of_terms[0], kovs_of_terms[1]);
    return NEVER_MATCH_SAYING_WHY_NOT;
}
```

This code is used in §3.

§**7. Assertion.** In general values differ, and cannot be equated by fiat. But an exception is setting a global variable.

```
int EQUALITY_KBP_assert(binary_predicate *bp,
        world_object *wo0, specification *spec0,
        world_object *wo1, specification *spec1) {
    if (spec_is_actual_NONLOCAL_VARIABLE(spec0)) {
        quantity *q = RETRIEVE_FROM_SPEC(spec0, quantity);
        inference_subject infs = inference_subject_q(q);
        inference *i;
        if (wo1) {
            i = create_global_inference(infs, world_object_to_OBJECT_spec(wo1));
            i->word_ref1 = wo1->word_ref1; i->word_ref2 = wo1->word_ref2;
        } else {
            i = create_global_inference(infs, spec1);
            i->word_ref1 = spec1->word_ref1; i->word_ref2 = spec1->word_ref2;
        }
        join_inference(i, infs);
        qty_now_has_initial_value_set(q);
        return TRUE;
    }
    return FALSE;
}
```

The function EQUALITY_KBP_assert is called from 5/bp.

§**8. Compilation.** Since we are compiling to I6, which is itself a C-level programming language, it looks at first as if we can compile *is* into `==` when testing equality and `=` when asserting it: thus

now the score is 10;

if the score is 10, ...

would compile to `score = 10;` and `if (score == 10) ...` respectively.

But there are three problems with this simplistic approach to "A is B".

(a) Sometimes "now A is B" must set a property of A, which does not change, rather than making A equal to B; and similarly for testing.
(b) Sometimes A is reference to a value stored in some data structure other than a local or global variable: for example, in "now entry 3 of the passenger list is 208", where A is "entry 3 of the passenger list". Access to this value is via I6 routines in the template, and the form of what we compile has to be different depending on whether we are reading or writing.
(c) Sometimes the values in question are block values, that is, they are stored as pointers to blocks of data on the heap at run-time. If we compile "now T is X", where T is an indexed text variable and X is some piece of text, we cannot simply copy the pointers: T needs to hold a fresh, independent copy of the text referred to by X.

Problem (a) is easily detected by looking at the kinds of value of A and B. To handle problems (b) and (c), we use a general framework in which the schema is a function of both the storage class of A and the kinds of value of both A and B.

```
int EQUALITY_KBP_compile(int task, binary_predicate *bp, annotated_i6_schema *asch) {
    kind_of_value *st[2];
    st[0] = cind_kind_of_value_of_term(asch->pt0);
    st[1] = cind_kind_of_value_of_term(asch->pt1);
    if ((is_kova(st[0], OBJECT_TY)) &&
        (kov_name_can_coincide_with_property(st[1])) && (kov_get_coinciding_property(st[1])))
        ⟨Handle the case of setting a property of A separately 9⟩;
    switch (task) {
        case TEST_ATOM_TASK:
            sch_write_to_existing(asch->schema, kov_interpret_test_equality(st[0], st[1]));
            return TRUE;
        case NOW_ATOM_FALSE_TASK:
            break;
        case NOW_ATOM_TRUE_TASK: {
            int storage_class = spec_get_storage_form(asch->pt0.constant);
            ⟨Forbid setting something to an action pattern 10⟩;
            if (storage_class != UNKNOWN) {
                ⟨Exceptional case of setting the "player" global variable 11⟩;
                sch_write_to_existing(asch->schema,
                    kov_interpret_store(storage_class, st[0], st[1]));
                return TRUE;
            }
            ⟨Issue problem message for being unable to set equal 12⟩;
            break;
        }
    }
    return FALSE;
}
```

The function EQUALITY_KBP_compile is called from 5/bp.

§**9.** So here is the exceptional case (a) mentioned above. Suppose we have:

if the lantern is bright, ...

where "bright" is one value of "luminance", which is both a kind of value and also a property. We then want to test if the luminance property of the lantern equals the constant value "bright"; and similarly for "now the lantern is bright".

⟨Handle the case of setting a property of A separately 9⟩ ≡

```
property_name *prn = kov_get_coinciding_property(st[1]);
switch (task) {
    case TEST_ATOM_TASK:
        sch_write_to_existing_1(asch->schema, "*1.%s == *2", prn_get_i6_identifier(prn));
        return TRUE;
    case NOW_ATOM_FALSE_TASK:
        break;
    case NOW_ATOM_TRUE_TASK:
        sch_write_to_existing_1(asch->schema, "*1.%s = *2", prn_get_i6_identifier(prn));
        return TRUE;
}
return FALSE;
```

This code is used in §8.

§**10.** Setting true exceptions. It could be argued that this ought to be a typechecking rule, but it applies only to setting true, so is here instead. The distinction is there because we can check whether an action is "taking a container" (say) but can't set a stored action equal to it, because it's too vague: what is the container to be taken?

⟨Forbid setting something to an action pattern 10⟩ ≡

```
if (is_kova(asch->pt1.term_checked_as_kov, DESCRIPTION_OF_ACTION_TY)) {
    sentence_problem(_P_(C6SetStoredAction),
        "that would use 'now' to set a value to an action",
        "and you've written this action as a condition: you need to use the "
        "form 'action of...' instead. For instance, 'now X is the action of "
        "taking the diamonds' is legal, but 'now X is taking the diamonds' "
        "is not.");
    asch->schema = NULL;
    return TRUE;
}
```

This code is used in §8.

§**11.** A little bit of support within Inform to help the template layer. It's very important that code compiled by Inform 7 not ever simply alter the value of `player`, as that if executed would break the invariants for the various I6 variables about the current situation. The correct thing is always to call the template routine `ChangePlayer`. But the user of I7 should not need to know that, so we take care of it here:

⟨Exceptional case of setting the "player" global variable 11⟩ ≡

```
if (spec_is_this_NONLOCAL_VARIABLE(asch->pt0.constant, player_quantity)) {
    sch_write_to_existing(asch->schema, "ChangePlayer(*2)");
    return TRUE;
}
```

This code is used in §8.

§**12.** Rather than just returning `FALSE` for a generic problem message, we issue one that's more helpfully specific and return `TRUE`.

⟨Issue problem message for being unable to set equal 12⟩ ≡

```
if (spec_get_constant_quantity_if_any(asch->pt0.constant))
    sentence_problem(_P_(C6CantChangeNamedConstant),
        "I can't change that",
        "because it is a name for a constant value. Some named values, "
        "like 'the score', can be changed, because they were defined "
        "as values that vary. But others are fixed. If we write 'The "
        "oak tree can be sturdy, lightning-struck or leaning.', for "
        "instance, then 'sturdy' is a name for a value which is fixed, "
        "just as the number '7' is fixed.");
else
    sentence_problem(_P_(C6CantEquateValues),
        "equality is not something I can change",
        "so either those are already the same or are different, and I "
        "can't alter matters.");
asch->schema = NULL;
return TRUE;
```

This code is used in §8.

§13. Problem message text.

```
int EQUALITY_KBP_describe_for_problems(binary_predicate *bp, char *problem_buffer) {
    return FALSE;
}
```

The function EQUALITY_KBP_describe_for_problems is called from 5/bp.

Assert Propositions 6/asp

Purpose

To declare that a given proposition is a true statement about the state of the world when play begins.

Definitions

¶1. Inside Inform, the term "the model" means the collection of the following:

(1) The objects, divided into:
  (1a) World objects, such as "Brazilian frog" or "Jungle Clearing"; each of which belongs to a kind;
  (1b) Kinds of (world) object, such as "animal" or "room"; each of which is a kind in turn of another kind, even if only "kind", which is its own kind;
  (1c) Named constant values, such as "red" or "Entire Game"; each of which belongs to a kind of value;
  (1d) Kinds of value whose values are all named, such as "colour" or "scene".

(2) The associated values, divided into:
  (2a) Properties, such as "open" or "carrying capacity", each of which is held by some subset of the objects – for instance, containers and people are permitted to have the "carrying capacity" property;
  (2b) Global variables, such as "time of day".

(3) Relations between the objects (or in some cases between objects and values outside the model), such as "containment".

Much else lies outside the model: values like `152` or `"alfalfa"`, rules, rulebooks, phrases, relations which can only be tested (such as "less than" or "visibility") and so on. Some of this is excluded because it is beyond the source text's power to decide facts about it (for instance, that 5 is less than 6); other matter is left out because it only relates to what happens to the world after its initial creation (for instances, actions and activities). The model is that which the source text can decide about the initial state of things.

¶2. The initial state of the world is built from the model. Properly speaking a few other things contribute, too – such as the entries initially found in table cells – but these don't need careful handling, since they are explicitly declared as literals in the source text: there is no need to analyse their meaning.

To build or change the model, we assert that propositions about it are true, using `prop_true_in_world_model` or `prop_true_in_world_model_about`. This is the *only* way to create kinds, world objects, kinds of value, global variables, and constant values, and also the *only* way to attach properties to objects, to set property values or the kind of a given object or the value of a global variable, or to declare that relationships hold.

However, creating new property names and new relations does not count as a change in the model world. (After all, we could create a new property called "scent" and a new relation called "admiring", but then choose not to attach scent to anything or to relate any objects by admiring. The model world would then not have changed at all.) So creating new properties and new relations is not done by asserting propositions.

¶3. `prop_true_in_world_model` asserts propositions in which all variables are bound (or which have no variables); `prop_true_in_world_model_about` asserts propositions in which x is free but all other variables are bound, and substitutes either an object O or a value V into x before asserting. These two procedures are the entire API, so to speak, for growing or changing the model. They are used by the detailed-look part of the A-parser, mostly found in Chapter 8, which takes assertion sentences in the source text and converts them into a series of propositions which it would like to make true.

Either way those requests come in, they all end up in the central `prop_true_in_model` procedure, one of the most important choke points within Inform. `prop_true_in_model` and its delegates – routines to assert the truth of various adjectives or relations – are allowed to call routines such as `set_kind` and `create_world_object` which are forbidden for use in the rest of Inform. These are guarded with the following macro, to ensure that we don't accidentally break this rule:

```
define PROTECTED_MODEL_PROCEDURE
    if (ptim_recursion_depth == 0)
        internal_error("protected model-affecting procedure used outside proposition assert");
```

```
int ptim_recursion_depth = 0;                                   depth of recursion of prop_true_in_model
```

---

§**1. Entrance.** This first entrance is a mere alias for the second.

```
void prop_true_in_world_model(pcalc_prop *prop) {
    prop_true_in_world_model_about(prop, NULL, NULL);
}
```

The function prop_true_in_world_model is called from 8/creat, 8/knowp, 8/relv, 9/prop, 9/pp and 10/tab.

§**2.** If we are working along a proposition and reach, say, $door(x)$, we can only assert that if we know what the value of x is. We therefore keep an array (or a pair of arrays) holding our current beliefs about the values of the variables – this is called the "identification slate".

```
world_object **current_interpretation_as_wo = NULL;             must point to a 26-element array
specification **current_interpretation_as_spec = NULL;          must point to a 26-element array
```

§**3.** The second entrance, then, keeps track of the recursion depth but also ensures that the identification slate is always correct, stacking them so that an inner `prop_true_in_model` has an independent slate from an outer one.

```
void prop_true_in_world_model_about(pcalc_prop *prop,
    world_object *ox, specification *ots) {
    world_object **saved_interpretation_as_wo = current_interpretation_as_wo;
    specification **saved_interpretation_as_spec = current_interpretation_as_spec;
    int saved_prevailing_mood = prevailing_mood;                which should not change, but just in case
    ⟨Establish a new identification slate for the variables in the proposition 4⟩;
    ptim_recursion_depth++;
    prop_true_in_model(prop);
    ptim_recursion_depth--;
    prevailing_mood = saved_prevailing_mood;
    current_interpretation_as_wo = saved_interpretation_as_wo;
    current_interpretation_as_spec = saved_interpretation_as_spec;
}
```

The function prop_true_in_world_model_about is called from 5/rel, 8/creat, 8/mass, 8/knowc, 8/knowp, 8/relv, 8/imp, 9/pp and 10/tab.

§**4.** The slate is initially blank unless substitutions for variable x have been supplied; x of course is variable number 0.

```
⟨Establish a new identification slate for the variables in the proposition 4⟩ ≡
    world_object *ciawo[26]; specification *ciats[26];
    int k;
    for (k=0; k<26; k++) { ciawo[k] = NULL; ciats[k] = NULL; }
    ciawo[0] = ox; ciats[0] = ots;
    if ((ox) && (ots)) internal_error("both ox and ots targets set");
    current_interpretation_as_wo = ciawo; current_interpretation_as_spec = ciats;
```

This code is used in §3.

§**5. Main procedure.** As can be seen, `prop_true_in_model` is a simple procedure. After a little fuss to check that everything is set up right, we simply run through the proposition one atom at a time.

This is a modest scheme. We are unable to assert any proposition other than $\exists$, so that we never see their attendant domain brackets. We are therefore left with a proposition in the form $P_1 \wedge P_2 \wedge ... \wedge P_n$ where each P_i is either a predicate-like atom, an $\exists v$ term for some variable v, or else $\neg(...)$ of a similar conjunction.

This is an ambiguous task if we have to assert $\neg(P \wedge Q)$, which is in effect a form of disjunction: $P \wedge Q$ fails if either is false, so which do we falsify? We choose both.

That means we can simply assert each atom in turn, with a parity depending on its nesting in negation brackets, which is nice and easy to write:

```
void prop_true_in_model(pcalc_prop *prop) {
    if (prop == NULL) return;
    ⟨Record the proposition in the debugging log 6⟩;
    if (prop_contains_nonexistence_quantifier(prop))
        ⟨Issue a problem message explaining that the proposition isn't exact enough 7⟩;
    ⟨Typecheck the proposition, in case this has not already been done 8⟩;
    ⟨Check the identification slate against variable usage in the proposition 9⟩;
    TRAVERSE_VARIABLE(pl);
    int now_negated = FALSE;
    TRAVERSE_PROPOSITION(pl, prop) {
        switch(pl->element) {
            case NEGATION_OPEN_ATOM: case NEGATION_CLOSE_ATOM:
                now_negated = (now_negated)?FALSE:TRUE; break;
            case QUANTIFIER_ATOM: ⟨Assert the truth or falsity of a QUANTIFIER atom 10⟩; break;
            case KIND_ATOM: ⟨Assert the truth or falsity of a KIND atom 14⟩; break;
            case PREDICATE_ATOM:
                switch (pl->arity) {
                    case 1: ⟨Assert the truth or falsity of a unary predicate 17⟩; break;
                    case 2: ⟨Assert the truth or falsity of a binary predicate 18⟩; break;
                }
                break;
            case HERE_ATOM: ⟨Assert the truth or falsity of a HERE atom 16⟩; break;
            case EVERYWHERE_ATOM: ⟨Assert the truth or falsity of an EVERYWHERE atom 15⟩; break;
            case ISAKIND_ATOM: case ISAKOV_ATOM: case ISAVAR_ATOM:
                if (now_negated) internal_error("ISA... atoms cannot be negated");
                break;
        }
    }
}
```

§**6.** The certainty, the initial interpretation slate, and the proposition are combined into a single line in the log:

⟨Record the proposition in the debugging log 6⟩ ≡

```
int i;
LOGIF(ASSERTIONS, "::");
switch(prevailing_mood) {
    case IMPOSSIBLE_CE: LOGIF(ASSERTIONS, " (impossible)"); break;
    case UNLIKELY_CE: LOGIF(ASSERTIONS, " (unlikely)"); break;
    case UNKNOWN_CE: LOGIF(ASSERTIONS, " (no certainty)"); break;
    case LIKELY_CE: LOGIF(ASSERTIONS, " (likely)"); break;
    case CERTAIN_CE: break;
    default: LOG(" (unknown certainty)"); break;
}
for (i=0; i<26; i++) {
    if (current_interpretation_as_wo[i]) {
        LOGIF(ASSERTIONS, " %c = $O", pcalc_vars[i], current_interpretation_as_wo[i]);
    } else if (current_interpretation_as_spec[i]) {
        LOGIF(ASSERTIONS, " %c = $S", pcalc_vars[i], current_interpretation_as_spec[i]);
    }
}
LOGIF(ASSERTIONS, " $D\n", prop);
```

This code is used in §5.

§**7.** It's surprisingly hard to get this problem message, because the assertion-maker rejects most of the obvious ways to try it with more direct problems. It took me about twenty sentences to get there ("The car is a vehicle in most rooms which are dark" will do it).

⟨Issue a problem message explaining that the proposition isn't exact enough 7⟩ ≡

```
sentence_problem(_P_(C6CantAssertQuantifier),
    "the relationship you describe is not exact enough",
    "so that I cannot be sure of the initial situation. A specific "
    "relationship would be something like 'the box is a container in "
    "the Attic', rather than 'the box is a container in a room which "
    "is dark' (fine, but which dark room? You must say).");
return;
```

This code is used in §5.

§**8.** Almost all propositions derive from sentences in the source text, but a crucial exception is the first one to be asserted: $\exists x : isakind(x)$, which creates the kind "kind". Type-checking problems never arise with this in any case, so it doesn't matter that we wouldn't know what text to use in them.

⟨Typecheck the proposition, in case this has not already been done 8⟩ ≡

```
int w1 = -1, w2 = -1;
if ((kind_kind) && (current_sentence)) { [[w1, w2 <-- current_sentence]]; }
if (prop_type_check(prop, tc_problem_reporting(w1, w2, "be asserting something"))
    != ALWAYS_MATCH)
    return;
```

This code is used in §5.

§**9.** This does nothing functional, except that it allows an interpretation as a world object to trump one as a specification; useful since the A-parser often specifies O and V as the object and value referred to by a given node in the parse tree, and since an object is also a value, this often means that both are given. If we have O, then, we cancel V.

As we shall see, it's permitted to interpret a bound variable *after* its quantifier, but not before, and in particular not at the start of the proposition. So we require that the slate identify exactly the free variables, and no others.

⟨Check the identification slate against variable usage in the proposition 9⟩ ≡

```
int i, valid = TRUE, var_states[26];
vars_determine_status(prop, var_states, &valid);
if (valid == FALSE) internal_error("tried to assert malformed proposition");
for (i=0; i<26; i++) {
    int set = FALSE;
    if (current_interpretation_as_spec[i]) set = TRUE;
    if (current_interpretation_as_wo[i]) {
        if (current_interpretation_as_spec[i]) current_interpretation_as_wo[i] = NULL;
        set = TRUE;
    }
    if ((var_states[i] == UNUSED_VST) && (set))
        internal_error("tried to set an unused variable");
    if ((var_states[i] == BOUND_VST) && (set))
        internal_error("tried to set a bound variable");
    if ((var_states[i] == FREE_VST) && (set == FALSE))
        internal_error("failed to set a free variable");
}
```

This code is used in §5.

§**10. Creations.** To assert the truth of $\exists x$, we must create an object to become x; that will provide a value in subsequent uses of x in the same proposition, so the new value has to be added to the identification slate.

⟨Assert the truth or falsity of a QUANTIFIER atom 10⟩ ≡

```
int v = pl->terms[0].variable; if (v == -1) internal_error("bad QUANTIFIER atom");
if (now_negated) internal_error("tried to negate existence");

int name1 = -1, name2 = -1;
int is_a_kind = FALSE, is_a_var = FALSE, is_a_kov = FALSE;
kind_of_value *kov = NULL;

⟨Scan subsequent atoms to find the name, nature and kind of what is to be created 11⟩;
⟨Create the object and add to the identification slate 12⟩;
⟨Record the new creation in the debugging log 13⟩;
```

This code is used in §5.

§**11.** Note that all five of these atoms are optional; the proposition might consist of just $\exists x$ alone, which creates a nameless object, since as usual we interpret no indication of a kind as meaning "object".

⟨Scan subsequent atoms to find the name, nature and kind of what is to be created 11⟩ ≡

```
TRAVERSE_VARIABLE(lookahead);
TRAVERSE_PROPOSITION(lookahead, pl)
    if ((lookahead->arity == 1) && (lookahead->terms[0].variable == v)) {
        switch(lookahead->element) {
            case KIND_ATOM: kov = lookahead->assert_kind_of_value; break;
            case CALLED_ATOM: atom_CALLED_get_name(lookahead, &name1, &name2); break;
            case ISAKIND_ATOM: is_a_kind = TRUE; break;
            case ISAKOV_ATOM: is_a_kov = TRUE; break;
            case ISAVAR_ATOM: is_a_var = TRUE; break;
        }
    }
```

This code is used in §10.

§**12.** There are really four cases: new KOV, new global variable, new world object or kind of object, and new constant value.

⟨Create the object and add to the identification slate 12⟩ ≡

```
if (is_a_kov) {
    kov = make_new_atomic_kind_of_value(name1, name2);
    current_interpretation_as_spec[v] = new_generic_CONSTANT_type(kov);
} else if (is_a_var) {
    if (kov == NULL) kov = kova(OBJECT_TY);
    quantity *q = new_model_quantity(name1, name2, kov, TRUE);
    current_interpretation_as_spec[v] = new_actual_NONLOCAL_VARIABLE_type(q);
} else {
    if ((kov == NULL) || (is_kova(kov, OBJECT_TY))) {
        current_interpretation_as_wo[v] = create_world_object(name1, name2, is_a_kind);
    } else {
        quantity *q = new_model_quantity(name1, name2, kov, FALSE);
        current_interpretation_as_spec[v] = spec_copy(qty_get_initial_value(q));
    }
}
```

This code is used in §10.

§**13.** It's useful to log the new creation, especially for objects which have duplicate names to others already made:

⟨Record the new creation in the debugging log 13⟩ ≡

```
if (current_interpretation_as_spec[v]) {
    LOGIF(ASSERTIONS, ":: %c <-- $S\n", pcalc_vars[v], current_interpretation_as_spec[v]);
} else if (current_interpretation_as_wo[v]) {
    LOGIF(ASSERTIONS, ":: %c <-- $O\n", pcalc_vars[v], current_interpretation_as_wo[v]);
}
```

This code is used in §10.

§14. Asserting kinds. Note that we never assert the KOV of anything. Typechecking won't allow such an atom to exist unless it states something already true, so there is no need. We are only interested if there is a kind of object `ko` here.

Once again, the problem messages in this section for negated attempts are really quite hard to generate, because the A-parser usually gets there first. "There is a banana which is something which is not a door." will fall through here, but it isn't exactly an everyday sentence.

⟨Assert the truth or falsity of a KIND atom 14⟩ ≡

```
world_object *ko = kovko_get_kind(pl->assert_kind_of_value);
world_object *ox = wo_of_term(pl->terms[0]);
if (now_negated) {
    sentence_problem(_P_(C6CantAssertNonKind),
        "that seems to say what kind something doesn't have",
        "which is too vague. You must say what kind it does have.");
    return;
}
if ((ko) && (ox)) set_kind(ox, ko);
```

This code is used in §5.

§15. Asserting HERE and EVERYWHERE. Two special cases. The first, EVERYWHERE, declares that something is found in every room; we generate a non-room-specific sort of `FOUNDIN_INF` inference. While we could simply deduce that the object must be a backdrop (and `set_kind` to make it so), this is such an extreme business, so rarely needed, that it seems better to make the user spell out that we're dealing with a backdrop. So we play dumb.

⟨Assert the truth or falsity of an EVERYWHERE atom 15⟩ ≡

```
world_object *ox = wo_of_term(pl->terms[0]);
if ((ox == NULL) || (wo_of_kind(ox, kind_backdrop) == FALSE)) {
    sentence_problem(_P_(C6EverywhereNonBackdrop),
        "only a backdrop can be everywhere",
        "and no other kind of object will do. For instance, 'The sky is "
        "a backdrop which is everywhere.' is allowed, but 'The travelator "
        "is a vehicle which is everywhere.' is not.");
    return;
}
if (now_negated) {
    sentence_problem(_P_(C6CantAssertNegatedEverywhere),
        "that seems to say that something isn't everywhere",
        "which is too vague. You must say where it is.");
    return;
}
inference *i = create_inference(FOUNDIN_INF, prevailing_mood);
join_inference(i, inference_subject_wo(ox));
```

This code is used in §5.

§**16.** HERE means "this object is in the current room", which is not as easy to resolve as it looks, because at this point we don't know for certain what will be a room and what won't. So we record a special inference and put the problem aside for now.

⟨Assert the truth or falsity of a HERE atom 16⟩ ≡

```
world_object *ox = wo_of_term(pl->terms[0]);
if (ox == NULL) internal_error("HERE on null object");                    typechecking prevents this
if (now_negated) {
    sentence_problem(_P_(BelievedImpossible),
        "that seems to say that something isn't here",
        "which is too vague. You must say where it is.");
    return;
}
inference *i = create_inference(PARENTAGE_HERE_INF, CERTAIN_CE);
inf_set_object_reference(i, get_current_subject());
join_inference(i, inference_subject_wo(ox));
i = create_inference(ISAROOM_INF, IMPOSSIBLE_CE);
join_inference(i, inference_subject_wo(ox));
```

This code is used in §5.

§**17. Asserting predicates.** First, asserting *adjective*(t). We know that t evaluates to a kind of value over which *adjective* is defined, or the proposition would not have survived type-checking. But only some adjectives can be asserted; "open" can, but "visible" can't, for instance. `aph_assert` returns a success flag.

Note that if the adjective applies to an object `ox` then we supply its kind of value to the adjective-asserter as `kovko(ox)`, not as `ox->kind`. This ensures that (a) if `ox` is an physical object then a definition applying to `ox` alone will be used, and (b) if `ox` is a kind, then a definition applying to that kind will be used.

⟨Assert the truth or falsity of a unary predicate 17⟩ ≡

```
adjective_list_entry *tr = RETRIEVE_POINTER_adjective_list_entry(pl->predicate);
adjectival_phrase *aph = get_adjective_from_list_entry(tr);
int parity = (now_negated)?FALSE:TRUE, found;
if (adjective_used_positively(tr) == FALSE) parity = (parity)?FALSE:TRUE;
world_object *ox = wo_of_term(pl->terms[0]);
specification *ots = spec_of_term(pl->terms[0]);
if (ox) found = aph_assert(aph, kovko(ox), ox, NULL, parity);
else found = aph_assert(aph, spec_get_kind_of_value(ots), NULL, ots, parity);
if (found == FALSE) {
    quote_source(1, current_sentence);
    quote_words(2, aph->word_ref1, aph->word_ref2);
    handmade_problem(_P_(C6CantAssertAdjective));
    issue_problem_segment(
        "In the sentence %1, you ask me to arrange for something to be '%2' "
        "at the start of play. There are some adjectives ('open' or 'dark', "
        "for instance) which I can fix, but others are just too vague. For "
        "example, saying 'Peter is visible.' isn't allowed, because it "
        "doesn't tell me where Peter is. Like 'visible', being '%2' is "
        "something I can test during play at any time, but not something "
        "I can arrange at the start.");
    issue_problem_end();
}
```

This code is used in §5.

§**18.** Binary predicates, unlike unary ones, can only be asserted positively. This is because $\neg P(x)$ tells you something fairly definite, whereas $\neg Q(x, y)$ gives no information about what z might exist, if any, such that $Q(x, z)$. For instance, knowing that X is not part of Y gives us no help in determining where X is.

Another difference is that $R(x, y)$ can give you definite information about the kinds of x and y, where they are objects, because binary predicates have single definitions. (Knowing *locked*(x), by contrast, doesn't tell you whether x is a door or a container – adjectives can have multiple domains in which they have differing definitions.) In the case of a proposition produced by sentence conversion, that information is redundant since appropriate kind atoms were added to the proposition anyway. But we also assert propositions generated from tree conversion, which don't have these kind atoms.

⟨Assert the truth or falsity of a binary predicate 18⟩ ≡

```
if (now_negated) {
    sentence_problem(_P_(C6CantAssertNegatedRelations),
        "that seems to make a negative statement about a relationship",
        "which is too vague. You must make positive assertions.");
    return;
}
binary_predicate *bp;
pcalc_term pt0, pt1;
⟨Determine the BP and terms to be asserted 19⟩;
specification *spec0 = spec_of_term(pt0), *spec1 = spec_of_term(pt1);
world_object *wo0 = wo_of_term(pt0), *wo1 = wo_of_term(pt1);
if (bp != a_region_contains_b_predicate) {
    if (bp_term_kind(bp, 0)) cautiously_set_kind(wo0, bp_term_kind(bp, 0));
    if (bp_term_kind(bp, 1)) cautiously_set_kind(wo1, bp_term_kind(bp, 1));
}
if (bp_assert(bp, wo0, spec0, wo1, spec1) == FALSE) {
    LOG("$2 on ($O, $S; $O, $S)\n", bp, wo0, spec0, wo1, spec1);
    sentence_problem(_P_(BelievedImpossible),
        "that is an assertion I can't puzzle out",
        "which seems to involve placing two things in some sort of "
        "relationship, but if so then I can't make it work. Perhaps the "
        "sentence is too complicatedly phrased, and could be broken up "
        "into two or more sentences?");
}
```

This code is used in §5.

§**19.** The "is" predicate is not usually assertable, but $is(x, f_R(y))$ can be asserted since it is equivalent to $R(x, y)$ – this is where we unravel that. We reject compound uses of functions in this way, but in practice they hardly ever arise, and could only do so with quite complex sentences where it seems reasonable to tell the user to write something simpler and clearer.

⟨Determine the BP and terms to be asserted 19⟩ ≡

```
bp = RETRIEVE_POINTER_binary_predicate(pl->predicate);
pt0 = pl->terms[0]; pt1 = pl->terms[1];
if (bp == a_is_b_predicate) {
    pcalc_func *the_fn = pt0.function; int side = 1;
    if (the_fn == NULL) { the_fn = pt1.function; side = 0; }
    if (the_fn) {
        if ((pl->terms[side].function) || (the_fn->fn_of.function)) {
            sentence_problem(_P_(BelievedImpossible),
                "that is too complicated an assertion",
                "and cannot be declared as part of the initial situation. (It "
                "does make sense, and could be tested with 'if' - it's just "
                "too difficult to get right as an instruction about the starting "
                "situation.");
            return;
        }
        bp = the_fn->bp; pt0 = pl->terms[side]; pt1 = the_fn->fn_of;
    }
}
```

This code is used in §18.

§**20.** As we've already seen, we have to be cautious about the mechanism to draw inferences about kinds based on the relationships which objects have. Some cases are easy: if A is worn by B, then B is a person. But "in" can be very problematic. When one region is in another, we want to suppress any inferences which might wrongly be drawn about "in": this is a different kind of containment from the three-dimensional spatial one suggested by containers and rooms. It also complicates things that a backdrop can be "in" a region. So we play very safe and make no guesses about regions or the first term of `a_region_contains_b_predicate`.

We also never deduce "thing" as the kind by this mechanism. This is because world objects with no apparent declared kind are made into things by default anyway; so there is no need to risk setting the kind here at this stage.

```
void cautiously_set_kind(world_object *wo, world_object *k) {
    if ((wo == NULL) || (k == NULL) || (wo->kind_flag)) return;
    if (k == kind_thing) return;
    if (wo_of_kind(wo, kind_region)) return;
    set_kind(wo, k);
}
```

§**21. Evaluating terms.** In asserting a proposition, we are in effect acting as an interpreter rather than a compiler. Given any term, we need to produce either an object O or a more general value V. Recall that a term can be

(a) a constant C,
(b) a variable v, or
(c) a function $f_R(t)$ for another term t.

We are unable, at compile-time, to evaluate $f_R(t)$ for any relation R, and won't even try. We can evaluate a variable using the interpretation slate – that was its whole purpose. So the only case left is a constant:

```
specification *spec_of_term(pcalc_term pt) {
    if (pt.function) return NULL;
    if (pt.variable >= 0) return current_interpretation_as_spec[pt.variable];
    return pt.constant;
}
```

§**22.** The analogous routine to extract a world object, which normally takes precedence, is more convoluted. First, we could be looking at the name of a kind – in "A door is usually closed", "door" will appear here as a `DESCRIPTION_SPC` type, and we need to extract the world object of the kind as our return value. Second, we want to divert all assertions about "the player" so that they refer to the player object, not to the global variable "the player".

Users tend to expect that they can talk about properties of things as values, when setting up the world, and since a property value might be an object, we are going to be careful to reject a `PROPERTY_VALUE_SPC` type with a problem message. In practice the A-parser gets there first, but just in case.

```
world_object *wo_of_term(pcalc_term pt) {
    if (pt.function) return NULL;
    if (pt.variable >= 0) return current_interpretation_as_wo[pt.variable];
    specification *spec = pt.constant;
    if (species_is(spec, CONSTANT_SPC)) return wo_of_CONSTANT_OBJECT_if_any(spec);
    if (species_is(spec, DESCRIPTION_SPC)) {
        if (spec_get_described_object(spec)) return spec_get_described_object(spec);
        if (spec_get_described_kind(spec)) return spec_get_described_kind(spec);
    }
    if ((species_is(spec, NONLOCAL_VARIABLE_SPC)) &&
        (RETRIEVE_FROM_SPEC(spec, quantity) == player_quantity))
        return wo_yourself;
    if (species_is(spec, PROPERTY_VALUE_SPC)) {
        if (problem_count == 0)
            sentence_problem(_P_(BelievedImpossible),
                "that seems to refer to a property of something in a way that I "
                "can't follow",
                "perhaps because I've misread the word 'of' here, or perhaps "
                "because verbs are missing which a human reader can manage "
                "without, but I can't.");
        return NULL;
    }
    return NULL;
}
```

I6 Schemas 6/sch

Purpose

To create, and later expand upon, short prototypes of I6 syntax for such run-time tasks as the setting, unsetting or testing of a relation.

Definitions

¶1. An I6 schema is an intermediate-level code for the final stages of compiling to Inform 6 syntax. Its "prototype" is a C string encoded as ISO Latin-1, in which the asterisk `*` acts as an escape character.

"Expanding" an I6 schema is essentially a form of macro expansion. A caller supplies us with a schema and a number of parameters, each of which is either a literal piece of text or a predicate calculus term. We then copy the schema's prototype into the output, except as specified:

(1) `**` expands to a literal asterisk.

(2) `*1`, `*2`, ..., `*9` expands to the 1st to 9th parameter, where a literal text parameter is copied straight through, whereas a term parameter is compiled as a value. Expanding a parameter which was not supplied does nothing, but is not an error.
   (a) The modifier `!`, as in `*!1` to `*!9`, enables the use of local variables in any text substitutions compiled in the course of the term.
   (b) The modifier `#`, as in `*#1` to `*#9`, causes us to expand to the ID number of the kind of value of the parameter, not to the parameter itself. (If the parameter is literal text rather than a term, we do nothing.)

(3) `*=-` and `*=+` turn "dereference pointers" mode off and on, respectively. This has effect only when compiling a value whose content is stored on the heap; Inform ordinarily compiles this by making a new copy of the value and using a pointer to it, but if "dereference pointers" is off then the pointer to the original data is used instead. It's a sort of macro-expansion version of the difference between call-by-value and call-by-reference. The effect lasts only while the expander is running; when the expander finishes, it restores the mode to its setting at the start.

(4) `*-` means a single character backspace, which allows us to erase the last character of the I6 source generated by the expansion of an immediately preceding `*1` or `*2`. (This can be used to tack an optional call parameter onto a routine: by deleting the close bracket in `Example(45)` and then continuing with `,7)` we effectively change the call to `Example(45,7)`.) If the chosen output is a file rather than a string in memory, then `*-` is unavailable, since we aren't able to undo having written a character. (Yes, on some platforms we could do this, but we don't need to, so we won't.)

(5) `*##` is reserved for the use of higher-level code in building schemas – it has to do with locations of data on the heap – but is not strictly speaking legal in a schema. Attempting to expand it will cause an internal error.

Any other occurrence of an asterisk is illegal, and will throw an internal error.

¶2. The I6 schema structure is very simple, then:

```
define MAX_I6_SCHEMA_LENGTH 128
```
in fact 40 is plenty, but to be on the safe side

```
typedef struct i6_schema {
    char prototype[MAX_I6_SCHEMA_LENGTH];
} i6_schema;
```

The structure i6_schema is private to this section.

---

§**1. Building schemas.** The numerous versions of essentially similar construction routines are intended to make the relation-creating code a little cleaner and easier to read. We could try to use variadic macros, or a variable argument list, but this would only invite compiler warnings on some platforms.

This is where all schemas originate:

```
i6_schema *sch_new(char *text) {
    i6_schema *sch = CREATE(i6_schema);
    sch_write_to_existing(sch, text);
    return sch;
}
void sch_write_to_existing(i6_schema *sch, char *text) {
    if (strlen(text) >= MAX_I6_SCHEMA_LENGTH) internal_error("i6 schema overlong");
    strcpy(sch->prototype, text);
}
```

The function sch_new is called from 9/spabp, 10/qnbp and 12/cinv.

The function sch_write_to_existing is called from 5/aph, 6/equal, 6/atoms, 6/cdefp, 7/kov, 7/data and 9/provr.

§**2.** Now for some convenient routines to create new schemas with given contents:

define MAX_I6_SCHEMA_ATTEMPT 512 *plenty of room for conjectural schema overruns*

```
i6_schema *sch_new_1(char *text, char *str1) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, str1);
    return sch_new(temp);
}
i6_schema *sch_new_1d(char *text, int d1) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, d1);
    return sch_new(temp);
}
i6_schema *sch_new_2(char *text, char *str1, char *str2) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, str1, str2);
    return sch_new(temp);
}
i6_schema *sch_new_3(char *text, char *str1, char *str2, char *str3) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, str1, str2, str3);
    return sch_new(temp);
}
```

The function sch_new_1 is called from 5/rel, 9/mapbp and 9/vpbp.

The function sch_new_1d is called from 5/rel and 10/libp.

The function sch_new_2 is called from 5/rel and 9/vpbp.

The function sch_new_3 is called from 9/cmpbp.

§**3.** And lastly some routines to write new contents to an existing schema:

```
void sch_write_to_existing_1(i6_schema *sch, char *text, char *str1) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, str1);
    sch_write_to_existing(sch, temp);
}
void sch_write_to_existing_1d(i6_schema *sch, char *text, int d1) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, d1);
    sch_write_to_existing(sch, temp);
}
void sch_write_to_existing_2(i6_schema *sch, char *text, char *str1, char *str2) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, str1, str2);
    sch_write_to_existing(sch, temp);
}
void sch_write_to_existing_1s(i6_schema *sch, char *text, char *str1) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, str1);
    sch_write_to_existing(sch, temp);
}
void sch_write_to_existing_1s1d(i6_schema *sch, char *text, char *str1, int d1) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, str1, d1);
    sch_write_to_existing(sch, temp);
}
void sch_write_to_existing_2s(i6_schema *sch, char *text, char *str1, char *str2) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, str1, str2);
    sch_write_to_existing(sch, temp);
}
void sch_write_to_existing_2sd(i6_schema *sch, char *text, char *str1, char *str2, int d1) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, str1, str2, d1);
    sch_write_to_existing(sch, temp);
}
void sch_write_to_existing_2d(i6_schema *sch, char *text, int d1, int d2) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, d1, d2);
    sch_write_to_existing(sch, temp);
}
void sch_write_to_existing_3d(i6_schema *sch, char *text, int d1, int d2, int d3) {
    char temp[MAX_I6_SCHEMA_ATTEMPT];
    sprintf(temp, text, d1, d2, d3);
    sch_write_to_existing(sch, temp);
}
```

The function sch_write_to_existing_1 is called from 6/equal, 6/atoms, 9/prop and 12/def.
The function sch_write_to_existing_1d is called from 6/atoms and 7/data.
The function sch_write_to_existing_2 is called from 9/cmpbp.
The function sch_write_to_existing_1s is called from 5/bp.
The function sch_write_to_existing_1s1d is called from 9/qty.
The function sch_write_to_existing_2s is called from 5/bp.
The function sch_write_to_existing_2sd is called from 9/madj.
The function sch_write_to_existing_2d is called from 7/kov and 9/prop.
The function sch_write_to_existing_3d is called from 5/aph and 9/qty.

§**4. Emptiness.** A schema is empty if its prototype is the null string.

```
int sch_empty(i6_schema *sch) {
    if (sch == NULL) return TRUE;
    if (sch->prototype[0] == 0) return TRUE;
    return FALSE;
}
```

The function sch_empty is called from 5/aph.

§**5. Expansion.** We provide two routines as a sort of API for expanding schemas. The user can either specify two parameters, both of them terms...

```
void sch_expand(i6_schema *sch, OUTPUT_STREAM, pcalc_term *pt1, pcalc_term *pt2) {
    STREAM *save_dl = dl;
    if (OUT == NULL) return;
    TEMPORARY_STREAM;
    if (OUT == dl) dl = TEMP;
    sch_expand_inner(sch, TEMP, pt1, NULL, pt2, NULL);
    STREAM_COPY(OUT, TEMP);
    CLOSE_TEMPORARY_STREAM;
    dl = save_dl;
}
```

The function sch_expand is called from 5/aph, 6/atoms, 6/cind, 6/cdefp and 12/cinv.

§**6.** ...or two parameters, both of them strings.

```
void sch_expand_textual(i6_schema *sch, OUTPUT_STREAM, char *str1, char *str2) {
    STREAM *save_dl = dl;
    if (OUT == NULL) return;
    TEMPORARY_STREAM;
    if (OUT == dl) dl = TEMP;
    sch_expand_inner(sch, TEMP, NULL, str1, NULL, str2);
    STREAM_COPY(OUT, TEMP);
    CLOSE_TEMPORARY_STREAM;
    dl = save_dl;
}
```

The function sch_expand_textual is called from 12/cinv and 13/gtok.

§**7.** The actual expander is here:

```
void sch_expand_inner(i6_schema *sch, OUTPUT_STREAM,
    pcalc_term *pt1, char *str1, pcalc_term *pt2, char *str2) {
    STREAM_MUST_BE_IN_MEMORY(OUT);
    if (sch == NULL) internal_error("Expanded null I6 schema");
    sch_type_parameter(pt1);
    sch_type_parameter(pt2);
    char *schematic_text = sch->prototype;
    int cmode = 0, i = 0;
    if ((schematic_text[0] == '*') && (schematic_text[1] == '=') &&
        (schematic_text[2] == '-')) {
        cmode = DEREFERENCE_POINTERS_CMODE; i = 3;
    }
    BEGIN_COMPILATION_MODE;
    COMPILATION_MODE_EXIT(cmode);
    for (; schematic_text[i]; i++) {
        if (schematic_text[i] == '*') {
            ⟨Act on an asterisk expansion in the schema 8⟩;
            continue;
        }
        WRITE("%c", schematic_text[i]);
    }
    END_COMPILATION_MODE;
}
```

§**8.** At present we never expand on terms 3 to 9, but they are reserved for possible future use.

⟨Act on an asterisk expansion in the schema 8⟩ ≡

```
    BEGIN_COMPILATION_MODE;
    int give_kov_id = FALSE;
    if ((schematic_text[i+2]) && (isdigit(schematic_text[i+2]))) {
        switch (schematic_text[i+1]) {
            case '!': COMPILATION_MODE_ENTER(PERMIT_LOCALS_IN_TEXT_CMODE); i++; break;
            case '#': give_kov_id = TRUE; i++; break;
        }
    }
    switch(schematic_text[++i]) {
        case '-': STREAM_BACKSPACE(OUT); break;
        case '1': sch_expand_parameter(OUT, pt1, str1, give_kov_id); break;
        case '2': sch_expand_parameter(OUT, pt2, str2, give_kov_id); break;
        case '3': case '4': case '5': case '6': case '7': case '8': case '9': break;
        case '#':
            if (schematic_text[++i] == '#') internal_error("can't expand *## in schema");
            else internal_error("schema syntax error");
            break;
        case '*': WRITE("*"); break;
        default: internal_error("schema syntax error");
    }
    END_COMPILATION_MODE;
```

This code is used in §7.

§**9.**

```
void sch_expand_parameter(OUTPUT_STREAM, pcalc_term *pt, char *str, int give_kov_id) {
    if (give_kov_id) {
        if (pt) {
            int id = kov_I6_ID(pt->term_checked_as_kov);
            WRITE("%d", id);
        }
    } else {
        if (str) WRITE("%s", str);
        else if (pt) term_compile(OUT, *pt);
    }
}
```

§**10.** Last and very much least: in case we receive an untypechecked term, we fill in its KOV.

```
void sch_type_parameter(pcalc_term *pt) {
    if ((pt) && (pt->constant) && (pt->term_checked_as_kov == NULL))
        pt->term_checked_as_kov = spec_evaluates_to(pt->constant);
}
```

§**11. Logging schemas.**

```
void log_i6_schema(i6_schema *sch) {
    if (sch == NULL) LOG("<null schema>");
    else LOG("<schema: %s>", sch->prototype);
}
void log_i6_schema_applied(i6_schema *sch, pcalc_term *pt1) {
    if (sch == NULL) { LOG("<null schema>"); return; }
    sch_expand(sch, dl, pt1, NULL);
}
```

The function log_i6_schema is called from 2/dl.
The function log_i6_schema_applied is called from 6/term.

Compile Atoms 6/atoms

Purpose

In this section, given an atom of a proposition we compile I6 code as required for any of three possible outcomes: (i) to test whether it is true, (ii) to make it henceforth true, or (iii) to make it henceforth false.

6/atoms.§2-4 Stage 1; §5 Stage 2; §6-14 Constructing the schema

---

Definitions

¶1. The compilation method is to look at the atom, work out a suitable I6 schema involving code to be applied to the one or two terms attaching to the atom, and then expand this. In some circumstances, the process of finding the schema will reveal that we need to apply it to different terms from those originally found in the atom, however, so we also need to keep track of that; and also of whether a condition is being regarded negatively.

```
typedef struct annotated_i6_schema {
    struct i6_schema *schema;
    int negate_schema;                    true if atom is to be tested with the opposite parity
    struct pcalc_term pt0;                terms on which the I6 schema is to be expanded
    struct pcalc_term pt1;
    int involves_action_variables;
} annotated_i6_schema;
```

The structure annotated_i6_schema is shared with 5/bp, 6/equal, 9/provr and 9/cmpbp.

¶2. Some constants to enumerate the three cases of what we are to do. This looks asymmetrical – shouldn't we also test to see whether an atom is false, a fourth case?

The answer is that there's no need, since "test false" can be done by compiling "test true" and applying the I7 negation operator ~~ to the result. No similar trick can be used to combine *making* something true or false into a single operation.

```
define TEST_ATOM_TASK 1
define NOW_ATOM_TRUE_TASK 2
define NOW_ATOM_FALSE_TASK 3
```

---

§**1.** So, then:

```
void atom_compile(OUTPUT_STREAM, int task, pcalc_prop *pl) {
    i6_schema sch;
    annotated_i6_schema asch;
    switch (task) {
        case TEST_ATOM_TASK: LOGIF(WORKINGS, "Compiling condition: $o\n", pl); break;
        case NOW_ATOM_TRUE_TASK: LOGIF(WORKINGS, "Compiling 'now': $o\n", pl); break;
        case NOW_ATOM_FALSE_TASK: LOGIF(WORKINGS, "Compiling 'now' false: $o\n", pl); break;
        default: internal_error("unknown compile task");
    }
    ⟨Stage 1: make an annotated schema from the atom 2⟩;
    ⟨Stage 2: expand that schema to the output stream 5⟩;
}
```

The function atom_compile is called from 6/defer and 6/cdefp.

§2. Stage 1.

⟨Stage 1: make an annotated schema from the atom 2⟩ ≡
```
int problem_count_before = problem_count;
asch = i6_schema_of_atom(&sch, pl, task);
if (asch.schema == NULL) {
    if (problem_count == problem_count_before)
        ⟨Issue a fallback problem message, since the schema-maker evidently didn't 3⟩;
    return;
}
⟨Reject all discussion of the action variables in the past tense 4⟩;
```

This code is used in §1.

§3.

⟨Issue a fallback problem message, since the schema-maker evidently didn't 3⟩ ≡
```
if (task == TEST_ATOM_TASK)
    sentence_problem(_P_(BelievedImpossible),
        "this is not a condition I am able to test",
        "or at any rate not during play.");
else
    sentence_problem(_P_(C6CantForceRelation),
        "this is not something I can make true with 'now'",
        "because it is too vague about the underlying cause which would "
        "need to be arranged.");
```

This code is used in §2.

§4. This is in the user's own best interest.

⟨Reject all discussion of the action variables in the past tense 4⟩ ≡
```
if ((asch.involves_action_variables) && (phsf_used_for_past_tense())) {
    if (problem_count == problem_count_before)
        sentence_problem(_P_(C6ActionVarsPastTense),
            "it is misleading to talk about the noun, the second noun "
            "or the person asked to do something in past tenses",
            "because in the past, those were different things and "
            "people, or may have been nothing at all. Writing "
            "'if the noun has been unlocked' tends not to do what we "
            "might hope because the value of 'noun' changes every turn. "
            "So such conditions are not allowed, although to get around "
            "this we can instead write 'if we have unlocked the noun', "
            "which uses a special mechanism to remember everything which "
            "has happened to every object.");
    return;
}
```

This code is used in §2.

§**5. Stage 2.** A valid I6 condition has to be bracketed, so we surround the output with brackets if testing; and a valid I6 statement has to end with a semicolon, so we terminate with that if making true or false.

⟨Stage 2: expand that schema to the output stream 5⟩ ≡

```
if (task == TEST_ATOM_TASK) {
    WRITE("(");
    if (asch.negate_schema) WRITE("~~(");
}
sch_expand(asch.schema, OUT, &(asch.pt0), &(asch.pt1));
if (task == TEST_ATOM_TASK) {
    if (asch.negate_schema) WRITE(")");
    WRITE(")");
} else {
    WRITE(";");
}
```

This code is used in §1.

§6. Constructing the schema.

```
annotated_i6_schema i6_schema_of_atom(i6_schema *sch, pcalc_prop *pl, int task) {
    annotated_i6_schema asch;
    sch_write_to_existing(sch, " ");                    a non-NULL return in case problems occur
    asch.schema = sch;
    asch.negate_schema = FALSE;
    asch.pt0 = pl->terms[0]; asch.pt1 = pl->terms[1];
    asch.involves_action_variables = atom_involves_action_variables(pl);
    switch(pl->element) {
        case CALLED_ATOM: ⟨Make an annotated schema for a CALLED atom 7⟩;
        case KIND_ATOM: ⟨Make an annotated schema for a KIND atom 8⟩;
        case EVERYWHERE_ATOM: ⟨Make an annotated schema for an EVERYWHERE atom 9⟩;
        case HERE_ATOM: ⟨Make an annotated schema for a HERE atom 10⟩;
        case PREDICATE_ATOM:
            switch(pl->arity) {
                case 1: ⟨Make an annotated schema for a unary predicate 11⟩;
                case 2: ⟨Make an annotated schema for a binary predicate 12⟩;
            }
    }
    asch.schema = NULL;                                 signal that the atom cannot be compiled simply
    return asch;
}
```

§**7.** We are now able to look at the different types of atom one at a time.

CALLED atoms cannot be asserted, and to test them, we simply copy the value into the local variable of the given name. Note then that here the I6 `=` (set equal) operator is being used in a condition context: there's a good chance that the value set is non-zero (since all objects and enumerated values are non-zero), but it isn't necessarily so – in Inform it's legal to quantify over times and truth states, for instance, where 0 is a legal I6 value. So we use the comma operator to throw away the result of the assignment, and evaluate the condition to `true`.

⟨Make an annotated schema for a CALLED atom 7⟩ ≡

```
    switch(task) {
        case TEST_ATOM_TASK: {
            int w1, w2;
            atom_CALLED_get_name(pl, &w1, &w2);
            sch_write_to_existing_1d(sch, "t_%d=(*1), true",
                ensure_called_local(w1, w2, pl->assert_kind_of_value));
            return asch;
        }
        default: asch.schema = NULL; return asch;
    }
```

This code is used in §6.

§**8.** In any type-checked proposition, a `KIND` atom can only exist where it is always at least sometimes true. In particular, if K is a kind of value, then the atom $K(v)$ can only exist where v is of that kind of value, so that the atom is always true when tested. But if K is a kind of object, then $K(O)$ may occur in the proposition for any object O, where O need not belong to K at all: so there is something substantive to check, which we do using the I6 `ofclass` operator.

⟨Make an annotated schema for a KIND atom 8⟩ ≡

```
switch(task) {
    case TEST_ATOM_TASK: {
        world_object *ko = kovko_get_kind(pl->assert_kind_of_value);
        if (ko) sch_write_to_existing_1(sch, "*1 ofclass %s", wo_get_I6_representation(ko));
        else sch_write_to_existing(sch, "true");
        return asch;
    }
    case NOW_ATOM_TRUE_TASK:
    case NOW_ATOM_FALSE_TASK:
        sentence_problem(_P_(C6CantChangeKind),
            "the kind of something is fixed",
            "and cannot be changed during play with a 'now'.");
        asch.schema = NULL; return asch;
}
```

This code is used in §6.

§**9.** An easy case. Note that `FoundEverywhere` is a template routine existing to provide a common value of the I6 `found_in` property – common that is to all backdrops which are currently everywhere.

⟨Make an annotated schema for an EVERYWHERE atom 9⟩ ≡

```
switch(task) {
    case TEST_ATOM_TASK:
        sch_write_to_existing(sch, "*1.found_in == FoundEverywhere");
        return asch;
    case NOW_ATOM_TRUE_TASK:
    case NOW_ATOM_FALSE_TASK:
        sentence_problem(_P_(C6CantChangeEverywhere),
            "being 'everywhere' is not something which can be changed during play",
            "so it cannot be brought about or cancelled out with 'now'.");
        asch.schema = NULL; return asch;
}
```

This code is used in §6.

§**10.** And another. (In fact, at present HERE atoms are never included in propositions to be compiled, so this code is never used.)

⟨Make an annotated schema for a HERE atom 10⟩ ≡

```
switch(task) {
    case TEST_ATOM_TASK:
        sch_write_to_existing(sch, "LocationOf(*1) == location");
        return asch;
    case NOW_ATOM_TRUE_TASK:
    case NOW_ATOM_FALSE_TASK:
        sentence_problem(_P_(BelievedImpossible),
            "being 'here' is not something which can be changed during play",
            "so it cannot be brought about or cancelled out with 'now'.");
        asch.schema = NULL; return asch;
}
```

This code is used in §6.

§**11.** The last unary atom is an adjective, for which we hand over to the general adjective apparatus.

⟨Make an annotated schema for a unary predicate 11⟩ ≡

```
int atask = 0;                                   redundant assignment to appease gcc -O2
adjective_list_entry *tr = RETRIEVE_POINTER_adjective_list_entry(pl->predicate);
adjectival_phrase *aph = get_adjective_from_list_entry(tr);
if (adjective_used_positively(tr) == FALSE) asch.negate_schema = TRUE;
if ((pl->terms[0].constant) && (pl->terms[0].term_checked_as_kov == NULL))
    pl->terms[0].term_checked_as_kov = spec_evaluates_to(pl->terms[0].constant);
switch(task) {
    case TEST_ATOM_TASK: atask = TEST_ADJECTIVE_TASK; break;
    case NOW_ATOM_TRUE_TASK: atask = NOW_ADJECTIVE_TRUE_TASK; break;
    case NOW_ATOM_FALSE_TASK: atask = NOW_ADJECTIVE_FALSE_TASK; break;
}
asch.schema = aph_get_i6_schema(aph, pl->terms[0].term_checked_as_kov, atask);
return asch;
```

This code is used in §6.

§**12.** Delegation is similarly the art of compiling a BP:

⟨Make an annotated schema for a binary predicate 12⟩ ≡

```
binary_predicate *bp = RETRIEVE_POINTER_binary_predicate(pl->predicate);
binary_predicate *bp_to_assert = NULL;
⟨Undo any functional simplification of the relation 13⟩;
asch.schema = bp_get_i6_schema(task, bp_to_assert, &asch);
return asch;
```

This code is used in §6.

§**13.** When a relation $R(x, y)$ has been simplified to $is(x, f_R(y))$ or $is(g_R(x), y)$, it can be tested but not asserted true or false; we have to re-establish $R(x, y)$ before we can proceed.

⟨Undo any functional simplification of the relation 13⟩ ≡

```
    if ((task != TEST_ATOM_TASK) && (bp == a_is_b_predicate)) {
        if (pl->terms[0].function) {
            bp_to_assert = pl->terms[0].function->bp;
            asch.pt0 = pl->terms[0].function->fn_of;
        } else if (pl->terms[1].function) {
            bp_to_assert = pl->terms[1].function->bp;
            asch.pt1 = pl->terms[1].function->fn_of;
        }
        if (bp_to_assert == a_is_b_predicate)
            internal_error("contraction of predicate applied to equality");
    }
    if (bp_to_assert == NULL) bp_to_assert = bp;
```

This code is used in §12.

§**14.**

```
int atom_involves_action_variables(pcalc_prop *pl) {
    int i;
    for (i=0; i<pl->arity; i++) {
        specification *operand = term_constant_underlying(&(pl->terms[i]));
        if (qty_is_an_I6_library_variable(operand)) return TRUE;
    }
    return FALSE;
}
```

Deciding to Defer 6/defer

Purpose

To decide whether a proposition can be compiled immediately, in the body of the current routine, or whether it must be deferred to a routine of its own, which is called from the current routine.

Template interpreter commands

```
1   {-callv:allow_no_further_deferrals}
```

Definitions

¶1. So far in this chapter, we have written code which suggests that there are basically only three ways to compile a proposition: as a test ("if the tree is blossoming"), as code forcing it to be true ("now the tree is blossoming") or as code forcing it to be false ("now the tree is not blossoming"). That's not quite true, and so deferral can happen for a number of different reasons, enumerated as follows:

```
define CONDITION_DEFER 1                          "if S", where S is a sentence
define NOW_ASSERTION_DEFER 2                      "now S"
define EXTREMAL_DEFER 3                           "the heaviest X", where X is a description
define LOOP_DOMAIN_DEFER 4                        "repeat with I running through X"
define NUMBER_OF_DEFER 5                          "the number of X"
define TOTAL_DEFER 6                              "the total P of X"
define RANDOM_OF_DEFER 7                          "a random X"
define MULTIPURPOSE_DEFER 100                     potentially any of the above

typedef struct pcalc_prop_deferral {
    int reason;                                   what we intend to do with it: one of the reasons above
    struct pcalc_prop *proposition_to_defer;      the proposition
    struct parse_node *deferred_from;             remember where it came from, for Problem reports
    struct general_pointer defn_ref;              sometimes we must remember other things too
    struct kind_of_value *cinder_kovs[16];        the kinds of value being cindered (see below)
    MEMORY_MANAGEMENT
} pcalc_prop_deferral;
```

The structure pcalc_prop_deferral is shared with 6/cind and 6/cdefp.

---

§**1. The guillotine.** We must be careful not to request a fresh deferral after the point at which all deferral requests are redeemed – they would then never be reached.

```
int no_further_deferrals = FALSE;
void allow_no_further_deferrals(void) {
    no_further_deferrals = TRUE;
}
```

The function allow_no_further_deferrals is invoked by a command in a `.i6t` template file.

§**2. Deferral requests.** The following fills out the paperwork to request a deferred proposition.

```
pcalc_prop_deferral *new_deferred_proposition(pcalc_prop *prop, int reason) {
    pcalc_prop_deferral *pdef = CREATE(pcalc_prop_deferral);
    pdef->proposition_to_defer = prop;
    pdef->reason = reason;
    pdef->deferred_from = current_sentence;
    if (no_further_deferrals) internal_error("Too late now to defer propositions");
    return pdef;
}
```

§**3.** It's worth cacheing deferral requests in the case of loop domains, because they are typically needed in the case of repeat-through loops where the same proposition is used three times in a row.

```
pcalc_prop *cache_loop_proposition = NULL;
pcalc_prop_deferral *cache_loop_pdef = NULL;
pcalc_prop_deferral *defer_loop_domain(pcalc_prop *prop) {
    pcalc_prop_deferral *pdef;
    if (prop == cache_loop_proposition) return cache_loop_pdef;
    pdef = new_deferred_proposition(prop, LOOP_DOMAIN_DEFER);
    cache_loop_proposition = prop;
    cache_loop_pdef = pdef;
    return pdef;
}
```

§**4.** The following shorthand routine takes a description SP, converts it to a proposition $\phi(x)$, then defers this and returns the number `n` such that the resulting routine will be called `Prop_n`.

```
int compile_deferred_description_test(specification *spec) {
    pcalc_prop_deferral *pdef;
    spec_convert_docket_to_proposition(spec);
    pdef = new_deferred_proposition(spec_get_proposition(spec), CONDITION_DEFER);
    return pdef->allocation_id;
}
```

The function compile_deferred_description_test is called from 11/av.

§**5. Testing, or deferring a test.** Given a proposition ϕ, and a value v, we compile a valid I6 condition to decide whether or not $\phi(v)$ is true. ϕ can either be a sentence with all variables bound, in which case v must be null, or can have just variable x free, in which case v must not be null.

We defer the proposition to a routine of its own if and only if it contains quantification.

```
void compile_test_of_proposition(OUTPUT_STREAM,
    specification *substitution, pcalc_prop *prop) {
    LOGIF(WORKINGS, "Compiling as test: $D\n", prop);
    WRITE("(");
    if (prop_contains_quantifier(prop)) {
        ⟨Defer test of proposition instead 6⟩;
    } else {
        if (substitution) substitute_var_0_in(prop, substitution);
        TRAVERSE_VARIABLE(pl);
        TRAVERSE_PROPOSITION(pl, prop) {
            if (implied_conjunction_between(pl_prev, pl)) WRITE(" && ");
            switch(pl->element) {
                case NEGATION_OPEN_ATOM: WRITE("~~("); break;
                case NEGATION_CLOSE_ATOM:
                    if (pl_prev->element == NEGATION_OPEN_ATOM) WRITE("true");
                    WRITE(")"); break;
                default: atom_compile(OUT, TEST_ATOM_TASK, pl); break;
            }
        }
    }
    WRITE(")");
}
```

The function compile_test_of_proposition is called from 7/cosp, 11/ap, 12/cinv and 13/nft.

§**6.** Since this is the first of our deferrals, let's take it slowly. We are compiling code in some outer routine – let's call it `R` – and the idea is to compile a function call `Prop_19(...)` into `R` where the test should be; this function will return either `true` or `false`, and its job is to test the proposition for us. (Deferred propositions are numbered in order of deferral; for the sake of example, we'll suppose ours in number 19.)

⟨Defer test of proposition instead 6⟩ ≡

```
    pcalc_prop_deferral *pdef;
    ⟨If the proposition is a negation, take care of that now 7⟩;
    pdef = new_deferred_proposition(prop, CONDITION_DEFER);
    ⟨Compile the call to the test-proposition routine 8⟩;
    ⟨Compile code to retrieve the values of any callings 9⟩;
```

This code is used in §5.

§**7.** This is done purely for the sake of compiling tidier code: if $\phi = \neg(\psi)$ then we defer ψ instead, negating the result of testing it.

⟨If the proposition is a negation, take care of that now 7⟩ ≡

```
    if (prop_is_a_group(prop, NEGATION_OPEN_ATOM)) {
        prop = prop_remove_topmost_group(prop);
        WRITE("~~");
    }
```

This code is used in §6.

§**8.** All of the subtlety here is to do with the fact that `R` and `Prop_19` have access to different values – in particular, they have different sets of local variables.

Because code in `Prop_19` cannot see the local variables of `R`, any such values needed must be passed from `R` to `Prop_19` as call parameters. These passed values are called "cinders".

In addition, `R` might not be able to evaluate the substitution value v for itself, so that must also be a call parameter. It will then become the initial value of the local variable `x` in `Prop_19`, and since x is free in the proposition, `x` never changes in `Prop_19`: thus we effect a substitution of $x = v$.

For example, `R` might contain the function call:

```
Prop_19(t_6, t_2, O13_sphinx)
```

and the function header of `Prop_19` might then look like so:

```
[ Prop_19 const_0 const_1 x;
```

The value of `cinder_count` would then be 2.

⟨Compile the call to the test-proposition routine 8⟩ ≡

```
WRITE("Prop_%d(", pdef->allocation_id);
int cinder_count = cind_find(OUT, prop, pdef);
if (substitution) {
    if (cinder_count > 0) WRITE(",");
    spec_compile(OUT, substitution);
}
WRITE(")");
```

This code is used in §6.

§**9.** When we defer a test, we make "called" more tricky to achieve. Suppose we are compiling a condition for

> if a woman (called the moll) has a weapon (called the gun), ...

This needs to set two local variables, "moll" and "gun", but those have to be locals in `R` – they are inaccessible to `Prop_19`. Somehow, they need to be return values, but I6 supports only a single return value from a routine, and that needs to be either `true` or `false`. What to do?

The answer is that `Prop_19` copies these values onto a special I6 array called the "deferred calling list". The very last thing that `Prop_19` does before it returns is to fill in this list; the very first thing we do on receiving that return is to extract what we want from it. (Because no other activity takes place in between, there is no risk that some recursive use of propositions will overwrite the list.)

For example, `R` might this time contain a call like so:

```
(Prop_19() && (t_2=deferred_calling_list-->0, t_3=deferred_calling_list-->1, true))
```

which safely transfers the values to locals `t_2` and `t_3` of `R`. Note that I6 evaluates conditions joined by `&&` from left to right, so we can be certain that `Prop_19` has been called and has returned before we get to the setting of `t_2` and `t_3`.

⟨Compile code to retrieve the values of any callings 9⟩ ≡

```
int calling_count=0;
TRAVERSE_VARIABLE(pl);
TRAVERSE_PROPOSITION(pl, prop) {
    switch(pl->element) {
        case CALLED_ATOM: {
            int local;
            ⟨Find which local variable in R needs the value, creating it if necessary 10⟩;
            if (calling_count == 0) WRITE(" && (");
            if (calling_count > 0) WRITE(", ");
```

```
                WRITE("t_%d=deferred_calling_list-->%d",
                    local, calling_count++);
                break;
            }
        }
    }
    if (calling_count > 0) WRITE(", true)");
```

This code is used in §6.

§**10.** `ensure_called_local` is so called because it ensures that a local of the right name and kind of value exists in `R`.

⟨Find which local variable in R needs the value, creating it if necessary 10⟩ ≡

```
    int w1, w2;
    atom_CALLED_get_name(pl, &w1, &w2);
    local = ensure_called_local(w1, w2, pl->assert_kind_of_value);
```

This code is used in §9.

§**11.** The following wrapper contributes almost nothing, but it checks some consistency assertions and writes to the debugging log.

```
void compile_test_if_var_matches_description(OUTPUT_STREAM,
    specification *var, specification *matches) {
    if (matches == NULL) internal_error("VMD against null description");
    if (var == NULL) internal_error("VMD on null variable");
    if ((spec_get_storage_form(var) != NONLOCAL_VARIABLE_SPC) &&
        (spec_get_storage_form(var) != LOCAL_VARIABLE_SPC))
        internal_error("VMD on non-variable");
    STREAM_INDENT(dl);
    STREAM_INDENT(dl);
    LOGIF(DESCRIPTION_COMPILATION, "[VMD: $S matches $S?]\n", var, matches);
    compile_test_of_proposition(OUT, var, prop_from_spec(matches, FALSE));
    STREAM_OUTDENT(dl);
    STREAM_OUTDENT(dl);
}
```

The function compile_test_if_var_matches_description is called from 12/cinv and 13/nft.

§**12. Forcing with now, or deferring the force.** For example, compiling code to achieve something like:

now the Marble Door is closed;

(which does not need to be deferred) or

now all the women are in the Dining Room;

(which does, since it contains a quantifier).

This is simpler than testing, because a "now" does not have callings, and because it always acts on whole sentences – no substitution is ever needed; the only call parameters for our `Prop_19` are the cinders, if any; and we never need to extract from the `deferred_calling_list`.

Once again the question arises of how to force $\neg(\phi \wedge \psi)$ to be true. It would be sufficient to falsify either one of ϕ or ψ alone, but for the sake of symmetry we falsify both. (We took the same decision when asserting propositions about the initial state of the model.)

```
void compile_now_proposition(OUTPUT_STREAM, pcalc_prop *prop) {
    int quantifier_count = 0;
    LOGIF(WORKINGS, "Compiling as 'now': $D\n", prop);
    ⟨Count quantifiers in, and generally vet, the proposition to be forced 13⟩;
    if (quantifier_count > 0) {
        pcalc_prop_deferral *pdef = new_deferred_proposition(prop, NOW_ASSERTION_DEFER);
        WRITE("Prop_%d(", pdef->allocation_id);
        cind_find(OUT, prop, pdef);
        WRITE(");");
    } else {
        int parity = TRUE;
        TRAVERSE_VARIABLE(pl);
        TRAVERSE_PROPOSITION(pl, prop) {
            switch (pl->element) {
                case NEGATION_OPEN_ATOM: case NEGATION_CLOSE_ATOM:
                    parity = (parity)?FALSE:TRUE;
                    break;
                default:
                    atom_compile(OUT, (parity)?NOW_ATOM_TRUE_TASK:NOW_ATOM_FALSE_TASK, pl);
                    break;
            }
        }
    }
}
```

The function compile_now_proposition is called from 7/cosp.

§**13.** We reject multiple quantifiers as too much work, and $\exists x$ because it would either require us to judge the x most likely to be meant – tricky – or to create an x out of nothing, which it's too late for, since Inform does not have run-time object or value creation.

⟨Count quantifiers in, and generally vet, the proposition to be forced 13⟩ ≡

```
TRAVERSE_VARIABLE(pl);
TRAVERSE_PROPOSITION(pl, prop) {
    switch(pl->element) {
        case QUANTIFIER_ATOM:
            if (atom_is_existence_quantifier(pl)) {
                sentence_problem(_P_(C6CantForceExistence),
                    "this is not explicit enough",
                    "and should set out definite relationships between specific "
                    "things, like 'now the cat is in the bag', not something "
                    "more elusive like 'now the cat is carried by a woman.' "
                    "(Which woman? That's the trouble.)");
                return;
            }
            if (atom_is_now_assertable_quantifier(pl) == FALSE) {
                sentence_problem(_P_(C6CantForceGeneralised),
                    "this can't be made true with 'now'",
                    "because it is too vague about what it applies to. It's fine "
                    "to say 'now all the doors are open' or 'now none of the doors "
                    "is open', because that clearly tells me which doors are "
                    "affected; but if you write 'now six of the doors are open' "
                    "or 'now almost all the doors are open', what am I to do?");
                return;
            }
            quantifier_count++;
            break;
        case CALLED_ATOM:
            sentence_problem(_P_(C6CantForceCalling),
                "a 'now' is not allowed to call names",
                "and it wouldn't really make sense to do so anyway. 'if "
                "a person (called the victim) is in the Trap Room' makes "
                "sense, because it gives a name - 'victim' - to someone "
                "whose identity we don't know. But 'now a person (called "
                "the victim) is in the Trap Room' won't be allowed, "
                "because 'now' can only talk about people or things whose "
                "identities we do know.");
            return;
    }
}
```

This code is used in §12.

§**14. Multipurpose descriptions.** Descriptions in the form $\phi(x)$, where x is free, are also sometimes converted into values – this is the kind of value "description". The I6 representation is (the address of) a routine `D` which, in general, performs task u on value v when called as `D(u, v)`, where u is expected to be one of the following values. (Note that v is only needed in the first two cases.)

These numbers *must* be negative, since they need to be different from every valid member of a quantifiable domain (objects, enumerated KOVs, truth states, times of day, and so on).

define `CONDITION_DUSAGE -1`	*return* `true` *iff* $\phi(v)$
define `LOOP_DOMAIN_DUSAGE -2`	*return the next* x *after* v *such that* $\phi(x)$
define `NUMBER_OF_DUSAGE -3`	*return the number of* w *such that* $\phi(w)$
define `RANDOM_OF_DUSAGE -4`	*return a random* w *such that* $\phi(w)$*, or 0 if none exists*
define `TOTAL_DUSAGE -5`	*return the total value of a property among* w *such that* $\phi(w)$
define `EXTREMAL_DUSAGE -6`	*return the maximal property value among such* w

§**15.** Multi-purpose description routines are pretty dandy, then, but they have one big drawback: they can't be passed cinders, because they might be called from absolutely anywhere. Hence the following:

```
void compile_multiple_use_proposition(OUTPUT_STREAM, specification *spec) {
    pcalc_prop *prop = prop_from_spec(spec, FALSE);
    if (prop_detect_locals(prop) > 0) {
        LOG("Offending proposition: $D\n", prop);
        sentence_problem(_P_(C6LocalInDescription),
            "descriptions used as values are not allowed to contain "
            "references to temporary variables defined by 'let' or by loops",
            "because they may very well not exist any more when the description "
            "needs to be used, in another time and another place.");
    } else {
        pcalc_prop_deferral *pdef = new_deferred_proposition(prop, MULTIPURPOSE_DEFER);
        WRITE("Prop_%d", pdef->allocation_id);
    }
}
```

The function compile_multiple_use_proposition is called from 7/vasp.

§**16.** Because multipurpose descriptions have this big drawback, we want to avoid them if we possibly can. Fortunately something much simpler will often do. For example, consider:

[1] the number of members of S

[2] the number of closed doors

where S, in [1], is a description which appears as a parameter in a phrase. In [1] we have no way of knowing what S might be, but we can safely assume that it has been compiled as a multi-purpose description routine, and therefore compile the function call:

```
D(NUMBER_OF_DUSAGE)
```

But in case [2] it is sufficient to take $\phi(x) = door(x) \wedge closed(x)$, defer it to a proposition with reason `NUMBER_OF_DEFER`, and then compile just

```
Prop_19()
```

to perform the calculation. We never need a multi-purpose description routine for $\phi(x)$ because it only occurs in this one context.

§**17.** We now perform this trick for "number of":

```
void compile_number_of_S(OUTPUT_STREAM, specification *spec) {
    if (spec_is_variable_of_KOV_description(spec)) {
        WRITE("(");
        spec_compile(OUT, spec);
        WRITE(")(%d)", NUMBER_OF_DUSAGE);
    } else {
        pcalc_prop *prop = prop_from_spec(spec, FALSE);
        prop_verify_descriptive(prop, "a number of things matching a description", spec);
        compile_call_to_deferred_desc(OUT, prop, NUMBER_OF_DEFER, NULL_GENERAL_POINTER);
    }
}
```

The function compile_number_of_S is called from 12/cinv.

§**18.** Where we employ:

```
int spec_is_variable_of_KOV_description(specification *spec) {
    if ((family_is(spec, STORAGE_FMY)) &&
        ((is_kova(spec_evaluates_to(spec), OBJECT_DESCRIPTION_TY)) ||
            (is_kova(spec_evaluates_to(spec), VALUE_DESCRIPTION_TY))))
        return TRUE;
    return FALSE;
}
void compile_call_to_deferred_desc(OUTPUT_STREAM, pcalc_prop *prop,
    int reason, general_pointer data) {
    pcalc_prop_deferral *pdef = new_deferred_proposition(prop, reason);
    pdef->defn_ref = data;
    WRITE("(Prop_%d(", pdef->allocation_id);
    cind_find(OUT, prop, pdef);
    WRITE("))");
}
```

§**19.** The pattern is repeated for "a random ...":

```
void compile_random_of_S(OUTPUT_STREAM, specification *spec) {
    if (spec_is_variable_of_KOV_description(spec)) {
        WRITE("(");
        spec_compile(OUT, spec);
        WRITE(")(%d)", RANDOM_OF_DUSAGE);
    } else {
        pcalc_prop *prop = prop_from_spec(spec, FALSE);
        prop_verify_descriptive(prop, "a random thing matching a description", spec);
        kind_of_value *kov = prop_describes_KOV(prop);
        if ((kov) && (kov_compile_domain_possible(kov) == FALSE))
            sentence_problem(_P_(C6RandomImpossible),
                "this asks to find a random choice from a range which is too "
                "large or impractical",
                "so can't be done. For instance, 'a random person' is fine - "
                "it's clear exactly who all the people are, and the supply is "
                "limited - but not 'a random text'.");
        else
```

```
            compile_call_to_deferred_desc(OUT, prop, RANDOM_OF_DEFER, NULL_GENERAL_POINTER);
    }
}
```

The function compile_random_of_S is called from 12/cinv.

§**20.** And similarly for "total of":

```
void compile_total_of_S(OUTPUT_STREAM, property_name *prn, specification *spec) {
    if (prn == NULL) internal_error("total of on non-property");
    if (spec_is_variable_of_KOV_description(spec)) {
        WRITE("(property_to_be_totalled=%s,(", prn_get_i6_identifier(prn));
        spec_compile(OUT, spec);
        WRITE(")(%d))", TOTAL_DUSAGE);
    } else {
        pcalc_prop *prop = prop_from_spec(spec, FALSE);
        prop_verify_descriptive(prop,
            "a total property value for things matching a description", spec);
        compile_call_to_deferred_desc(OUT, prop, TOTAL_DEFER,
            STORE_POINTER_property_name(prn));
    }
}
```

The function compile_total_of_S is called from 12/cinv.

§**21.** Also for the occasionally useful task of seeing if the current value of the I6 variable `subst__v` (the "substitution variable") is within the domain of $\phi(x)$.

```
void compile_test_of_subst__v(OUTPUT_STREAM, specification *spec) {
    if (spec_is_variable_of_KOV_description(spec)) {
        WRITE("(");
        spec_compile(OUT, spec);
        WRITE(")(%d,subst__v)", CONDITION_DUSAGE);
    } else {
        compile_test_of_proposition(OUT,
            new_QUANTITY_spec(subst_variable_quantity),
            prop_from_spec(spec, FALSE));
    }
}
```

The function compile_test_of_subst__v is called from 7/cosp and 12/cinv.

§**22.** And the extremal case is pretty well the same, too, with only some fuss over identifying which superlative is meant. We get here from code like

```
let X be the heaviest thing in the wooden box;
```

where there has previously been a definition of "heavy".

```
void compile_extremal_of_S(OUTPUT_STREAM, specification *spec,
    property_name *prn, int sign) {
    if (prn == NULL) internal_error("extremal of on non-property");
    if (spec_is_variable_of_KOV_description(spec)) {
        WRITE("(property_to_be_totalled=%s,property_loop_sign=%d,(",
            prn_get_i6_identifier(prn), sign);
        spec_compile(OUT, spec);
        WRITE(")(%d))", EXTREMAL_DUSAGE);
    } else {
        measurement_definition *mdef_found = NULL, *mdef;
        LOOP_OVER(mdef, measurement_definition) {
            property_name *def_prn;
            int def_prn_sign;
            validate_mdef(mdef);
            mdef_read_property_details(mdef, &def_prn, &def_prn_sign);
            if ((def_prn == prn) && (def_prn_sign == sign)) {
                LOGIF(WORKINGS, "Def: prop $Y, sign %d\n", def_prn, def_prn_sign);
                mdef_found = mdef; break;
            }
        }
        if (mdef_found) {
            pcalc_prop *prop = prop_from_spec(spec, FALSE);
            prop_verify_descriptive(prop,
                "an extreme case of something matching a description", spec);
            validate_mdef(mdef);
            compile_call_to_deferred_desc(OUT, prop, EXTREMAL_DEFER,
                STORE_POINTER_measurement_definition(mdef_found));
        } else internal_error("Unable to match extremal to definition");
    }
}
```

The function compile_extremal_of_S is called from 12/cinv.

§23. Domains of loops. Here we define an I6 `for` loop header to handle a repeat loop through all of the x matching a given description $\phi(x)$.

We are allowed to use two local variables in the current stack frame: `t_v1` and `t_v2`, where the numbers v_1 and v_2 are supplied to us. We mark them as available for reuse once the loop has been exited, by setting their scope to the code block for the loop.

We use v_1 as the current value and v_2 as the one which will follow it. Always evaluating one step ahead protects us in case the body of the loop takes action which moves v_1 out of the domain – e.g., in the case of

```
repeat with T running through items on the table: now T is in the box.
```

This is the famous "broken `objectloop`" hazard of Inform 6, which typically occurs because the mechanism to move from one value to the next uses `sibling` in the I6 object tree, and that relies on v_1 being an object which is still in the same location at the end of the loop as at the beginning.

Thus a typical loop header has the form

```
for (t_1=D(0), t_2=D(t_1): t_1: t_1=t_2, t_2=D(t_1))
```

where `D` is a routine such that at 0 it produces the first element of the domain, and then given `x` in the domain, `D(x)` produces the next element until it returns 0, when the domain is exhausted.

```
void compile_repeat_over_S(OUTPUT_STREAM, specification *spec, int v1, int v2) {
    pcalc_prop *domain_prop = NULL; int use_as_is = FALSE;
    set_scope_of_local_to_current_block(v1);
    set_scope_of_local_to_current_block(v2);
    if (spec_is_variable_of_KOV_description(spec)) use_as_is = TRUE;
    else {
        domain_prop = prop_from_spec(spec, FALSE);
        if (prop_contains_callings(domain_prop))
            sentence_problem(_P_(C6CalledInRepeat),
                "this tries to use '(called ...)' to give names to values "
                "arising in the course of working out what to repeat through",
                "but this is not allowed. (Sorry: it's too hard to get right.)");
    }
    WRITE("for (t_%d=", v1);
    if (use_as_is) compile_repeat_call(OUT, spec, -1);
    else compile_repeat_domain(OUT, domain_prop, -1);
    WRITE(", t_%d=", v2);
    if (use_as_is) compile_repeat_call(OUT, spec, v1);
    else compile_repeat_domain(OUT, domain_prop, v1);
    WRITE(": t_%d: t_%d=t_%d, t_%d=", v1, v1, v2, v2);
    if (use_as_is) compile_repeat_call(OUT, spec, v2);
    else compile_repeat_domain(OUT, domain_prop, v2);
    WRITE(")");
}
```

The function compile_repeat_over_S is called from 12/cinv.

§**24.** If the description D is not explicitly known – because it sits inside a variable – then the following compiles code to call `D` in order to calculate the next value in the domain after the one stored in `fromv`.

Here, once again, we know that D has been compiled to a general-purpose deferred description routine, and we simply call that routine with the `LOOP_DOMAIN_DUSAGE` task.

```
void compile_repeat_call(OUTPUT_STREAM, specification *spec, int fromv) {
    WRITE("(");
    spec_compile(OUT, spec);
    WRITE(")(%d,", LOOP_DOMAIN_DUSAGE);
    if (fromv >= 0) WRITE("t_%d", fromv); else WRITE("0");
    WRITE(")");
}
```

§**25.** But if the description $D = \phi(x)$ is an explicitly known proposition, then we defer it to a routine specifically tailored to loop domains – it will never be needed for anything else.

```
void compile_repeat_domain(OUTPUT_STREAM, pcalc_prop *prop, int fromv) {
    pcalc_prop_deferral *pdef = defer_loop_domain(prop);
    WRITE("Prop_%d(", pdef->allocation_id);
    int nc = cind_find(OUT, prop, pdef);
    if (nc > 0) WRITE(",");
    if (fromv >= 0) WRITE("t_%d", fromv); else WRITE("0");
    WRITE(")");
}
```

§**26. Checking the validity of a description.** The following utility routine checks that a proposition contains exactly one unbound variable, producing problem messages if not, and that it passes type-checking successfully.

```
void prop_verify_descriptive(pcalc_prop *prop, char *billing,
    specification *constructor) {
    if (constructor == NULL) internal_error("description with null constructor");
    best guess at the text to quote in any problem message
    int ew1 = constructor->word_ref1, ew2 = constructor->word_ref2;
    if ((ew1 < 0) && (spec_get_argc(constructor) >= 1) && (spec_get_argument(constructor, 0))) {
        ew1 = spec_get_argument(constructor, 0)->word_ref1;
        ew2 = spec_get_argument(constructor, 0)->word_ref2;
    }
    if (prop_is_well_formed(prop) == FALSE)
        internal_error("malformed proposition in description verification");
    int N = vars_number_free(prop);
    if (N == 1)
        prop_type_check(prop,
            tc_problem_reporting(ew1, ew2,
            "involve a range of objects matching a description"));
    if (N > 1) {
        quote_source(1, current_sentence);
        quote_text(2, billing);
        quote_words(3, ew1, ew2);
        handmade_problem(_P_(BelievedImpossible));
```

```
        issue_problem_segment(
            "In %1, you are asking for %2, but this should range over a "
            "simpler description than '%3', please - it should not include any "
            "determiners such as 'at least three', 'all' or 'most'. "
            "(The range is always taken to be all of the things matching "
            "the description.)");
        issue_problem_end();
        return;
    }
    if (N < 1) {
        quote_source(1, current_sentence);
        quote_text(2, billing);
        quote_words(3, ew1, ew2);
        handmade_problem(_P_(BelievedImpossible));
        issue_problem_segment(
            "In %1, you are asking for %2, but '%3' looks as if it ranges "
            "over only a single specific object, not a whole collection of "
            "objects.");
        issue_problem_end();
    }
}
```

The function prop_verify_descriptive is called from 11/ap.

Cinders and Deferrals 6/cind

Purpose

To compile terms, having carefully preserved any constants which might have been lost in the process of deferring a proposition (such tricky constants being called "cinders").

---

§**1. About cinders.** Any proposition which includes quantification will, in general, need to be deferred: the quantifier will become a loop, and the loop variable will be a local variable in the deferred routine. Instead of compiling explicit code in the current routine `R`, we will compile a call to, say, `Prop_19`.

It is both good and bad that `Prop_19` has its own set of local variables. Good because this means it can have loop variables and counters to help it to implement quantifiers, without having to use up scarce locals in `R`. (Recall that I6 allows only 15 locals in any one routine.) But bad because the proposition may itself involve mention of local variables in `R`, which do not exist inside `Prop_19`.

There are two issues here. Suppose we have:

> if a dark room contains the painting

This compiles to code quantifying over rooms x, requiring a loop, so it will be deferred to a routine. "The painting" is, in predicate calculus terms, a constant – in that it does not depend on x – but that does not mean it's a value known at compile time. If it is a local variable in the current routine, then it won't exist in the deferred one, as mentioned above. But it might also be a phrase to decide something, which has side-effects, so that it might be important not to evaluate it more than once.

Any such constants are compiled into the call to `Prop_19` as parameters: then, when the latter is compiled, they become locals `const_0`, `const_1`, ..., whose evaluation will be rapid and without side-effects.

§**2.** As part of the deferral process, then, we scan through a proposition to look for constants which might cause trouble. These are called "cinders", which is a contraction of "constants in deferred routines".

Within any given proposition, the cinders are numbered 0, 1, 2, ...; these numbers are recorded in the `cinder` field of the relevant `pcalc_term` structure. A constant term with `cinder` set to -1 is harmless, and can be compiled in `Prop_19` as it stands: a literal number, for instance.

At any given moment, we can only be working on the compilation of a single deferred proposition routine. The following records the information noted down at the time when the proposition was deferred:

```
pcalc_prop_deferral *current_pdef = NULL;
```
used only in this section

§**3. Finding cinders.** In this operation, conducted when we defer a proposition, we look for constant terms which need to be cindered, do so, compile their values as a comma-separated list of I6 expressions (which can be used in a function call), note down their kinds of value in the record of the deferral, and return the number of cinders made.

```
int cind_find(OUTPUT_STREAM, pcalc_prop *prop, pcalc_prop_deferral *pdef) {
    TRAVERSE_VARIABLE(pl);
    int i, cinder_number = 0;
    current_pdef = pdef;
    TRAVERSE_PROPOSITION(pl, prop)
        for (i=0; i<pl->arity; i++)
            cinder_number =
                cind_find_in_term(OUT, &(pl->terms[i]), cinder_number);
    return cinder_number;
}
int cind_find_in_term(OUTPUT_STREAM, pcalc_term *pt, int cinder_number) {
    do not clear the local I6 stream
    if (pt->function)
        return cind_find_in_term(OUT, &(pt->function->fn_of), cinder_number);
    if (pt->constant) {
        if (spec_needs_to_be_cindered(pt->constant)) {
            pt->cinder = cinder_number++;
            if (STREAM_EXTENT(OUT) > 0) WRITE(",");
            spec_compile(OUT, pt->constant);
            current_pdef->cinder_kovs[pt->cinder] =
                spec_evaluates_to(pt->constant);
        } else pt->cinder = -1;
    }
    return cinder_number;
}
```

The function cind_find is called from 6/defer.

§**4.** Which leaves us to decide, given any specification, whether it represents a value needing to be cindered. If in doubt, we should cinder, but we don't want to go mad since there's a limit to how many cinders can be passed as parameters. Currently we do cinder these:

(a) phrases to decide values, cindered because they might be slow or have side-effects to evaluate;
(b) stacked non-local variables, such as variables attached to actions or activities, cindered because they are only allowed in certain routines, and the eventual deferred proposition routine might not qualify;
(c) local variables, cindered since they won't exist in the deferred routine;
(d) list and table entries, cindered since they are relatively slow to look up.

But we do not cinder these, since their evaluation never depends on context and never needs more than a single array entry lookup at run-time:

(a) constants from literal numbers to names of scenes, rulebooks, and so on;
(b) global variables.

```
int spec_needs_to_be_cindered(specification *spec) {
    if (species_is(spec, CONSTANT_SPC)) return FALSE;
    if (spec_is_global_variable(spec)) return FALSE;
    return TRUE;
}
```

§**5. Declaring cinder parameters.** Symmetrically, when we come to compiled our deferred proposition routine `Prop_19`, we need to place suitable cinder variables in its I6 header. We print their names, separated by spaces, since that's the somewhat assembler-like syntax used by I6 routine headers.

We also set `current_pdef`, since this is the first action taken in starting a new deferred routine.

```
void cind_declare(OUTPUT_STREAM, pcalc_prop *prop, pcalc_prop_deferral *pdef) {
    TRAVERSE_VARIABLE(pl);
    int i, cinder_number = 0;
    current_pdef = pdef;
    TRAVERSE_PROPOSITION(pl, prop)
        for (i=0; i<pl->arity; i++)
            cinder_number = cind_declare_in(OUT, cinder_number, &(pl->terms[i]));
}
int cind_declare_in(OUTPUT_STREAM, int cinder_number, pcalc_term *pt) {
    if (pt->function)
        return cind_declare_in(OUT, cinder_number, &(pt->function->fn_of));
    if ((pt->constant) && (pt->cinder >= 0))
        if (species_is(pt->constant, CONSTANT_SPC) == FALSE) {
            WRITE(" const_%d", cinder_number++);
        }
    return cinder_number;
}
```

The function cind_declare is called from 6/cdefp.

§**6. The KOV of terms.** We are now finally able to say what the kind of value of a term to be compiled is. The only troublesome case is when the term is a cinder; its KOV is then part of the information recorded at deferral time.

```
kind_of_value *cind_kind_of_value_of_term(pcalc_term pt) {
    if (pt.variable >= 0) return kova(OBJECT_TY);
    if (pt.constant) {
        if (pt.cinder >= 0) return current_pdef->cinder_kovs[pt.cinder];
        if (spec_is_phrasal(pt.constant)) typecheck_without_expectations(pt.constant);
        return spec_evaluates_to(pt.constant);
    }
    if (pt.function) return kova(OBJECT_TY);
    internal_error("Broken pcalc term");
    return NULL;
}
```

The function cind_kind_of_value_of_term is called from 6/equal, 9/provr and 9/cmpbp.

§**7. Compiling terms.** We are now ready to compile a general predicate-calculus term, which may be a constant (perhaps with a cinder marking), a variable or a function of another term.

Variables are compiled to I6 locals `x`, `y`, `z`, ...; cindered constants to `const_0`, `const_1`, ... These will only be valid inside a deferred routine like `Prop_19`, but that is fine because they cannot arise anywhere else. If we are compiling an undeferred proposition then all constants are uncindered and there are no variables (if there were, it would have been deferred).

Functions $f_R(t)$ are compiled by expanding an I6 schema for f_R with t as parameter.

One small wrinkle is that we type-check any use of a phrase to decide a value, because this might not yet have been checked otherwise.

```
void term_compile(OUTPUT_STREAM, pcalc_term pt) {
    if (logging_to_I6_text) { log_pcalc_term(&pt); return; }
    if (pt.variable >= 0) {
        WRITE("%c", pcalc_vars[pt.variable]);
        return;
    }
    if (pt.constant) {
        if (pt.cinder >= 0) {
            WRITE("const_%d", pt.cinder);
        } else {
            if (spec_is_phrasal(pt.constant)) typecheck_without_expectations(pt.constant);
            spec_compile(OUT, pt.constant);
        }
        return;
    }
    if (pt.function) {
        i6_schema *fn;
        binary_predicate *bp = (pt.function)->bp;
        fn = bp_get_term_as_function_of_other(bp, 0);
        if (fn == NULL) fn = bp_get_term_as_function_of_other(bp, 1);
        if (fn == NULL) internal_error("Function of non-functional predicate");
        sch_expand(fn, OUT, &(pt.function->fn_of), NULL);
        return;
    }
    internal_error("Broken pcalc term");
}
```

The function term_compile is called from 6/sch and 6/cdefp.

§**8. Detect locals.** Properly speaking, this has nothing to do with either cinders or deferrals, but it solves essentially the same problem in another context.

Here we search a proposition to look for any term involving a local variable. This is used to verify past tense propositions, which cannot rely on local values because their contents may have been wiped and reused many times since the time with which the proposition is concerned.

```
int prop_detect_locals(pcalc_prop *prop) {
    TRAVERSE_VARIABLE(pl);
    int i, locals_count = 0;
    TRAVERSE_PROPOSITION(pl, prop)
        for (i=0; i<pl->arity; i++)
            locals_count =
                detect_local_in_term(&(pl->terms[i]), locals_count);
    return locals_count;
}
int detect_local_in_term(pcalc_term *pt, int locals_count) {
    if (pt->function)
        locals_count += detect_local_in_term(&(pt->function->fn_of), locals_count);
    if (pt->constant)
        locals_count += detect_local_in_spec(pt->constant, locals_count);
    return locals_count;
}
int detect_local_in_spec(specification *spec, int locals_count) {
    int i;
    if (spec_get_storage_form(spec) == LOCAL_VARIABLE_SPC) return ++locals_count;
    if (spec_is_phrasal(spec)) {
        INVOCATION_VARIABLE(inv);
        LOOP_THROUGH_INVOCATION_LIST(inv, spec_invocation_list(spec)) {
            specification *param;
            LOOP_THROUGH_TOKENS_PARSED_IN_INV(inv, param)
                locals_count +=
                    detect_local_in_spec(param, locals_count);
        }
    }
    for (i=0; i<spec_get_argc(spec); i++)
        locals_count += detect_local_in_spec(spec_get_argument(spec, i), locals_count);
    return locals_count;
}
```

The function prop_detect_locals is called from 5/mlc, 6/defer and 11/chron.

Compile Deferred Propositions 6/cdefp

Purpose

To compile the I6 routines needed to perform the tests or tasks deferred as being too difficult in their original contexts.

Template interpreter commands

```
2   {-callv:compile_deferred_propositions}
```

§**1. Comment.** The following compiles an I6 comment noting the reason for a deferral.

```
void compile_comment_about_deferral_reason(OUTPUT_STREAM, int reason) {
    switch(reason) {
        case CONDITION_DEFER:
            WRITE("! True or false?\n"); break;
        case NOW_ASSERTION_DEFER:
            WRITE("! Force this to be true via 'now':\n"); break;
        case EXTREMAL_DEFER:
            WRITE("! Find the extremal x satisfying:\n"); break;
        case LOOP_DOMAIN_DEFER:
            WRITE("! Find next x satisfying:\n"); break;
        case NUMBER_OF_DEFER:
            WRITE("! How many x satisfy this?\n"); break;
        case TOTAL_DEFER:
            WRITE("! Find a total property value over all x satisfying:\n"); break;
        case RANDOM_OF_DEFER:
            WRITE("! Find a random x satisfying:\n"); break;
        case MULTIPURPOSE_DEFER:
            WRITE("! Abstraction for set of x such that:\n"); break;
        default: internal_error("Unknown proposition deferral reason");
    }
}
```

§**2. Preliminaries.** We have seen that propositions are deferred for diverse reasons. Here we take our medicine, and actually compile the deferred propositions into routines. The basic structure of a proposition routine is the same for all of the various reasons, but with considerable variations affecting (mainly) the initial setup and the returned value.

Note that the unchecked array bounds of 26 are safe here because propositions may only use 26 different variables at most (`x`, `y`, `z`, `a`, ..., `w`). There therefore can't be more than 26 callings, or 26 quantifiers, either.

```
void compile_deferred_propositions(OUTPUT_STREAM) {
    pcalc_prop_deferral *pdef;
    LOOP_OVER(pdef, pcalc_prop_deferral) {
        int ct_locals_problem_thrown = FALSE, negated_quantifier_found = FALSE;
        current_sentence = pdef->deferred_from;
        pcalc_prop *proposition = prop_copy(pdef->proposition_to_defer);
        int multipurpose_routine = (pdef->reason == MULTIPURPOSE_DEFER)?TRUE:FALSE;
        int reason = CONDITION_DEFER;                    redundant assignment to appease gcc -O2
        ⟨Simplify the proposition by flipping negated quantifiers, if possible 3⟩;
        phsf_create_nonphrase_stack_frame();
        ⟨Write some explanatory comments into the output 4⟩;
        WRITE("[ Prop_%d ", pdef->allocation_id);
        ⟨Declare the I6 local variables which will be needed by this deferral routine 7⟩;
        INDENT;
        ⟨Compile the code inside this deferral routine 8⟩;
        ⟨Issue a problem message if the table-lookup locals were needed 5⟩;
        ⟨Issue a problem message if a negated quantifier was needed 6⟩;
        OUTDENT; WRITE("];\n\n");
        phsf_remove_nonphrase_stack_frame();
    }
}
```

The function compile_deferred_propositions is invoked by a command in a `.i6t` template file.

§**3.** Just in case this hasn't already been done:

⟨Simplify the proposition by flipping negated quantifiers, if possible 3⟩ ≡

```
    int changed = FALSE;
    proposition = simp_negated_determiners(proposition, &changed, TRUE);
    if (changed) {
        LOGIF(CALCULUS, "simp_negated_determiners: $D\n", proposition);
    }
```

This code is used in §2.

§**4.** The following does nothing except to log details of the routine to the debugging log, and also in I6 comments in the generated I6 code.

⟨Write some explanatory comments into the output 4⟩ ≡

```
LOGIF(CALCULUS, "compile_deferred_propositions: %d: $D\n",
    pdef->allocation_id, proposition);
compile_comment_about_deferral_reason(OUT, pdef->reason);
STREAM *save_dl = dl; dl = OUT; logging_to_I6_text = TRUE;
LOG("! $D", proposition);
dl = save_dl; logging_to_I6_text = FALSE;
WRITE("\n");
if ((current_sentence) && (current_sentence->word_ref1 >= 0)) {
    save_dl = dl; dl = OUT;
    LOG("! in '$W'\n", current_sentence->word_ref1, current_sentence->word_ref2);
    dl = save_dl;
}
```

This code is used in §2.

§**5.** While unfortunate in a way, this is for the best, because a successful match on a condition looking up a table would record the table and row in local variables within the deferred proposition: they would then be wrong in the calling routine, where they are needed.

⟨Issue a problem message if the table-lookup locals were needed 5⟩ ≡

```
if ((do_we_need_ct_locals()) && (!ct_locals_problem_thrown)) {
    ct_locals_problem_thrown = TRUE;
    sentence_problem(_P_(C6CantLookUpTableInDeferred),
        "I am not able to look up table entries in this complicated "
        "condition",
        "which seems to involve making a potentially large number "
        "of checks in rather few words (and may perhaps result from "
        "a misunderstanding such as writing the name of a kind where "
        "an individual object is intended?).");
}
```

This code is used in §2.

§**6.** This looks like a horrible restriction, but in fact propositions are built and simplified in such a way that it never bites. (Quantifiers are always moved outside of negation where possible, and it is almost always possible.)

⟨Issue a problem message if a negated quantifier was needed 6⟩ ≡

```
if (negated_quantifier_found) {
    sentence_problem(_P_(BelievedImpossible),
        "this involves a very complicated negative thought",
        "which I'm not able to untangle. Perhaps you could rephrase "
        "this more simply, or split it into more than one sentence?");
}
```

This code is used in §2.

§**7.** Recall that an I6 function header consists of a `[`, then an identifier name for the function – in this case always `Prop_N` for some number `N` – and then a space-delimited list of local variable names, the initial of which are used to receive function arguments. The order of the variables is: any cinders (constants evaluated back at deferral time and being handed forward on the stack as function arguments); then any variables in the predicate calculus sense, of which the first may or may not be being used as a function argument, depending on whether or not it is bound; then the enumeration variables needed to compile generalised quantifiers, if any; and finally any oddball variables needed by code specific to particular deferral reasons.

⟨Declare the I6 local variables which will be needed by this deferral routine 7⟩ ≡

```
int j, var_states[26], no_extras;
if (multipurpose_routine) WRITE("reason");                    no cinders exist here
else cind_declare(OUT, proposition, pdef);
WRITE(" ");
vars_determine_status(proposition, var_states, NULL);
for (j=0; j<26; j++)
    if (var_states[j] != UNUSED_VST)
        WRITE("%c %c_ix ", pcalc_vars[j], pcalc_vars[j]);
no_extras = 0;
TRAVERSE_VARIABLE(pl);
TRAVERSE_PROPOSITION(pl, proposition)
    if (pl->element == DOMAIN_OPEN_ATOM) {
        WRITE("qcy_%d qcn_%d ", no_extras, no_extras);
        no_extras++;
    }
⟨Declare the I6 locals needed by adaptations to particular deferral cases 29⟩;
WRITE(";\n");
```

This code is used in §2.

§**8.**

⟨Compile the code inside this deferral routine 8⟩ ≡

```
if (multipurpose_routine) {
    int use;
    WRITE("if (reason >= 0) { x = reason; reason = %d; }\n",
        CONDITION_DUSAGE);
    WRITE("switch (reason) {\n");
    INDENT;
    pcalc_prop *safety_copy = prop_copy(proposition);
    for (use = EXTREMAL_DUSAGE; use <= CONDITION_DUSAGE; use++) {
        if (use > EXTREMAL_DUSAGE) proposition = prop_copy(safety_copy);
        switch (use) {
            case CONDITION_DUSAGE: reason = CONDITION_DEFER; break;
            case LOOP_DOMAIN_DUSAGE: reason = LOOP_DOMAIN_DEFER; break;
            case NUMBER_OF_DUSAGE: reason = NUMBER_OF_DEFER; break;
            case RANDOM_OF_DUSAGE: reason = RANDOM_OF_DEFER; break;
            case TOTAL_DUSAGE: reason = TOTAL_DEFER; break;
            case EXTREMAL_DUSAGE: reason = EXTREMAL_DEFER; break;
        }
        WRITE("%d: ", use);
        compile_comment_about_deferral_reason(OUT, reason);
        INDENT;
        ⟨Compile body of deferred proposition for the given reason 9⟩;
```

```
        OUTDENT;
    }
    OUTDENT;
    WRITE("}\n");
} else {
    reason = pdef->reason;
    ⟨Compile body of deferred proposition for the given reason 9⟩;
}
```

This code is used in §2.

§**9.** From here on, we compile the body of a routine to handle the deferral case in the variable `reason`.

What these different cases have in common is that each is basically a search of all possible values of the bound variables in the expression. There will be some initialisation, something to do with each successfully found combination, and eventually some winding-up code. For example, "number of..." initialises by setting `counter` to 0, on each success it performs `counter++`, and at the end of the search it performs `return counter`.

⟨Compile body of deferred proposition for the given reason 9⟩ ≡

```
property_name *prn = NULL;
property_name *def_prn = NULL;
int def_prn_sign = 0;
switch(reason) {
    case NOW_ASSERTION_DEFER: break;
    case CONDITION_DEFER: ⟨Initialisation before CONDITION search 30⟩; break;
    case EXTREMAL_DEFER: ⟨Initialisation before EXTREMAL search 42⟩; break;
    case LOOP_DOMAIN_DEFER: ⟨Initialisation before LOOP search 46⟩; break;
    case NUMBER_OF_DEFER: ⟨Initialisation before NUMBER search 33⟩; break;
    case TOTAL_DEFER: ⟨Initialisation before TOTAL search 39⟩; break;
    case RANDOM_OF_DEFER: ⟨Initialisation before RANDOM search 36⟩; break;
}
⟨Compile code to search for valid combinations of variables 11⟩;
switch(reason) {
    case NOW_ASSERTION_DEFER: break;
    case CONDITION_DEFER: ⟨Winding-up after CONDITION search 32⟩; break;
    case EXTREMAL_DEFER: ⟨Winding-up after EXTREMAL search 44⟩; break;
    case LOOP_DOMAIN_DEFER: ⟨Winding-up after LOOP search 48⟩; break;
    case NUMBER_OF_DEFER: ⟨Winding-up after NUMBER search 35⟩; break;
    case TOTAL_DEFER: ⟨Winding-up after TOTAL search 41⟩; break;
    case RANDOM_OF_DEFER: ⟨Winding-up after RANDOM search 38⟩; break;
}
```

This code is used in §8.

§**10. The Search.** We can now begin the real work. Given ϕ, we compile I6 code which contains a magic position M (for "match") such that M is visited exactly once for every combination of possible substitutions into the bound variables such that ϕ is true. For example,

$$\exists x : door(x) \wedge open(x) \wedge \exists y : room(y) \wedge in(x, y)$$

might compile to code in the form:

```
blah, blah, blah {
    M
} rhubarb, rhubarb
```

such that execution reaches `M` exactly once for each combination of open door x and room y such that x is in y. (Position `M` is where we will place the case-dependent code for what to do on a successful match.) In the language of model theory, this is a loop over all interpretations in which ϕ is true.

We will do this by compiling the proposition from left to right. If there are k atoms in ϕ, then there are $k + 1$ positions between atoms, counting the start and the end. Then:

Invariant. Let ψ be any syntactically valid subproposition of ϕ (that is, a contiguous sequence of atoms from ψ which would be a valid proposition in its own right). Then there are before and after positions `B` and `A` in the compiled I6 code for searching ϕ such that

(a) `A` cannot be reached except from `B`, and
(b) at execution time, on every occasion `B` is reached, `A` is then reached exactly once for each combination of possible substitutions into the $\exists$-bound variables of ψ such that ψ is then true.

In particular, in the case when $\psi = \phi$, `B` is the start of our compiled I6 code (before anything is done) and `A` is the magic match position `M`.

The restriction to syntactically valid subpropositions is important. Suppose ϕ arises from "all doors are open" and is stored in memory as:

```
Forall x IN[ door(x) IN] open(x)
```

Then `door(x)` and `Forall x IN[ door(x) IN]` are valid, for instance, but `IN] open(x)` is not.

Lemma. If the Invariant holds for two adjacent syntactically valid subpropositions μ and ν, then it holds for the subproposition $\mu\nu$.

Proof. There are now three positions in the code: `B1`, before μ; `B2`, before ν, which is the same position as after μ; and `A`, after ν. Execution reaches `B2` m times for each visit to `B1`, where m is the number of combinations of viable bound variable values in μ. Execution reaches `A` n times for each visit to `B2`, where n is the similar number for ν. Therefore execution reaches `A` a total of nm times for each visit to `B1`, the product of the number of variable combinations in μ and ν, which is exactly the number of combinations in total.

Corollary. If the Invariant holds for subpropositions in each of the following forms, then it will hold overall.

(a) `Exists v`, for some variable v, or `Q v IN[ ... IN]`, for some quantifier other than $\exists$.
(b) `NOT[ ... NOT]`.
(c) any single predicate-like atom.

Proof. Because all valid subpropositions are concatenations of these, and we then apply the Lemma.

It follows that if we can prove our algorithm maintains the invariant in cases (a) to (d), we can be sure it will correctly construct code leading to the match point `M`.

§**11.** We will make use of three stacks:

(a) The R-stack, which holds the current "reason": the goal being pursued by the I6 code currently being compiled.
(b) The Q-stack, which holds details of quantifiers being searched on.
(c) The C-stack, which holds details of callings of variables.

Since each is tied to a quantifier, each of which is tied to a distinct variable, and there are at most 26 variables, we need a worst-case capacity of 27 slots on the R-stack (counting the initial `reason`) and 26 on the Q-stack and C-stack.

⟨Compile code to search for valid combinations of variables 11⟩ ≡

```
int block_nesting = 0;                          how many { ... } blocks are open in I6 code being compiled
```

The R-stack

```
int R_stack_reason[27];
int R_stack_parity[27];
int R_sp = 0;
```

The Q-stack

```
quantifier *Q_stack_quantifier[26];
int Q_stack_parameter[26];
int Q_stack_C_stack_level[26];
int Q_stack_block_nesting[26];
int Q_sp = 0;
```

The C-stack

```
pcalc_term C_stack_term[26];                    the term to which a called-name is being given
int C_stack_index[26];                          its index in the deferred_calling_list
int C_sp = 0;
```

⟨Push initial reason onto the R-stack 12⟩;
we now begin compiling the search code
⟨Compile the proposition into a search algorithm 15⟩;
`while (Q_sp > 0)` ⟨Pop the Q-stack 24⟩;
`while (C_sp > 0)` ⟨Pop the C-stack 26⟩;
we are now at the magic match point `M` *in the search code*
⟨Pop the R-stack 14⟩;
`while (block_nesting > 0)`
    ⟨Close a block in the I6 code compiled to perform the search 28⟩;
we have now finished compiling the search code

```
if (R_sp != 0) internal_error("R-stack failure");
if (Q_sp != 0) internal_error("Q-stack failure");
if (C_sp != 0) internal_error("C-stack failure");
```

This code is used in §9.

§**12. The R-stack.** This is a sort of "split goals into sub-goals" mechanism. In order to determine

if all but one of the closed doors are unlocked, ...

our main goal is to determine the truth of the "are unlocked" part. This is reason `CONDITION_DEFER`, and it is pushed onto the R-stack at the start of the compilation:

⟨Push initial reason onto the R-stack 12⟩ ≡
```
R_stack_reason[R_sp] = reason;
R_stack_parity[R_sp] = TRUE;
R_sp++;
```
This code is used in §11.

§**13.** But in order to work this out, we have to work out which doors are closed, and this is a subgoal to which we give the pseudo-reason `FILTER_DEFER`. We push this new sub-goal onto the R-stack, leaving the original to be resumed when we're done.

define `FILTER_DEFER 10000` *pseudo-reason value used only inside this routine*

⟨Push domain-opening onto the R-stack 13⟩ ≡
```
R_stack_reason[R_sp] = FILTER_DEFER;
R_stack_parity[R_sp] = TRUE;
R_sp++;
```
This code is used in §15.

§**14.** The R-stack is then popped when the goal is accomplished (or rather, when the I6 code we are compiling has reached a point which will be executed when its goal has been accomplished).

In the case of `FILTER_DEFER`, when scanning domains of quantifiers, we increment the count of the domain set size – the number of closed doors, in the above example. (See below.)

⟨Pop the R-stack 14⟩ ≡
```
R_sp--; if (R_sp < 0) internal_error("R stack underflow");
switch(R_stack_reason[R_sp]) {
    case FILTER_DEFER:
        WRITE("qcn_%d++;\n", Q_sp-1);
        break;
    case NOW_ASSERTION_DEFER: break;
    case CONDITION_DEFER: ⟨Act on successful match in CONDITION search 31⟩; break;
    case EXTREMAL_DEFER: ⟨Act on successful match in EXTREMAL search 43⟩; break;
    case LOOP_DOMAIN_DEFER: ⟨Act on successful match in LOOP search 47⟩; break;
    case NUMBER_OF_DEFER: ⟨Act on successful match in NUMBER search 34⟩; break;
    case TOTAL_DEFER: ⟨Act on successful match in TOTAL search 40⟩; break;
    case RANDOM_OF_DEFER: ⟨Act on successful match in RANDOM search 37⟩; break;
}
```
This code is used in §11,15,11,15,11.

§15. Compiling the search. In the following we run through the proposition from left to right, compiling I6 code as we go, but preserving the Invariant.

```
⟨Compile the proposition into a search algorithm 15⟩ ≡
    TRAVERSE_VARIABLE(pl);
    int run_of_conditions = FALSE;
    int no_deferred_callings = 0;                    how many CALLED atoms have been found to date
    TRAVERSE_PROPOSITION(pl, proposition) {
        switch (pl->element) {
            case NEGATION_OPEN_ATOM:
            case NEGATION_CLOSE_ATOM:
                ⟨End a run of predicate-like conditions, if one is under way 18⟩;
                R_stack_parity[R_sp-1] = (R_stack_parity[R_sp-1])?FALSE:TRUE;        reverse parity
                break;
            case QUANTIFIER_ATOM:
                ⟨End a run of predicate-like conditions, if one is under way 18⟩;
                if (R_stack_parity[R_sp-1] == FALSE) negated_quantifier_found = TRUE;
                quantifier *quant = RETRIEVE_POINTER_quantifier(pl->predicate);
                int param = pl->calling_data;
                if (quant != exists_quantifier) ⟨Push the Q-stack 22⟩;
                ⟨Compile a loop through possible values of the variable quantified 20⟩;
                break;
            case DOMAIN_OPEN_ATOM:
                ⟨End a run of predicate-like conditions, if one is under way 18⟩;
                ⟨Push domain-opening onto the R-stack 13⟩;
                break;
            case DOMAIN_CLOSE_ATOM:
                ⟨End a run of predicate-like conditions, if one is under way 18⟩;
                ⟨Pop the R-stack 14⟩;
                break;
            case CALLED_ATOM:
                ⟨Push the C-stack 25⟩;
                break;
            default:
                if (R_stack_reason[R_sp-1] == NOW_ASSERTION_DEFER)
                    ⟨Compile code to force the atom 19⟩
                else
                    ⟨Compile code to test the atom 17⟩;
                break;
        }
    }
    ⟨End a run of predicate-like conditions, if one is under way 18⟩;
```

This code is used in §11.

§16. Predicate runs and their negations. Or, cheating Professor de Morgan.

If we have a run of predicate-like atoms – say X, Y, Z – then this amounts to a conjunction: $X \wedge Y \wedge Z$. The obvious way to compile code for this would be to take one term at a time:

```
if (X)
    if (Y)
        if (Z)
```

That satisfies the Invariant, and is clearly correct. But we want to use the same mechanism when looking at a negation, and then it would go wrong.

Note that if ϕ contains $\neg(\psi)$ then ψ must be a conjunction of predicate-like atoms. (Otherwise a problem message would be issued and in that case it doesn't matter what code we compile, so long as we don't crash: it will never be run.) Thus we can assume that between `NEGATION_OPEN_ATOM` and `NEGATION_CLOSE_ATOM` is a predicate run.

Between negation brackets, then, we must interpret X, Y, Z as $\neg(X \wedge Y \wedge Z)$, and we need to compile that to

```
if (~~(X && Y && Z))
```

rather than

```
if (~~X)
    if (~~Y)
        if (~~Z)
```

which gets de Morgan's laws wrong.

§**17.** That means a little fancy footwork to start and finish the compound `if` statement properly:

⟨Compile code to test the atom 17⟩ ≡

```
if (run_of_conditions == FALSE) {
    WRITE("if ");
    if (R_stack_parity[R_sp-1] == FALSE) WRITE("(~~");
    WRITE("(");
    run_of_conditions = TRUE;
} else WRITE(" && ");
atom_compile(OUT, TEST_ATOM_TASK, pl);
```

This code is used in §15.

§**18.**

⟨End a run of predicate-like conditions, if one is under way 18⟩ ≡

```
if (run_of_conditions) {
    if (R_stack_parity[R_sp-1] == FALSE) WRITE(")");
    WRITE(")");
    run_of_conditions = FALSE;
    ⟨Open a block in the I6 code compiled to perform the search 27⟩;
}
```

This code is used in §15.

§**19.** The `NOW_ASSERTION_DEFER` reason is different from all of the others, because rather than searching for a given situation it tries force it to happen (or not to). Forcing rather than testing is easy here: we just supply a different task when calling `atom_compile`.

In the negated case, we again cheat de Morgan, by falsifying ϕ more aggressively than we need: we force $\neg(X) \wedge \neg(Y) \wedge \neg(Z)$ to be true, though strictly speaking it would be enough to falsify X alone. (We do it that way for consistency with the same convention when asserting about the model world.)

We don't need to consider runs of predicates for that; we can take the atoms one at a time.

⟨Compile code to force the atom 19⟩ ≡

```
atom_compile(OUT, (R_stack_parity[R_sp-1])?NOW_ATOM_TRUE_TASK:NOW_ATOM_FALSE_TASK, pl);
WRITE("\n");
```

This code is used in §15.

§**20. Quantifiers and the Q-stack.** It remains to deal with quantifiers, and to show that the Invariant is preserved by them. There are two cases: $\exists$, and everything else.

The existence case is the easiest. Given $\exists v : \psi(v)$ we compile

```
loop header for v to run through its domain set {
    ...
```

and note that execution reaches the start of the loop body once for each possible choice of v, as required by the Invariant – indeed the Invariant pretty much requires that this is what we compile.

⟨Compile a loop through possible values of the variable quantified 20⟩ ≡

```
pl = compile_loop_header(OUT, pl->terms[0].variable, pl,
    (R_stack_reason[R_sp-1] == NOW_ASSERTION_DEFER)?TRUE:FALSE,
    (quant != exists_quantifier)?TRUE:FALSE);
⟨Open a block in the I6 code compiled to perform the search 27⟩;
```

This code is used in §15.

§**21.** Generalised quantifiers – "at least three", "all but four", and so on – make quantitative statements about the number of valid or invalid cases over a domain set. These need more elaborate code. Suppose we have $\phi = Qv \in \{v \mid \psi(v)\} : \theta(v)$, which in memory looks like this:

```
QUANTIFIER --> DOMAIN_OPEN --> psi --> DOMAIN_CLOSE --> theta
```

We compile that to code in the following shape:

```
set count of domain size to 0
set count of valid cases to 0
loop header for v to run through its domain set {
    if psi holds {
        increment count of domain size
        if theta holds {
            increment count of valid cases
        }
    }
}
if the counts are such that the quantifier is satisfied {
    ...
```

We don't always need both counts. For instance, to handle "at least three doors are unlocked" we count both the domain size (the number of doors) and the number of valid cases (the number of unlocked doors), but only need the latter. This might be worth optimising some day, to save local variables.

§**22.** The domain size and valid case counts are stored in locals called `qcn_N` and `qcy_N` respectively, where `N` is the index of the quantifier – 0 for the first one in the proposition, 1 for the second and so on.

On reading a non-existence `QUANTIFIER` atom, we compile code to zero the counts, and push details of the quantifier onto the Q-stack, so that we can recover them later. We then compile a loop header exactly as above.

The test of ψ, which acts as a filter on the domain set – e.g., only doors, not all objects – is handled by pushing a suitable goal onto the R-stack, but we don't need to do anything to make that happen here, because the `DOMAIN_OPEN` atom does it.

⟨Push the Q-stack 22⟩ ≡

```
if (R_stack_reason[R_sp-1] == NOW_ASSERTION_DEFER)
    ⟨Handle "not exists" as "for all not" 23⟩;
Q_stack_quantifier[Q_sp] = quant;
Q_stack_parameter[Q_sp] = param;
Q_stack_block_nesting[Q_sp] = block_nesting;
Q_stack_C_stack_level[Q_sp] = C_sp;
WRITE("qcy_%d = 0;\n", Q_sp);
WRITE("qcn_%d = 0;\n", Q_sp);
Q_sp++;
```

This code is used in §15.

§**23.** It is always true that $\nexists x : \psi(x)$ is equivalent to $\forall x : \neg(\phi(x))$, so the following seems pointless. We do this, in the case of "now" only, in order to make $\nexists$ legal in a "now", which it otherwise wouldn't be. Most quantifiers aren't, because they are too vague: "now fewer than six doors are open", for instance, is not allowed. But we do want to allow "now nobody likes Mr Wickham", say, which asserts $\nexists x : person(x) \wedge likes(x, W)$.

⟨Handle "not exists" as "for all not" 23⟩ ≡

```
if (quant == not_exists_quantifier) {
    R_stack_parity[R_sp-1] = (R_stack_parity[R_sp-1])?FALSE:TRUE;
    quant = for_all_quantifier;
}
```

This code is used in §22.

§**24.** To resume the narrative of what happens when we read:

```
QUANTIFIER --> DOMAIN_OPEN --> psi --> DOMAIN_CLOSE --> theta
```

We zeroed the counters, compiled the loop headers and pushed details to the Q-stack at the `QUANTIFIER` atom; pushed a filtering goal onto the R-stack at the `DOMAIN_OPEN` atom; popped it again as accomplished at `DOMAIN_CLOSE`, compiling a line which increments the domain size to celebrate; and then compiled code to test θ.

Now we are at the end of the line, and still have the quantifier code half-done, as we know because the Q-stack is not empty. We first compile an increment of the valid cases count, because if execution of the I6 code gets to the end of testing θ then it must have found a valid case: in the "at least three doors are unlocked" example, it will have found an unlocked one among the doors making up the domain. We then need to record any "called" values for later retrieval by whoever called this proposition routine: see below. That leaves just this part:

```
            }
        }
    }
    if the counts are such that the quantifier is satisfied {
        ...
```

left to compile, and we will be done: execution will reach the ... if and only if it is true at run-time that three or more of the doors is unlocked.

Thus this elaborate generalised-quantifier case satisfies the Invariant because it transfers execution from before to ... either 0 times (if the counts don't satisfy us), or once. Unlike in the $\exists v$ case, it's not a question of enumerating which v work and which do not; the whole thing works, or doesn't, and is more like testing a single `if`.

⟨Pop the Q-stack 24⟩ ≡

```
Q_sp--; if (Q_sp < 0) internal_error("Q stack underflow");
WRITE("qcy_%d++;\n", Q_sp);
while (C_sp > Q_stack_C_stack_level[Q_sp])
    ⟨Pop the C-stack 26⟩;
while (block_nesting > Q_stack_block_nesting[Q_sp])
    ⟨Close a block in the I6 code compiled to perform the search 28⟩;
WRITE("if (");
quant_compile_test(OUT, Q_stack_quantifier[Q_sp], Q_sp, Q_stack_parameter[Q_sp]);
WRITE(")");
⟨Open a block in the I6 code compiled to perform the search 27⟩;
```

This code is used in §11.

§25. The C-stack. When a CALLED atom in the proposition gives a name to a variable, we have to transcribe that to the `deferred_calling_list` for the benefit of the code calling this proposition routine. Each time we discover that a term t is to be given a name, we stack it up. These are not always variables:

> if a person (called the dupe) is in a dark room (called the lair), ...

gives names to x ("dupe") and $f(x)$ ("lair"), because simplification has eliminated the variable y which appears to be being given a name.

⟨Push the C-stack 25⟩ ≡

```
C_stack_term[C_sp] = pl->terms[0];
C_stack_index[C_sp] = no_deferred_callings++;
C_sp++;
```

This code is used in §15.

§26. When does the compiled search code record values into `deferred_calling_list`? In two situations:

(a) when a domain-search has successfully found a viable case for a quantifier, the values of any variables called in that domain are recorded;
(b) and otherwise the values of called variables are recorded just before point `M`, that is, immediately before acting on a successful match.

For example, when reading:

> if a person (called the dupe) is in a lighted room which is adjacent to exactly one dark room (called the lair), ...

the value of "dupe" is transferred just before `M`, but the value of "lair" is transferred as soon as a dark room is found. The code looks like this:

```
set count of domain size to 1
loop through domain (i.e., dark rooms adjacent to the person's location) {
    increment count of domain size
    record the lair value
}
if the count of domain size is 1 {
```

```
        record the dupe value
        M
    }
```

If we waited until point `M` to record the lair value, it would have disappeared, because `M` is outside the loop which searches the domain of the "exactly one" quantifier.

⟨Pop the C-stack 26⟩ ≡

```
    C_sp--; if (C_sp < 0) internal_error("C stack underflow");
    WRITE("deferred_calling_list-->%d = ", C_stack_index[C_sp]);
    term_compile(OUT, C_stack_term[C_sp]);
    WRITE(";\n");
```

This code is used in §11,24,11,24,11,24.

§**27.** That just leaves the blocking, which follows the One True Brace Style. Thus:

⟨Open a block in the I6 code compiled to perform the search 27⟩ ≡

```
    WRITE("{\n");
    INDENT;
    block_nesting++;
```

This code is used in §18,20,24,18,20,24,18,20,24.

§**28.** and:

⟨Close a block in the I6 code compiled to perform the search 28⟩ ≡

```
    OUTDENT;
    WRITE("}\n");
    block_nesting--;
```

This code is used in §11,24,11,24,11,24.

§29. Adaptations. That completes the general pattern of searching according to the proposition's instructions. It remains to adapt it to different needs, by providing, in each case, some setting-up code; some code to execute when a viable set of variable values is found; and some winding-up code.

In some of the cases, additional local variables are needed within the `Prop_N` routine, to keep track of counters or totals. These are they:

⟨Declare the I6 locals needed by adaptations to particular deferral cases 29⟩ ≡

```
if (multipurpose_routine) WRITE("total counter selection best best_with");
else {
    switch (pdef->reason) {
        case NUMBER_OF_DEFER: WRITE("counter"); break;
        case RANDOM_OF_DEFER: WRITE("counter selection"); break;
        case TOTAL_DEFER: WRITE("total"); break;
        case EXTREMAL_DEFER: WRITE("best best_with"); break;
    }
}
```

This code is used in §7.

§30. Adaptation to CONDITION. The first and simplest of our cases to understand: where ϕ is a sentence, with all variables bound, and we have to return `true` if it is true and `false` if it is false. There is no initialisation:

⟨Initialisation before CONDITION search 30⟩ ≡

```
;
```

This code is used in §9.

§31. As soon as we find any valid combination of the variables, we return `true`:

⟨Act on successful match in CONDITION search 31⟩ ≡

```
WRITE("rtrue;\n");
```

This code is used in §14.

§32. So we only reach winding-up if every case failed, and then we return `false`:

⟨Winding-up after CONDITION search 32⟩ ≡

```
WRITE("rfalse;\n");
```

This code is used in §9.

§33. Adaptation to NUMBER. In the remaining cases, ϕ has variable x (only) left free, but the use we want to make will be a loop over all objects x, and we compile this "outer loop" here: the loop opens in the initialisation code, closes in the winding-up code, and therefore completely encloses the code generated by the searching mechanism above.

In the first case, we want to count the number of x for which $\phi(x)$ is true. The local `counter` holds the count so far; it starts out automatically at 0, since all I6 locals do.

⟨Initialisation before NUMBER search 33⟩ ≡

```
proposition = compile_loop_header(OUT, 0, proposition, FALSE, FALSE);
WRITE("{\n");
INDENT;
```

This code is used in §9.

§**34.** Recall that we get here for *each possible way* that $\phi(x)$ could be true, that is, once for each viable set of values of bound variables in ϕ. But we only want to increment `counter` once, so having done so, we exit the searching code and continue the outer loop.

The `jump` to a label is forced on us since I6, unlike, say, Perl, has no syntax to break or continue a loop other than the innermost one.

⟨Act on successful match in NUMBER search 34⟩ ≡
```
WRITE("counter++;\n");
WRITE("jump NextOuterLoop_%d;\n", reason);
```
This code is used in §14.

§**35.** The continue-outer-loop labels are marked with the reason number so that if code is compiled for each reason in turn within a single routine – which is what we do for multipurpose deferred propositions – the labels do not have clashing names.

⟨Winding-up after NUMBER search 35⟩ ≡
```
WRITE(".NextOuterLoop_%d;\n", reason);
OUTDENT;
WRITE("}\n");
WRITE("return counter;\n");
```
This code is used in §9.

§**36. Adaptation to RANDOM.** To choose a random x such that $\phi(x)$, we essentially run the same code as for NUMBER searches, but twice over: first to count how many such x there are, then to run through again to find the nth of these, where n is a uniformly random number such that $1 \le n \le x$.

This avoids needing to store the full list of matches anywhere, which would be impossible since (a) it's potentially a lot of storage and (b) it can only safely live on the current stack frame, and I6 does not allow arrays on the current stack frame (because of restrictions in the Z-machine). This means that, on average, the compiled code takes 50% longer to find its random x than it ideally would, but we accept the trade-off.

⟨Initialisation before RANDOM search 36⟩ ≡
```
WRITE("selection = -1;\n");
WRITE("while (true) {\n");
INDENT;
WRITE("counter = 0;\n");
proposition = compile_loop_header(OUT, 0, proposition, FALSE, FALSE);
WRITE("{\n");
INDENT;
```
This code is used in §9.

§**37.** Again we exit the searcher as soon as a match is found, since that guarantees that $\phi(x)$.

Note that we can only return here on the second pass, since `selection` is -1 throughout the first pass, whereas `counter` is non-negative.

⟨Act on successful match in RANDOM search 37⟩ ≡
```
WRITE("counter++;\n");
WRITE("if (counter == selection) return %c;\n", pcalc_vars[0]);
WRITE("jump NextOuterLoop_%d;\n", reason);
```
This code is used in §14.

§**38.** We return `nothing` – the non-object – if `counter` is zero, since that means the set of possible x is empty. But we also return if `selection` has been made already, because that means that the second pass has been completed without a return – something which in theory cannot happen, but just might do if testing part of the proposition had some side-effect changing the state of the objects and thus the size of the set of possibilities.

⟨Winding-up after RANDOM search 38⟩ ≡

```
WRITE(".NextOuterLoop_%d;\n", reason);
OUTDENT;
WRITE("}\n");
WRITE("if ((counter == 0) || (selection >= 0)) return nothing;\n");
WRITE("selection = random(counter);\n");
OUTDENT;
WRITE("}\n");
```

This code is used in §9.

§**39. Adaptation to TOTAL.** Here the task is to sum the values of property P attached to each object in the domain $\{x \mid \phi(x)\}$. (At the moment, this is guarded so that the domain set must indeed contain world objects, not values.)

⟨Initialisation before TOTAL search 39⟩ ≡

```
proposition = compile_loop_header(OUT, 0, proposition, FALSE, FALSE);
WRITE("{\n");
INDENT;
```

This code is used in §9.

§**40.** The only wrinkle here is the way the compiled code knows which property it should be totalling. If we know that ourselves, we can compile in a direct reference. But if we are compiling a multipurpose deferred proposition, then it might be used to total any property over the domain, and we won't know which until runtime – when its identity will be found in the I6 variable `property_to_be_totalled`.

⟨Act on successful match in TOTAL search 40⟩ ≡

```
char *source_of_property;
if (multipurpose_routine)
    source_of_property = "property_to_be_totalled";
else {
    prn = RETRIEVE_POINTER_property_name(pdef->defn_ref);
    source_of_property = prn_get_i6_identifier(prn);
}
if (source_of_property == NULL) internal_error("No property name");
WRITE("total = total + %c.%s;\n", pcalc_vars[0], source_of_property);
WRITE("jump NextOuterLoop_%d;\n", reason);
```

This code is used in §14.

§**41.**

⟨Winding-up after TOTAL search 41⟩ ≡

```
WRITE(".NextOuterLoop_%d;\n", reason);
OUTDENT;
WRITE("}\n");
WRITE("return total;\n");
```

This code is used in §9.

§**42. Adaptation to EXTREMAL.** This is rather similar. We find the member of $\{x \mid \phi(x)\}$ which either minimises, or maximises, the value of some property P. We use two local variables: `best`, the extreme P value found so far; and `best_with`, the member of the domain set which achieves that.

If two or more x achieve the optimal P-value, it is deliberately left undefined which one is returned. The user may be typing "the heaviest thing on the table", but what he gets is "a heaviest thing on the table".

We open the search with `best_with` equal to `nothing`, the non-object, which is what we will return if the domain set turns out to be empty; and with `best` set to the furthest-from-optimal value possible. For a search maximising P, `best` starts at the lowest number representable in the virtual machine; for a minimisation, it starts at the highest. That way, if any member of the domain is found, its P-value must be at least as good as the starting value of `best`.

Again the only nuisance is that sometimes we know P, and whether we are maximising or minimising, at compile time; but for a multipurpose routine we don't, and have to look that up at run-time.

⟨Initialisation before EXTREMAL search 42⟩ ≡

```
if (multipurpose_routine) {
    WRITE("if (property_loop_sign>0) best=MIN_NEGATIVE_NUMBER;\n");
    WRITE("else best=MAX_POSITIVE_NUMBER;\n");
} else {
    measurement_definition *mdef =
        RETRIEVE_POINTER_measurement_definition(pdef->defn_ref);
    mdef_read_property_details(mdef, &def_prn, &def_prn_sign);
    if (def_prn_sign == 1) {
        WRITE("best = MIN_NEGATIVE_NUMBER;\n");
    } else {
        WRITE("best = MAX_POSITIVE_NUMBER;\n");
    }
}
proposition = compile_loop_header(OUT, 0, proposition, FALSE, FALSE);
WRITE("{\n");
INDENT;
```

This code is used in §9.

§**43.** It might look as if we could speed up the multipurpose case by multiplying by `property_loop_sign`, thus combining the max and min versions into one, and saving an `if`. But (a) the multiplication is as expensive as the `if` (remember that on a VM there's no real branch penalty), and (b) we need to watch out because -1 times -32768, on a 16-bit machine, is -1, not 32768: so it is *not* always true that multiplying by -1 is order-reversing.

⟨Act on successful match in EXTREMAL search 43⟩ ≡

```
char *source_of_property;
if (multipurpose_routine) source_of_property = "property_to_be_totalled";
else source_of_property = prn_get_i6_identifier(def_prn);
if (multipurpose_routine) {
    WRITE("if (property_loop_sign>0) {\n");
    INDENT;
    WRITE("if (%c.%s >= best) {\n", pcalc_vars[0], source_of_property);
    INDENT;
    WRITE("best = %c.%s;\n", pcalc_vars[0], source_of_property);
    WRITE("best_with = %c;\n", pcalc_vars[0]);
    OUTDENT;
    WRITE("}\n");
    OUTDENT;
```

```
        WRITE("} else {\n");
        INDENT;
        WRITE("if (%c.%s <= best) {\n", pcalc_vars[0], source_of_property);
        INDENT;
        WRITE("best = %c.%s;\n", pcalc_vars[0], source_of_property);
        WRITE("best_with = %c;\n", pcalc_vars[0]);
        OUTDENT;
        WRITE("}\n");
        OUTDENT;
        WRITE("}\n");
    } else {
        WRITE("if (%c.%s %s best) {\n",
            pcalc_vars[0], source_of_property, (def_prn_sign == 1)?">=":"<=");
        INDENT;
        WRITE("best = %c.%s;\n", pcalc_vars[0], source_of_property);
        WRITE("best_with = %c;\n", pcalc_vars[0]);
        OUTDENT;
        WRITE("}\n");
    }
```

This code is used in §14.

§**44.**

⟨Winding-up after EXTREMAL search 44⟩ ≡

```
    WRITE(".NextOuterLoop_%d;\n", reason);
    OUTDENT;
    WRITE("}\n");
    WRITE("return best_with;\n");
```

This code is used in §9.

§**45. Adaptation to LOOP.** Here the proposition is used to iterate through the members of the domain set $\{x \mid \phi(x)\}$. Two local variables exist: `x` and `x_ix`. One of the following is true:

(1) The domain set contains only objects, so that `x` is non-zero if it represents a member of that set. In this case `x_ix` may or may not be used, and we will not rely on it.
(2) The domain set contains only values, and then `x` might easily be zero, but `x_ix` is always the index within the domain set: 1 if `x` is the first value, 2 for the second and so on.

The proposition is called with a pair of values `x`, `x_ix` and returns the next value `x` in the domain set, or 0 if the domain is exhausted. (In case (2) it's not safe to regard 0 as an end-of-set sentinel value because 0 can be a valid member of the set; so in looping through (2) we should first find the size of the set using NUMBER OF, then keep calling for members until the index reaches the size.) There is no need to return the next `x_ix` value since it is always the present value plus 1.

If the proposition is called with `x` set to `nothing`, in case (1), or with `x_ix` equal to 0, in case (2), it returns the first value in the domain.

§**46.** Snarkily, this is how we do it:

```
if we're called with a valid member of the domain, go to Z
loop x over members of the domain {
    return x
    label Z is here
}
```

Which is not really a loop at all, but is a cheap way to extract either the initial value or the successor value from a loop header. (The trick actually caused some consternation for I6 hackers when early drafts of I7 came out, because they had been experimenting with a patch to I6 which protected `objectloop` from object-tree rearrangements but which assumed that nobody ever used `jump` to enter a loop body bypassing its header. But the DM4, which defines I6, doesn't forbid this. The designer of I6 has learned his lesson, though: I7 has no goto or jump instruction, and I7 loops can be proved to be entered and exited cleanly.)

⟨Initialisation before LOOP search 46⟩ ≡

```
WRITE("if (%c_ix > 0) {\n", pcalc_vars[0]);
INDENT;
WRITE("%c_ix--;\n", pcalc_vars[0]);
WRITE("jump NextOuterLoop_%d;\n", reason);
OUTDENT;
WRITE("}\n");
WRITE("if (%c) jump NextOuterLoop_%d;\n", pcalc_vars[0], reason);
proposition = compile_loop_header(OUT, 0, proposition, FALSE, FALSE);
WRITE("{\n");
INDENT;
```

This code is used in §9.

§**47.**

⟨Act on successful match in LOOP search 47⟩ ≡

```
WRITE("return %c;\n", pcalc_vars[0]);
```

This code is used in §14.

§**48.**

⟨Winding-up after LOOP search 48⟩ ≡

```
WRITE(".NextOuterLoop_%d;\n", reason);
OUTDENT;
WRITE("}\n");
WRITE("return nothing;\n");
```

This code is used in §9.

§**49. Compiling loop headers.** The final task of this entire chapter is to compile an I6 loop header which causes a given variable v to range through a domain set D – which we have to deduce by looking at the proposition ψ in front of us.

We want this loop to run as quickly as possible: efficiency here makes a very big difference to the running time of compiled I7 code. Most optimisations aren't worth the risk in added complexity – but these are.

Loops through kinds of value are not in general optimisable. The problem cases involve loops through objects. Consider:

> if everyone in the Dining Room can see an animal, ...

Code like this will run very slowly:

```
loop over objects (x)
    loop over objects (y)
        if x is a person
            if x is in the Dining Room
                if y is an animal
                    if x can see y
                        success!
```

This is folly in so many ways. Most objects aren't people or animals, so almost all combinations of x and y are wasted. We test the eligibility of x for every possible y. And there are quick ways to find what is in the Dining Room, so we're missing a trick there, too. What we want is:

```
loop over objects in the Dining Room (x)
    if x is a person
        loop over animals (y)
            if x can see y
                success!
```

§**50.** Part of the work is done already: we generate propositions with quantifiers as far forwards as they can be, so we won't loop over y before checking the validity of x. The rest of the work comes from two basic optimisations:

(1) if a loop over v is such that $K(v)$ holds in every case, where K is a kind, then loop v over K rather than all objects, and
(2) if a loop over v is such that $R(v, t)$ holds in every case, then loop over all v such that $R(v, t)$ in cases where R has a run-time representation making this quick and easy.

In each case we can then delete $K(v)$ or $R(v, t)$ from the proposition as redundant, since the loop header has taken care of it.

Case (1) is called "kind optimisation"; case (2), "parent optimisation", because the prototype case is R being containment – we exploit that the object-tree parent of v is known, and that I6 has a fast form of `objectloop` to visit the children of a given node in the object tree. Case (2) has to be avoided if we are compiling code to force a proposition, rather than test it, because then $R(v, t)$ is not an accomplished fact but is something we have yet to make come true. This is why the loop-compiler takes a flag `avoid_parent_optimisation`. Case (1) doesn't suffer from this since kinds cannot be changed at run-time.

§**51.** So here is the code. We write an I6 schema for the loop into `loop_schema`, then expand it into the output.

```
i6_schema loop_schema;
pcalc_prop *compile_loop_header(OUTPUT_STREAM, int var, pcalc_prop *proposition,
    int avoid_parent_optimisation, int grouped) {
    kind_of_value *kov = NULL;
    pcalc_prop *kind_position = NULL;
    pcalc_term var_term = term_new_variable(var);
    pcalc_term second_term = term_new_variable(var);          this value is never used, but just in case
    int parent_optimised = FALSE;
    the default, if we are unable to provide either kind or parent optimisation
    sch_write_to_existing(&loop_schema, "objectloop (*1 ofclass Object)");
    ⟨Scan the proposition to find the domain of the loop, and look for opportunities 52⟩;
    if ((kov) && (parent_optimised == FALSE)) {          parent optimisation is stronger, so we prefer that
        kov_write_loop_schema(&loop_schema, kov, TRUE);
        proposition = prop_delete_atom(proposition, kind_position);
    }
    sch_expand(&loop_schema, OUT, &var_term, &second_term);
    return proposition;
}
```

§**52.** The following looks more complicated than it really is. Sometimes it's called to compile a loop arising from a quantifier with a domain, in which case `grouped` is set and `proposition` points to:

```
QUANTIFIER --> DOMAIN_OPEN --> psi --> DOMAIN_CLOSE --> ...
```

so that ψ, the part in the domain group, defines the range of the variable. But sometimes the call is to compile a loop not arising from a quantifier, so there is no domain group to scan; instead the whole proposition makes up ψ, and now `grouped` is clear.

⟨Scan the proposition to find the domain of the loop, and look for opportunities 52⟩ ≡

```
int bl = 0, enabled = FALSE;
TRAVERSE_VARIABLE(pl);
TRAVERSE_PROPOSITION(pl, proposition) {
    switch (element_get_group(pl->element)) {
        case OPEN_OPERATORS_GROUP: bl++; break;
        case CLOSE_OPERATORS_GROUP: bl--; break;
    }
    if (grouped) {
        if (pl->element == DOMAIN_OPEN_ATOM) enabled = TRUE;
        if (pl->element == DOMAIN_CLOSE_ATOM) enabled = FALSE;
        if (bl < 0) break;
        if (enabled == FALSE) continue;
        if (bl != 1) continue;
    } else {
        if (bl < 0) break;
        if (bl > 0) continue;
    }
    ⟨Scan $\psi$, the part of the proposition establishing the domain 53⟩;
}
```

This code is used in §51.

§**53.** In either case, we scan ψ looking for $K(v)$ atoms, which would tell us the domain set for the variable v, or for $R(v,t)$ atoms for parent-optimisable relations R.

⟨Scan ψ, the part of the proposition establishing the domain 53⟩ ≡

```
if ((pl->element == KIND_ATOM) && (pl->terms[0].variable == var)) {
    kov = pl->assert_kind_of_value;
    kind_position = pl_prev;
}
if ((avoid_parent_optimisation == FALSE) &&
    (pl->element == PREDICATE_ATOM) && (pl->arity == 2))
    ⟨Consider parent optimisation on this binary predicate 54⟩;
```

This code is used in §52.

§**54.** We give the relation R an opportunity to write a loop which runs v through all possible x such that $R(x,t)$, by writing a schema for the loop in which `*1` denotes the variable v and `*2` the term t.

For example, the worn-by relation writes the schema:

```
objectloop (*1 in *2) if (WearerOf(*1)==parent(*1))
```

where v runs quickly through the object-tree children of t, but items carried rather than worn are skipped.

We have to check three possible cases: $R(v,t)$ direct, and then $is(f_R(v),t)$ or $is(t,f_R(v))$, which can arise from simplifications. We set `optimise_on` to R and `parent` to t.

⟨Consider parent optimisation on this binary predicate 54⟩ ≡

```
binary_predicate *bp = RETRIEVE_POINTER_binary_predicate(pl->predicate);
if (bp == a_is_b_predicate) {
    int chk;
    for (chk=0; chk<=1; chk++) {
        pcalc_func *pf = pl->terms[chk].function;
        if ((pf) && (pf->fn_of.variable == var) &&
            (bp_write_optimised_loop_schema(&loop_schema, pf->bp))) {
            second_term = pl->terms[1-chk];
            parent_optimised = TRUE;
            proposition = prop_delete_atom(proposition, pl_prev);
            break;
        }
    }
} else if ((pl->terms[0].variable == var) &&
    (bp_write_optimised_loop_schema(&loop_schema, bp))) {
    second_term = pl->terms[1];
    parent_optimised = TRUE;
    proposition = prop_delete_atom(proposition, pl_prev);
}
```

This code is used in §53.

§**55.** And that concludes the predicate-calculus engine at the heart of Inform, and with it Chapter 6.

7 Kinds and Specifications

`7/itsp`: *Introduction to Specifications.w* A general introduction to kinds of value and specifications.

`7/tids`: *Table of ID Numbers.w* To manage a table of details about the ID numbers used to enumerate families and species of specifications, and kinds of value.

`7/kov`: *Kinds of Value.w* To manage the `kind_of_value` structure, representing the possible categories into which eventual I6 values can be placed; to determine which KOVs contain which others, and what their characteristics are.

`7/data`: *Data Types.w* To manage the built-in set of data types, which are used as the atomic kinds of value and therefore characterise fundamental kinds of data – numbers, times, texts, and so on – and to create new ones as required.

`7/dti`: *Data Type Interpreter.w* To read in details of the built-in data types from the template file "Types.i6t", setting them up ready for use.

`7/dim`: *Dimensions.w* To keep a small database indicating the physical dimensions of numerical values, and how they combine: for instance, allowing us to specify that a length times a length is an area.

`7/kix`: *Kinds Index.w* To produce most of the Kinds page in the Index for a project: the chart at the top, and the detailed entries below.

`7/spec`: *Specifications.w* To create, manage and compare specifications.

`7/ttts`: *Text to Specifications.w* Complete forms of parsing from excerpts of text to specifications.

`7/specv`: *Specification Versions of Other Structures.w* To see how specifications correspond to other structures which they conceptually include.

`7/cfsp`: *Compiling from Specifications.w* To compile specifications into Inform 6 values, conditions or void expressions.

`7/masp`: *MATCHING Specifications.w* Documentation on and functions handling specifications which fall into the MATCHING family.

`7/vasp`: *VALUE Specifications.w* Documentation on and functions handling specifications which fall into the VALUE family.

`7/stsp`: *STORAGE Specifications.w* Documentation on and functions handling specifications which fall into the STORAGE family.

`7/cosp`: *CONDITION Specifications.w* Documentation on and functions handling specifications which fall into the CONDITION family.

`7/cmsp`: *COMMAND Specifications.w* Documentation on and functions handling specifications which fall into the COMMAND family.

`7/tc`: *Type Checking.w* To decide whether a given specification, representing a value or construct found in the source text, can be used in a given context: the possible answers being yes, no, or "possibly, contingent on run-time type checking".

`7/inv`: *Invocations.w* Specifications which ask to use a phrase (which are "phrasal") indicate which phrase they intend by means of a list of "invocations". This list goes on to record the outcome of type-checking and provides instructions for code generation, as we see here.

Introduction to Specifications 7/itsp

Purpose

A general introduction to kinds of value and specifications.

§**1. Specifications.** A "specification" describes a single unambiguous meaning. It is a very general concept, covering not only what other programming languages would call "values" but also what they would call "types". It is a Swiss Army knife of representation.

Specifications arise in two ways. First, they are the outcome of successful parsing of an excerpt of text. "201", "the player", "a number", "a closed door", "award 6 points" and "an existing variable" all produce specifications. Each one is marked with a reference to the words it came from, so that wherever it travels through Inform it can always be used to construct a useful problem message related back to its origin.

Second, though this happens less frequently, they are created internally to describe something – usually to describe what is needed in a particular context. A specification provides a natural way to say "I would like a phrase to decide a value here".

§**2.** The reader might be feeling a certain déjà vu here. Weren't meaning lists also data structures which unified a great range of possible meanings for excerpts of texts? Why do we need this new concept?

There are several answers. One is simply that specifications are much more compact in memory than meaning lists, which are bloated by having to hold rival interpretations of the same text. (This is why meaning lists are thrown away when parsing is done.) Another is that meaning lists are basically linguistic in their structure – they make trees like those drawn by grammarians. Specifications, on the other hand, are organised around the basic concepts of Inform. So they are much easier for us to use.

A more satisfying answer is that specifications are *unambiguous*, but we need to define that word carefully. In the case of phrases, they can contain multiple possible "invocations". For instance, the phrase

```
let exemplum be 10;
```

has a subtly different meaning if the value "exemplum" already exists than if it doesn't. The specification (SP) will be unambiguous that this is a phrase, but will carry a list of alternative invocations describing different possible ways it could be carried out at run-time. The reason such ambiguities are allowed to persist into the SP, rather than being resolved at the meaning list stage, is that some ambiguities are impossible to sort out at compile time: instead, run-time code must be compiled to get matters right when the code is executed. The SP must therefore be allowed to contain exactly that range of ambiguities potentially insoluble until run-time, and no other. In design terms, this is the principle behind the meaning list/specification boundary.

§**3. Kinds of value.** A value is something which could eventually exist as a value in some variable at run-time. The number 17, the time "3:15 PM", the every turn rulebook, etc., are all values. A value may either be self-contained when stored at run-time (a "word value", occupying one word of memory) or be a pointer to a larger array of data (a "pointer value"). For instance, times are word values; lists are pointer values.

A kind of value (often abbreviated "KOV") categorises values. (KOVs are closely analogous to what C-like programming languages call "types".) For instance, the KOV of 17 is "number", the KOV of 3:15 PM is "time", and so on. If the source text defines

```
Paint colour is a kind of value. The paint colours are mauve, duck-white and peach.
```

then "duck-white" is a value and "paint colour" is its KOV.

§**4.** Some kinds of value contain each other – for instance a "rulebook" is a special case of a "rule" – so that it is possible for a value V to belong to more than one kind of value at once. When we talk about its own kind of value, we implicitly mean the narrowest kind of value containing it. If two kinds of value K_1 and K_2 are such that K_1 contains K_2, we say that K_2 "implicitly casts" to K_1 – meaning that by doing nothing at all at run-time, a K_2 value can simply be treated as if it were a K_1 value, because it already is one. Of course, most pairs of KOV are simply incomparable – "number" and "text", for instance – so that neither can implicitly cast to the other.

KOVs thus form a hierarchy. At the top is "value", which matches any value at all, so that every other KOV is contained within it. The other special point in the hierarchy is the kind of value "object", which matches all rooms, things, directions, regions and so on. If the source text says

```
A wheelbarrow is a kind of vehicle. The blue garden barrow is a wheelbarrow.
```

then the value "blue garden barrow" has kind "wheelbarrow", which is within "vehicle", within "thing", within "object", within "value". The "object" KOV is unusual because values with any kind at or below it in the hierarchy – in other words objects – are mutually distinguishable at run-time. The I6 value of a specific wheelbarrow can never coincide with that of a specific door, for instance. This allows us to be much looser in type-checking below "object", and we get a great deal of flexibility that way. When a KOV represents a kind of object, it is called a KOVKO ("kind of value which is a kind of object").

Kinds of value are stored in a `kind_of_value` structure, which is usually a lightweight packet of data, but can in principle be quite a large tree. "List of lists of numbers", for instance, is represented by a `kind_of_value` marked "list of" which points to another `kind_of_value` marked "list of" which points to a `kind_of_value` marked "number". This is an example of a "constructed KOV" or KOVCON, since it has been made out of other KOVs. A more basic one is called an "atomic KOV" or KOVA.

`kind_of_value` structures are identified internally with numerical constants called "data type IDs" and written with the `_TY` suffix: thus, `TEXT_TY`, `NUMBER_TY`, `LIST_OF_TY`, for instance.

§**5.** Even on a minimal run, Inform's KOV hierarchy is already quite rich, setting up the hierarchy which is the main display in the Kinds index. There are about 45 basic KOVs built in – about 15 KOVKOs (kinds of object), declared in the Standard Rules, and about 30 KOVAs used for data, whose relationships and specifics are laid out in the `Types.i6t` template file. The format of this file amounts to a mini-language just to set up KOVAs, and much of the section "Data Types.w" consists of the interpreter for that mini-language.

But the hierarchy is not *all that* rich. Inform also supports much more complicated ways to describe values than any KOV can represent. "Door" is a KOV, in fact a KOVKO, but "open door" is not a KOV, and nor is "even number", or

```
a woman in a lighted room
```

It seems at first appealing to decide that "open door" (say) ought to be a KOV, but no variable with this KOV would be safe: merely by opening a door somewhere during play, the player could make its current value no longer fit its KOV.

In order to preserve type safety, Inform forces its storage objects (variables, table entries, and so on) to have fixed KOVs decided at compile time. We are allowed to create a "number that varies", but not an "even number that varies", because the former is a KOV and the latter is not. This is rather like C, where variables might be declared like so:

```
int width, height; char *name;
```

In C, the range of types which can be stored is the same as the range of types which can be function arguments, so for instance:

```
int area(int width, int height) { return width*height; }
```

But whereas Inform's variables are much like C's, Inform's phrase definitions are not like C's functions, and they can have much more general arguments:

```
To jemmy open (the barrier - a closed door): ...
```

To repeat (P - a phrase) indefinitely: ...

We therefore need a much more general way to describe what text can appear in phrase arguments than a KOV can provide, and this is what leads us to the idea of a "specification". In fact, though, even these examples are not general enough. Natural language is very slippery about concepts which, in most computing theories, are formally separate. Consider for instance the overloaded definition of a phrase like this one:

To appreciate (composer - a person): ...

To appreciate (composer - Alessandro Striggio): ...

The early design of Inform did not permit this, since it clearly confused values and types, but this lack was in fact reported as a bug – always a sign that users considered it a natural thing to do. Specifications therefore needed to be able to represent not only vague descriptions like "person", but also explicit identities like "Alessandro Striggio".

§**6.** The enormous variety of specifications means that we need some sort of taxonomy to sort them out.

(a) Each SP is either "actual" or "generic". Actual SPs describe structures explicitly referred to in the source. Generic SPs are patterns against which these are matched: "number", for instance, or "existing text variable". Actual SPs contain specifics, often including pointers to further actual SPs. The actual SP for "award 6 points" includes a pointer to an actual SP for "6"; it matches the generic SP "phrase", which has no specifics.

(b) Like the animal kingdom, the realm of SPs is classified on several levels.

(b1) There are five "families", identified with a `*_FMY` constant. For instance, the family `STORAGE_FMY` contains all forms of storage item – variables local and global, table entries and the like.

(b2) Within each family are a number of "species", identified with a constant in the form `*_SPC`. For instance, local variables belong to the species `LOCAL_VARIABLE_SPC` of family `STORAGE_FMY`.

(b3) Some species are still further subdivided according to a kind of value associated with them, though most are not. For instance, "number that varies" is a generic member of the species `NONLOCAL_VARIABLE_SPC` of family `STORAGE_FMY` with the KOV "number".

While the source text can and almost always does create many new KOVs, the families and species are fixed, and built into Inform.

It's useful to have a shorthand notation for writing these distinctions, especially for the debugging log. So: `(G)VALUE_FMY/CONSTANT_SPC` means "a generic SP of family `VALUE_FMY` and species `CONSTANT_SPC`", and so forth.

§**7.** Whereas source text can create new KOVs, this hierarchy of SPs is hard-wired into Inform and is the same for every run.

The following table shows that every combination of actual/generic and of family is used in Inform, with just one exception.

Family	**Generic**	**Actual**
`MATCHING_FMY`	nonexisting number variable	*no actual matching SPs exist*
`VALUE_FMY`	number	102
`STORAGE_FMY`	time that varies	time of day
`CONDITION_FMY`	condition	the player is in a dark room
`COMMAND_FMY`	phrase	award 5 points

Values and their kinds form only a small part of this picture. A constant value is an `(A)VALUE_FMY/CONSTANT_SPC`, a kind of value a `(G)VALUE_FMY/CONSTANT_SPC`.

§**8.** Lastly, it seems worth recording the fact that the classification of these ideas into families and species was constantly revised in the course of writing and testing Inform. While the actual semantics of the Inform language were not changing, the SP representations used for it were in constant flux between public builds, in a process which amounted to a sort of investigation into what the "true" concepts of the language were. Numerous species and even whole families have come and gone. (Who now remembers `KIND_OF_VARIABLE_SPC`, or `OBJECT_SPEC_FMY` of which the only documentation once read "this enigmatic data type holds things like property numbers"?)

Broadly, though, there is a choice in setting up the classification of SPs between grouping together those with similar data in their structures, and on the other hand grouping together SPs which play a similar semantic role in the Inform language. For instance, the former regards "award 5 points" and "location of the harpsichord" as similar (both are phrases to be compiled), but the latter strategy sees them as very different – one performs a command, whereas the other determines a value.

For the first four years of its life, Inform classified SPs around data structures, which made parsing simpler but resulted in a type-checker festooned with exceptional cases; now it classifies SPs by semantic purpose, which makes parsing harder but simplifies type-checking. The change is not as odd as it might seem – in the early years, it was the parsing apparatus which gave the most trouble; but in later years, with the type system becoming richer, type-checking was the biggest source of "difficult" bugs.

§**9. Architecture of the type-checker.** The type-checker is the part of Inform which examines a "found SP" against an "expected SP", and decides whether this is permissible. In some cases, it issues problem messages if not: in others, it simply reports back. And in a few cases, it performs a "coercion" to modify the found SP to match the expected one: the type-checker is not a neutral observer, but actively tries to make the source fit.

The type-checker has three concentric levels, each of which can be independently used by the rest of Inform:

(1) The innermost level checks SPs which evaluate to values. The test is made by `can_we_cast_kovs` in "Kinds of Value.w", which tests whether one KOV can be used where another KOV is expected. There is no coercion at this level, and no problem messages are issued: failures are simply reported back to the calling routine.

(2) The middle level checks SPs which evaluate or describe values. For instance, this level might deduce that "Marble Door" can be used where "closed openable door" is expected. The tests are performed in "Value Checking.w". Again, there is no coercion at this level, and no problem messages are issued: failures are simply reported back.

(3) The outermost level checks SPs in general. This is done in "Type Checking.w", and is a much more active process. SPs are coerced if need be, and problem messages issued to the user.

§**10.** All three levels use a set of three constants to describe outcomes:

```
define ALWAYS_MATCH    2                 provably correct at compile time
define SOMETIMES_MATCH 1                 provably reduces to a check feasible at run-time
define NEVER_MATCH     0                 neither of the above
```

§**11.** The term "type-checking" is perhaps a little misleading – "specification matching" might be better, since level (1) alone is the equivalent of a conventional programming language's type-checker. But it's a familiar term and, after all, the whole process is analogous.

Table of ID Numbers 7/tids

Purpose

To manage a table of details about the ID numbers used to enumerate families and species of specifications, and kinds of value.

Template interpreter commands

```
1   {-callv:make_type_IDs_table}
12   {-callv:complete_type_IDs_table}
21   {-callv:compile_I6_constants_for_typenames}
28   {-callv:report_pairs_observed}
29   {-callv:report_pairs_allowed}
```

Definitions

¶1. As the Introduction showed, a "specification" can hold a very large range of concepts. Each individual SP will have its own "family" and "species", and some will also have a kind of value (KOV) attached. KOVs can themselves describe quite a range of values, and these are in turn are sorted out with "data type IDs".

That gives us three sets of ID numbers, and given how complex all this is, we want to be careful never to confuse these different categorisations. We therefore choose the IDs so that none of them overlap, i.e., no family ID is numerically the same as a species ID, and so on: this makes it easier to catch bugs where IDs have been confused together.

In general we call these numerical codes "type IDs", and they divide up as follows:

(i) The `UNKNOWN` special value, which can be used only to indicate that no family or species is given.

(ii) The `*_FMY` family identifiers.

(iii) The `*_SPC` species identifiers.

(iv) The `*_TY` data type identifiers, which can only be used to mark KOVAs. (`UNKNOWN` is not valid here.)

The values of these constants are not significant, except that they must all be different, and that some routines require `FALSE` to be a distinct value from all type IDs, so it is important that 0 never be used.

We conventionally use the range 1 to 49 for the families and species, 50 to 99 for built-in data type IDs, and 100 upwards for data type IDs arising from newly created kinds of value in the source text.

¶2. The UNKNOWN type ID. Uniquely, this can be used both as a family and as a species. As a family, it means that the meaning of the SP is unknown: such a SP is sometimes returned by a routine to mean "could not understand this text". As a species, `UNKNOWN` means nothing: it simply indicates that no species is supplied. It is also used as a default return value by some routines which would otherwise return a valid data type ID.

define `UNKNOWN 1` *"arfle barfle gloop"*

¶3. The MATCHING family. No actual SPs have this family: it is used only for generic SPs matching against raw textual material, something we do very little of, but which is conceptually so different that it earns a family of its own. Since all SPs are generic, there are never any specific details.

define MATCHING_FMY 5
define NEW_LOCAL_VARIABLE_NAME_SPC 6 *"nonexisting number variable"*

¶4. The VALUE family. VALUE specifications are "rvalues", that is, they express I6 values – numbers, objects, text and so on – but cannot be assigned to, so that in an assignment of the form "change L to R" they can be used only as R, not L. This is not the same thing as a constant: for instance, "location of the player" evaluates differently at different times, but cannot be changed in an assignment and so qualifies as a VALUE.

define VALUE_FMY 10
define CONSTANT_SPC 11 *"7", "the can't lock a locked door rule", etc.*
define PHRASE_TO_DECIDE_VALUE_SPC 12 *"holder of the black box"*

¶5. The STORAGE family. STORAGE specifications represented stored I6 data at run-time, which means that they can be used not only as rvalues but also lvalues: they can be assigned to, that is. For instance, a table entry qualifies because it can be both read and changed. To qualify as STORAGE, a type must exactly specify the storage location referred to: "Table of Corvettes" only indicates a table, not an entry in a table, so is merely a VALUE. Similarly, "carrying capacity" (as a property name not indicating an owner) is a mere VALUE.

define STORAGE_FMY 20
define LOCAL_VARIABLE_SPC 21 *"the running total", say*
define NONLOCAL_VARIABLE_SPC 22 *"the location"*
define PROPERTY_VALUE_SPC 23 *"the carrying capacity of the cedarwood box"*
define TABLE_ENTRY_SPC 24 *"tonnage in row X of the Table of Corvettes"*
define LIST_ENTRY_SPC 25 *"item 4 in L"*

¶6. The CONDITION family. This is an I7 condition which can be compiled into an I6 condition: for instance, "the bronze gate is open", or "eating the biscuit".

define CONDITION_FMY 30
define LOGICAL_AND_SPC 31 *"A and B"*
define LOGICAL_OR_SPC 32 *"A or B"*
define TEST_PROPOSITION_SPC 33 *if "the cat is on the mat"*
define TEST_PHRASE_OPTION_SPC 34 *"giving full details", say*
define TEST_ACTION_SPC 35 *"going nowhere in the Confusing Labyrinth"*
define TEST_PAST_ACTION_SPC 36 *"we have gone nowhere"*
define NOW_PROPOSITION_SPC 37 *now "the cat is on the mat"*
define PHRASE_TO_DECIDE_IF_SPC 38 *"in darkness"*
define DESCRIPTION_SPC 39 *"open door which is lockable", "a container"*

¶7. The COMMAND family. Compiles to one or more I6 statements in void context inside a routine. The most obvious examples are "to..." phrases, but there are other possibilities too.

define COMMAND_FMY 40
define TO_PHRASE_SPC 41 *"award 5 points"*
define END_BLOCK_SPC 42 *"end while"*
define OTHERWISE_SPC 43 *"otherwise"*
define CASE_SPC 44 *"– V" value in a switch*
define RULEBOOK_OUTCOME_PHRASE_SPC 45 *"persuasion succeeds"*
define TRY_ACTION_SPC 46 *"try dropping the eustace diamonds"*

¶8. That only leaves the data type IDs. Those come from two sources: built-in ones like "number", set out in the template file "Types.i6t" and given IDs in the range 50 to 99, and new ones created in the source text (by text like "Air pressure is a kind of value").

There are broadly five groups of data type ID:
(1) built-in data types used in matching, but which have no instances as such – for instance ANY_VALUE_TY: any value matches this, but no value has this as its own kind;
(2) built-in data types whose instances are stored as word-value data, where a single I6 value holds the whole thing, like NUMBER_TY;
(3) built-in data types whose instances are stored as pointers to larger blocks of data on the heap, like INDEXED_TEXT_TY;
(4) enumerations added in the source text, with a finite known number of named constant values, internally numbered upwards from 1; and
(5) units added in the source text, with literal patterns to give them notation.

The "Types.i6t" template file consists almost entirely of commands for the Data Type Interpreter, which sets up groups (1), (2) and (3) accordingly. This template file contains as much as possible of the specifics about the various types, so that a great deal can be changed about the interplay of types without altering the compiler itself.

Some of the built-in data types need to referred to in the Inform source code, and these have defined constant values below. Others never do, and these are shown as comments in the discussion below – they could be deleted from "Types.i6t" or given different meanings and Inform would have no difficulty.

¶9. Group 1: Data types used in matching.

ANY_VALUE. This is a superhero among data types: it exists only for type-checking purposes, and matches any value of any data type. Use of it basically makes the type-checking machinery switch off, so in fact when it is used other arrangements need to be made to validate what is going on. In the Standard Rules, the data type for w in the following:

    To change (tr - table-reference) to (w - value): ...

...is ANY_VALUE: but that does not mean it is necessarily sensible to write any value to a given table entry, so we need to be careful.

ASSIGNABLE_VALUE_TY represents any value except the name of a property: it is needed because of the ambiguity arising in a phrase like

    change the current safe to unlocked

where we clearly intend not an assignment to the variable "current safe" but a modification of an either/or property held by the object which is the present value of that variable.

The finer distinctions ANY_WORD_VALUE and ANY_POINTER_VALUE divide all values into those which are word data in themselves, and those which are pointers to blocks of data on the heap.

To handle some of the casts from values to pointers-to-block-structures, we also need to store kinds of value at run-time as values in their own right: that is, we need to store code numbers like NUMBER_TY as I6 values.

The two kinds of value KIND_OF_WORD_VALUE and KIND_OF_POINTER_VALUE represent these, though they are used only for matching purposes.

define ANY_VALUE_TY 50

ANY_WORD_VALUE_TY
ANY_POINTER_VALUE_TY
define ASSIGNABLE_VALUE_TY 51
define KIND_OF_WORD_VALUE_TY 52
define KIND_OF_POINTER_VALUE_TY 53

¶10. DESCRIPTION_OF_ACTION is used in type-checking to represent gerunds: that is, actions described by what appear to be participles but which are in context nouns ("if taking something, ...").

define DESCRIPTION_OF_ACTION_TY 54

¶11. VALUE_DESCRIPTION is used only in type-checking to distinguish value from object descriptions; it's the difference between "even **number**" and "open **door**".

define VALUE_DESCRIPTION_TY 55

¶12. A little bit of voidology theory here: when the source text uses the notation { }, that refers to the empty list. But this is a constant whose kind of value, uniquely, can't be determined: what is it an empty list of – numbers, texts, rooms? So it acquires the NO_ENTRIES (it means, no entries: it isn't an abbreviation for number of entries) kind of value.

define NO_ENTRIES_TY 56

¶13. CLASSIFIED. Not in any sense of being secret, though, at the same time, I don't really want people to find out about it and use it. I6 is not a typed language but it has what amounts to a data type in its generalised sense of object: a value which can be any of 0, an object, a class, a packed string or a routine. Given an arbitrary value of this type, I6 can determine at run-time which of these five possibilities is the case. We want to exploit this ability to perform run-time type-checking, so we need an umbrella I7 data type which means "anything stored at run-time in one of these I6 formats". This data type is called CLASSIFIED since it can be resolved by looking at the I6 metaclass at run time. (Okay, so perhaps it ought to be called "metaclassified".)

We cannot decide simply to ignore CLASSIFIED in the interests of a simpler life since, though its use is deprecated, we need it to deal with the return value of a rulebook – something currently rather fuzzy in Inform's design: eventually we shall probably have rulebooks which specify explicitly the kind of value they return, and then CLASSIFIED can at last go.

/* CLASSIFIED_TY */

¶14. Group 2: Data types for basic word-sized values.

NUMBER. An integer. Constants can be written either in words (up to twelve), or signed decimal numbers.

define NUMBER_TY 57

¶15. TIME. "Men can do nothing without the make-believe of a beginning. Even Science, the strict measurer, is obliged to start with a make-believe unit, and must fix on a point in the stars' unceasing journey when his sidereal clock shall pretend that time is at Nought. His less accurate grandmother Poetry has always been understood to start in the middle; but on reflection it appears that her proceeding is not very different from his; since Science, too, reckons backwards as well as forwards, divides his unit into billions, and with his clock-finger at Nought really sets off in medias res" (George Eliot, *Daniel Deronda*). Our make-believe here is midnight, our unit is divided not into billions but into 1240, and a value of this kind holds one of two possibilities:

(i) an absolute time, measured as the number of minutes since midnight;
(ii) a relative time, measured in minutes.

Thus the value 70 might mean 1:10 AM, or it might mean 70 minutes, and type-checking does not try to distinguish the two. (This is so that arithmetic will be easier – we can add 70 minutes to 1:10 AM to get 2:20 AM, but if they had different kinds, this would be illegal.) The ambiguity is occasionally unhelpful, though: we have to supplement the "[a time]" Understand token, which parses an absolute time, with a special "[a time period]" one, so that users are able to parse relative times as well.

define TIME_TY 58

¶16. TEXT, TEXT_ROUTINE, QUOT, SNIPPET. Inform 6 code was designed to be compiled for the Z-machine, a virtual computer with very unusual memory access rules but with support for several kinds of textual matter. This means that string-handling in I6 code is a messy business. Many things which would be routine in most languages are essentially impossible in I6 code (for instance, concatenating arbitrary strings of text, or extracting a character at a given position in an arbitrary string). The I7 type system tries to hide some of this nightmare vision from us, but as a result is pretty complicated.

TEXT is the native type of a piece of double-quoted text without substitutions, and for type checking purposes it also matches a TEXT-ROUTINE, which is double-quoted text which does have substitutions. (These compile into I6 packed strings and routines respectively: since they can be distinguished using `metaclass`, I6 can treat them as if they were a single type without breaking type safety, so it is safe for variables to have the kind of value TEXT, say.) A QUOT is the double-quoted text used for displaying boxed quotations, which has different line-breaking conventions, and has to be handled in a type of its own. Finally, a SNIPPET represents a selection of words from the player's command, recording both the word number to start at and the number of words included in the snippet. This must be handled with care, because when the player's command changes, all old SNIPPET values become meaningless. It therefore cannot be stored meaningfully in global variables, except for a couple defined by I7's own code and which are automatically updated each time the command changes.

define TEXT_TY 59
define TEXT_ROUTINE_TY 60
define QUOT_TY 61
define SNIPPET_TY 62

¶17. UNICODECHAR. A character referred to by its Unicode number, or by a name defined in the Unicode standard: for instance, "unicode 65" is a capital A. Characters are not really a data type in I7: rather, we deal with one-character strings if we need to access individual letters, and this type exists to refer to constants only.

define UNICODECHAR_TY 63

¶18. OBJECT. An object (a thing, room, direction, etc.): something which compiles to either an Object in I6 code. (Thus, not a kind, which compiles to a Class in I6 code, even though objects and kinds of object share the same structure, world_object, in NI.)

There is normally one FORMLESS reference: a world_object * pointer to the object in question. If this is absent (i.e., there are no references), then one of the flags is used...

```
define OBJECT_TY 64
```

¶19. I6_DESCRIPTION_ROUTINE corresponds to the "description" kind of value, and compiles as an I6 value to the routine name for a general-purpose deferred proposition.

```
define OBJECT_DESCRIPTION_TY 65
```

¶20. USEOPTION. The named use options can actually be values at run-time, though the Inform documentation does not exactly stress this point. We need the possibility in order to be able to define the phrase

To decide whether using the (UO - use-option): ...

VALUE/USEOPTION SPs have one FORMLESS reference: a use_option * pointer.

```
define USEOPTION_TY 66
```

¶21. PROPERTY. The name of a property can itself be used as a value on occasion, as for instance in the "if X provides P" condition. An actual property has a single reference, to the relevant property_name structure.

```
define PROPERTY_TY 67
```

¶22. RULE, RULEBOOK, RULE_OUTCOME, RULEBOOK_OUTCOME. Rules are values at run-time, and names such as "can't reach inside closed containers rule" are constants evaluating to RULE_TY. (They are compiled to names of routines in I6 code.) Rulebooks are also values, but are numbered from 1 upwards.

Each rule has a unique identifying number at run-time. If this ID number is less than or equal to the number of rulebooks created then the number refers to a rulebook, which are numbered 0, 1, 2, ... in order of creation. An ID number above this threshold value is taken as the Z-machine address of the routine to which a rule has been compiled. In principle there could be a problem if any genuine rule had a Z-machine address so low as to clash with the range of rulebook numbers, but in practice the presence of the I6 library routines, the veneer, and other Z-machine impedimenta mean that this cannot happen.

RULE constant SPs have one FORMLESS reference: a booked_rule * pointer to the rule.

RULEBOOK constant SPs have one FORMLESS reference: a rulebook * pointer to the rulebook.

RULEBOOK_OUTCOME constants are similar values to possible outcomes of a rulebook, with a FORMLESS reference to a rulebook_outcome * pointer.

RULE_OUTCOME is a bit bogus as a data type: no variables or properties ever exist at run-time to hold data of this type. It represents the data type of the return code of a routine implementing a rule, and exists only evanescently on the run-time stack. We need it because we need a type name in order to parse the "outcome" notation in phrase definitions like:

To continue the action: (- rfalse; -) - outcome.

```
define RULE_TY 68
RULE_OUTCOME_TY
define RULEBOOK_TY 69
define RULEBOOK_OUTCOME_TY 70
```

¶23. `ACTION_NAME`. For those rare occasions where we need to identify the basic underlying action but none of the nouns, etc., thereto. At run-time, this stores as the I6 action number: e.g. `##Go` for the going action.

define `ACTION_NAME_TY 71`

¶24. `ACTIVITY`. Activities are also numbered, and this data type holds the identifying number of an activity. Names like "reading a command" evaluate to `ACTIVITY` constants, and the data type is mostly useful in order for the Standard Rules to define phrases which run activities.

`ACTIVITY` constant SPs have one `FORMLESS` reference: an `activity *` pointer to the activity.

define `ACTIVITY_TY 72`

¶25. `SCENE`. Scenes are similarly numbered and stored in their own data type: actually, they are for practical purposes a built-in enumeration type, as the `Types.i6t` specification shows.

`SCENE` constant SPs use the `index` to store the allocation ID number of the scene referred to. (That is, scenes are numbered upwards from 0 as they are created, and this is the number stored here. The first scene created, with number 0, is always the "Entire Game", defined in the Standard Rules: this furnishes us with a convenient default value.)

define `SCENE_TY 73`

¶26. `TABLE`, `TABLE_COLUMN`. Table names and table column names are both values at run-time. Tables are stored as their byte addresses in I6, table columns as code numbers in such a way that each different column name has its own number, regardless of the table it belongs to. (This is so that we can cross-reference between two tables, using a common column as a key.)

Constant `TABLE` and `TABLE_COLUMN` SPs each have one `FORMLESS` reference: a `table *` and `table_column *` pointer respectively.

define `TABLE_TY 74`
define `TABLE_COLUMN_TY 75`

¶27. `EQUATION`. And similarly for the name of an equation.

define `EQUATION_TY 76`

¶28. `RELATION`. The different relations can, again, be used as values at run time. This data type is the type of the parameter R in the phrase:

> To decide which object is next step via (R - abstract-relation) from (O1 - object) to (O2 - object): ...

(which is defined in the Standard Rules). Relation numbers are indexes to the table of relation information in the compiled I6 code.

Constant `RELATION` SPs have one `FORMLESS` reference: a `binary_predicate *` pointer.

define `RELATION_TY 77`

¶29. `UNDERSTANDING`. Again, no variables or properties have this type. It is used to refer to a token of parsing grammar for the I6 parser to handle, and needs to be a data type because literal text in double quotation marks must always result in a `VALUE` specification. The data type depends on context. When we parse grammar, text like "take [something]" comes out as `UNDERSTANDING`, whereas in other contexts it would be `TEXT_ROUTINE`. To make this flexibility possible, `UNDERSTANDING` needs to be a data type, not a family.

This type is also used in grammar tables, to hold the result of a grammar token. In that event there is one formless reference to a constant `UNDERSTANDING`: a `grammar_verb *` pointer.

define `UNDERSTANDING_TY 78`

¶30. `FIGURENAME`, `SOUNDNAME`. A resource ID number for a figure (i.e., a picture) or a sound effect in the eventual blorb, or for use in Glulx within the application.

define `FIGURENAME_TY 79`
define `SOUNDNAME_TY 80`

¶31. `EXTERNALFILE`. A `-->` array to a run-time data structure associated with an external file, read or written by the story file during play.

define `EXTERNALFILE_TY 81`

¶32. `TRUTH_STATE`. This is a boolean value type: it can have only two values at run-time, `true` and `false` in I6.

define `TRUTH_STATE_TY 82`

¶33. Group 3: Data types for block-sized values.

¶34. `INDEXED_TEXT`, added in August 2007, is the long-awaited flexible length, character-indexed string type, and was the first pointer-valued data type. See "Indexed Text.i6t" for details.

define `INDEXED_TEXT_TY 83`

¶35. `STORED_ACTION` is just what it says it is: a stored action, which can be tried later. Again, this is a pointer value; see "StoredAction.i6t".

define `STORED_ACTION_TY 84`

¶36. Strictly speaking there is no data type `LIST_OF`: it is a constructor, referring to the act of taking K and producing "list of K". But for many practical purposes we can treat it as a data type, since the block on the heap used to represent "list of K" is much the same whatever K is. See "Lists.i6t".

define `LIST_OF_TY 85`

¶37. The data type `INTERMEDIATE` is also fictitious: this is a different sort of composition.

define `INTERMEDIATE_TY 86`

¶38. And this is the starting point for the type interpreter to create further built-in types, of which there will only ever be few:

```
int next_Types_dot_i6_number = 87;
```

¶39. Now for the data types created by new kinds of value in the source text. These are given IDs from 100 upwards, as the following constant defines:

```
define BASE_OF_DATA_TYPE_IDS 100                    base value for KOVAs of new KOVs in the source text
define LOWEST_DESIGNED_TY (BASE_OF_DATA_TYPE_IDS)
define HIGHEST_DESIGNED_TY (next_free_data_type_ID-1)
define LOOP_OVER_DESIGNED_TYPE_IDS(var)
    for (var=LOWEST_DESIGNED_TY; var<=HIGHEST_DESIGNED_TY; var++)
define LOOP_OVER_DATA_TYPE_IDS(var)
    for (var=1; var<next_free_data_type_ID; var++)
        if (type_IDs_table[var].data_type_rules)
define LOOP_OVER_POSSIBLE_TYPE_IDS(var)
    for (var=1; var<next_free_data_type_ID; var++)
int next_free_data_type_ID = BASE_OF_DATA_TYPE_IDS;     next new kind of value gets this
```

¶40. The table. That gives us quite a mountain of type IDs, with various relationships between them. For instance, TEST_ACTION_SPC can only be used as a species ID, and only when subordinate to the family ID CONDITION_FMY. It turns out to be useful to record some details for each ID, which we do with the following structure.

```
typedef struct type_id_rules {
    int type_number;                                    the ID number which this structure describes
    int type_id_flags;                                  a bitmap of *_SPC flags for properties of species
    struct vocabulary_entry *parsing_name;              built-in types have one word names
    struct vocabulary_entry *parsing_plural;
    int word_ref1, word_ref2;                           but new kinds of value do not
    int plural_ref1, plural_ref2;
    char *source_name;                                  used in the debugging log: e.g. "NUMBER_TY"
    char *description;                                  used in Problem messages: e.g. "a number"
    char numerical_source_name[32];                     used to make a debugging log name
    int no_possible_specific_types;                     or 0 if not a family
    int main_type_for_which_this_is_specific;           or UNKNOWN if it isn't
    struct data_type_id_rules *data_type_rules;         or null if not a data type
    struct kind_of_value *kova_of_this;                 cached result of kova(T)
} type_id_rules;
```

The structure type_id_rules is shared with 7/kix.

¶41. SP contexts. Specifications are used right across Inform, but there are five main situations in which they turn up, as follows.

(1) As something which we expect to see when type-checking. For instance, if we are trying to match "award 5 points" against the definition of the phrase "award (N - a number) points", then the type we expect the middle word to have is `(G)VALUE_FMY/CONSTANT_SPC-NUMBER_TY`.

(2) As something we actually find when type-checking. For instance, the "5" is found as a `(A)VALUE_FMY/CONSTANT_SPC-NUMBER_`

(3) As a value or other construction to be compiled. For instance, the "5" found above would eventually be sent to `spec_compile` as a `(A)VALUE_FMY/CONSTANT_SPC-NUMBER_TY` specification.

(4) As an expression parsed when reading in the source text. Again, "5" is an example, but so is the entire phrase "award 5 points", which is parsed as a `(A)PHRASE_FMY/TO_PHRASE_SPC`.

(5) As the name of a type, which we parse when reading in a definition: for instance, "a number" would be parsed as a type and stored in a SP as `(G)VALUE_FMY/CONSTANT_SPC-NUMBER_TY`.

We define constants for these five contexts solely because, with type specifications being so complex, Inform takes the precaution of checking that they are correctly used: a precaution, that is, against bugs in Inform itself. We use these contexts to form bitmaps, so they are defined as powers of 2.

```
define NO_SPEC_CONTEXTS 8                    we store bitmaps of these in char (i.e. 8-bit) arrays
define TYPE_EXPECTED_SPCONTEXT 1
define TYPE_FOUND_SPCONTEXT 2
define COMPILED_SPCONTEXT 4
define PARSED_SPCONTEXT 8
define TYPE_PARSED_SPCONTEXT 16
```

¶42. Each SP has a pair of ID numbers attached: family and species. The following little structure is used to record in which contexts it is legal for a SP to exist with a given pair of ID numbers (M, S) attached.

```
typedef struct type_pair_rules {
    char legal_contexts;
    struct specification *observations[NO_SPEC_CONTEXTS];
} type_pair_rules;
```

The structure type_pair_rules is private to this section.

¶43. An even more minimal data structure, the need for which is explained below:

```
typedef struct single_integer {
    int the_integer;
} single_integer;
```

The structure single_integer is private to this section.

§**1. Starting up.** Early in Inform's run, we create the table, and populate it with details of the main and species IDs. We don't add the data type IDs yet, but we do signal the data type interpreter (see "Data Types.w") to get ready for that.

Altogether about 80 IDs – for main types, specific types and data types – are created in even the smallest run, and each new kind of value created by the source text adds another.

```
define INITIAL_ID_TABLE_SIZE 150          must be at least 50, to cover main and specific types
```

```
void make_type_IDs_table(void) {
    type_ID_enlarge_types_table(INITIAL_ID_TABLE_SIZE);
    type_ID_set_source_name(UNKNOWN, "UNKNOWN");
    type_ID_set_description(UNKNOWN, "something unknown");
    type_ID_permit_pair(UNKNOWN, UNKNOWN,
        PARSED_SPCONTEXT + TYPE_PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + COMPILED_SPCONTEXT);
    create_MATCHING_species();
    create_VALUE_species();
    create_STORAGE_species();
    create_CONDITION_species();
    create_COMMAND_species();
    data_type_initialise_type_interpreter();
}
```

The function make_type_IDs_table is invoked by a command in a `.i6t` template file.

§**2.** We want to store the table of details about type IDs as an array, not a linked list, for speed of access. But the size of this array has to increase as new type IDs are added – while main types and specific types are fixed, new data type IDs are added often, and we don't want to impose any upper limit on how many the user can make.

```
int extent_of_type_tables = 0;                                   call this number X
type_id_rules *type_IDs_table = NULL;                            1-dimensional array of X entries
type_pair_rules *table_of_observations = NULL;                   2-dimensional array of X^2 entries
```

§**3.** Accessing a two-dimensional array of a non-constant size means that we have to perform a calculation to look up the entry we want:

```
type_pair_rules *type_ID_pair_rules(int r, int s) {
    if (table_of_observations == NULL) internal_error("no table of observations");
    return &(table_of_observations[r*extent_of_type_tables + s]);
}
```

§**4.** So the table is dynamically allocated and expanded as need be, doubling in size whenever we break the bounds of the old one.

```
int type_ID_claim_new_ID(void) {
    if (next_free_data_type_ID >= extent_of_type_tables)
        type_ID_enlarge_types_table(2*extent_of_type_tables);
    return next_free_data_type_ID++;
}
```

The function type_ID_claim_new_ID is called from 7/data.

§5. Expanding the types table. The following enlarges the table, possibly from 0 entries (in which case the arrays are both NULL), possibly from an already positive size (in which case we migrate the existing entries to ensure their survival).

```
void type_ID_enlarge_types_table(int to_size) {
    int from_size = extent_of_type_tables;
    type_id_rules *tt;                                    the new larger type ID table
    type_pair_rules *tc;                                  the new larger table of observations
    ⟨Allocate new larger tables for the type IDs 6⟩;
    ⟨Transfer the contents of the existing tables to the new larger ones 7⟩;
    ⟨Initialise the new cells in the now-enlarged tables 8⟩;
    ⟨Free the memory holding the current type ID tables 9⟩;
    ⟨Make the new larger tables of type IDs the current ones 10⟩;
}
```

§6.

⟨Allocate new larger tables for the type IDs 6⟩ ≡

```
    if (to_size <= from_size)
        internal_error("type tables not increased in size");
    tt = (type_id_rules *)
        (I7_calloc(to_size, sizeof(type_id_rules), TYPE_TABLES_MREASON));
    tc = (type_pair_rules *)
        (I7_calloc(to_size*to_size, sizeof(type_pair_rules), TYPE_TABLES_MREASON));
```

This code is used in §5.

§7.

⟨Transfer the contents of the existing tables to the new larger ones 7⟩ ≡

```
    int i, j;
    for (i=0; i<from_size; i++)
        tt[i] = type_IDs_table[i];
    for (i=0; i<from_size; i++)
        for (j=0; j<from_size; j++)
            tc[i*to_size + j]
                = table_of_observations[i*from_size + j];
```

This code is used in §5.

§8.

⟨Initialise the new cells in the now-enlarged tables 8⟩ ≡

```
    int i, j;
    for (i=from_size; i<to_size; i++)
        type_ID_details_initialise(&(tt[i]), i);
    for (i=0; i<to_size; i++)
        for (j=0; j<to_size; j++)
            if ((i>=from_size) || (j>=from_size))
                tpr_initialise(&(tc[i*to_size + j]));
```

This code is used in §5.

§**9.** This is used when expanding the table, but not at the end of the run; our automatic memory manager takes care of deallocating the table then.

⟨Free the memory holding the current type ID tables 9⟩ ≡
```
    if (from_size > 0) {
        I7_free(type_IDs_table, TYPE_TABLES_MREASON);
        I7_free(table_of_observations, TYPE_TABLES_MREASON);
    }
```
This code is used in §5.

§**10.**

⟨Make the new larger tables of type IDs the current ones 10⟩ ≡
```
    type_IDs_table = tt;
    table_of_observations = tc;
    extent_of_type_tables = to_size;
```
This code is used in §5.

§**11.** Slots in the table are initialised (immediately after allocation) by the following:

```
void type_ID_details_initialise(type_id_rules *tidr, int i) {
    tidr->type_number = i;
    tidr->type_id_flags = 0;
    sprintf(tidr->numerical_source_name, "<type-%d>", i);
    tidr->source_name = tidr->numerical_source_name;
    tidr->description = "<a type which should not exist>";
    tidr->parsing_name = NULL;
    tidr->parsing_plural = NULL;
    tidr->word_ref1 = -1; tidr->word_ref2 = -1;
    tidr->plural_ref1 = -1; tidr->plural_ref2 = -1;
    tidr->no_possible_specific_types = 0;
    tidr->main_type_for_which_this_is_specific = UNKNOWN;
    tidr->data_type_rules = NULL;
    tidr->kova_of_this = NULL;
}
void tpr_initialise(type_pair_rules *tpr) {
    int i;
    tpr->legal_contexts = 0;
    for (i=0; i<NO_SPEC_CONTEXTS; i++) tpr->observations[i] = NULL;
}
```

§**12. Completing the built-in types.** Once the data type interpreter has finished creating data types from the template file `Types.i6t`, the built-in stock of type IDs is complete. (Other type IDs will probably arrive later as a result of the source text – by the built-in ones are done.)

That provides a useful opportunity to register single-word type names with the vocabulary banks so that they can be parsed in an efficient way.

```
void complete_type_IDs_table(void) {
    int i;
    for (i=0; i<extent_of_type_tables; i++) {
        if (type_IDs_table[i].parsing_name) {
            vocab_set_flags(type_IDs_table[i].parsing_name, BUILTINTYPE_MC);
            vocab_set_literal_number_value(type_IDs_table[i].parsing_name, i);
        }
        if (type_IDs_table[i].parsing_plural) {
            vocab_set_flags(type_IDs_table[i].parsing_plural, BUILTINTYPE_MC);
            vocab_set_literal_number_value(type_IDs_table[i].parsing_plural, i);
        }
    }
}
```

The function complete_type_IDs_table is invoked by a command in a `.i6t` template file.

§**13. Access to the table.** We can now assume that the table of type IDs exists, and we need to provide facilities to the rest of Inform, answering questions that anyone might have. Most basic of all:

```
int type_ID_exists(int t) {
    if ((t < 0) || (t >= next_free_data_type_ID)) return FALSE;
    return TRUE;
}
```

The function type_ID_exists is called from 7/spec.

§**14.** Note that `UNKNOWN` passes the `type_ID_is_family_ID` test, since it has exactly one possible species: `UNKNOWN`. Otherwise, the only cases which pass ought to be the `*_FMY` constants.

```
int type_ID_is_family_ID(int t) {
    if ((type_ID_exists(t)) && (type_IDs_table[t].no_possible_specific_types > 0))
        return TRUE;
    return FALSE;
}
```

The function type_ID_is_family_ID is called from 7/specv.

§**15.** Here `UNKNOWN` returns `UNKNOWN`, and otherwise t returns a value other than `UNKNOWN` if and only if it is a species ID.

```
int type_ID_get_family_of_species(int t) {
    if (type_ID_exists(t)) return type_IDs_table[t].main_type_for_which_this_is_specific;
    return UNKNOWN;
}
```

The function type_ID_get_family_of_species is called from 7/specv.

§**16.** The flags:

```
void type_ID_set_flag(int t, int f) {
    if (type_ID_exists(t)) type_IDs_table[t].type_id_flags |= f;
    else internal_error("tried to set type ID flags for nonexistent type ID");
}
int type_ID_test_flag(int t, int f) {
    if (type_ID_exists(t) == FALSE) return FALSE;
    if ((type_IDs_table[t].type_id_flags) & f) return TRUE;
    return FALSE;
}
```

The function type_ID_set_flag is called from 7/masp, 7/vasp, 7/stsp, 7/cosp and 7/cmsp.
The function type_ID_test_flag is called from 7/spec.

§**17. KOVA cache.** This is managed by the "Kinds of Value" section, and explained there.

```
kind_of_value **type_ID_get_kova_cache_location(int t) {
    if (type_ID_exists(t)) return &(type_IDs_table[t].kova_of_this);
    return NULL;
}
```

The function type_ID_get_kova_cache_location is called from 7/kov.

§**18. Source names.** Source names at first look like only internal conveniences. We want the debugging log to be legible and not to say what's going on in terms of ID numbers we would have to memorise, so we print text like `NUMBER_TY` rather than, oh, `68`. In fact, though, this is more than a debugging convenience, because certain I6 runtime activities need access to data type IDs too, and so we define them all as I6 constants with the same values in the I6 compiler run as they have now in the C run compiling NI. We also want to use them as constant names in the data type minilanguage, so we need to be able to convert back from source name to its ID number, too.

```
void log_data_type_ID(int id) {
    type_ID_write_source_name_to_file(dl, id);
}
void type_ID_write_source_name_to_file(OUTPUT_STREAM, int id) {
    if ((id <= 0) || (id >= next_free_data_type_ID)) {
        WRITE("ILLEGAL-SUBTYPE-%d", id);
    } else if (id == 1) WRITE("UNKNOWN");
    else WRITE("%s", type_IDs_table[id].source_name);
}
```

The function log_data_type_ID is called from 2/dl.
The function type_ID_write_source_name_to_file is called from 7/kov.

§**19.**

```
void type_ID_set_source_name(int id, char *d) {
    type_IDs_table[id].source_name = d;
}
char *type_ID_get_source_name(int id) {
    return type_IDs_table[id].source_name;
}
```

The function type_ID_set_source_name is called from 7/dti, 7/masp, 7/vasp, 7/stsp, 7/cosp and 7/cmsp.
The function type_ID_get_source_name is called from 7/kov and 9/rsdt.

§**20.** And logging:

```
void log_data_type_name(int t) {
    if (type_IDs_table[t].word_ref1 >= 0) {
        log_lowercase_dtid(t);
    } else {
        if (type_IDs_table[t].parsing_name != NULL)
            LOG("%s", vocab_get_exemplar(type_IDs_table[t].parsing_name, FALSE));
        else data_type_print_long_name(dl, t);
    }
}
```

The function log_data_type_name is called from 7/kov and 7/dim.

§**21.** Note that these routines assume that an invalid TSID can be detected by having a source name begin with an angle bracket: the one from `<type-N>` written in the initialisation code above.

```
int type_ID_get_ID_for_source_name(char *sn) {
    int t;
    for (t=1; t<next_free_data_type_ID; t++) {
        char *p = type_IDs_table[t].source_name;
        if ((p == NULL) || (p[0] == '<')) continue;
        if (strcmp(p, sn) == 0) return t;
    }
    return UNKNOWN;
}
void compile_I6_constants_for_typenames(OUTPUT_STREAM) {
    int t;
    for (t=1; t<next_free_data_type_ID; t++) {
        char *p = type_IDs_table[t].source_name;
        if (p[0] == '<') continue;
        if (t == UNKNOWN) WRITE("Constant UNKNOWN_TY = %d;\n", UNKNOWN);
        else WRITE("Constant %s = %d;\n", p, t);
    }
}
```

The function type_ID_get_ID_for_source_name is called from 7/dti and 12/cinv.
The function compile_I6_constants_for_typenames is invoked by a command in a `.i6t` template file.

§**22. The description of a type.** Each type ID has associated with it a textual description suitable for use in messages for the end user: so, not cryptic debugging-log-speak like `(G)STORAGE_FMY`, but text such as "a table entry".

```
void type_ID_set_description(int t, char *d) {
    type_IDs_table[t].description = d;
}
char *type_ID_get_description(int t) {
    return type_IDs_table[t].description;
}
```

The function type_ID_set_description is called from 7/dti, 7/masp, 7/vasp, 7/stsp, 7/cosp and 7/cmsp.
The function type_ID_get_description is called from 2/index and 7/kov.

§23. Single integers. This is a pretty foolish dodge. Registered excerpt meanings carry with them pointers to structures embodying their meanings. But for names of KOVs (see below) the only meaning we want is a single integer – because this acts as a key into the table of type IDs. It might seem more natural to register KOV names with a pointer to the table entry itself. But that's not safe, because the table moves around when being reallocated as the number of KOVs grows – this was discovered, er, empirically.

```
int sint_get(single_integer *sint) {
    if (sint == NULL) return 0;
    return sint->the_integer;
}
single_integer *sint_new(int t) {
    single_integer *sint = CREATE(single_integer);
    sint->the_integer = t;
    return sint;
}
```

The function sint_get is called from 5/candd.

§24. Names used in parsing. The type ID rules structure provides two alternative ways to specify the name of a type: as a word-range pair, used for types created in the main text, and as a single word, used for types built in. (Though the data type interpreter is able to make longer names if need be, the single word names are the most efficient to parse quickly, which is worth it for the commonly occurring type names.)

Because `type_ID_set_name` is used only when new KOVs are created in the source text, and none of the built-in types are made that way, any such textual name is registered with the `DESIGNED_TYPE_MC` meaning code. Because such names are registered with pointers to the relevant `type_id_rules` structure, whereas we want to deal with type IDs by their numbers, we need to provide a routine to translate back.

```
void type_ID_copy_name(int t, int *w1, int *w2) {
    *w1 = type_IDs_table[t].word_ref1;
    *w2 = type_IDs_table[t].word_ref2;
}
void type_ID_set_name(int t, int w1, int w2) {
    single_integer *sint = sint_new(t);
    type_IDs_table[t].word_ref1 = w1;
    type_IDs_table[t].word_ref2 = w2;
    register_excerpt_meaning(DESIGNED_TYPE_MC, 0, w1, w2,
        STORE_POINTER_single_integer(sint));
    make_plural_of(w1, w2, NULL);
    type_IDs_table[t].plural_ref1 = plw1;
    type_IDs_table[t].plural_ref2 = plw2;
    register_excerpt_meaning(DESIGNED_TYPE_MC, 0, plw1, plw2,
        STORE_POINTER_single_integer(sint));
}
vocabulary_entry *type_ID_get_singular_parsing_name(int t) {
    return type_IDs_table[t].parsing_name;
}
void type_ID_set_parsing_names(int t,
    vocabulary_entry *singular, vocabulary_entry *plural) {
    if (singular) type_IDs_table[t].parsing_name = singular;
    if (plural) type_IDs_table[t].parsing_plural = plural;
}
```

The function type_ID_copy_name is called from 2/index, 4/edoc, 7/kov and 7/data.
The function type_ID_set_name is called from 7/data.
The function type_ID_get_singular_parsing_name is called from 7/kov and 7/data.
The function type_ID_set_parsing_names is called from 7/dti, 7/stsp, 7/cosp and 7/cmsp.

§**25. Interface with data types.** Those IDs used to identify data types (in atomic KOVs) have far more information associated with them than main types or specific types have, but this information is stored in the form of a pointer to a structure managed in the "Data Types.w" section. Here we only manage the pointers connecting the type IDs table to these further structures.

```
int type_ID_is_data_type(int t) {
    if ((t<=0) || (t>=next_free_data_type_ID)) return FALSE;
    if (type_IDs_table[t].data_type_rules) return TRUE;
    return FALSE;
}
void type_ID_make_into_data_type(int t, data_type_id_rules *dtid) {
    if ((t<=0) || (t>=next_free_data_type_ID))
        internal_error("tried to register type id out of range");
    if (type_IDs_table[t].data_type_rules)
        internal_error("data type doubly registered");
    type_IDs_table[t].data_type_rules = dtid;
}
data_type_id_rules *type_ID_get_dtid(int t) {
    if ((t<=0) || (t>=next_free_data_type_ID)) internal_error("type id out of range");
    if (type_IDs_table[t].data_type_rules == NULL)
        internal_error("type id unregistered as data type");
    return type_IDs_table[t].data_type_rules;
}
data_type_id_rules *type_ID_get_dtid_or_null(int t) {
    if ((t<=0) || (t>=next_free_data_type_ID)) internal_error("type id out of range");
    return type_IDs_table[t].data_type_rules;
}
int type_ID_with_this_dtid(data_type_id_rules *dtid) {
    int t;
    if (dtid == NULL) internal_error("null dtid");
    for (t=1; t<next_free_data_type_ID; t++)
        if (type_IDs_table[t].data_type_rules == dtid)
            return t;
    return UNKNOWN;
}
```

The function type_ID_is_data_type is called from 5/candd, 7/kov and 7/specv.
The function type_ID_make_into_data_type is called from 7/data.
The function type_ID_get_dtid is called from 7/data.
The function type_ID_get_dtid_or_null is called from 7/data.
The function type_ID_with_this_dtid is called from 7/data.

§26. Checking SP invariants. NI contains detailed code to check that no improperly built SP lasts long inside it without an internal error being produced: so while the routines here have no practical effect on the work done by NI (and testing shows that they make no appreciable difference to speed), they do offer some assurance that many complex mechanisms in NI are at least getting some things right. Recall that there are a number of named contexts in which a SP can be used in NI: here, we give these names which can be used in any internal error messages which should turn up.

```
char *name_of_spec_context(int context) {
    switch(context) {
        case TYPE_EXPECTED_SPCONTEXT: return "type expected";
        case TYPE_FOUND_SPCONTEXT: return "type found";
        case COMPILED_SPCONTEXT: return "value compiled";
        case PARSED_SPCONTEXT: return "value parsed";
        case TYPE_PARSED_SPCONTEXT: return "type parsed";
        default: return "miscellaneous";
    }
}
void type_ID_permit_pair(int a, int b, int map) {
    type_ID_pair_rules(a, b)->legal_contexts = map;
    type_IDs_table[a].no_possible_specific_types++;
    type_IDs_table[b].main_type_for_which_this_is_specific = a;
}
```

The function type_ID_permit_pair is called from 7/masp, 7/vasp, 7/stsp, 7/cosp and 7/cmsp.

§**27.** "Observation" of a SP involves noting its first appearance, if any, and causing an internal error if it appears to be in the wrong place (or simply has an illegal pair of type IDs).

```
int observed_before = FALSE;
void type_ID_observe(specification *spec, int context) {
    int fmy = spec_get_family(spec), spc = spec_get_species(spec);
    int i, j, k, ix;
    ix=0; j=context; while (j != 1) { ix++; j = j/2; }
    if ((fmy <= 0) || (fmy >= 50) ||
        (spc <= 0) || (spc >= next_free_data_type_ID)) {
        LOG("Illegal type ID (context %d): %d/%d\n", context, fmy, spc);
        LOG("Error on: $X", spec);
        internal_error("type ID out of legal range");
    }
    if ((type_ID_pair_rules(fmy, spc)->legal_contexts
        & context) == 0) {
        LOG("SP ID combination illegal in context %s: %s/%s\n",
            name_of_spec_context(context),
            type_IDs_table[fmy].source_name,
            type_IDs_table[spc].source_name);
        LOG("Error on: $X (allows %04x)\n", spec,
            type_ID_pair_rules(fmy, spc)->legal_contexts);
        internal_error("Illegal SP ID combination");
    }
    if (observed_before == FALSE) {
        observed_before = TRUE;
        for (i=0; i<extent_of_type_tables; i++)
            for (j=0; j<extent_of_type_tables; j++)
```

```
                for (k=0; k<NO_SPEC_CONTEXTS; k++)
                    type_ID_pair_rules(i, j)->observations[k] = NULL;
    }
    type_ID_pair_rules(fmy, spc)->observations[ix] = spec;
}
```

The function type_ID_observe is called from 7/ttts, 7/cfsp and 7/tc.

§**28.** At the end of a run, we sometimes write a note about the specifications actually used into the debugging log. This is particularly useful if we need to find out whether certain combinations of family and species are ever observed in the wild, which we might do, for instance, with the following command:

```
intest.pl --compile-only --log=type-usage '--keep-log=SP used' all
```

...which concatenates the report from the following routine over all of the hundreds of test runs available. The resulting accumulated debugging log can then be passed through `sort`: alternatively, we could look for specific cases thus:

```
intest.pl --compile-only --log=type-usage '--keep-log=SP used: CONDITION' all
```

```
void report_pairs_observed(void) {
    int i, j, k, context;
    if (dl_this(TYPE_USAGE_DA) == FALSE) return;
    for (i=0; i<extent_of_type_tables; i++)
        for (j=0; j<extent_of_type_tables; j++)
            for (k=0, context=1; k<NO_SPEC_CONTEXTS; k++, context = context*2) {
                if (type_ID_pair_rules(i, j)->observations[k]) {
                    LOG("SP used: %s/%s in %s  ",
                        type_IDs_table[i].source_name,
                        type_IDs_table[j].source_name,
                        name_of_spec_context(context));
                    if ((type_ID_pair_rules(i, j)->legal_contexts
                        & context) == 0)
                        LOG("!!!!  ");
                    LOG("$S\n", type_ID_pair_rules(i, j)->observations[k]);
                }
            }
}
```

The function report_pairs_observed is invoked by a command in a `.i6t` template file.

§**29.** Similarly, we can write out a systematic picture of the legal type ID combinations into the debugging log.

```
void report_pairs_allowed(void) {
    int i, j;
    if (dl_this(TYPE_PERMISSIONS_DA) == FALSE) return;
    for (i=0; i<extent_of_type_tables; i++) {
        int flag = FALSE;
        for (j=0; j<extent_of_type_tables; j++)
            if (type_ID_pair_rules(i, j)->legal_contexts) {
                if (flag == FALSE) {
                    LOG("%s - family with specific types:\n",
                        type_IDs_table[i].source_name);
                    flag = TRUE;
                }
                LOG("    %s\n", type_IDs_table[j].source_name);
            }
        flag = FALSE;
        for (j=0; j<extent_of_type_tables; j++)
            if (type_ID_pair_rules(j, i)->legal_contexts) {
                if (flag == FALSE) {
                    LOG("%s - species of",
                        type_IDs_table[i].source_name);
                    flag = TRUE;
                }
                LOG(" %s", type_IDs_table[j].source_name);
            }
        if (flag) { LOG("\n"); }
    }
}
```

The function report_pairs_allowed is invoked by a command in a `.i6t` template file.

Kinds of Value 7/kov

Purpose

To manage the `kind_of_value` structure, representing the possible categories into which eventual I6 values can be placed; to determine which KOVs contain which others, and what their characteristics are.

Definitions

¶1. As we have seen, "type" is an enormous category inside Inform, taking in actual data as well as general descriptions of data, and actions and phrases as well as data. "Kind of value" is a much smaller category, and more like the idea of "type" in a C-like language.

The `kind_of_value` structure is always referred to by means of pointers, so that in practice a KOV is described inside Inform using a `kind_of_value *`. A `NULL` pointer means that the KOV is unknown. The structure pointed to unifies the four ways that KOVs are formed:

(i) Each data type ID, such as `NUMBER_TY`, makes an atomic KOV (a "KOVA"), such as "number".
(ii) Each kind of object, such as "vehicle", makes a KOVKO (a "KOV which is also a kind of object"). Note that "object" itself is a KOVA, not a KOVKO.
(iii) Each kind constructor, such as "list of", makes a new KOVCON out of existing ones: for instance "list of numbers".
(iv) Each intermediate result, partway through an arithmetic calculation, which can't be represented as a KOVA, is given a specially made KOVIV.

¶2. KOVs are static structures. Once created, their fields are never touched. It would be neat if there were exactly one KOV structure somewhere in memory for each different kind of value – and that is true for KOVAs and KOVKOs, but is not true for KOVCONs or KOVIVs, because it would be too much trouble to arrange. (In practice KOVCONs and KOVIVs are rarely needed, so even though they cause duplication, the amount of memory wasted is typically only about 2K.)

Therefore to see if two pointers `kov1` and `kov2` represent the same KOV, it's not good enough to test `kov1 == kov2`; we have to use the `kov_compare` routine.

define `MAX_KC_ARGS 1` *at present the kind constructors are unary*

```
typedef struct kind_of_value {
    int atomic_kov;
    struct world_object *atomic_kind;
    struct unit_sequence *intermediate_result;
    struct kind_of_value *kc_args[MAX_KC_ARGS];
} kind_of_value;
```

The structure kind_of_value is private to this section.

¶3. We keep track of the newest-created KOVA here, along with some statistics:

```
kind_of_value *latest_atomic_kov = NULL;
int no_kovas_created = 0;
int no_kovkos_created = 0;
int no_kovivs_created = 0;
int no_kovcons_created = 0;
```

---

§**1.** The four fundamental constructions. All KOV structures are obtained by one of the following. First, `kova(D)` returns the KOV for a value of data type ID `D`. This is something we need very often, so we create it only on the first request, and cache the pointer to it in the table of data about `D`; we can then use that same pointer on all subsequent requests.

Note that `kova(NUMBER_TY)` doesn't create the idea of "number", or bring the data type into existence, or anything like that: it simply returns a token which indicates that a number is meant rather than, say, a time.

```
kind_of_value *kova(int data_type_id) {
    if (data_type_id == UNKNOWN) internal_error("made unknown as atomic kova");
    kind_of_value **cache = type_ID_get_kova_cache_location(data_type_id);
    if (cache) { if (*cache) return *cache; }
    kind_of_value *kov;
    ⟨Create a raw KOV structure 5⟩;
    kov->atomic_kov = data_type_id;
    if (cache) *cache = kov;
    no_kovas_created++;
    return kov;
}
```

The function kova is called from 2/mem, 5/bp, 5/rel, 5/aph, 5/lit, 5/candd, 5/mlc, 6/simp, 6/tcpr, 6/asp, 6/cind, 7/data, 7/dim, 9/rsdt, 7/kix, 7/specv, 7/vasp, 7/stsp, 7/cmsp, 7/tc, 8/refpt, 8/creat, 8/mass, 8/knowp, 9/qty, 9/prop, 9/cmpbp, 10/qnbp, 10/list, 10/tab, 10/eqns, 10/libp, 10/bib, 11/act, 11/ap, 11/los, 11/av, 12/ph, 12/phtd, 12/phsf, 12/cinv, 13/gtok and 13/gprv.

§**2.** Second, `kovko(K)` returns the KOV for an object whose kind is `K`. Once again this is cached, so that each time we ask for `kovko(K)` for the same `K` we will get a pointer to the same structure.

```
kind_of_value *kovko(world_object *kind) {
    if (kind == NULL) internal_error("made null as atomic kovko");
    kind_of_value **cache = wo_get_kova_cache_location(kind);
    if (cache) { if (*cache) return *cache; }
    kind_of_value *kov;
    ⟨Create a raw KOV structure 5⟩;
    kov->atomic_kov = OBJECT_TY;
    kov->atomic_kind = kind;
    if (cache) *cache = kov;
    no_kovkos_created++;
    return kov;
}
```

The function kovko is called from 5/aph, 5/candd, 5/mlc, 6/aprop, 6/treec, 6/simp, 6/asp, 7/vasp, 7/tc, 8/creat, 8/mass, 9/mapbp, 9/pp, 10/list, 11/act, 11/av, 12/phtd, 12/stv and 12/rb.

§**3.** Third, `kovcon(C, K)` performs construction `C` on the existing kind of value `K` to produce a more elaborate one: for instance, `kovcon(LIST_OF_TY, kova(NUMBER_TY))` produces a KOV structure meaning "list of numbers". This is not cached anywhere, so a second request for the same thing will produce a different copy of the KOV, though with identical contents.

```
kind_of_value *kovcon(int constructor, kind_of_value *base) {
    kind_of_value *kov;
    ⟨Create a raw KOV structure 5⟩;
    kov->atomic_kov = constructor;
    kov->kc_args[0] = base;
    no_kovcons_created++;
    return kov;
}
```

The function kovcon is called from 5/candd and 10/list.

§**4.** Fourth, `koviv(US)` produces a KOV for the intermediate result given by the unit sequence US. Unit sequences are explained in "Dimensions", but for instance one might represent "area times pressure divided by velocity". Again, these are not cached.

```
kind_of_value *koviv(unit_sequence *ik) {
    if (ik == NULL) internal_error("made unknown as koviv");
    kind_of_value *kov;
    ⟨Create a raw KOV structure 5⟩;
    kov->atomic_kov = INTERMEDIATE_TY;
    kov->intermediate_result = CREATE(unit_sequence);
    *(kov->intermediate_result) = *ik;
    no_kovivs_created++;
    return kov;
}
```

The function koviv is called from 7/dim.

§**5.** These four constructions, and these four alone, have the power to create KOV structures, like so:

⟨Create a raw KOV structure 5⟩ ≡

```
    kov = CREATE(kind_of_value);
    kov->atomic_kov = UNKNOWN;
    kov->atomic_kind = NULL;
    kov->intermediate_result = NULL;
    int i;
    for (i=0; i<MAX_KC_ARGS; i++) kov->kc_args[i] = NULL;
```

This code is used in §1,2,3,4,1,2,3,4,1,2,3,4.

§**6. Comparison.** The following determines whether or not two KOV pointers represent the same kind of value.

```
int kov_compare(kind_of_value *k1, kind_of_value *k2) {
    int i;
    if (k1 == NULL) { if (k2 == NULL) return TRUE; return FALSE; }
    if (k2 == NULL) return FALSE;
    if (k1->atomic_kov != k2->atomic_kov) return FALSE;
    if (k1->atomic_kind != k2->atomic_kind) return FALSE;
    if ((k1->intermediate_result) && (k2->intermediate_result == NULL)) return FALSE;
    if ((k1->intermediate_result == NULL) && (k2->intermediate_result)) return FALSE;
    if ((k1->intermediate_result) &&
        (compare_unit_sequences(k1->intermediate_result, k2->intermediate_result)
            == FALSE)) return FALSE;
    for (i=0; i<MAX_KC_ARGS; i++)
        if (kov_compare(k1->kc_args[i], k2->kc_args[i]) == FALSE)
            return FALSE;
    return TRUE;
}
```

The function kov_compare is called from 5/aph, 5/litp, 7/dim, 9/rsdt, 7/kix, 7/vasp, 8/mass, 9/qty, 9/prop, 9/pp, 9/inf, 10/tab, 10/eqns, 11/ap and 12/cinv.

§**7. Quicker tests.** Three short forms. Firstly, if `D` is a data type other than `OBJECT_TY` then `is_kova(K, D)` is a quick way to test `kov_compare(K, kova(D))`, a test we need to perform often.

```
int is_kova(kind_of_value *kov, int data_type_id) {
    if (kov == NULL) return FALSE;
    if (kov->atomic_kov == data_type_id) return TRUE;
    return FALSE;
}
```

The function is_kova is called from 2/prob2, 4/iext, 5/rel, 5/aph, 5/lit, 5/tandv, 6/treec, 6/simp, 6/tcpr, 6/equal, 6/asp, 6/defer, 7/dim, 7/specv, 7/vasp, 7/stsp, 7/cmsp, 7/tc, 8/refpt, 8/creat, 8/mass, 9/qty, 9/prop, 9/pp, 9/provr, 9/inf, 10/list, 10/tab, 10/eqns, 10/bib, 10/fig, 10/sfx, 10/isin, 11/act, 11/ap, 11/av, 12/phud, 12/phtd, 12/cinv, 12/rb, 13/tfg, 13/gl, 13/gtok and 13/gprv.

§**8.** Now in principle `kova(D)` can be created for any type ID `D`, and not just data types, though this will generally only happen when problems have occurred. A careful test of a genuine KOVA, in such tricky cases, can be performed like this:

```
int kov_represents_data(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    if (kov->atomic_kind) return TRUE;
    if (data_type_is_constructor(kov->atomic_kov)) return TRUE;
    if (type_ID_is_data_type(kov->atomic_kov)) return TRUE;
    return FALSE;
}
```

The function kov_represents_data is called from 8/creat.

§**9.** `K` is a KOVKO if and only if `kovko_get_kind(K)` returns a non-`NULL` pointer. Note that `is_kova(K, OBJECT_TY)` is not quite enough, because `kova(OBJECT_TY)`, representing the KOV "object", is strictly speaking a KOVA and not a KOVKO.

```
world_object *kovko_get_kind(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    if (kov->atomic_kov != OBJECT_TY) return NULL;
    return kov->atomic_kind;
}
```

The function kovko_get_kind is called from 2/index, 2/prob2, 5/aph, 6/aprop, 6/vars, 6/treec, 6/simp, 6/asp, 6/atoms, 8/knowp, 9/model and 10/list.

§**10.** KOVCONs are easy to detect:

```
int is_kovcon(kind_of_value *kov) {
    if ((kov) && (data_type_is_constructor(kov->atomic_kov))) return TRUE;
    return FALSE;
}
```

The function is_kovcon is called from 7/dim, 7/vasp, 7/tc, 10/tab and 11/act.

§**11.** Given, say, "list of numbers", the following returns "number"...

```
kind_of_value *kovcon_get_base(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    return kov->kc_args[0];
}
```

The function kovcon_get_base is called from 7/stsp, 7/tc, 10/tab and 12/cinv.

§**12.** ...and the following returns `LIST_OF_TY`:

```
int kov_I6_construct_ID(kind_of_value *kov) {
    if (kov == NULL) return 0;
    if (data_type_is_constructor(kov->atomic_kov) == FALSE) return 0;
    return kov_I6_ID(kov->kc_args[0]);
}
```

The function kov_I6_construct_ID is called from 9/rsdt and 10/tab.

§**13. Weaker integer representations of a KOV.** Sometimes a KOV has to be stored as an I6 integer value at run-time. I6 is typeless, so some of the routines and data structures in the I6 template need these integer values to tell them what they are looking at. For instance, the `ActionData` table records the KOVs of the noun and second noun to which an action applies.

Clearly a single 16-bit integer can't faithfully represent the full range of KOVs. We could get closer to this if we used a trick like the one attributed to Ritchie and Johnson in chapter 6.3 of the Dragon book (Aho, Sethi and Ullman, *Compilers*, 1986), where lower bits of a word store `atomic_kov` for the underlying data and upper bits record KOV constructors applied to this.

But it turns out to be more fruitful to store the `atomic_kov`, thus recording a deliberately weaker version of the KOV. Any two lists look the same, because at run-time each is stored as `LIST_OF_TY`. All KOVKOs look the same – they all merge together as `OBJECT_TY`. This is positively useful, because –

Dogma. If a value v has KOV K, and we want to use it as a value of KOV W, and they are not KOVIVs, then

(a) if the `kov_I6_ID` of K is different from that of W then this is impossible;
(b) if the `kov_I6_ID` of K is the same as that of W then run-time code can tell from v alone whether this is possible.

For instance, if we know that the `kov_I6_ID` of v is `OBJECT_TY` then we can tell if it's a "vehicle" or not by testing the I6 condition `(v ofclass K27_vehicle)`. And if v is a `LIST_OF_TY` then it points to a block of data on the stack which includes the `kov_I6_ID` of the entries in the list.

It does not matter that KOVIVs do not conform to Dogma, because they are made to order, and are never assigned to storage objects like variables. That's what makes them intermediate.

```
int kov_I6_ID(kind_of_value *kov) {
    if (kov == NULL) return UNKNOWN;
    return kov->atomic_kov;
}
```

The function kov_I6_ID is called from 5/aph, 6/sch, 9/rsdt, 7/stsp, 9/qty, 9/prop, 10/list, 10/tab, 11/act, 12/cinv and 13/gprv.

§**14. Dimensions and scaling.** In theory, our polymorphic system of arithmetic allows us to add, multiply, etc., any KOVs according to rules provided in the source text. In fact, though, this only applies to atomic data types, and the dimensions system uses the following routine to pierce through the veil of the KOV and see the data type beneath. (Returning `UNKNOWN` implies that arithmetic cannot legally be applied to this KOV.) See "Dimensions" for much more on this.

```
int kov_get_dimensional_data_type(kind_of_value *kov) {
    if (kov == NULL) return UNKNOWN;
    return kov->atomic_kov;
}
int kov_is_dimensionless(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_dimensionless(kov->atomic_kov);
}
unit_sequence *kov_get_dimensional_form(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    if (kov_quasinumerical(kov) == FALSE) return NULL;
    if (kov->atomic_kov == INTERMEDIATE_TY)
        return kov->intermediate_result;
    return data_type_get_unit_sequence(kov->atomic_kov);
}
int kov_scale_factor(kind_of_value *kov) {
    if (kov == NULL) return 1;
```

```
    if (kov->intermediate_result) return us_get_scaling_factor(kov->intermediate_result);
    return data_type_scale_factor(kov->atomic_kov);
}
```

The function kov_get_dimensional_data_type is called from 7/dim.
The function kov_is_dimensionless is called from 7/dim and 10/eqns.
The function kov_get_dimensional_form is called from 7/dim.
The function kov_scale_factor is called from 7/dim and 13/test.

§**15. Compiling default values.** When we create a new variable of a given KOV, but never say what its value is to be, Inform initialises it to the default value for that KOV, and similarly for other storage objects.

The following should compile a default value for K, and return
(a) `TRUE` if it succeeded,
(b) `FALSE` if it failed (because K had no values or no default could be chosen), and no problem message has been issued about this, or
(c) `NOT_APPLICABLE` if it failed and issued a specific problem message.

```
int compile_default_value(OUTPUT_STREAM, kind_of_value *kov,
    int w1, int w2, char *storage_name) {
    if (kov_get_default_value_of_type(kov)) {
        WRITE("%s\n", kov_get_default_value_of_type(kov));
        return TRUE;
    }
    we need to issue a problem message: try to find an appropriate one –
    if (kovko_get_kind(kov))
        ⟨The kind must have no instances, or it would have worked 16⟩;
    if (is_kova(kov, ANY_VALUE_TY))
        ⟨"Value" is too vague to be the kind of a variable 17⟩;
    if (kov_can_be_stored_in_variables(kov) == FALSE)
        ⟨This is a KOV not intended for end users at all 18⟩;
    – but failing that, we leave it up to the caller to say something
    return FALSE;
}
```

The function compile_default_value is called from 9/rsdt, 9/qty, 9/pp, 10/tab, 12/phtd and 12/cinv.

§**16.**

⟨The kind must have no instances, or it would have worked 16⟩ ≡

```
    quote_words(1, w1, w2);
    quote_object(2, kovko_get_kind(kov));
    quote_text(3, storage_name);
    handmade_problem(_P_(C7EmptyKind));
    issue_problem_segment(
        "I am unable to put any value into the %3 '%1', "
        "which needs to be a %2, because there is no %2 in the world.");
    issue_problem_end();
    return NOT_APPLICABLE;
```

This code is used in §15.

§17.

⟨"Value" is too vague to be the kind of a variable 17⟩ ≡

```
quote_words(1, w1, w2);
handmade_problem(_P_(C7KindlessValue));
issue_problem_segment(
    "I am unable to start '%1' off with any value, because the "
    "instructions do not tell me what kind of value it should be "
    "(a number, a time, some text perhaps?).");
issue_problem_end();
return NOT_APPLICABLE;
```

This code is used in §15.

§18.

⟨This is a KOV not intended for end users at all 18⟩ ≡

```
quote_words(1, w1, w2);
quote_kov(2, kov);
handmade_problem(_P_(C7BadVarKind));
issue_problem_segment(
    "I am unable to create '%1' with the kind of value '%2', "
    "because this is a kind of value which is not allowed as "
    "something to be stored in properties, variables and the "
    "like. (See the Kinds index for which kinds of value "
    "are available. The ones which aren't available are really "
    "for internal use by Inform.)");
issue_problem_end();
return NOT_APPLICABLE;
```

This code is used in §15.

§19. **Logging KOVs.** As always, we want a way of representing this fundamental data structure in the debugging log:

```
void log_kind_of_value(kind_of_value *kov) {
    int i;
    if (kov == NULL) LOG("NULL-KOV");
    else {
        if (kov->atomic_kind) {
            LOG("$O", kov->atomic_kind);
        } else if (kov->intermediate_result) {
            LOG("["); log_unit_sequence(kov->intermediate_result); LOG("]");
        } else {
            log_data_type_name(kov->atomic_kov);
        }
        if (!(logging_to_I6_text)) LOG("-KOV");
        if ((kov->atomic_kov) && (data_type_is_constructor(kov->atomic_kov)))
            for (i=0; i<MAX_KC_ARGS; i++)
                LOG("($u)", kov->kc_args[i]);
    }
}
```

The function log_kind_of_value is called from 2/dl, 9/prop and 13/test.

§**20. Creating new KOVAs.** The following is called in response to sentences like "Texture is a kind of value", and allocates a new data type ID for the new name ("texture").

```
kind_of_value *make_new_atomic_kind_of_value(int w1, int w2) {
    PROTECTED_MODEL_PROCEDURE;
    int d = new_data_type(w1, w2);
    if (d < 0) return NULL;
    latest_atomic_kov = kova(d);
    return kova(d);
}
```

The function make_new_atomic_kind_of_value is called from 6/asp.

§**21. Wrapper routines.** The whole of the rest of this section consists of routines which answer questions about KOVs.

These almost always have the form:

```
int kov_silly_question(kind_of_value *kov) {
    handle the NULL case somehow
    do something interesting to handle KOVKOs if the kind of object matters
    return data_type_silly_question(kov->atomic_kov);
}
```

Very often the question is such that the same answer applies to all KOVKOs, so that nothing needs to be done in that case.

§**22. Questions about origin.** Some KOVs are built in (in that the template file "Types.i6t" creates them, using the type interpreter), while others arise from "X is a kind of value" sentences in the source text.

Note that a KOVKO counts as built-in by this test, even though it might be a kind of object created in the source text, because at the end of the day "object" is built in.

```
int kov_is_built_in(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_is_built_in(kov->atomic_kov);
}
```

The function kov_is_built_in is called from 5/litp and 10/tab.

§**23.** A few types support modifying adjectives when created. These adjectives are created in "Types.i6t" file, and parsed here; in an ideal world there would be none of them, and we would simply use ordinary Inform adjectives.

```
int kov_parse_modifying_adjectives(kind_of_value *kov, int w1, int w2) {
    if (kov == NULL) return -1;
    return data_type_parse_modifying_adjectives(kov->atomic_kov, w1, w2);
}
```

The function kov_parse_modifying_adjectives is called from 8/refpt.

§**24. Questions about constants and notation.** Some KOVs have named constants, others use a quasi-numerical notation: for instance "maroon" might be a named constant for KOV "colour", while "234 kilopascals" might be a notation for a KOV where constants are not named.

```
int kov_has_named_constant_values(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_has_named_constant_values(kov->atomic_kov);
}
```

The function kov_has_named_constant_values is called from 5/rel, 8/creat and 9/qty.

§**25.** The following returns the compilation method (a constant in the form `*_CCM`, defined in "Data Types.w") used when compiling an `(A)VALUE/CONSTANT` specification of this KOV: in other words, when compiling an I6 value for a constant of this kind.

```
int kov_get_constant_compilation_method(kind_of_value *kov) {
    if (kov == NULL) return NONE_CCM;
    return data_type_get_constant_compilation_method(kov->atomic_kov);
}
```

The function kov_get_constant_compilation_method is called from 7/vasp.

§**26.** On reading "5 feet 4 inches specifies a height", Inform parses "5 feet 4 inches" into a `literal_pattern` structure and then calls this routine to attach it to the KOV "height". (Multiple patterns can be attached to the same KOV, and they become alternative syntaxes.)

```
void kov_add_literal_pattern(kind_of_value *kov, literal_pattern *lp) {
    if (kov == NULL) internal_error("can't add LP to null KOV");
    data_type_add_literal_form(kov->atomic_kov, lp);
}
```

The function kov_add_literal_pattern is called from 5/litp.

§**27.** And here we find the list of such notations.

```
literal_pattern *kov_list_of_literal_forms(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    return data_type_list_of_literal_forms(kov->atomic_kov);
}
```

The function kov_list_of_literal_forms is called from 5/litp, 9/rsdt and 7/kix.

§**28. Questions about range and behaviour.** Unsigned comparisons are slower in the VM we're compiling for, so we use them only if we have to.

```
int kov_uses_signed_comparisons(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_uses_signed_comparisons(kov->atomic_kov);
}
```

The function kov_uses_signed_comparisons is called from 10/tab.

§**29.** See "Dimensions":

```
int kov_quasinumerical(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_is_quasinumerical(kov->atomic_kov);
}
```

The function kov_quasinumerical is called from 7/dim, 7/kix, 10/eqns and 13/test.

§**30.** See "Tables.i6t"; the issue is whether the value `IMPROBABLE_VALUE` can, despite its improbability, be valid for this KOV. If we can prove that it is not, we should return `FALSE`; if in any doubt, we must return `TRUE`.

```
int kov_requires_blanks_bitmap(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_requires_blanks_bitmap(kov->atomic_kov);
}
```

The function kov_requires_blanks_bitmap is called from 10/tab.

§**31.** The following is used only when the KOV has named constant values.

```
int kov_get_highest_valid_value_as_integer(kind_of_value *kov) {
    if (kov == NULL) return 0;
    return data_type_get_highest_valid_value_as_integer(kov->atomic_kov);
}
```

The function kov_get_highest_valid_value_as_integer is called from 5/rel and 9/rsdt.

§**32.** Is a variable allowed to have this KOV?

```
int kov_can_be_stored_in_variables(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_can_be_stored_in_variables(kov->atomic_kov);
}
```

The function kov_can_be_stored_in_variables is called from 7/kix and 12/phtd.

§**33.** Because of the way "let" variables are ambiguously defined – by initial value rather than by an explicit type description – it is not always clear what their KOV should be. When Inform reads:

```
let X be "look, it's [time of day]";
```

it sees that the initial value of X is a `kova(TEXT_ROUTINE_TY)`, but this is not permitted as the type of a "let" variable: it is upgraded to the more general `kova(TEXT_TY)`. The following routine takes the KOV K_I of the initial value and returns K_V, a suitable KOV for the "let" variable, or else `NULL` to say that the value cannot be stored in a "let". Provided K_V is not null, we always have $K_I \subseteq K_V$, so it is certain to be safe to store the initial value in the variable that results.

```
kind_of_value *kov_for_storage_to_hold(kind_of_value *kov) {
    int d;
    if (kov == NULL) return NULL;
    if (data_type_is_constructor(kov->atomic_kov)) return kov;
    if (kov->atomic_kov == INTERMEDIATE_TY) return kov;
    d = data_type_for_storage_to_hold(kov->atomic_kov);
    if (d == UNKNOWN) return NULL;
    return kova(d);
}
```

The function kov_for_storage_to_hold is called from 7/tc.

§**34. Questions about the default value.** This returns either valid I6 code for the value which is the default for K, or else `NULL` if K has no values, or no default can be chosen.

```
char *kov_get_default_value_of_type(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    if (kovko_get_kind(kov)) {
        world_object *wo, *kind_wo = kovko_get_kind(kov);
        LOOP_OVER(wo, world_object)
            if ((wo_of_kind(wo, kind_wo)) && (wo->kind_flag == FALSE))
                return wo_get_I6_representation(wo);
        if (kind_wo == kind_room)                    for roomless texts used to rerelease I6 story files
            return "nothing";
        return NULL;
    }
    if (kov_uses_pointer_values(kov)) return "0";          meaning: not allocated on heap yet
    if (kov_is_an_enumeration(kov)) {
        quantity *q2;
        LOOP_OVER(q2, quantity)
            if ((kov_compare(qty_kind_of_value(q2), kov))
                && (qty_is_a_variable(q2) == FALSE))
                return qty_get_I6_representation(q2);
        return NULL;
    }
    return data_type_get_default_value_of_type(kov->atomic_kov);
}
```

§**35.** The following is used in the Kinds index, in the table showing the default values for each kind:

```
char *kov_get_index_default_value(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    return data_type_get_index_default_value(kov->atomic_kov);
}
```

The function kov_get_index_default_value is called from 7/kix.

§**36. Questions about quantification.** Some KOVs are such that all legal values can efficiently be looped through at run-time, some are not: we can sensibly loop over all scenes, but not over all texts. We use the term "domain" to mean the set of values which a loop traverses.

```
int kov_compile_domain_possible(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    if (kovko_get_kind(kov)) return TRUE;
    return data_type_compile_domain_possible(kov->atomic_kov);
}
```

The function kov_compile_domain_possible is called from 6/defer and 7/kix.

§**37.** If the domain is a KOVKO, say "things", then we use a linked list compiled into the object tree for exactly this purpose. The list starts with the constant `IK2_First` (2 being the kind number for "things" in the usual arrangement), and then each instance of a thing has a property `IK2_Link` whose value is the next thing; this is `nothing` for the last in the list.

If it happens that no instances exist – unlikely for things, but often true of more unusual kinds – then a loop header which never executes the block following it is compiled.

If the domain `kov` is not a KOVKO, then we loop through the known constants which make up this kind of value; each KOV is allowed to provide its own loop syntax for this. For "time", for instance, it becomes a `for` loop running from 0 (midnight) to 1439 (one minute to midnight).

In all cases, we copy a valid schema to `sch` if the loop can be made, and return `TRUE` or `FALSE` to indicate success.

```
int kov_write_loop_schema(i6_schema *sch, kind_of_value *kov, int use_ix) {
    if (kov == NULL) return FALSE;
    world_object *k = kovko_get_kind(kov);
    if (k) {
        if (k->instance_list_length == 0)
            sch_write_to_existing(sch,
                "for (*1=nothing: false: )");
        else
            sch_write_to_existing_2d(sch,
                "for (*1=IK%d_First: *1: *1=*1.IK%d_Link)",
                k->allocation_id, k->allocation_id);
        return TRUE;
    }
    return data_type_write_loop_schema(sch, kov->atomic_kov, use_ix);
}
```

The function kov_write_loop_schema is called from 6/cdefp and 12/cinv.

§**38. Questions about run-time storage.** Recall that values are stored at run-time either as "word values" – a single I6 word – or "pointer values" (sometimes "block values"), where the I6 word is a pointer to a block of data on the heap. Numbers and times are word values, indexed texts (i.e., editable strings) are pointer values. Which form a value takes depends on its KOV:

```
int kov_uses_pointer_values(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_uses_pointer_values(kov->atomic_kov);
}
```

The function kov_uses_pointer_values is called from 9/rsdt, 7/cfsp, 7/vasp, 7/tc, 9/prop, 10/tab, 12/phsf, 12/stv and 12/cinv.

§**39.** A reasonable estimate of how large the heap block needs to be, for a pointer-value KOV, in bytes.

```
int kov_get_heap_size_estimate(kind_of_value *kov) {
    if (kov == NULL) return 0;
    return data_type_get_heap_size_estimate(kov->atomic_kov);
}
```

The function kov_get_heap_size_estimate is called from 9/rsdt.

§**40.** Some pointer-value KOVs have literal constants in the source text – lists, for instance. Others do not, because they accept their constants by casts from word values (indexed text, for instance, since any quoted matter written in the source is parsed as text – a word value).

```
int kov_has_pointer_value_constants(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_has_pointer_value_constants(kov->atomic_kov);
}
```

The function kov_has_pointer_value_constants is called from 9/rsdt and 9/prop.

§**41.** The following decides whether word-values of kind R can be cast into pointer-values of kind L – for instance, text values can be cast into indexed text, but rulebooks can't.

```
int kov_allow_word_as_pointer(kind_of_value *left, kind_of_value *right) {
    if ((left == NULL) || (right == NULL)) return FALSE;
    return data_type_allow_word_as_pointer(left->atomic_kov, right->atomic_kov);
}
```

The function kov_allow_word_as_pointer is called from 12/cinv.

§**42. Questions about copying and comparing.** For most word-value KOVs, it's easy to compare two values to see if they are equal: all we need is the `==` operator. But for pointer-value KOVs, that would simply tell us whether they point to the same block of data on the heap, whereas we need in fact to compare the blocks they point to. So the KOV system makes it possible for each individual KOV to decide how values should be compared, returning an I6 schema prototype to compare `*1` and `*2`.

```
char *kov_interpret_test_equality(kind_of_value *left, kind_of_value *right) {
    if ((left == NULL) || (right == NULL))
        return data_type_interpret_test_equality(UNKNOWN, UNKNOWN);
    return data_type_interpret_test_equality(left->atomic_kov, right->atomic_kov);
}
char *kov_interpret_store(int storage_class, kind_of_value *left, kind_of_value *right) {
    if ((left == NULL) || (right == NULL))
        return data_type_interpret_store(storage_class, UNKNOWN, UNKNOWN);
    return data_type_interpret_store(storage_class, left->atomic_kov, right->atomic_kov);
}
```

The function kov_interpret_test_equality is called from 6/equal.
The function kov_interpret_store is called from 6/equal and 12/cinv.

§**43.** And the following returns the name of an I6 routine to determine if two values of K are different from each other; or `NULL` to say that it's sufficient to apply `~=` to the values.

```
char *kov_get_distinguisher(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    return data_type_get_distinguisher(kov->atomic_kov);
}
```

The function kov_get_distinguisher is called from 13/gpr.

§**44.** Can values of this KOV be serialised out to a file and read back in again?

```
int kov_can_exchange(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_can_exchange(kov->atomic_kov);
}
```

The function kov_can_exchange is called from 10/tab.

§**45. Questions about implicit casting.** Just one question, really. If K_F and K_T are kinds of value, what values do they have in common? `can_we_cast_kovs` returns

(a) `ALWAYS_MATCH` if $K_F \subseteq K_T$,
(b) `SOMETIMES_MATCH` if $K_F \supseteq K_T$, or
(c) `NEVER_MATCH` if $K_F \cap K_T = \emptyset$.

That exhausts the possibilities: the hierarchy of kinds is not a lattice, and if two kinds overlap at all then one must be a specialisation of the other.

"Casting" is using a value of one kind where another is expected – for instance, in C, writing `N = 'x';` where `N` is an integer variable. This is an implicit cast because no explicit conversion is requested: `N = (int) 'x';` would be explicit. Inform only has implicit casts; casting is always possible when $K_F \subseteq K_T$, sometimes possible when $K_T \subseteq K_F$, never possible when $K_F \cap K_T = \emptyset$.

To put this still a third way, we call `can_we_cast_kovs` when we believe we have data of a given data type (`from` below), and want to know if it is safe to use in I6 code where data of type `to` is expected. There are three possible answers: yes, no or yes but only subject to run-time checking.

The outer routine is just a logging mechanism for the real routine. Cases where $K_F = K_T$ are frequent and not interesting enough to be logged.

```
int can_we_cast_kovs(kind_of_value *from, kind_of_value *to) {
    if (kov_compare(from, to)) return ALWAYS_MATCH;
    LOGIF(TYPE_CASTS, "(Can we cast from $u to $u?", from, to);
    switch(can_we_cast_kovs_inner(to, from)) {
        case NEVER_MATCH: LOGIF(TYPE_CASTS, " No)\n"); return NEVER_MATCH;
        case ALWAYS_MATCH: LOGIF(TYPE_CASTS, " Yes)\n"); return ALWAYS_MATCH;
        case SOMETIMES_MATCH: LOGIF(TYPE_CASTS, " Sometimes)\n"); return SOMETIMES_MATCH;
    }
    internal_error("bad return value from can_we_cast_kovs"); return NEVER_MATCH;
}
```

The function can_we_cast_kovs is called from 5/aph, 6/simp, 6/tcpr, 6/equal, 7/cmsp, 7/tc, 8/mass, 8/knowp, 9/mapbp, 9/prop, 9/madj, 9/cmpbp, 9/vpbp, 10/qnbp, 10/list, 10/tab, 11/act and 12/phtd.

§**46.** KOV marker variables are special tricks used when matching phrase prototypes like this one:

> To add (N - value = kov reference 1) to (L - list of values = list of kov marker 1) ...

This matches "add 23 to L" if "L" is a list of numbers, but not if it is a list of times; that is, "N" can be any value, but "L" has to be a list of whatever kind of value "N" has. Clearly we can only achieve this if the KOVs noticed while checking "N" are still remembered somehow while checking "L", but we also don't want to assume anything about the order in which "N" and "L" occur in the prototype. So in fact we check phrase prototypes twice, in what we call pass 1 and then pass 2, and the KOVs we need to remember during this process are stored in "marker variables".

Phrase prototype matching isn't recursive, so we only need one set of KOV marker variables inside Inform. In fact at present "Types.i6t" uses only two at a time, so this is plenty:

```
define MAX_KOV_MARKERS 8

kind_of_value *kov_marker_variables[MAX_KOV_MARKERS];
int prototype_checking_pass = 0;
```

§**47.** With that background covered, here is the main routine:

```
int can_we_cast_kovs_inner(kind_of_value *to, kind_of_value *from) {
    if ((to == NULL) || (from == NULL)) return NEVER_MATCH;
    if (data_type_template_variable_number(to->atomic_kov) != 0)
        ⟨Deal separately with matches against template variables 50⟩;
    if (from->atomic_kov == NO_ENTRIES_TY) return ALWAYS_MATCH;
    if ((to->atomic_kov == OBJECT_TY) && (from->atomic_kov == OBJECT_TY))
        ⟨Deal separately with casting within the objects hierarchy 48⟩;
    if (data_type_is_constructor(to->atomic_kov))
        ⟨Deal separately with constructions 49⟩;
    return data_type_allow_cast(to->atomic_kov, from->atomic_kov);
}
```

§**48.** Here both our KOVs are `OBJECT_TY`. In each case either `atomic_kind` is set, and then we are dealing with a subset of the objects (say, all the vehicles), or it is `NULL` and we are dealing with the set of all objects.

⟨Deal separately with casting within the objects hierarchy 48⟩ ≡

```
    world_object *to_kind = to->atomic_kind, *from_kind = from->atomic_kind;
    if (from_kind == to_kind) return ALWAYS_MATCH;
    if (to_kind == NULL) return ALWAYS_MATCH;
    if (from_kind == NULL) return SOMETIMES_MATCH;
    if (wo_of_kind(from_kind, to_kind)) return ALWAYS_MATCH;
    if (wo_of_kind(to_kind, from_kind)) return SOMETIMES_MATCH;
    return NEVER_MATCH;
```

This code is used in §47.

§**49.** In general we are pretty strict with constructions. If $K_F \subseteq K_T$ then we will allow "list of K_F" to cast to "list of K_T", but we won't allow any "sometimes" outcomes: so a list of objects will not be allowed at the sometimes level to cast to a list of vehicles (say), even though sufficiently clever run-time code might be able to police this.

⟨Deal separately with constructions 49⟩ ≡

```
    int i;
    if (from->atomic_kov != to->atomic_kov) return NEVER_MATCH;
    for (i=0; i<MAX_KC_ARGS; i++)
        if (can_we_cast_kovs_inner(to->kc_args[i], from->kc_args[i]) != ALWAYS_MATCH)
            return NEVER_MATCH;
    return ALWAYS_MATCH;
```

This code is used in §47.

§**50.** The template variable number marks out the special matching KOVs described above; it's set in "Types.i6t". For all ordinary data types, this number is 0. For "kov reference 1" and "kov reference 2", it is 1 or 2; for the corresponding "kov marker 1" and "kov marker 2", it is −1 to −2.

⟨Deal separately with matches against template variables 50⟩ ≡
```
int vn = data_type_template_variable_number(to->atomic_kov);
if (prototype_checking_pass == 0)
    internal_error("KOV marker variable occurs outside phrase typechecking");
if ((vn >= MAX_KOV_MARKERS) || (vn <= -MAX_KOV_MARKERS))
    internal_error("too many KOV marker variables in phrase typechecking");
if (vn > 0)
    ⟨A KOV reference target is always achieved, and the KOV achieved is recorded on pass 1 51⟩;
if (vn < 0)
    ⟨A KOV marker target is always achieved on pass 1, but its value must be matched on pass 2 52⟩;
```

This code is used in §47.

§**51.** For instance, given our example:

> To add (N - value = kov reference 1) to (L - list of values = list of kov marker 1) ...

When the KOV of "N" is matched against "kov reference 1" for the first time, it is stored in marker variable 1; on both passes, the match is always successful. ("Kov reference" types match anything.)

⟨A KOV reference target is always achieved, and the KOV achieved is recorded on pass 1 51⟩ ≡
```
switch(prototype_checking_pass) {
    case 1:
        LOGIF(INVOCATIONS,
        "can_we_cast_kovs_inner: setting KOV marker variable %d to $u\n", vn, from);
        kov_marker_variables[vn] = from; return ALWAYS_MATCH;
    case 2: return ALWAYS_MATCH;
}
```

This code is used in §50.

§**52.** Meanwhile, matching "list of kov marker 1" begins by matching "list of...", which takes a recursive call to `can_we_cast_kovs_inner`. The top-level call checks that "L" is indeed a list of *something*, say boojums. The inner call then compares boojums against "kov marker 1". On pass 1 this is always a successful match – we can't be sure that "kov reference 1" has been set yet, which depends on which way round they are in the prototype. But on pass 2 we can be certain that "kov reference 1" has been set to the KOV K of "K", so now the token matches if and only if boojums are K.

⟨A KOV marker target is always achieved on pass 1, but its value must be matched on pass 2 52⟩ ≡
```
switch(prototype_checking_pass) {
    case 1: return ALWAYS_MATCH;
    case 2:
        LOGIF(INVOCATIONS,
        "can_we_cast_kovs_inner: checking $u against KOV marker variable %d (=$u)\n",
            from, -vn, kov_marker_variables[-vn]);
        if (can_we_cast_kovs_inner(kov_marker_variables[-vn], from) == NEVER_MATCH)
            return NEVER_MATCH;
        else
            return ALWAYS_MATCH;
}
```

This code is used in §50.

§**53. Questions about units and enumerations.** Let's start with uncertainly defined KOVs, and what happens to them. When we read "Colour is a kind of value.", "colour" is uncertainly defined at first. Later we read either "The colours are blue and pink.", say, and then "colour" becomes an enumeration; or "450 nanometers specifies a colour.", say, and then it becomes a unit.

```
int kov_is_uncertainly_defined(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_is_uncertainly_defined(kov->atomic_kov);
}
```

The function kov_is_uncertainly_defined is called from 10/tab.

§**54.** Here we test for being a unit or an enumeration:

```
int kov_is_a_unit(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_is_a_unit(kov->atomic_kov);
}
int kov_is_an_enumeration(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_is_an_enumeration(kov->atomic_kov);
}
```

The function kov_is_a_unit is called from 9/rsdt, 7/kix and 8/creat.
The function kov_is_an_enumeration is called from 2/lexi, 4/edoc, 5/rel, 9/rsdt, 7/kix and 7/specv.

§**55.** And here we perform the conversion to a unit. (We don't need conversion to an enumeration since that happens when new enumerated values are created – see below.) The return value is `TRUE` if the KOV was already a unit or was successfully converted into one, `FALSE` if it's now too late.

```
int kov_convert_to_unit(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_convert_to_unit(kov->atomic_kov);
}
```

The function kov_convert_to_unit is called from 5/litp.

§**56.** The instances of an enumeration have the values $1, 2, 3, ..., N$ at run-time; the following returns $N+1$, that is, a value which can be held by the next instance to be created.

```
int kov_new_enumerated_value(kind_of_value *kov) {
    if (kov == NULL) return 0;
    data_type_convert_to_enumeration(kov->atomic_kov);
    return data_type_new_enumerated_value(kov->atomic_kov);
}
```

The function kov_new_enumerated_value is called from 9/qty.

§**57. Questions about properties.** Some values can have properties attached – scenes, for instance – while others can't – numbers or times, for instance. In general v can have properties only if its KOV passes this test:

```
int kov_has_properties(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    if (kov_is_an_enumeration(kov)) return TRUE;
    if (is_kova(kov, OBJECT_TY)) return TRUE;
    if (is_kova(kov, SCENE_TY)) return TRUE;
    return FALSE;
}
```

The function kov_has_properties is called from 7/kix, 8/mass, 9/pp and 9/vpbp.

§**58.** Inform allows vague general statements to be made about everything of a given KOV, as here:

A scene is usually recurring.

("Recurring" is an either/or property of scenes.) Knowledge derived from sentences like this needs to be stored somewhere, and this is where:

```
inference **kov_get_knowledge(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    return data_type_get_knowledge(kov->atomic_kov);
}
```

The function kov_get_knowledge is called from 9/inf.

§**59.** When a KOV has the same name as a property, the two are closely identified in Inform, and special handling is needed. This is how we find out:

```
int kov_name_can_coincide_with_property(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_name_can_coincide_with_property(kov->atomic_kov);
}
```

The function kov_name_can_coincide_with_property is called from 6/equal, 8/mass, 9/prop and 9/cmpbp.

§**60.** And here we read, or write, the correspondence between K and P:

```
property_name *kov_get_coinciding_property(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    return data_type_get_coinciding_property(kov->atomic_kov);
}
void kov_set_coinciding_property(kind_of_value *kov, property_name *prn) {
    if (kov == NULL) return;
    data_type_set_coinciding_property(kov->atomic_kov, prn);
}
```

The function kov_get_coinciding_property is called from 2/lexi, 5/mlc, 6/term, 6/pform, 6/equal, 7/tc, 8/refpt, 8/mass, 9/qty, 9/cmpbp and 12/cinv.
The function kov_set_coinciding_property is called from 9/qty.

§**61. Questions about printing.** Each KOV can provide its own I6 routine to print out a value onto screen, in some human-readable format.

```
char *kov_get_name_of_printing_rule(kind_of_value *kov) {
    if (kov == NULL) return "DecimalNumber";
    return data_type_get_name_of_printing_rule(kov->atomic_kov);
}
```

The function kov_get_name_of_printing_rule is called from 5/rel, 5/litp, 9/rsdt, 9/inpw, 12/phsf and 13/test.

§**62.** And it can also provide a variant used when printing out text about an action, where K is the KOV of either the noun or second noun for that action. (This is used in practice to tweak the use of articles slightly when describing objects, and for a few other variations.)

```
char *kov_get_name_of_printing_rule_ACTIONS(kind_of_value *kov) {
    if (kov == NULL) return "DA_Name";
    return data_type_get_name_of_printing_rule_ACTIONS(kov->atomic_kov);
}
```

The function kov_get_name_of_printing_rule_ACTIONS is called from 11/act.

§**63. Questions about understanding.** Some KOVs can be used as tokens in Understand sentences, others can't: thus "[time]" is a valid Understand token, but "[stored action]" is not.

Some KOVs provide have a GPR ("general parsing routine", an I6 piece of jargon) defined somewhere in the template: if so, this returns its name; if not, it returns `NULL`.

```
char *kov_get_explicit_I6_GPR(kind_of_value *kov) {
    if (kov == NULL) internal_error("kov_get_explicit_I6_GPR on null KOV");
    if (is_kova(kov, OBJECT_TY)) internal_error("wrong way to handle object grammar");
    return data_type_get_explicit_I6_GPR(kov->atomic_kov);
}
```

The function kov_get_explicit_I6_GPR is called from 13/gtok and 13/gpr.

§**64.** Can the KOV have a GPR of any kind in the final code?

```
int kov_offers_I6_GPR(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_offers_I6_GPR(kov->atomic_kov);
}
```

The function kov_offers_I6_GPR is called from 7/kix, 13/gtok and 13/gpr.

§**65.** Request that a GPR be compiled for this KOV; the return value tell us whether this will be allowed or not.

```
int kov_request_I6_GPR(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    if (is_kova(kov, OBJECT_TY)) internal_error("wrong way to handle object grammar");
    return data_type_request_I6_GPR(kov->atomic_kov);
}
```

The function kov_request_I6_GPR is called from 13/tfg and 13/gtok.

§**66.** Do we need to compile a GPR of our own for this KOV?

```
int kov_needs_I6_GPR(kind_of_value *kov) {
    if (kov == NULL) return FALSE;
    return data_type_needs_I6_GPR(kov->atomic_kov);
}
```

The function kov_needs_I6_GPR is called from 13/gprv.

§**67.** If the user writes lines in the source text such as

Understand "eleventy-one" as 111.

then grammar lines will have to be attached to a KOV; in fact, a KOV can have its own `grammar_verb` structure attached, which holds a sequence of such grammar lines. (These are possibilities in addition to those provided by any GPR existing because of the above routines.)

```
void kov_set_parsing_grammar(kind_of_value *kov, grammar_verb *gv) {
    if (kov == NULL) return;
    if (is_kova(kov, OBJECT_TY)) internal_error("wrong way to handle object grammar");
    data_type_set_parsing_grammar(kov->atomic_kov, gv);
}
grammar_verb *kov_get_parsing_grammar(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    if (is_kova(kov, OBJECT_TY)) internal_error("wrong way to handle object grammar");
    return data_type_get_parsing_grammar(kov->atomic_kov);
}
```

The function kov_set_parsing_grammar is called from 13/gv.

The function kov_get_parsing_grammar is called from 13/gv and 13/gprv.

§**68.** For the following, see the explanation in "Indexed Texts.i6t" in the template: a recognition-only GPR is used for matching specific data in the course of parsing names of objects, but not as a grammar token in its own right.

```
char *kov_get_recognition_only_GPR(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    return data_type_get_recognition_only_GPR(kov->atomic_kov);
}
```

The function kov_get_recognition_only_GPR is called from 13/tfg and 13/gpr.

§69. Questions about naming. The source name is that of the data type ID: for instance, `LIST_OF_TY` or `NUMBER_TY`.

```
char *kov_get_NI_source_name(kind_of_value *kov) {
    if (kov == NULL) return "UNKNOWN";
    return type_ID_get_source_name(kov->atomic_kov);
}
void kov_write_source_name_to_file(OUTPUT_STREAM, kind_of_value *kov) {
    if (kov == NULL) WRITE("UNKNOWN");
    else type_ID_write_source_name_to_file(OUT, kov->atomic_kov);
}
```

The function kov_get_NI_source_name is called from 10/tab and 11/act.
The function kov_write_source_name_to_file is called from 12/phsf.

§**70.** The name as used in the source text is the one which users are more familiar with: "table", "scene", "indexed text" and so on.

For efficiency reasons (and to help disambiguate matters since words like "table" are also often used in other contexts) each KOV can have either a single-word name (the "parsing name") or a multi-word name (the "long name"), and these are stored in different forms.

```
vocabulary_entry *kov_get_parsing_name(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    return type_ID_get_singular_parsing_name(kov->atomic_kov);
}
kind_of_value *kov_parse_long_name(int w1, int w2) {
    int t = data_type_parse_long_name(w1, w2);
    if (t != UNKNOWN) return kova(t);
    return NULL;
}
```

The function kov_get_parsing_name is called from 9/qty and 11/act.
The function kov_parse_long_name is called from 5/candd.

§**71.** That means we need an easy way to read the name without having to know which version is used, and it's convenient to combine that with the option to read a pluralised form:

```
void kov_get_name(kind_of_value *kov, int *w1, int *w2, int plural_form) {
    world_object *k;
    if (kov == NULL) { *w1 = -1; *w2 = -1; return; }
    k = kovko_get_kind(kov);
    if ((k) && (k->word_ref1 >= 0)) {
        *w1 = k->word_ref1; *w2 = k->word_ref2;
        if (plural_form) { make_plural_of(*w1, *w2, NULL); *w1 = plw1; *w2 = plw2; }
        return;
    }
    type_ID_copy_name(kov->atomic_kov, w1, w2);
    if (plural_form) { make_plural_of(*w1, *w2, NULL); *w1 = plw1; *w2 = plw2; }
}
```

The function kov_get_name is called from 2/index, 2/lexi, 5/litp, 9/rsdt, 9/qty, 9/pp and 11/act.

§**72.** These human-readable names are also the ones used in the index and in problem messages.

```
void copy_kov_to_string(kind_of_value *kov, char *to, int plural_form) {
    TEMPORARY_STREAM;
    copy_kov_to_string_inner(kov, TEMP, plural_form);
    truncated_strcpy(to, STREAM_TEXT(TEMP), 200);
    CLOSE_TEMPORARY_STREAM;
}
void copy_kov_to_string_inner(kind_of_value *kov, OUTPUT_STREAM, int plural_form) {
    int w1, w2;
    if (kov == NULL) { WRITE("something"); return; }
    if (is_kova(kov, ANY_VALUE_TY)) {
        if (plural_form) WRITE("values"); else WRITE("a value");
        return;
    }
    if ((is_kova(kov, OBJECT_TY)) && (kovko_get_kind(kov) == NULL)) {
        if (plural_form) { WRITE("objects"); return; }
    }
    kov_get_name(kov, &w1, &w2, plural_form);
    if (w1 >= 0) {
        int i, f = FALSE;
        if (plural_form == FALSE) {
            char c = *(lw_array[w1].lw_text);
            if ((c=='a')||(c=='e')||(c=='i')||(c=='o')||(c=='u')) WRITE("an ");
            else WRITE("a ");
        }
        for (i=w1; i<=w2; i++) {
            if (f) WRITE(" "); f = TRUE;
            WRITE("%s", lw_array[i].lw_text);
        }
    } else WRITE("%s", kov_get_description(kov));
    if (data_type_is_constructor(kov->atomic_kov)) {
        WRITE(" ");
        copy_kov_to_string_inner(kov->kc_args[0], OUT, TRUE);
    }
}
```

The function copy_kov_to_string is called from 5/bp, 5/aph, 7/vasp and 7/stsp.

§**73. Questions about indexing.** Every KOV can provide a more human-readable description of what it is: for instance, “a number”.

```
char *kov_get_description(kind_of_value *kov) {
    if (kov == NULL) return "something";
    return type_ID_get_description(kov->atomic_kov);
}
```

The function kov_get_description is called from 2/index.

§**74.** And every KOV is allowed to have the specification pseudo-property – a little text used only on the index pages, and not existing at run-time. This is set explicitly in the source text, but initialised for built-in KOVs from the template file “Types.i6t”.

```
void kov_set_specification_text(kind_of_value *kov, char *desc) {
    data_type_set_specification_text(kov->atomic_kov, desc);
}
char *kov_get_specification_text(kind_of_value *kov) {
    if (kov == NULL) internal_error("can't get specification of null KOV");
    return data_type_get_specification_text(kov->atomic_kov);
}
```

The function kov_set_specification_text is called from 8/mass.
The function kov_get_specification_text is called from 7/kix.

Data Types 7/data

Purpose

To manage the built-in set of data types, which are used as the atomic kinds of value and therefore characterise fundamental kinds of data – numbers, times, texts, and so on – and to create new ones as required.

Definitions

¶1. Data types are ID numbers used to provide a meaning for a `kind_of_value` structure – see "Table of ID Numbers". In this section, we maintain a database of the characteristics associated with each data type. This provides a back-end service for the section managing kinds of value, which is what the rest of Inform communicates with when it wants to know about the nature of values.

When we talk about a SP below as a constant `NUMBER_TY`, say, what we mean is that it is an actual not generic specification, that it has family `VALUE_FMY` and species `CONSTANT_SPC`, and that its associated kind of value is `kova(NUMBER_TY)`, that is, the atomic kind of value corresponding to `NUMBER_TY`.

¶2. Coincidence of data types and properties occurs where a data type has the same name exactly as a property, allowing the same name to be used grammatically in two different contexts. We say that the data type and the property "coincide".

None of the built-in data types coincides with any of the built-in properties, but designed ones often do. This arises from source text such as

> Brightness is a kind of value. The brightnesses are guttering, weak, radiant and blazing. The lantern has a brightness. The lantern is blazing.

Here "brightness" becomes the name of a new type ID; there are only four possible values for this type, with names as given; but "brightness" also becomes the name of a property held by the lantern, whose value is, of course, always a brightness. Inform 7 therefore understands "The lantern is blazing" as a statement about the brightness *property* of the lantern, not as an attempt to equate two incomparable values.

¶3. There are four basic constant compilation methods used:

define `NONE_CCM 1` *constant values of this kind cannot exist*
define `LITERAL_CCM 2` *the literal-parser is used to resolve text of SP to a number*
define `QUANTITATIVE_CCM 3` *a quantity structure is pointed to*
define `SPECIAL_CCM 4` *special code specific to the kind of value is needed*

¶4. Different data types have different abilities: some can be stored in local variables, some cannot, and so forth. This is a great big bag of properties, almost all set by commands to the type interpreter.

```
define MAX_MODIFYING_ADJECTIVES 10
define MAX_WORDS_IN_TYPE_NAME 10
define LOWEST_INDEX_PRIORITY 100

typedef struct data_type_id_rules {
    int data_type_these_apply_to;                                   standard I7 type number
    int defined_in_source_text;                                     rather than by the Types.i6t file
    int named_values_created_with_assertions;                       such as "Train Arrival is a scene."
    int is_incompletely_defined;                                    newly defined type ambiguous as yet
    int is_constructor;                                             is a constructor like "list of..." rather than a base type
    int is_template_variable;                                       is a variable type used in a phrase specification
    int template_variable_number;                                   1, 2, 3, ... for variables; −1, −2, −3, ... references
    int is_a_unit;                                                  a seminumerical data type with literal constant patterns
    int is_an_enumeration;                                          a data type with only named enumerated values
    int quasinumerical;                                             can be used in arithmetic
    int used_in_let_without_casts;                                  can a let variable always have this as its kind?
    int can_be_type_of_global_variable;                             similarly
    int unassignable;                                               can the typechecker allow this to be assigned to storage?
    int constant_compilation_method;                                one of the values below
    int uses_signed_comparisons;                                    when sorting table entries
    int can_exchange;                                               with external files and therefore other story files
    char *default_value;                                            if so, then an I6 value, for built-in types
    char *index_default_value;                                      and its description in the Kinds index
    int index_priority;                                             from 1 (highest) to LOWEST_INDEX_PRIORITY (lowest)
    int indexed_grey_if_empty;                                      shaded grey in the Kinds index
    int block_data;                                                 TRUE for large values stored on the heap
    int multiple_block;                                             TRUE for flexible-size values stored on the heap
    int native_constants;                                           TRUE if uncast constants can be parsed of this type
    int heap_size_estimate;                                         typical number of bytes used
    int next_free_value;                                            to make distinguishable constants belonging to the type
    int has_i6_GPR;                                                 a general parsing routine exists in the I6 code for this
    int I6_GPR_needed;                                              and is actually required
    char *explicit_i6_GPR;                                          routine name, when not compiled automatically
    char *recognition_only_GPR;                                     for recognising an explicit value as preposition
    char *distinguisher;                                            I6 routine to see if values distinguishable in parsing
    struct grammar_verb *understand_as_values;                      used when parsing such values
    struct literal_pattern *ways_to_write_literals;                 list of ways to write this
    int can_coincide_with_property;                                 allowed to coincide in name with a property
    struct property_name *coinciding_property;                      property of the same name, if any
    char dt_I6_identifier[32];                                      an I6 identifier used for compiling printing rules
    char name_of_printing_rule_ACTIONS[32];                         ditto but for ACTIONS testing command
    struct dimensional_rules dim_rules;                             how arithmetic operations work here
    char *specification_text;                                       text for specification
    int no_words_in_exotic_name;                                    where by exotic we mean multi-word but built in
    struct vocabulary_entry *words_in_exotic_name[MAX_WORDS_IN_TYPE_NAME];   ditto
    int no_words_in_exotic_plural;                                  where by exotic we mean multi-word but built in
    struct vocabulary_entry *words_in_exotic_plural[MAX_WORDS_IN_TYPE_NAME];  ditto
    struct data_type_casting_rule *first_casting_rule;              list of these
    struct unit_sequence dimensional_form;                          dimensions of this KOV
    int dimensional_form_fixed;                                     whether they are derived
    struct data_type_comparison_schema *first_comparison_schema;    list of these
```

```
    int no_modifying_adjectives;                                    which can be used in creation
    struct vocabulary_entry *modifying_adjectives[MAX_MODIFYING_ADJECTIVES];   ditto
    struct inference *dt_knowledge;                                 inferences about properties
    char *loop_domain_schema;                                       how to compile an I6 loop over the instances
    char *documentation_reference;                                  documentation symbol, if any
    MEMORY_MANAGEMENT
} data_type_id_rules;
```

The structure data_type_id_rules is shared with 7/dti.

---

§**1.** Take a deep breath and...

```
data_type_id_rules *dtid_new(int t) {
    data_type_id_rules *dtid = CREATE(data_type_id_rules);
    type_ID_make_into_data_type(t, dtid);
    dtid->data_type_these_apply_to = t;
    dtid->block_data = FALSE;
    dtid->can_be_type_of_global_variable = FALSE;
    dtid->can_coincide_with_property = FALSE;
    dtid->can_exchange = FALSE;
    dtid->coinciding_property = NULL;
    dtid->constant_compilation_method = NONE_CCM;
    dtid->default_value = NULL;
    dtid->defined_in_source_text = FALSE;
    dtid->distinguisher = NULL;
    dtid->documentation_reference = NULL;
    dtid->dt_knowledge = NULL;
    dtid->explicit_i6_GPR = NULL;
    dtid->first_casting_rule = NULL;
    dtid->first_comparison_schema = NULL;
    dtid->has_i6_GPR = FALSE;
    dtid->heap_size_estimate = 0;
    dtid->I6_GPR_needed = FALSE;
    dtid->index_default_value = "--";
    dtid->index_priority = LOWEST_INDEX_PRIORITY;
    dtid->is_a_unit = FALSE;
    dtid->is_an_enumeration = FALSE;
    dtid->is_constructor = FALSE;
    dtid->is_incompletely_defined = FALSE;
    dtid->is_template_variable = FALSE;
    dtid->loop_domain_schema = NULL;
    dtid->multiple_block = FALSE;
    dtid->named_values_created_with_assertions = FALSE;
    dtid->native_constants = FALSE;
    dtid->next_free_value = 1;
    dtid->no_modifying_adjectives = 0;
    dtid->no_words_in_exotic_name = 0;
    dtid->no_words_in_exotic_plural = 0;
    dtid->quasinumerical = FALSE;
    dtid->recognition_only_GPR = NULL;
    dtid->specification_text = NULL;
    dtid->template_variable_number = 0;
```

```
    dtid->unassignable = FALSE;
    dtid->understand_as_values = NULL;
    dtid->used_in_let_without_casts = FALSE;
    dtid->uses_signed_comparisons = FALSE;
    dtid->ways_to_write_literals = NULL;

    dtid->dimensional_form = data_spec_to_unit_sequence(t);
    dtid->dimensional_form_fixed = FALSE;

    strcpy(dtid->dt_I6_identifier, "DecimalNumber");
    strcpy(dtid->name_of_printing_rule_ACTIONS, "DecimalNumber");
    dim_initialise(&(dtid->dim_rules));
    play_back_type_macro(parse_type_macro_name("#DEFAULTS"), dtid);
    return dtid;
}
```

The function dtid_new is called from 7/dti.

§2. Two special sorts of data type.

```
int data_type_is_constructor(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(t);
    if (dtid == NULL) return FALSE;
    if (dtid->is_constructor) return TRUE;
    return FALSE;
}
```

The function data_type_is_constructor is called from 7/kov and 7/kix.

§3. See the discussion of KOV casting in "Kinds of Value".

```
int data_type_template_variable_number(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(t);
    if (dtid == NULL) return 0;
    return dtid->template_variable_number;
}
```

The function data_type_template_variable_number is called from 7/kov.

§**4. Initial creation of new data types.** The following routine performs the initial, incomplete creation of a new data type: it is what happens when an assertion like "Weight is a kind of value." is read. The `#NEW` type macro defined in "Types.i6t" sets all of the properties of the kind except those pertaining to its name. (That includes the question of whether it coincides with a property, since this is specified by giving a common name to both.)

```
int new_data_type(int w1, int w2) {
    if (is_word_intermediate(of_V, w1, w2) >= 0)
        ⟨Weasel out at the thought of "of" in a KOV 5⟩;
    int new_id = type_ID_claim_new_ID();
    data_type_id_rules *dtid = dtid_new(new_id);
    type_ID_set_name(new_id, w1, w2);
    isn_compose_identifier(dtid->dt_I6_identifier, 'T', (dtid->allocation_id)+1, w1, w2);
    strcpy(dtid->name_of_printing_rule_ACTIONS, dtid->dt_I6_identifier);
    play_back_type_macro(parse_type_macro_name("#NEW"), dtid);
    property_name *prn = parse_property_name(w1, w2);
    if (prn) {
        kind_of_value *kov = kova(new_id);
        prn_set_kind_of_value(prn, kov);
        make_type_coincident(kov, prn);
    }
    LOGIF(TYPE_CREATIONS, "Created designed type %d: $W\n", new_id, w1, w2);
    return new_id;
}
```

The function new_data_type is called from 7/kov.

§**5.** This is just a tiny bit pusillanimous. It avoids ambiguities arising if we create a property called, say, "point of view" and then make that coincide with a KOV. But in an ideal world, we ought to allow such things.

⟨Weasel out at the thought of "of" in a KOV 5⟩ ≡

```
    quote_source(1, current_sentence);
    quote_words(2, w1, w2);
    handmade_problem(_P_(C7KindOfValueWithOf));
    issue_problem_segment(
        "You wrote %1, but that would mean creating a kind of value "
        "'%2' which contains the word 'of' - this is disallowed because "
        "it creates just too many grammatical ambiguities down the road. "
        "Apologies for the inconvenience.");
    issue_problem_end();
    return -1;
```

This code is used in §4.

§**6. Questions about origin.** Some data types are built in (in that the template file "Types.i6t" creates them, using the type interpreter), while others arise from "X is a kind of value" sentences in the source text:

```
int data_type_is_built_in(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    if (dtid->defined_in_source_text) return FALSE;
    return TRUE;
}
```

The function data_type_is_built_in is called from 7/kov.

§**7.** A few types support modifying adjectives when created. These adjectives are created in the "Types.i6t" file, and parsed here.

```
int data_type_parse_modifying_adjectives(int t, int w1, int w2) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(t);
    int i;
    if (dtid == NULL) return -1;
    if (w1 != w2) return -1;
    for (i=0; i<dtid->no_modifying_adjectives; i++)
        if (compare_word(w1, dtid->modifying_adjectives[i]))
            return i;
    return -1;
}
```

The function data_type_parse_modifying_adjectives is called from 7/kov.

§**8. Questions about constants of this kind.** Some KOVs have named constants, others use a quasi-numerical notation: for instance "maroon" might be a named constant for KOV "colour", while "234 kilopascals" might be a notation for a KOV where constants are not named.

```
int data_type_has_named_constant_values(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    if (dtid->named_values_created_with_assertions) return TRUE;
    return FALSE;
}
```

The function data_type_has_named_constant_values is called from 7/kov.

§**9.** The following returns the compilation method: a constant in the form `*_CCM`, defined above.

```
int data_type_get_constant_compilation_method(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->constant_compilation_method;
}
```

The function data_type_get_constant_compilation_method is called from 7/kov.

§**10.** Declares that the given notation can be used to write constants of this data type.

```
void data_type_add_literal_form(int t, literal_pattern *lp) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    dtid->ways_to_write_literals =
        lp_list_add(dtid->ways_to_write_literals, lp);
}
```

The function data_type_add_literal_form is called from 7/kov.

§**11.** Returns the list of LPs attached.

```
literal_pattern *data_type_list_of_literal_forms(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->ways_to_write_literals;
}
```

The function data_type_list_of_literal_forms is called from 5/litp, 7/kov, 7/dim and 7/kix.

§**12. Questions about range and behaviour.** For explanation, see the corresponding paragraphs of "Kinds of Value".

```
int data_type_uses_signed_comparisons(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->uses_signed_comparisons;
}
int data_type_requires_blanks_bitmap(int t) {
    if (t == OBJECT_TY) return FALSE;
    if (data_type_is_an_enumeration(t)) return FALSE;
    return TRUE;
}
int data_type_get_highest_valid_value_as_integer(int t) {
    switch (t) {
        case ACTION_NAME_TY: return NUMBER_CREATED(action_name);
        case ACTIVITY_TY: return NUMBER_CREATED(activity);
        case EXTERNALFILE_TY: return NUMBER_CREATED(external_file);
        case FIGURENAME_TY: return NUMBER_CREATED(blorb_figure);
        case RULE_TY: return NUMBER_CREATED(booked_rule);
        case RULEBOOK_TY: return NUMBER_CREATED(rulebook);
        case SOUNDNAME_TY: return NUMBER_CREATED(blorb_sound);
        case TABLE_TY: return NUMBER_CREATED(table) + 1;
        case TRUTH_STATE_TY: return 2;
    }
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->next_free_value - 1;
}
```

The function data_type_uses_signed_comparisons is called from 7/kov and 9/rsdt.
The function data_type_requires_blanks_bitmap is called from 7/kov.
The function data_type_get_highest_valid_value_as_integer is called from 7/kov and 7/kix.

§**13.** The dimensional rules for t are the conventions on whether arithmetic operations can be applied, and if so, what KOV the result has.

```
dimensional_rules *dtid_get_dim_rules(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(t);
    if (dtid == NULL) return NULL;
    return &(dtid->dim_rules);
}
int data_type_is_quasinumerical(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(t);
    if (dtid == NULL) return FALSE;
    return dtid->quasinumerical;
}
unit_sequence *data_type_get_unit_sequence(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(t);
    if (dtid == NULL) return NULL;
    return &(dtid->dimensional_form);
}
int data_type_test_if_derived(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(t);
    if (dtid == NULL) return TRUE;
```

```
    return dtid->dimensional_form_fixed;
}

void data_type_now_derived(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(t);
    if (dtid == NULL) internal_error("can't fix dim form for null data type");
    dtid->dimensional_form_fixed = TRUE;
}
```

The function dtid_get_dim_rules is called from 7/dim.
The function data_type_is_quasinumerical is called from 7/kov and 7/dim.
The function data_type_get_unit_sequence is called from 7/kov and 7/dim.
The function data_type_test_if_derived is called from 7/dim.
The function data_type_now_derived is called from 7/dim.

§**14.** Some data types can be stored in global variables (and properties), others not: this is a straightforward matter, with the specification of each type explicitly giving a yes/no response. The issue is simple here because global variables and properties have their types declared exactly by name, so that Inform is never unsure what type they have. The routine below is used to decided whether creation of a global variable of kind t should be permitted. (Once it exists, casting means that it may well contain values of other kinds – for instance, a `TEXT` variable may well contain a `TEXT_ROUTINE` value, but a `TEXT_ROUTINE` variable is not permitted to exist.)

```
int data_type_can_be_stored_in_variables(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->can_be_type_of_global_variable;
}
```

The function data_type_can_be_stored_in_variables is called from 7/kov and 9/rsdt.

§**15.** Similarly, some data can be stored in local ("let") variables. But because of the way let variables are ambiguously defined – by initial value rather than by an explicit type description – we need to know which are the more likely possibilities, and that likelihood order is provided by the global array defined below. (There will typically only be seven such options, so the array below is plenty.)

Suppose we read "let X be V", where X is an unknown name: Inform creates X as a variable and takes the type from V. Or does it? If the KOV K of V has "used-in-let-without-casts" set, X is immediately set to K. Otherwise Inform attempts to cast V to one of the types with a "used-in-let-with-cast-priority" set, starting with the lowest priority and working up, until the first match. For instance:

```
let X be "look, it's [time of day]";
```

has K equal to `kova(TEXT_ROUTINE_TY)`, but this is not permitted as the type of a let variable, so Inform tries various possibilities, and the lowest-priority-number match is `kova(TEXT_TY)`.

It follows that, as with `kova(TEXT_ROUTINE)`, any KOV for a data type with neither property specified is never the type of any let variable.

define `MAX_LET_CAST_OPTIONS 64`

```
int no_let_cast_options = 0;
int data_types_in_let_variables[MAX_LET_CAST_OPTIONS];
int let_cast_priority_levels[MAX_LET_CAST_OPTIONS];

void initialise_let_casting_options(void) {
    register_let_cast_option(-1, 10000);
}

void register_let_cast_option(int t, int priority) {
    int i, j;
```

```
    if (no_let_cast_options == MAX_LET_CAST_OPTIONS)
        internal_error("too many let cast options");
    for (i=0; i<no_let_cast_options; i++)
        if (priority < let_cast_priority_levels[i]) {
            for (j=no_let_cast_options; j>i; j--) {
                data_types_in_let_variables[j] =
                    data_types_in_let_variables[j-1];
                let_cast_priority_levels[j] =
                    let_cast_priority_levels[j-1];
            }
            data_types_in_let_variables[i] = t;
            let_cast_priority_levels[i] = priority;
            no_let_cast_options++;
            return;
        }
    data_types_in_let_variables[no_let_cast_options] = t;
    let_cast_priority_levels[no_let_cast_options++] = priority;
}
int data_type_for_storage_to_hold(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    int j;
    if (dtid->used_in_let_without_casts) return t;
    for (j=0; data_types_in_let_variables[j] >= 0; j++) {
        if (data_type_allow_cast(data_types_in_let_variables[j], t)
            == ALWAYS_MATCH)
            return data_types_in_let_variables[j];
    }
    return UNKNOWN;
}
```

The function initialise_let_casting_options is called from 7/dti.
The function register_let_cast_option is called from 7/dti.
The function data_type_for_storage_to_hold is called from 7/kov.

§**16.** Most values are such that the typechecker will allow at least an attempt to assign them to variables, even if that later leads to a problem message; but in a few cases we forbid this even as an interpretation of the source text's wishes. (This enables us to avoid ambiguities.)

```
int data_type_unassignable(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->unassignable;
}
```

§**17. Questions about the default value.** For explanation, see the corresponding paragraphs of "Kinds of Value".

```
char *data_type_get_default_value_of_type(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->default_value;
}
char *data_type_get_index_default_value(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->index_default_value;
}
```

The function data_type_get_default_value_of_type is called from 7/kov.
The function data_type_get_index_default_value is called from 7/kov.

§18. Questions about quantification. For explanation, see the corresponding paragraphs of "Kinds of Value".

```
int data_type_compile_domain_possible(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    if (data_type_is_an_enumeration(t)) return TRUE;
    if (dtid->loop_domain_schema) return TRUE;
    return FALSE;
}
```

The function data_type_compile_domain_possible is called from 7/kov.

§19. At some stage these should probably move into the "Types.i6t" template file.

```
define MAX_LOOP_DOMAIN_SCHEMA_LENGTH 1000

int data_type_write_loop_schema(i6_schema *sch, int t, int use_ix) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    char *p = dtid->loop_domain_schema;
    char q[MAX_LOOP_DOMAIN_SCHEMA_LENGTH+4];
    if (data_type_is_an_enumeration(t)) {
        sch_write_to_existing_1d(sch,
            "for (*1=1: *1<=%d: *1++)", data_type_get_highest_valid_value_as_integer(t));
        return TRUE;
    }
    if (p == NULL) return FALSE;
    if (use_ix) ⟨Convert all references to term 2 to term 1 plus "ix" 20⟩
    else strcpy(q, p);
    sch_write_to_existing(sch, q);
    return TRUE;
}
```

The function data_type_write_loop_schema is called from 7/kov.

§20. Oh, but this is a vulgar little trick. The domain schemas set in "Types.i6t" refer to two variables, `*1` and `*2`. `*1` is the current instance in the loop body, and `*2` is free for the loop header to use – typically it is a counter iterating through some table of values from which `*1` is drawn. But sometimes we need to expand to `*1` and `*1` with `_ix` suffixed, instead of general terms `*1` and `*2`. (For instance, to `x` and `x_ix`: the suffix is supposed to mean "indexer".) Rather than fix this in any elegant way, we just hack the schema as we write it.

⟨Convert all references to term 2 to term 1 plus "ix" 20⟩ ≡

```
    int i, j;
    for (i=0, j=0; (p[i]) && (i<MAX_LOOP_DOMAIN_SCHEMA_LENGTH); i++)
        if ((use_ix) && (i>0) && (p[i-1] == '*') && (p[i] == '2')) {
            q[j++] = '1'; q[j++] = '_'; q[j++] = 'i'; q[j++] = 'x';
        } else q[j++] = p[i];
    q[j] = 0;
```

This code is used in §19.

§**21. Questions about run-time storage.** For explanation, see the corresponding paragraphs of "Kinds of Value".

```
int data_type_has_pointer_value_constants(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(t);
    if (dtid == NULL) return FALSE;
    if (dtid->native_constants) return TRUE;
    return FALSE;
}

int data_type_uses_pointer_values(int t) {
    data_type_id_rules *dtid;
    if (t == UNKNOWN) return FALSE;
    dtid = type_ID_get_dtid(t);
    return dtid->block_data;
}

int data_type_get_heap_size_estimate(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->heap_size_estimate;
}

int data_type_allow_word_as_pointer(int left, int right) {
    if (data_type_uses_pointer_values(left) == FALSE) return FALSE;
    if (data_type_uses_pointer_values(right) == TRUE) return FALSE;
    if (data_type_allow_cast(left, right) == ALWAYS_MATCH) return TRUE;
    return FALSE;
}
```

The function data_type_has_pointer_value_constants is called from 7/kov.

The function data_type_uses_pointer_values is called from 7/kov and 9/rsdt.

The function data_type_get_heap_size_estimate is called from 7/kov.

The function data_type_allow_word_as_pointer is called from 7/kov.

§**22. Questions about copying and comparing.** What happens at run-time when we test to see if value V equals value W, or change storage object S so that it now contains value T, depends on the kind of values we are discussing. If there were only word-based values in Inform (as was the case until September 2007), there would be little to do here, as the comparison would simply compile to `V == W`, while the storage would be a matter of either `S = W;` or some more exotic case along the lines of `StorageRoutineWrite(S, W);`.

But once pointers to blocks are allowed, this becomes more interesting. Now the comparison needs to be a deep one, that is, we want to test whether two indexed texts (say) have the same textual content – not whether we are holding two pointers to the same blocks in memory, which is what a simple comparison would achieve. Such a test is called "deep comparison", and similarly, we must assign by transferring the contents of the blocks of data, not merely the pointer to them, which is a "deep copy". This turns out to be one of the convenient places to perform any explicit casts between mere word values and block data, too: see above. (The other place explicit casting is needed is during transfer of arguments to functions and other phrases, and this happens during invocation compilation.)

```
char *data_type_interpret_test_equality(int left, int right) {
    LOGIF(TYPE_CASTS, "Finding ST == with kovs $s, $s\n", left, right);
    char *special_rule = data_type_find_comparison_schema(left, right);
    if (special_rule) return special_rule;
    if (data_type_uses_pointer_values(left)) {
        if (data_type_allow_word_as_pointer(left, right)) {
            pointer_allocation *pall =
                phsf_add_allocation(kova(left),
                    "*=-BlkValueCompare(*1, BlkValueCast(*##, *#1, *#2, *!2))==0");
            return pall_get_expanded_schema(pall);
        }
        return "*=-BlkValueCompare(*1, *2)==0";
    }
    return "*=-*1 == *2";
}
char *data_type_interpret_store(int storage_class, int L, int R) {
    if (data_type_uses_pointer_values(L)) {
        if (data_type_allow_word_as_pointer(L, R))
            return storage_class_schema(storage_class, STORE_WORD_TO_POINTER);
        return storage_class_schema(storage_class, STORE_POINTER_TO_POINTER);
    }
    return storage_class_schema(storage_class, STORE_WORD_TO_WORD);
}
```

The function data_type_interpret_test_equality is called from 7/kov.
The function data_type_interpret_store is called from 7/kov.

§**23.** And the following returns an I6 routine to determine if two values differ:

```
char *data_type_get_distinguisher(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->distinguisher;
}
```

The function data_type_get_distinguisher is called from 7/kov.

§**24.** Can values be written out to an external file and retrieved by some other Inform story file, or by this one running on a different day?

```
int data_type_can_exchange(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->can_exchange;
}
```

The function data_type_can_exchange is called from 7/kov.

§25. Questions about implicit casting. For explanation, see the corresponding paragraphs of "Kinds of Value".

Inform uses the term "casting" in a slightly more specific way than its usual definition in compiler theory. We say that X casts to Y if a value of kind X can be used as if it were of kind Y. The cast is said to be implicit if no change is required to the run-time value of X to make it into a Y; explicit if some conversion code must be run. Clearly any type X casts implicitly to itself.

Given Y (the left-hand type) and X (the right-hand – thinking of this as being a way to check an assignment to a variable), the following decides whether a cast can be made always, never or sometimes. Sometimes means that the compiler will allow an implicit cast, but will generate run-time code to check the validity then. (For instance, rulebook always casts to rule because all rulebook values at run-time are valid as rule values: but rule only sometimes casts to rulebook, because only some rule values are rulebook values.)

Inform's convention is that all casts between word values must be implicit, while all casts from word values to block values are explicit. The routine below does not concern itself with this.

Finally, note that this implementation does not automatically make the casting relation transitive: if X casts to Y, and Y casts to Z, then it ought to be true that X casts to Z, but such a deduction is not made. So we need to set up the casting definitions in the `Types.i6t` file carefully to ensure this by hand.

```
int data_type_allow_cast(int left, int right) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(left);
    data_type_casting_rule *dtcr;
    if (left == right) return ALWAYS_MATCH;
    if (dtid == NULL) return NEVER_MATCH;
    dtcr = dtid->first_casting_rule;
    while (dtcr) {
        switch(dtcr->cast_domain) {
            case CAST_SPECIFIC:
                if (right == dtcr->single_value_casts) return dtcr->casting_certainty;
                break;
            case CAST_UNIVERSAL:
                return ALWAYS_MATCH;
            case CAST_ASSIGNABLE:
                if (data_type_unassignable(right)) return NEVER_MATCH;
                return ALWAYS_MATCH;
            case CAST_ANY_WORD:
                if (data_type_uses_pointer_values(right) == FALSE) return ALWAYS_MATCH;
                break;
            case CAST_ANY_POINTER:
                if (data_type_uses_pointer_values(right) == TRUE) return ALWAYS_MATCH;
                break;
        }
        dtcr = dtcr->next_casting_rule;
    }
    return NEVER_MATCH;
}
```

The function data_type_allow_cast is called from 7/kov.

§**26.** And similarly for extracting any special schema needed to test whether a value of K_L "is" a value of K_R:

```
char *data_type_find_comparison_schema(int left, int right) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(left);
    data_type_comparison_schema *dtcs;
    if (dtid == NULL) return NULL;
    dtcs = dtid->first_comparison_schema;
    while (dtcs) {
        if (right == dtcs->comparator) return dtcs->comparison_schema;
        dtcs = dtcs->next_comparison_schema;
    }
    return NULL;
}
```

§**27. Questions about units and enumerations.** On creation, a new data type has not yet decided what it is going to be like: perhaps an enumeration, perhaps a unit, and in future perhaps other possibilities too. Initially, then, it has an incompletely defined flag set: once one of the conversion routines has been used, the matter is settled and there is no going back.

```
int data_type_is_uncertainly_defined(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->is_incompletely_defined;
}
```

The function data_type_is_uncertainly_defined is called from 7/kov.

§**28.** Here we test for being a unit or an enumeration:

```
int data_type_is_a_unit(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->is_a_unit;
}
int data_type_is_an_enumeration(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(t);
    if (dtid == NULL) return FALSE;                    to facilitate a problem message elsewhere
    return dtid->is_an_enumeration;
}
```

The function data_type_is_a_unit is called from 7/kov.
The function data_type_is_an_enumeration is called from 7/kov.

§**29.** Conversions to a unit or enumeration require running macros in the type interpreter:

```
int data_type_convert_to_unit(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    if (dtid->is_incompletely_defined == TRUE) {
        play_back_type_macro(parse_type_macro_name("#UNIT"), dtid);
        return TRUE;
    }
    if (dtid->is_a_unit) return TRUE;
    return FALSE;
}
int data_type_convert_to_enumeration(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    if (dtid->is_incompletely_defined == TRUE) {
        play_back_type_macro(parse_type_macro_name("#ENUMERATION"), dtid);
        return TRUE;
    }
    if (dtid->is_an_enumeration) return TRUE;
    return FALSE;
}
```

The function data_type_convert_to_unit is called from 7/kov.
The function data_type_convert_to_enumeration is called from 7/kov.

§**30.** Allocating a new instance number:

```
int data_type_new_enumerated_value(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->next_free_value++;
}
```

The function data_type_new_enumerated_value is called from 7/kov.

§**31. Questions about properties.** For explanation, see the corresponding paragraphs of "Kinds of Value".

```
inference **data_type_get_knowledge(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid_or_null(t);
    if (dtid == NULL) return NULL;
    return &(dtid->dt_knowledge);
}
```

The function data_type_get_knowledge is called from 7/kov.

§**32.** When a KOV coincides with a property:

```
int data_type_name_can_coincide_with_property(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->can_coincide_with_property;
}
void data_type_set_coinciding_property(int t, property_name *pn) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    dtid->coinciding_property = pn;
}
property_name *data_type_get_coinciding_property(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->coinciding_property;
}
```

The function data_type_name_can_coincide_with_property is called from 7/kov.
The function data_type_set_coinciding_property is called from 7/kov.
The function data_type_get_coinciding_property is called from 7/kov.

§**33. Questions about printing.** An I6 routine to print out a value onto screen:

```
char *data_type_get_name_of_printing_rule(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->dt_I6_identifier;
}
```

The function data_type_get_name_of_printing_rule is called from 7/kov and 9/rsdt.

§**34.** And one to describe a value in an action:

```
char *data_type_get_name_of_printing_rule_ACTIONS(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->name_of_printing_rule_ACTIONS;
}
```

The function data_type_get_name_of_printing_rule_ACTIONS is called from 7/kov.

§35. Questions about understanding. For explanation, see the corresponding paragraphs of "Kinds of Value".

```
char *data_type_get_explicit_I6_GPR(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->explicit_i6_GPR;
}
```

The function data_type_get_explicit_I6_GPR is called from 7/kov.

§36. Can this have a GPR of any kind in the final code?

```
int data_type_offers_I6_GPR(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->has_i6_GPR;
}
```

The function data_type_offers_I6_GPR is called from 7/kov.

§37. Request that a GPR be compiled; the return value tell us whether this will be allowed or not.

```
int data_type_request_I6_GPR(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    if (dtid->has_i6_GPR == FALSE) return FALSE;                    can't oblige
    dtid->I6_GPR_needed = TRUE;                                     make note to oblige later
    return TRUE;
}
```

The function data_type_request_I6_GPR is called from 7/kov.

§38. Do we need to compile a GPR of our own?

```
int data_type_needs_I6_GPR(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->I6_GPR_needed;
}
```

The function data_type_needs_I6_GPR is called from 7/kov.

§39. The GV attached to a data type, and from which its GPR will be compiled.

```
void data_type_set_parsing_grammar(int t, grammar_verb *gv) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    dtid->understand_as_values = gv;
}
grammar_verb *data_type_get_parsing_grammar(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->understand_as_values;
}
```

The function data_type_set_parsing_grammar is called from 7/kov.
The function data_type_get_parsing_grammar is called from 7/kov.

§**40.** A recognition-only GPR.

```
char *data_type_get_recognition_only_GPR(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->recognition_only_GPR;
}
```

The function data_type_get_recognition_only_GPR is called from 7/kov.

§41. Questions about naming. For explanation, see the corresponding paragraphs of "Kinds of Value". For the most part, a kind of value has a singular and a plural form of its name. This is stored either as a single word (where we can use a vocabulary flag to spot it quickly in parsing), or, exceptionally, in an array of single words, or, for source-defined kinds of value, in a standard Inform word range. The following routine parses an exceptional case. This all looks like a lot of fuss for not much gain, but is done so that the `Types.i6t` file can specify a multiple-word name for a type: note that the contents of said file are not in the source text, so can't have source text word numbers, which is why we can't specify the name in the usual way.

```
int recognise_type_template_variables = FALSE;
int data_type_parse_long_name(int w1, int w2) {
    data_type_id_rules *dtid;
    int i;
    LOOP_OVER(dtid, data_type_id_rules) {
        if (dtid->no_words_in_exotic_name == 0) continue;
        if (dtid->is_constructor) continue;
        if ((dtid->is_template_variable) &&
            (recognise_type_template_variables == FALSE)) continue;
        if ((w2-w1+1) != dtid->no_words_in_exotic_name) continue;
        for (i=0; i<dtid->no_words_in_exotic_name; i++) {
            if (lw_array[w1+i].lw_identity == dtid->words_in_exotic_name[i])
                continue;
            goto ContinueOuter;
        }
        return type_ID_with_this_dtid(dtid);
        ContinueOuter: ;
    }
    LOOP_OVER(dtid, data_type_id_rules) {
        if (dtid->no_words_in_exotic_plural == 0) continue;
        if (dtid->is_constructor) continue;
        if ((dtid->is_template_variable) &&
            (recognise_type_template_variables == FALSE)) continue;
        if ((w2-w1+1) != dtid->no_words_in_exotic_plural) continue;
        for (i=0; i<dtid->no_words_in_exotic_plural; i++) {
            if (lw_array[w1+i].lw_identity == dtid->words_in_exotic_plural[i])
                continue;
            goto ContinueOuterP;
        }
        return type_ID_with_this_dtid(dtid);
        ContinueOuterP: ;
    }
    return UNKNOWN;
}
```

The function data_type_parse_long_name is called from 7/kov.

§**42.** The S-parser needs the following, which looks at a word range and tries to find text making a kind construction: if it does, it returns the ID code for the constructor, and adjusts the word range to the kind being constructed on; if it fails, it returns `UNKNOWN`. For instance, given "list of marbles", it adjusts the word range to "marbles" and returns `LIST_OF_TY`.

```
int parse_constructor_name(int *w1, int *w2) {
    data_type_id_rules *dtid;
    int i;
    LOOP_OVER(dtid, data_type_id_rules) {
        if (dtid->is_constructor == FALSE) continue;
        if ((*w2-*w1+1) < dtid->no_words_in_exotic_name) continue;
        for (i=0; i<dtid->no_words_in_exotic_name; i++)
            if (lw_array[*w1+i].lw_identity != dtid->words_in_exotic_name[i])
                goto ContinueOuter;
        *w1 = *w1 + (dtid->no_words_in_exotic_name);
        return type_ID_with_this_dtid(dtid);
        ContinueOuter: ;
    }
    LOOP_OVER(dtid, data_type_id_rules) {
        if (dtid->is_constructor == FALSE) continue;
        if ((*w2-*w1+1) < dtid->no_words_in_exotic_plural) continue;
        for (i=0; i<dtid->no_words_in_exotic_plural; i++)
            if (lw_array[*w1+i].lw_identity != dtid->words_in_exotic_plural[i])
                goto ContinueOuterP;
        *w1 = *w1 + (dtid->no_words_in_exotic_plural);
        return type_ID_with_this_dtid(dtid);
        ContinueOuterP: ;
    }
    return UNKNOWN;
}
```

The function parse_constructor_name is called from 5/candd.

§**43.** And for the benefit of the index:

```
void data_type_print_long_name(OUTPUT_STREAM, int t) {
    int i;
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    for (i=0; i<dtid->no_words_in_exotic_name; i++) {
        if (i>0) WRITE(" ");
        WRITE("%s", vocab_get_exemplar(dtid->words_in_exotic_name[i], FALSE));
    }
}
void data_type_print_long_plural(OUTPUT_STREAM, int t) {
    int i;
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    for (i=0; i<dtid->no_words_in_exotic_plural; i++) {
        if (i>0) WRITE(" ");
        WRITE("%s", vocab_get_exemplar(dtid->words_in_exotic_plural[i], FALSE));
    }
}
```

The function data_type_print_long_name is called from 7/tids and 7/kix.

The function data_type_print_long_plural is called from 7/kix.

§**44.** And for the benefit of the data type interpreter:

```
void data_type_transcribe_name(char *buffer, data_type_id_rules *dtid, int lower_case) {
    buffer[0] = 0;
    int w1, w2;
    type_ID_copy_name(dtid->data_type_these_apply_to, &w1, &w2);
    if (w1 >= 0) {
        if (lower_case) print_raw_text_to_string(w1, w2, buffer);
        else print_text_to_string(w1, w2, buffer);
    } else {
        int l;
        if (type_ID_get_singular_parsing_name(dtid->data_type_these_apply_to)) {
            sprintf(buffer, "%s",
                vocab_get_exemplar(type_ID_get_singular_parsing_name(
                    dtid->data_type_these_apply_to), TRUE));
        } else
        for (l=0; l<dtid->no_words_in_exotic_name; l++) {
            sprintf(buffer+strlen(buffer), "%s ",
                vocab_get_exemplar(dtid->words_in_exotic_name[l], TRUE));
        }
    }
    if (dtid->is_constructor) sprintf(buffer+strlen(buffer), " value");
}
```

The function data_type_transcribe_name is called from 7/dti.

§**45. Questions about indexing.** For explanation, see the corresponding paragraphs of "Kinds of Value".

```
void data_type_set_specification_text(int t, char *desc) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    dtid->specification_text = desc;
}
char *data_type_get_specification_text(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->specification_text;
}
int data_type_get_index_priority(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->index_priority;
}
char *data_type_get_documentation_reference(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->documentation_reference;
}
int data_type_indexed_grey_if_empty(int t) {
    data_type_id_rules *dtid = type_ID_get_dtid(t);
    return dtid->indexed_grey_if_empty;
}
```

The function data_type_set_specification_text is called from 7/kov.
The function data_type_get_specification_text is called from 7/kov.
The function data_type_get_index_priority is called from 7/kix.
The function data_type_get_documentation_reference is called from 7/kix.
The function data_type_indexed_grey_if_empty is called from 7/kix.

Data Type Interpreter 7/dti

Purpose

To read in details of the built-in data types from the template file "Types.i6t", setting them up ready for use.

Template interpreter commands

```
22  {-callv:include_templates_for_types}
```

Definitions

¶1. Everyone loves a mini-language, so here is one. At the top level:

(a) Lines consisting of white space or whose first non-white space character is `!` are ignored as comments.

(b) A line ending with a colon `:` opens a new block. The text before the colon is the title of the block, except that the first character indicates its type:

(1) An asterisk means that the block is a template definition: for instance, `*PRINTING-ROUTINE:` says the block defines a template called `PRINTING-ROUTINE`. A template consists of Inform 7 source text which extends as far as the next `*END` line.

(2) A sharp sign `#` means that the block is a macro definition. For instance, `#UNIT:` says the block defines a template called `UNIT`. A macro is just a sequence of lines holding data type commands which continues to the beginning of the next block, or the end of the file.

(3) And otherwise the block is a type definition, but the optional opening character `+` marks the type as one which Inform requires the existence of. Thus `+NUMBER_TY:`, since Inform will crash if "Types.i6t" doesn't set this data type up, but `BOOJUMS_TY:` would validly declare a new data type called `BOOJUMS_TY` which isn't special to any of Inform's internals. The `+` signs are there as a help for hackers looking at "Types.i6t" and wondering what they can safely monkey with.

¶2. The body of a type definition is a sequence of one-line commands setting what properties the type has. These commands take the form of a name, a colon, and an operand; for instance,

```
i6-printing-routine-actions:DA_Number
used-in-let-with-cast-priority:10
can-be-type-of-global-variable:yes
```

As can be seen, the operands have different data types, and the possibilities are given here:

define `NO_TCA -1` *there's no operand*

define `BOOLEAN_TCA 1` *must be* `yes` *or* `no`

define `CCM_TCA 2` *a constant compilation method*

define `TEXT_TCA 3` *any text (no quotation marks or other delimiters are used)*

define `VOCABULARY_TCA 4` *any single word*

define `NUMERIC_TCA 5` *any decimal number*

define `TYPENAME_TCA 6` *any valid type number, such as* `NUMBER_TY`

define `TEMPLATE_TCA 7` *the name of a template whose definition is given in the file*

define `MACRO_TCA 8` *the name of a macro whose definition is given in the file*

¶3. When processing a command, we parse it into one of the following structures:

```
typedef struct single_type_command {
    struct type_command_definition *which_type_command;
    int boolean_argument;                                          where appropriate
    int numeric_argument;                                          where appropriate
    char *textual_argument;                                        where appropriate
    int ccm_argument;                                              where appropriate
    int no_vocabulary_arguments;                                   where appropriate
    struct vocabulary_entry *vocabulary_arguments[10];
    int domain_argument;                                           where appropriate
    int typename_argument;                                         where appropriate
    struct type_template_definition *template_argument;            where appropriate
    struct type_macro_definition *macro_argument;                  where appropriate
} single_type_command;
```

The structure single_type_command is private to this section.

¶4. A few of the commands connect pairs of data types together: for instance, when we write

```
cast:TEXT_ROUTINE_TY
```

in the definition block for TEXT_TY, we're saying that every text routine can always be cast implicitly to a text. There can be any number of these in the definition block, so we need somewhere to store details, and the following structure provides an entry in a linked list.

```
define CAST_SPECIFIC 1
define CAST_UNIVERSAL 2
define CAST_ASSIGNABLE 3
define CAST_ANY_WORD 4
define CAST_ANY_POINTER 5
```

```
typedef struct data_type_casting_rule {
    int cast_domain;                                          one of the above forms
    int single_value_casts;                                   when the form is CAST_SPECIFIC
    int casting_certainty;                                    ALWAYS_MATCH or only SOMETIMES_MATCH
    struct data_type_casting_rule *next_casting_rule;
} data_type_casting_rule;
```

The structure data_type_casting_rule is shared with 7/data.

¶5. And this is the analogous structure for giving I6 schemas to compare data of two different types:

```
typedef struct data_type_comparison_schema {
    int comparator;
    char *comparison_schema;
    struct data_type_comparison_schema *next_comparison_schema;
} data_type_comparison_schema;
```

The structure data_type_comparison_schema is shared with 7/data.

¶6. And, to cut to the chase, here is the complete table of commands:

```
type_command_definition table_of_type_commands[] = {
    { "block-data", block_data_TCC, BOOLEAN_TCA },
    { "can-be-type-of-global-variable", can_be_type_of_global_variable_TCC, BOOLEAN_TCA },
    { "can-coincide-with-property", can_coincide_with_property_TCC, BOOLEAN_TCA },
    { "can-exchange", can_exchange_TCC, BOOLEAN_TCA },
    { "defined-in-source-text", defined_in_source_text_TCC, BOOLEAN_TCA },
    { "has-i6-GPR", has_i6_GPR_TCC, BOOLEAN_TCA },
    { "has-native-constants", native_constants_TCC, BOOLEAN_TCA },
    { "indexed-grey-if-empty", indexed_grey_if_empty_TCC, BOOLEAN_TCA },
    { "is-a-unit", is_a_unit_TCC, BOOLEAN_TCA },
    { "is-an-enumeration", is_an_enumeration_TCC, BOOLEAN_TCA },
    { "is-constructor", is_constructor_TCC, BOOLEAN_TCA },
    { "is-incompletely-defined", is_incompletely_defined_TCC, BOOLEAN_TCA },
    { "is-template-variable", is_template_variable_TCC, BOOLEAN_TCA },
    { "multiple-block", multiple_block_TCC, BOOLEAN_TCA },
    { "named-values-created-with-assertions",
        named_values_created_with_assertions_TCC, BOOLEAN_TCA },
    { "quasinumerical", quasinumerical_TCC, BOOLEAN_TCA },
    { "unassignable", unassignable_TCC, BOOLEAN_TCA },
    { "used-in-let-without-casts", used_in_let_without_casts_TCC, BOOLEAN_TCA },
    { "uses-signed-comparisons", uses_signed_comparisons_TCC, BOOLEAN_TCA },
    { "constant-compilation-method", constant_compilation_method_TCC, CCM_TCA },
    { "default-value", default_value_TCC, TEXT_TCA },
    { "description", description_TCC, TEXT_TCA },
    { "distinguisher", distinguisher_TCC, TEXT_TCA },
    { "documentation-reference", documentation_reference_TCC, TEXT_TCA },
    { "explicit-i6-GPR", explicit_i6_GPR_TCC, TEXT_TCA },
    { "i6-printing-routine", i6_printing_routine_TCC, TEXT_TCA },
    { "i6-printing-routine-actions", i6_printing_routine_actions_TCC, TEXT_TCA },
    { "index-default-value", index_default_value_TCC, TEXT_TCA },
    { "loop-domain-schema", loop_domain_schema_TCC, TEXT_TCA },
    { "recognition-only-GPR", recognition_only_GPR_TCC, TEXT_TCA },
    { "specification-text", specification_text_TCC, TEXT_TCA },
    { "cast", cast_TCC, TYPENAME_TCA },
    { "comparison-schema", comparison_schema_TCC, TYPENAME_TCA },
    { "sometimes-cast", sometimes_cast_TCC, TYPENAME_TCA },
    { "modifying-adjective", modifying_adjective_TCC, VOCABULARY_TCA },
    { "plural", plural_TCC, VOCABULARY_TCA },
    { "singular", singular_TCC, VOCABULARY_TCA },
    { "heap-size-estimate", heap_size_estimate_TCC, NUMERIC_TCA },
    { "index-priority", index_priority_TCC, NUMERIC_TCA },
    { "template-variable-number", template_variable_number_TCC, NUMERIC_TCA },
    { "used-in-let-with-cast-priority", used_in_let_with_cast_priority_TCC, NUMERIC_TCA },
    { "apply-template", apply_template_TCC, TEMPLATE_TCA },
    { "apply-macro", apply_macro_TCC, MACRO_TCA },
    { NULL, -1, NO_TCA }
};
```

¶7. Where each legal command is defined with a block like so:

```
typedef struct type_command_definition {
    char *text_of_command;
    int opcode_number;
    int operand_type;
} type_command_definition;
```

The structure type_command_definition is private to this section.

¶8. Macros and templates have their definitions stored in structures thus:

```
typedef struct type_template_definition {
    char *template_name;                                  including the asterisk, e.g., "*PRINTING-ROUTINE"
    char *template_text;
    MEMORY_MANAGEMENT
} type_template_definition;

typedef struct type_macro_definition {
    char *type_macro_name;                                including the sharp, e.g., "#UNIT"
    int type_macro_line_count;
    struct single_type_command type_macro_line[MAX_TYPE_MACRO_LENGTH];
    MEMORY_MANAGEMENT
} type_macro_definition;
```

The structure type_template_definition is private to this section.
The structure type_macro_definition is private to this section.

¶9. And this makes a note to insert the relevant chunk of I7 source text later on. (We do this because type definitions are read very early on in Inform's run, whereas I7 source text can only be lexed later.)

```
typedef struct type_template_obligation {
    struct type_template_definition *remembered_template;        I7 source to insert...
    struct data_type_id_rules *remembered_dtid;                  ...concerning this data type
    MEMORY_MANAGEMENT
} type_template_obligation;
```

The structure type_template_obligation is private to this section.

¶10. Errors and limitations. In implementing the interpreter, we have to ask: who is it for? It occupies a strange position in being not quite for end users – the average Inform user will never know what the template is, let only how "Types.i6t" works – and yet not quite for internal use only, either. The main motivation for moving properties of data types out of Inform's program logic and into an external text file was to make it easier to verify that they were correctly described; but it was certainly also meant to give future Inform hackers – users who like to burrow into internals – scope for play.

The copy of "Types.i6t" supplied with Inform's standard distribution is, of course, correct. So how forgiving should we be, if errors are found in it? (These must result from mistakes by hackers.) To what extent should we allow arbitrarily complex constructions in "Types.i6t", as we would if it were a feature intended for end users?

We strike a sort of middle position. Inform will probably not crash if an incorrect "Types.i6t" is supplied, but it is free to throw internal errors or generate I6 code which fails to compile through I6. Inform will allow "Types.i6t" to be larger and more complex than the one supplied, but it is free to impose limits on this:

(a) The file can only be 32K long. (The standard distribution is about 20K.) Each line can be up to 1K long.
(b) The file must define 36 data types with fixed names (see below) which have relevance to the Inform internals, and is allowed to define up to 14 additional ones for use in the template layer. (In the standard distribution, it defines 8 of these extras.)
(c) The file can define any number of macros of up to 20 type commands each, but it must define at least the macros `#DEFAULTS`, `#NEW`, `#UNIT` and `#ENUMERATION`.
(d) The file can define any number of templates of I7 source text, with no restrictions (beyond that they have to fit into the file).

```
define MAX_TYPE_COMMAND_FILE_LENGTH 32768          the total length of "Types.i6t"
define MAX_DATA_TYPE_COMMAND_LENGTH 1024           length of any single line
define MAX_TYPE_MACRO_LENGTH 20                    maximum number of commands in any one macro
```

§1. Setting up the interpreter.

```
void data_type_initialise_type_interpreter(void) {
    initialise_type_text_archiver();
    initialise_let_casting_options();
    ⟨Assign source names to data types which are required to exist 2⟩;
}
```

The function data_type_initialise_type_interpreter is called from 7/tids.

§2. Here we assign textual names to correspond to internal NI source code constants, but it does not in itself create any types: all of that is done via type commands from the `Types.i6t` file, as fed into the type interpreter below. These are the ones marked with a `+` in the `Types.i6t` file.

```
define INFORM_KNOWS(name) type_ID_set_source_name(name, #name);
```

⟨Assign source names to data types which are required to exist 2⟩ ≡

```
    INFORM_KNOWS(ACTION_NAME_TY); INFORM_KNOWS(ACTIVITY_TY); INFORM_KNOWS(ANY_VALUE_TY);
    INFORM_KNOWS(ASSIGNABLE_VALUE_TY); INFORM_KNOWS(DESCRIPTION_OF_ACTION_TY);
    INFORM_KNOWS(EQUATION_TY); INFORM_KNOWS(EXTERNALFILE_TY); INFORM_KNOWS(FIGURENAME_TY);
    INFORM_KNOWS(OBJECT_DESCRIPTION_TY); INFORM_KNOWS(INDEXED_TEXT_TY);
    INFORM_KNOWS(INTERMEDIATE_TY);
    INFORM_KNOWS(KIND_OF_POINTER_VALUE_TY); INFORM_KNOWS(KIND_OF_WORD_VALUE_TY);
    INFORM_KNOWS(LIST_OF_TY); INFORM_KNOWS(NO_ENTRIES_TY); INFORM_KNOWS(NUMBER_TY);
    INFORM_KNOWS(OBJECT_TY); INFORM_KNOWS(PROPERTY_TY); INFORM_KNOWS(QUOT_TY);
    INFORM_KNOWS(RELATION_TY); INFORM_KNOWS(RULE_TY); INFORM_KNOWS(RULEBOOK_OUTCOME_TY);
    INFORM_KNOWS(RULEBOOK_TY); INFORM_KNOWS(SCENE_TY); INFORM_KNOWS(SNIPPET_TY);
    INFORM_KNOWS(SOUNDNAME_TY); INFORM_KNOWS(STORED_ACTION_TY);
    INFORM_KNOWS(TABLE_COLUMN_TY); INFORM_KNOWS(TABLE_TY); INFORM_KNOWS(TEST_ACTION_SPC);
    INFORM_KNOWS(TEXT_ROUTINE_TY); INFORM_KNOWS(TEXT_TY); INFORM_KNOWS(TIME_TY);
    INFORM_KNOWS(TRUTH_STATE_TY); INFORM_KNOWS(UNDERSTANDING_TY);
    INFORM_KNOWS(UNICODECHAR_TY); INFORM_KNOWS(USEOPTION_TY);
    INFORM_KNOWS(VALUE_DESCRIPTION_TY);
```

This code is used in §1.

§3. **The type command despatcher.** And this is where textual commands are received. (They come in from the template interpreter in Chapter 14; while they will almost always just be lines from "Types.i6t", they can in theory come from other template files as well.) Comments and blank lines have already been stripped out.

A template absorbs the raw text of its definition, and ends with `*END`; whereas a macro absorbs the parsed form of its commands, and continues to the next new heading. (Templates can't use the same end syntax because they often need to contain I7 phrase definitions, where lines end with colons.)

```
data_type_id_rules *current_dtid_described = NULL;
int total_type_command_lengths = 0;
void despatch_type_command(char *command) {
    ⟨Police the limits on type file and command length 4⟩;
    if (recording_a_type_template()) {
        if (strcmp(command, "*END") == 0) end_type_template();
        else record_into_type_template(command);
        return;
    }
    if (command[strlen(command)-1] == ':') {
```

```
        if (recording_a_type_macro()) end_type_macro();
        command[strlen(command)-1] = 0;                    remove the terminal colon
        ⟨Deal with the heading at the top of a type command block 5⟩;
        return;
    }
    single_type_command stc = parse_type_command(command);
    if (recording_a_type_macro()) record_into_type_macro(stc);
    else if (current_dtid_described) apply_type_command(stc, current_dtid_described);
    else internal_error("type command describes unspecified type");
}
```

The function despatch_type_command is called from 14/i6t.

§**4.**

⟨Police the limits on type file and command length 4⟩ ≡

```
    total_type_command_lengths += strlen(command) + 1;
    if (total_type_command_lengths >= MAX_TYPE_COMMAND_FILE_LENGTH)
        type_command_error(command, "too great an extent of type commands");
    if (strlen(command) >= MAX_DATA_TYPE_COMMAND_LENGTH)
        type_command_error(command, "type command too long");
```

This code is used in §3.

§**5.** The limit on the number of data types is reached when we collide with `BASE_OF_DATA_TYPE_IDS`. In practice we start at 50 and that's set to 100, which is why the limit comes out at 50.

⟨Deal with the heading at the top of a type command block 5⟩ ≡

```
    if (command[0] == '#') begin_type_macro(command);
    else if (command[0] == '*') begin_type_template(command);
    else if (command[0] == '+') {
        int x = type_ID_get_ID_for_source_name(command+1);
        if (x == UNKNOWN) internal_error("type command describes type with no known name");
        current_dtid_described = dtid_new(x);
    } else {
        int x = type_ID_get_ID_for_source_name(command);
        if (x != UNKNOWN) internal_error("type command describes already-known type");
        x = next_Types_dot_i6_number++;
        if (x >= BASE_OF_DATA_TYPE_IDS)
            internal_error("too many data types created by template commands");
        type_ID_set_source_name(x, archive_type_text(command));
        current_dtid_described = dtid_new(x);
    }
```

This code is used in §3.

§**6. Parsing single type commands.** Each command is read in as text, parsed and stored into a modest structure.

```
single_type_command parse_type_command(char *command) {
    char *argument = NULL;
    single_type_command stc;
    ⟨Parse line into command and argument, divided by a colon 8⟩;
    ⟨Initialise the STC to a blank command 7⟩;
    ⟨Identify the command being used 9⟩;
    switch(stc.which_type_command->operand_type) {
        case BOOLEAN_TCA: ⟨Parse a boolean argument for a type command 10⟩; break;
        case CCM_TCA: ⟨Parse a CCM argument for a type command 11⟩; break;
        case MACRO_TCA: ⟨Parse a macro name argument for a type command 17⟩; break;
        case NUMERIC_TCA: ⟨Parse a numeric argument for a type command 14⟩; break;
        case TEMPLATE_TCA: ⟨Parse a template name argument for a type command 16⟩; break;
        case TEXT_TCA: ⟨Parse a textual argument for a type command 12⟩; break;
        case TYPENAME_TCA: ⟨Parse a typename argument for a type command 15⟩; break;
        case VOCABULARY_TCA: ⟨Parse a vocabulary argument for a type command 13⟩; break;
    }
    return stc;
}
```

§**7.**

⟨Initialise the STC to a blank command 7⟩ ≡

```
stc.which_type_command = NULL;
stc.boolean_argument = NOT_APPLICABLE;
stc.numeric_argument = 0;
stc.textual_argument = NULL;
stc.ccm_argument = -1;
stc.no_vocabulary_arguments = 0;
stc.domain_argument = 0;
stc.typename_argument = UNKNOWN;
stc.macro_argument = NULL;
stc.template_argument = NULL;
```

This code is used in §6.

§**8.** Spaces and tabs after the colon are skipped; so a textual argument cannot begin with those characters, but that doesn't matter for the things we need.

⟨Parse line into command and argument, divided by a colon 8⟩ ≡

```
int i;
for (i=0; command[i]; i++)
    if (command[i] == ':') {
        command[i++]=0;
        while ((command[i] == ' ') || (command[i] == '\t')) i++;
        argument = command + i;
        break;
    }
if (argument == NULL) type_command_error(command, "type command without argument");
```

This code is used in §6.

§**9.** The following is clearly inefficient, but is not worth optimising. It makes about 20 calls to `strcmp` per command, and there are about 600 commands in a typical run of Inform, so the total cost is about 12,000 calls to `strcmp` with quite small strings as arguments – which is negligible for our purposes, so we neglect it.

⟨Identify the command being used 9⟩ ≡
```
int i;
for (i=0; table_of_type_commands[i].text_of_command; i++)
    if (strcmp(command, table_of_type_commands[i].text_of_command) == 0)
        stc.which_type_command = &(table_of_type_commands[i]);
if (stc.which_type_command == NULL) type_command_error(command, "no such type command");
```

This code is used in §6.

§**10.**

⟨Parse a boolean argument for a type command 10⟩ ≡
```
if (strcmp(argument, "yes") == 0) stc.boolean_argument = TRUE;
else if (strcmp(argument, "no") == 0) stc.boolean_argument = FALSE;
else type_command_error(command, "boolean type command takes yes/no argument");
```

This code is used in §6.

§**11.**

⟨Parse a CCM argument for a type command 11⟩ ≡
```
if (strcmp(argument, "none") == 0) stc.ccm_argument = NONE_CCM;
else if (strcmp(argument, "literal") == 0) stc.ccm_argument = LITERAL_CCM;
else if (strcmp(argument, "quantitative") == 0) stc.ccm_argument = QUANTITATIVE_CCM;
else if (strcmp(argument, "special") == 0) stc.ccm_argument = SPECIAL_CCM;
else type_command_error(command, "type command with unknown constant-compilation-method");
```

This code is used in §6.

§**12.**

⟨Parse a textual argument for a type command 12⟩ ≡
```
stc.textual_argument = archive_type_text(argument);
```

This code is used in §6.

§13.

⟨Parse a vocabulary argument for a type command 13⟩ ≡

```
char *latest_word;
int i;
stc.no_vocabulary_arguments = 0;
argument = archive_type_text(argument);
latest_word = argument;
for (i=0; argument[i]; i++)
    if (argument[i] == ' ') {
        argument[i++] = 0;
        stc.vocabulary_arguments[stc.no_vocabulary_arguments++]
            = vocab_entry_for_text(latest_word);
        latest_word = argument+i;
    }
stc.vocabulary_arguments[stc.no_vocabulary_arguments++] =
    vocab_entry_for_text(latest_word);
if (stc.no_vocabulary_arguments > MAX_WORDS_IN_TYPE_NAME) {
    type_command_error(command, "too many words in type command");
    stc.no_vocabulary_arguments = MAX_WORDS_IN_TYPE_NAME;
}
```

This code is used in §6.

§14.

⟨Parse a numeric argument for a type command 14⟩ ≡

```
stc.numeric_argument = atoi(argument);
```

This code is used in §6.

§15.

⟨Parse a typename argument for a type command 15⟩ ≡

```
int i;
for (i=0; argument[i]; i++)
    if ((argument[i] == '>') && (argument[i+1] == '>') && (argument[i+2] == '>')) {
        argument[i] = 0;
        stc.textual_argument = archive_type_text(argument+i+3);
        break;
    }
stc.typename_argument = UNKNOWN;
stc.domain_argument = CAST_SPECIFIC;
if (strcmp(argument, "<all>") == 0) stc.domain_argument = CAST_UNIVERSAL;
if (strcmp(argument, "<assignable>") == 0) stc.domain_argument = CAST_ASSIGNABLE;
if (strcmp(argument, "<word>") == 0) stc.domain_argument = CAST_ANY_WORD;
if (strcmp(argument, "<pointer>") == 0) stc.domain_argument = CAST_ANY_POINTER;
if (stc.domain_argument == CAST_SPECIFIC) {
    stc.typename_argument = type_ID_get_ID_for_source_name(argument);
    if (stc.typename_argument == UNKNOWN)
        type_command_error(command, "unknown type name in type command");
}
```

This code is used in §6.

§**16.**

⟨Parse a template name argument for a type command 16⟩ ≡

```
stc.template_argument = parse_type_template_name(argument);
if (stc.template_argument == NULL)
    type_command_error(command, "unknown template name in type command");
```

This code is used in §6.

§**17.**

⟨Parse a macro name argument for a type command 17⟩ ≡

```
stc.macro_argument = parse_type_macro_name(argument);
if (stc.macro_argument == NULL)
    type_command_error(command, "unknown template name in type command");
```

This code is used in §6.

§**18. Source text templates.** These are passages of I7 source text which can be inserted into the main source text at the request of any data type. An example would be:

```
*UNDERSTOOD-VARIABLE:
<type> understood is a <type> which varies.
*END
```

The template `*UNDERSTOOD-VARIABLE` contains only a single sentence of source text, and the idea is to make a new global variable associated with a given data type. Note that the text is not quite literal, because it can contain wildcards like `<type>`, which expands to the name of the kind of value in question: for instance, we might get

number understood is a number which varies.

There are a few limitations on what template text can include. Firstly, nothing with angle brackets in, except where a wildcard appears. Secondly, each sentence must end at the end of a line, and similarly the colon for any rule or other definition. Thus this template would fail:

```
*UNDERSTOOD-VARIABLE:
<type> understood is a <type> which
varies. To judge <type>: say "I judge [<type> understood]."
*END
```

because the first sentence ends in the middle of the second line, and the colon dividing the phrase header from its definition is also mid-line. The template must be reformatted thus to work:

```
*UNDERSTOOD-VARIABLE:
<type> understood is a <type> which varies.
To judge <type>:
    say "I judge [<type> understood]."
*END
```

§**19.** So, to begin:

```
type_template_definition *new_type_template(char *name) {
    type_template_definition *ttd = CREATE(type_template_definition);
    ttd->template_name = archive_type_text(name);
    return ttd;
}
type_template_definition *parse_type_template_name(char *name) {
    type_template_definition *ttd;
    LOOP_OVER(ttd, type_template_definition)
        if (strcmp(name, ttd->template_name) == 0) return ttd;
    return NULL;
}
```

§**20.** Here is the code which records templates, reading them as one line of plain text at a time. (In the above example, `record_into_type_template` would be called just once, with the single source text line.)

```
type_template_definition *current_type_template = NULL;                    the one now being recorded
int recording_a_type_template(void) {
    if (current_type_template) return TRUE;
    return FALSE;
}
void begin_type_template(char *name) {
    if (current_type_template) internal_error("first stt still recording");
    if (parse_type_template_name(name))
        internal_error("duplicate definition of source text template");
    current_type_template = new_type_template(name);
    current_type_template->template_text = begin_recording_type_text();
}
void record_into_type_template(char *line) {
    record_type_text(line);
}
void end_type_template(void) {
    if (current_type_template == NULL) internal_error("no stt currently recording");
    end_recording_type_text();
    current_type_template = NULL;
}
```

§**21.** So much for recording a template. To "play back", we need to take its text and squeeze it into the main source text, but there's a timing issue: when we read type commands it is very early in Inform's run, and the lexer may not even have started yet. So we simply remember our intention to insert the text:

```
void remember_to_transcribe_spec_template(type_template_definition *ttd, data_type_id_rules *dt) {
    type_template_obligation *tto = CREATE(type_template_obligation);
    tto->remembered_template = ttd;
    tto->remembered_dtid = dt;
}
```

§**22.** ...until now, when it's later on and the source text does indeed exist.

```
void include_templates_for_types(void) {
    type_template_obligation *tto;
    LOOP_OVER(tto, type_template_obligation)
        transcribe_type_template(tto->remembered_template, tto->remembered_dtid);
}
void transcribe_type_template(type_template_definition *ttd, data_type_id_rules *dtid) {
    if (ttd == NULL) internal_error("tried to transcribe missing source text template");
    char *p = ttd->template_text;
    int i = 0;
    while (p[i]) {
        char template_line_buffer[MAX_DATA_TYPE_COMMAND_LENGTH], terminator = 0;
        if ((p[i] == '\n') || (p[i] == ' ')) { i++; continue; }
        ⟨Transcribe one line of the template into the line buffer 23⟩;
        if (template_line_buffer[0]) {
            int x1 = lexer_wordcount, x2;
            feed_into_lexer(template_line_buffer, FALSE, FALSE);
            x2 = lexer_wordcount-1;
            if (terminator != 0) make_sentence_node(x1, x2, terminator);
        }
    }
    register_recently_lexed_phrases();
}
```

The function include_templates_for_types is invoked by a command in a `.i6t` template file.

§**23.** Inside template text, anything in angle brackets `<...>` is a wildcard. These cannot be nested and cannot include newlines. All other material is copied verbatim into the line buffer.

The only sentence terminators we recognise are full stop and colon; in particular we wouldn't recognise a stop inside quoted matter. This does not matter, since such things never come into type definitions.

⟨Transcribe one line of the template into the line buffer 23⟩ ≡

```
    int j=0;
    while ((p[i] != 0) && (p[i] != '\n')) {
        if (p[i] == '<') {
            char template_wildcard_buffer[MAX_DATA_TYPE_COMMAND_LENGTH];
            int k=0;
            i++;
            while ((p[i] != 0) && (p[i] != '\n') && (p[i] != '>'))
                template_wildcard_buffer[k++] = p[i++];
            template_wildcard_buffer[k++] = 0;
            i++;
            ⟨Transcribe the template wildcard 24⟩;
        } else template_line_buffer[j++] = p[i++];
    }
    template_line_buffer[j] = 0;
    if ((j>0) && ((template_line_buffer[j-1] == '.') || (template_line_buffer[j-1] == ':'))) {
        terminator = template_line_buffer[j-1];
        template_line_buffer[j-1] = 0;
    }
```

This code is used in §22.

§**24.** Only four wildcards are recognised:

⟨Transcribe the template wildcard 24⟩ ≡
```
if (strcmp(template_wildcard_buffer, "type") == 0)
    ⟨Transcribe the data type's name 25⟩
else if (strcmp(template_wildcard_buffer, "lower-case-type") == 0)
    ⟨Transcribe the data type's name in lower case 26⟩
else if (strcmp(template_wildcard_buffer, "type-number") == 0)
    ⟨Transcribe the data type's number 27⟩
else if (strcmp(template_wildcard_buffer, "printing-routine") == 0)
    ⟨Transcribe the data type's I6 printing routine 28⟩
else internal_error("no such source text template wildcard");
j = strlen(template_line_buffer);
```

This code is used in §23.

§**25.**

⟨Transcribe the data type's name 25⟩ ≡
```
data_type_transcribe_name(template_line_buffer+j, dtid, FALSE);
```

This code is used in §24.

§**26.**

⟨Transcribe the data type's name in lower case 26⟩ ≡
```
data_type_transcribe_name(template_line_buffer+j, dtid, TRUE);
```

This code is used in §24.

§**27.**

⟨Transcribe the data type's number 27⟩ ≡
```
sprintf(template_line_buffer+j, "%d", dtid->data_type_these_apply_to);
```

This code is used in §24.

§**28.**

⟨Transcribe the data type's I6 printing routine 28⟩ ≡
```
sprintf(template_line_buffer+j, "%s", dtid->dt_I6_identifier);
```

This code is used in §24.

§**29. Type macros.** These are much simpler, and are just lists of type commands grouped together under names.

```
type_macro_definition *current_type_macro = NULL;                    the one now being recorded
type_macro_definition *new_type_macro(char *name) {
    type_macro_definition *tmd = CREATE(type_macro_definition);
    tmd->type_macro_line_count = 0;
    tmd->type_macro_name = archive_type_text(name);
    return tmd;
}
type_macro_definition *parse_type_macro_name(char *name) {
    type_macro_definition *tmd;
    LOOP_OVER(tmd, type_macro_definition)
        if (strcmp(name, tmd->type_macro_name) == 0) return tmd;
    return NULL;
}
```

The function parse_type_macro_name is called from 7/data.

§**30.** And here once again is the code to record macros:

```
int recording_a_type_macro(void) {
    if (current_type_macro) return TRUE;
    return FALSE;
}
void begin_type_macro(char *name) {
    if (parse_type_macro_name(name))
        internal_error("duplicate definition of type command macro");
    current_type_macro = new_type_macro(name);
}
void record_into_type_macro(single_type_command stc) {
    if (current_type_macro == NULL)
        internal_error("type macro not being recorded");
    if (current_type_macro->type_macro_line_count >= MAX_TYPE_MACRO_LENGTH)
        internal_error("type macro contains too many lines");
    current_type_macro->type_macro_line[current_type_macro->type_macro_line_count++] = stc;
}
void end_type_macro(void) {
    if (current_type_macro == NULL) internal_error("ended type macro outside one");
    current_type_macro = NULL;
}
```

§**31.** Playing back is easier, since it's just a matter of despatching the stored commands in sequence to the relevant data type.

```
void play_back_type_macro(type_macro_definition *macro, data_type_id_rules *dtid) {
    int i;
    if (macro == NULL) internal_error("no such type macro to play back");
    for (i=0; i<macro->type_macro_line_count; i++)
        apply_type_command(macro->type_macro_line[i], dtid);
}
```

The function play_back_type_macro is called from 7/data.

§32. The type text archiver. Large chunks of the text in "Types.i6t" will need to exist permanently in memory, and this is where we make room for them. We can either archive a single C string, or go into recording mode and accept a series of them, concatenated with newlines dividing them. The worst-case storage requirement is that we need to store the entire file, so that's how much memory we put aside.

```
char archive_for_data_type_text[MAX_TYPE_COMMAND_FILE_LENGTH];                    text storage area
char *top_of_afdtt = archive_for_data_type_text;                                  next free character
int afdtt_recording_mode = FALSE;

void initialise_type_text_archiver(void) {
    archive_for_data_type_text[0] = 0;
}

char *archive_type_text(char *original) {
    if (afdtt_recording_mode) internal_error("can't archive while recording");
    char *new_location = top_of_afdtt;
    top_of_afdtt = top_of_afdtt + strlen(original) + 1;
    strcpy(new_location, original);
    return new_location;
}
```

§**33.** And here is recording mode:

```
char *begin_recording_type_text(void) {
    top_of_afdtt[0] = 0;
    afdtt_recording_mode = TRUE;
    return top_of_afdtt;
}

void record_type_text(char *line) {
    if (afdtt_recording_mode == FALSE) internal_error("can't record outside recording");
    sprintf(top_of_afdtt + strlen(top_of_afdtt), "%s\n", line);
}

void end_recording_type_text(void) {
    top_of_afdtt = top_of_afdtt + strlen(top_of_afdtt) + 1;
    afdtt_recording_mode = FALSE;
}
```

§34. Error messages.

```
void type_command_error(char *command, char *error) {
    LOG("Command error found at: %s\n", command);
    internal_error(error);
}
```

§**35. Applying type commands.** We take a single type command and apply it to a given data type.

```
define apply_macro_TCC 1
define apply_template_TCC 2
define block_data_TCC 3
define can_be_type_of_global_variable_TCC 4
define can_coincide_with_property_TCC 5
define can_exchange_TCC 6
define cast_TCC 7
define comparison_schema_TCC 8
define constant_compilation_method_TCC 9
define default_value_TCC 10
define defined_in_source_text_TCC 11
define description_TCC 12
define distinguisher_TCC 13
define documentation_reference_TCC 14
define explicit_i6_GPR_TCC 15
define has_i6_GPR_TCC 16
define heap_size_estimate_TCC 17
define i6_printing_routine_actions_TCC 18
define i6_printing_routine_TCC 19
define index_default_value_TCC 20
define indexed_grey_if_empty_TCC 21
define index_priority_TCC 22
define is_a_unit_TCC 23
define is_an_enumeration_TCC 24
define is_constructor_TCC 25
define is_incompletely_defined_TCC 26
define is_template_variable_TCC 27
define loop_domain_schema_TCC 28
define modifying_adjective_TCC 29
define multiple_block_TCC 30
define named_values_created_with_assertions_TCC 31
define native_constants_TCC 32
define plural_TCC 33
define quasinumerical_TCC 34
define recognition_only_GPR_TCC 35
define singular_TCC 36
define sometimes_cast_TCC 37
define specification_text_TCC 38
define template_variable_number_TCC 39
define unassignable_TCC 40
define used_in_let_with_cast_priority_TCC 41
define used_in_let_without_casts_TCC 42
define uses_signed_comparisons_TCC 43

void apply_type_command(single_type_command stc, data_type_id_rules *dtid) {
    if (stc.which_type_command == NULL) internal_error("null STC command");
    int tcc = stc.which_type_command->opcode_number;
    ⟨Apply type macros or transcribe type templates on request 36⟩;
    ⟨Most type commands simply set a field in the DTID structure 37⟩;
    ⟨A few type commands contribute to linked lists in the DTID structure 38⟩;
    ⟨And the rest fill in fields in the DTID structure in miscellaneous other ways 39⟩;
```

```
    internal_error("unimplemented type command");
}
```

§36.

⟨Apply type macros or transcribe type templates on request 36⟩ ≡

```
    switch (tcc) {
        case apply_template_TCC:
            if (text_loaded_from_source) transcribe_type_template(stc.template_argument, dtid);
            else remember_to_transcribe_spec_template(stc.template_argument, dtid);
            return;
        case apply_macro_TCC:
            play_back_type_macro(stc.macro_argument, dtid);
            return;
    }
```

This code is used in §35.

§37.

```
define SET_BOOLEAN_FIELD(field) case field##_TCC: dtid->field = stc.boolean_argument; return;
define SET_INTEGER_FIELD(field) case field##_TCC: dtid->field = stc.numeric_argument; return;
define SET_TEXTUAL_FIELD(field) case field##_TCC: dtid->field = stc.textual_argument; return;
define SET_CCM_FIELD(field) case field##_TCC: dtid->field = stc.ccm_argument; return;
```

⟨Most type commands simply set a field in the DTID structure 37⟩ ≡

```
    switch (tcc) {
        SET_BOOLEAN_FIELD(block_data)
        SET_BOOLEAN_FIELD(can_be_type_of_global_variable)
        SET_BOOLEAN_FIELD(can_coincide_with_property)
        SET_BOOLEAN_FIELD(can_exchange)
        SET_BOOLEAN_FIELD(defined_in_source_text)
        SET_BOOLEAN_FIELD(has_i6_GPR)
        SET_BOOLEAN_FIELD(indexed_grey_if_empty)
        SET_BOOLEAN_FIELD(is_a_unit)
        SET_BOOLEAN_FIELD(is_an_enumeration)
        SET_BOOLEAN_FIELD(is_constructor)
        SET_BOOLEAN_FIELD(is_incompletely_defined)
        SET_BOOLEAN_FIELD(is_template_variable)
        SET_BOOLEAN_FIELD(multiple_block)
        SET_BOOLEAN_FIELD(named_values_created_with_assertions)
        SET_BOOLEAN_FIELD(native_constants)
        SET_BOOLEAN_FIELD(quasinumerical)
        SET_BOOLEAN_FIELD(unassignable)
        SET_BOOLEAN_FIELD(used_in_let_without_casts)
        SET_BOOLEAN_FIELD(uses_signed_comparisons)
        SET_INTEGER_FIELD(heap_size_estimate)
        SET_INTEGER_FIELD(index_priority)
        SET_INTEGER_FIELD(template_variable_number)
        SET_CCM_FIELD(constant_compilation_method)
        SET_TEXTUAL_FIELD(default_value)
        SET_TEXTUAL_FIELD(distinguisher)
        SET_TEXTUAL_FIELD(documentation_reference)
        SET_TEXTUAL_FIELD(explicit_i6_GPR)
```

```
    SET_TEXTUAL_FIELD(index_default_value)
    SET_TEXTUAL_FIELD(loop_domain_schema)
    SET_TEXTUAL_FIELD(recognition_only_GPR)
    SET_TEXTUAL_FIELD(specification_text)
}
```

This code is used in §35.

§**38.** The DTID has two linked lists relating it to DTIDs of other data types: one for casting, one for comparisons.

⟨A few type commands contribute to linked lists in the DTID structure 38⟩ ≡

```
if ((tcc == cast_TCC) || (tcc == sometimes_cast_TCC)) {
    data_type_casting_rule *dtcr = CREATE(data_type_casting_rule);
    dtcr->next_casting_rule = dtid->first_casting_rule;
    dtid->first_casting_rule = dtcr;
    dtcr->cast_domain = stc.domain_argument;
    if (tcc == cast_TCC) dtcr->casting_certainty = ALWAYS_MATCH;
    else dtcr->casting_certainty = SOMETIMES_MATCH;
    if (dtcr->cast_domain == CAST_SPECIFIC)
        dtcr->single_value_casts = stc.typename_argument;
    return;
}
if (tcc == comparison_schema_TCC) {
    data_type_comparison_schema *dtcs = CREATE(data_type_comparison_schema);
    dtcs->next_comparison_schema = dtid->first_comparison_schema;
    dtid->first_comparison_schema = dtcs;
    dtcs->comparator = stc.typename_argument;
    dtcs->comparison_schema = stc.textual_argument;
    return;
}
```

This code is used in §35.

§**39.**

⟨And the rest fill in fields in the DTID structure in miscellaneous other ways 39⟩ ≡

```
switch (tcc) {
    case description_TCC:
        type_ID_set_description(dtid->data_type_these_apply_to, stc.textual_argument);
        return;
    case i6_printing_routine_TCC:
        if (strlen(stc.textual_argument) > 31) internal_error("overlong I6 identifier");
        else strcpy(dtid->dt_I6_identifier, stc.textual_argument);
        return;
    case i6_printing_routine_actions_TCC:
        if (strlen(stc.textual_argument) > 31) internal_error("overlong I6 identifier");
        else strcpy(dtid->name_of_printing_rule_ACTIONS, stc.textual_argument);
        return;
    case singular_TCC:
        if (stc.no_vocabulary_arguments == 1) {
            type_ID_set_parsing_names(dtid->data_type_these_apply_to,
                stc.vocabulary_arguments[0], NULL);
        } else {
```

```
            int i;
            for (i=0; i<stc.no_vocabulary_arguments; i++) {
                dtid->words_in_exotic_name[i] = stc.vocabulary_arguments[i];
                vocab_set_flags(stc.vocabulary_arguments[i], DESIGNED_TYPE_MC);
            }
            dtid->no_words_in_exotic_name = stc.no_vocabulary_arguments;
        }
        return;
    case plural_TCC:
        if (stc.no_vocabulary_arguments == 1) {
            type_ID_set_parsing_names(dtid->data_type_these_apply_to,
                NULL, stc.vocabulary_arguments[0]);
        } else {
            int i;
            for (i=0; i<stc.no_vocabulary_arguments; i++) {
                dtid->words_in_exotic_plural[i] = stc.vocabulary_arguments[i];
                vocab_set_flags(stc.vocabulary_arguments[i], DESIGNED_TYPE_MC);
            }
            dtid->no_words_in_exotic_plural = stc.no_vocabulary_arguments;
        }
        return;
    case modifying_adjective_TCC:
        if (stc.no_vocabulary_arguments != 1)
            internal_error("modifying adjectives must be single words");
        if (dtid->no_modifying_adjectives >= MAX_MODIFYING_ADJECTIVES)
            internal_error("too many modifying adjectives");
        dtid->modifying_adjectives[dtid->no_modifying_adjectives++]
            = stc.vocabulary_arguments[0];
        return;
    case used_in_let_with_cast_priority_TCC:
        register_let_cast_option(dtid->data_type_these_apply_to,
            stc.numeric_argument);
        return;
}
```

This code is used in §35.

§**40.** And that completes the data type interpreter.

Dimensions 7/dim

Purpose

To keep a small database indicating the physical dimensions of numerical values, and how they combine: for instance, allowing us to specify that a length times a length is an area.

Definitions

¶1. Dimension in this sense is a term drawn from physics. The idea is that when quantities are multiplied together, their natures are combined as well as the actual numbers involved. For instance, in

$$v = f\lambda$$

if the frequency f of a wave is measured in Hz (counts per second), and the wavelength λ in m, then the velocity v must be measured in m/s: and that is indeed a measure of velocity, so this looks right. We can tell that the formula

$$v = f^2\lambda$$

must be wrong because it would result in an acceleration. Physicists use the term "dimensions" much as Inform uses the term "kinds of value".

Inform applies dimension-checking to all "quasinumerical" KOVs – those which can be expressed numerically. The choice of which KOVs are quasinumerical is all done in "Types.i6t", not built into Inform at the compiler level, but the standard setup makes number, time, intermediate results of calculations (see below), and what the Inform documentation calls "units" – kinds of value specified by literal patterns.

¶2. Inform divides quasinumerical KOVs into three: base units, derived units with dimensions, and dimensionless units. In the default setup provided by "Types.i6t", a typical run has one base unit ("time"), one dimensionless unit ("number") and – unless the source text does something very strange – no derived units.

It would no doubt be cool to distinguish these by applying Buckingham's π-theorem to all the equations we need to use, but this is a tricky technique and does not always produce the "natural" results which people expect. Instead, Inform requires the writer to specify explicitly how units combine.

Number and time are built-in special cases. Further base units are created every time source text like this is read:

> Mass is a kind of value. 1kg specifies a mass.

Derived units only come about when the source text specifies a multiplication rule. For instance, when Inform reads

> A mass times an acceleration specifies a force.

it chooses one of the three units – say, force – and derives that from the others.

Multiplication rules are stored in a linked list associated with left operand; so that the rule A times B specifies C causes (B, C) to be stored in the list of `multiplications` belonging to A.

```
typedef struct dimensional_rules {
    struct dimensional_rule *multiplications;
} dimensional_rules;
typedef struct dimensional_rule {
    int word_ref1, word_ref2;
    int right;
    int outcome;
    struct dimensional_rule *next;
} dimensional_rule;
```

The structure dimensional_rules is private to this section.
The structure dimensional_rule is private to this section.

¶3. The derivation process can be seen in action by feeding Inform definitions of the SI units (see the test case `SIUnits-G`) and looking at the output of:

Test dimensions (internal) with –.

(The dash is meaningless – this is a test with no input.) In the output, we see that

```
Base units: time, length, mass, elapsed time, electric current, temperature, luminosity
Derived units:
frequency = (elapsed time)-1
force = (length).(mass).(elapsed time)-2
energy = (length)2.(mass).(elapsed time)-2
pressure = (length)-1.(mass).(elapsed time)-2
power = (length)2.(mass).(elapsed time)-3
electric charge = (elapsed time).(electric current)
voltage = (length)2.(mass).(elapsed time)-3.(electric current)-1
```

...and so on. Those expressions on the right hand sides are "derived units", where the numbers are powers, so that negative numbers mean division. It's easy to see why we want to give names and notations for some of these derived units – imagine going into a cycle shop and asking for a $5\mathrm{m}^2 \cdot \mathrm{kg} \cdot \mathrm{s}^{-3} \cdot \mathrm{A}^{-1}$ battery.

¶4. A "dimensionless" quantity is one which is just a number, and is not a physical measurement as such. In an equation like

$$K = \frac{mv^2}{2}$$

the 2 is clearly dimensionless, but other possibilities also exist. The arc length of part of a circle at radius r drawn out to angle θ (if measured in radians) is given by:

$$A = \theta r$$

Here A and r are both lengths, so the angle θ must be dimensionless. But clearly it's not quite conceptually the same thing as an ordinary number. Inform creates new dimensionless quantities this way, too:

Angle is a kind of value. 1 rad specifies an angle. Length times angle specifies a length.

Inform is not quite so careful about distinguishing dimensionless quantities as some physicists might be. The official SI units distinguish angle, measured in radians, and solid angle, in steradians, writing them as having units $\mathrm{m} \cdot \mathrm{m}^{-1}$ and $\mathrm{m}^2 \cdot \mathrm{m}^{-2}$ respectively – one is a ratio of lengths, the other of areas. Inform cancels the units and sees them as dimensionally equal. So if we write:

Solid angle is a kind of value. 1 srad specifies an solid angle. Area times solid angle specifies an area.

then Inform treats angle and solid angle as having the same multiplicative properties – but it still allows variables to have either one as a kind of value, and prints them differently.

Note that a dimensionless unit (other than `NUMBER_TY`) can only get that way by derivation, so it is always a derived unit, never a base unit.

¶5. In the process of calculations, we often need to create other and nameless units as partial answers of calculations. Consider the kinetic energy equation

$$K = \frac{mv^2}{2}$$

being evaluated the way a computer does it, one step at a time. One way takes the mass, multiplies by the velocity to get a momentum, multiplies by the velocity again to get energy, then divides by a dimensionless constant. But another way would be to square the velocity first, then multiply by mass to get energy, then halve. If we do it that way, what units are the squared velocity in? The answer has to be

```
(length)2.(elapsed time)-2
```

but that's a unit which isn't useful for much, and doesn't have any everyday name. Inform creates what are called "intermediate KOVs" like this in order to be able to represent the kinds of intermediate values which turn up in calculation. They use the atomic KOV number `INTERMEDIATE_TY`, they are nameless, and the user isn't allowed to store the results permanently. (They can't be the KOV of a global variable, a table column, and so on.) If the user wants to deal with such values on a long-term basis, he must give them a name, like this:

```
Funkiness is a kind of value. 1 Claude is a funkiness. A velocity times a velocity specifies a funkiness.
```

¶6. Expressions like $\mathrm{m}^2 \cdot \mathrm{kg}$ are stored inside Inform as sequences of ordered pairs in the form

$$((B_1, p_1), (B_2, p_2), ..., (B_k, p_k))$$

where each B_i is the type ID of a base unit, each p_i is a non-zero integer, and $B_1 < B_2 < ... < B_k$. For instance, energy would be

$$((\text{length}, 2), (\text{mass}, 1), (\text{elapsed time}, -2)).$$

Every physically different derived unit has a unique and distinct sequence. This is only true because a unit sequence is forbidden to contain derived units. For instance, specific heat capacity looks as if it is written with two different units in physics:

$$\mathrm{J} \cdot \mathrm{K}^{-1} \cdot \mathrm{kg}^{-1} \quad = \quad \mathrm{m}^2 \cdot \mathrm{s}^{-2} \cdot \mathrm{K}^{-1}$$

But this is because the Joule is a derived unit. Substituting $\mathrm{J} = \mathrm{m}^2 \cdot \mathrm{kg} \cdot \mathrm{s}^{-2}$ to get back to base units shows that both sides would be computed as the same unit sequence.

The case $k = 0$, the empty sequence, is not only legal but important: it is the derivation for a dimensionless unit. (As discussed above, Inform doesn't see different dimensionless units as being physically different.)

```
typedef struct unit_pair {
    int base_unit;                                   a type ID number
    int power;                                       a non-zero integer
} unit_pair;
```

The structure unit_pair is private to this section.

¶7. The following is a hard limit, but really not a problematic one. The entire SI system has only 7 base units, and the only named scientific unit I've seen which has even 5 terms in its derivation is molar entropy, a less than everyday chemical measure ($\mathrm{kg} \cdot \mathrm{m}^2 \cdot \mathrm{s}^{-2} \cdot \mathrm{K}^{-1} \cdot \mathrm{mol}^{-1}$, if you're taking notes).

```
define MAX_BASE_UNITS_IN_SEQUENCE 16

typedef struct unit_sequence {
    int no_unit_pairs;                                    in range 0 to MAX_BASE_UNITS_IN_SEQUENCE
    struct unit_pair unit_pairs[MAX_BASE_UNITS_IN_SEQUENCE];
    int scaling_factor;                                   see discussion of scaling below
} unit_sequence;
```

The structure unit_sequence is shared with 7/dim.

¶8. Manipulating units like $\mathrm{m}^2 \cdot \mathrm{kg} \cdot \mathrm{s}^{-2}$ looks a little like manipulating formal polynomials in several variables, and of course that isn't an accident. Another way of thinking of the above is that we have a ring R of underlying numbers but that all arithmetic is done in a larger ring. For each unit extend by R by a pair of formal variables U_i and U_i^{-1}, and then quotient by the ideal generated by $U_j U_j^{-1}$ (so that they are indeed reciprocals of each other, as the notation suggests) and also by all of the derivations we know of. Thus Inform calculates in the ring:

$$I = R[U_1, U_2, ..., U_n, U_1^{-1}, ..., U_n^{-1}]/(U_1U_1^{-1}, U_2U_2^{-1}, ..., U_nU_n^{-1}, D_1, D_2, ..., D_i).$$

It does that in practice by eliminating all of the U_i and U_i^{-1} which are derived, so that it's left with just

$$I = R[U_1, U_2, ..., U_k, U_1^{-1}, ..., U_k^{-1}]/(U_1U_1^{-1}, U_2U_2^{-1}, ..., U_kU_k^{-1}).$$

For instance, given seconds, Watts and Joules,

$$I = R[\mathrm{s}, \mathrm{s}^{-1}, \mathrm{W}, \mathrm{W}^{-1}, \mathrm{J}, \mathrm{J}^{-1}]/(\mathrm{ss}^{-1} = 1, \mathrm{WW}^{-1} = 1, \mathrm{JJ}^{-1} = 1, \mathrm{sW} = \mathrm{J})$$

which by substituting all occurrences of J can be reduced to:

$$I = R[\mathrm{s}, \mathrm{s}^{-1}, \mathrm{W}, \mathrm{W}^{-1}]/(\mathrm{ss}^{-1} = 1, \mathrm{WW}^{-1} = 1).$$

Of course there are other ways to calculate I – we could have eliminated any of the three units and kept the other two.

If the derivations were ever more complex than $AB = C$, we might have to use some elegant algorithms for calculating Gröbner bases in order to determine I. But Inform's syntax is such that the writer of the source text gives us the simplest possible description of the ideal, so no such fun occurs.

What does this ring look like? Because we are not allowed to add terms with different powers of the variables, we only ever deal with monomials. Thus we can form $2\mathrm{Ws}^{-1} + 7\mathrm{Ws}^{-1}$, but Inform forbids us to form (say) $6\mathrm{J} + 7\mathrm{W}$. We can therefore picture the ring I as a great mass of parallel copies of R. Dimensionless values all live in R itself, while energies all live in R.s.W, powers in R.W and so on. Addition and subtraction slide values around within their own parallel copies, but multiplication and division move them from one to another. The computation $v_1 v_2$ is done in general by calculating the numerical part (in R) at run-time, and the units (the choice of which copy of R) at compile-time.

¶9. But enough abstraction: time for some arithmetic. Inform performs checking whenever values from two different KOVS are combined by any of eight arithmetic operations, numbered as follows. The numbers must *not* be changed without amending the definitions of "plus" and so on in the Standard Rules.

```
define NO_OPERATIONS 8
define PLUS_OPERATION 0                    addition
define MINUS_OPERATION 1                   subtraction
define TIMES_OPERATION 2                   multiplication
define DIVIDE_OPERATION 3                  division
define REMAINDER_OPERATION 4               remainder after division
define APPROXIMATION_OPERATION 5           "X to the nearest Y"
define ROOT_OPERATION 6                    square root – a unary operation
define CUBEROOT_OPERATION 7                cube root – similarly unary
define EQUALS_OPERATION 8                  set equal – used only in equations
define POWER_OPERATION 9                   raise to integer power – used only in equations
define UNARY_MINUS_OPERATION 10            unary minus – used only in equations
```

¶10. The following is associated with "total...", as in "the total weight of things on the table", but that's a dodge used in the Standard Rules, and for dimensional purposes we ignore it.

```
define TOTAL_OPERATION 11                  not really one of the above
```

¶11. There are two reasons why Inform monitors arithmetic: to keep track of how it changes KOVs, and to preserve scaling factors.

We start from the principle that not every arithmetic operation can be done, and that even when it can, the result may have a different KOV than the operand(s) had. For one thing, every arithmetic operation requires that its operands are quasinumerical – Inform won't allow a text to be multiplied by a sound effect. (Occasionally we have thought about allowing text to be duplicated by multiplication – 2 times "zig" would be "zigzig", and maybe similarly for lists – but it always seemed a pose, fleetingly cool but seldom useful, and more likely to be used by mistake than intentionally.)

Other restrictions are also applied. For instance, a time cannot be added to a number, or vice versa; addition, subtraction and approximation require both operands to have the same units.

¶12. Finally, scaling. `NUMBER_TY` is straightforwardly an integer data type: it holds whole numbers. But other quasinumerical data types can be stored using scaled, fixed-point arithmetic. In general for each named unit U (base or derived) there is a positive integer k_U such that the true value v is stored at run-time as the I6 integer $k_U v$. We call this the scaled value.

For example, if the text reads:

```
Force is a kind of value. 1N specifies a force scaled up by 1000.
```

then $k = 1000$ and the value 1N will be stored at run-time as `1000`; forces can thus be calculated to a true value accuracy of at best 0.001N, stored at run-time as `1`.

It must be emphasised that this is scaled, fixed-point arithmetic: there are no mantissas or exponents. In such schemes the scale factor is usually 2^{16} or some similar power of 2, but here we want to use exactly the scale factors laid out by the source text – partly because the user knows best, partly so that it is unambiguous how to print values, partly so that source text like "0.001N" determines an exact value rather than being approximated by a binary equivalent.

§**1. Multiplication lists.** The linked lists of multiplication rules begin empty for every data type:

```
void dim_initialise(dimensional_rules *dimrs) {
    dimrs->multiplications = NULL;
}
```

The function dim_initialise is called from 7/data.

§**2.** And this adds a new one to the relevant list:

```
void record_multiplication_rule(int left, int right, int outcome) {
    dimensional_rules *dimrs = dtid_get_dim_rules(left);
    dimensional_rule *dimr;
    for (dimr = dimrs->multiplications; dimr; dimr = dimr->next)
        if (dimr->right == right) {
            sentence_problem(_P_(C7DimensionRedundant),
                "multiplication rules can only be given once",
                "and this combination is already established.");
            return;
        }
    dimensional_rule *dimr_new = CREATE(dimensional_rule);
    dimr_new->right = right;
    dimr_new->outcome = outcome;
    dimr_new->word_ref1 = -1; dimr_new->word_ref2 = -1;
    if (current_sentence) {
        dimr_new->word_ref1 = current_sentence->word_ref1;
        dimr_new->word_ref2 = current_sentence->word_ref2;
    }
    dimr_new->next = dimrs->multiplications;
    dimrs->multiplications = dimr_new;
}
```

§**3.** The following loop-header macro iterates through the possible triples (L, R, O) of multiplication rules $L \times R = O$.

```
define LOOP_OVER_MULTIPLICATIONS(left_operand, right_operand, outcome_type, wn)
    dimensional_rules *dimrs;
    dimensional_rule *dimr;
    LOOP_OVER_POSSIBLE_TYPE_IDS(left_operand)
        for (dimrs = dtid_get_dim_rules(left_operand),
            dimr = (dimrs)?(dimrs->multiplications):NULL,
            wn = (dimr)?(dimr->word_ref1):-1,
            right_operand = (dimr)?(dimr->right):0,
            outcome_type = (dimr)?(dimr->outcome):0;
            dimr;
            dimr = dimr->next,
            wn = (dimr)?(dimr->word_ref1):-1,
            right_operand = (dimr)?(dimr->right):0,
            outcome_type = (dimr)?(dimr->outcome):0)
```

§4. And this is where the user asks for a multiplication to come out in a particular way:

```
void dim_set_multiplication(kind_of_value *left, kind_of_value *right,
    kind_of_value *outcome) {
    if ((is_kovcon(left)) || (is_kovcon(right)) || (is_kovcon(outcome))) {
        sentence_problem(_P_(C7DimensionNotBaseKOV),
            "multiplication rules can only involve simple kinds of value",
            "rather than complicated ones such as lists of other values.");
        return;
    }
    if ((kov_quasinumerical(left) == FALSE) ||
        (kov_quasinumerical(right) == FALSE) ||
        (kov_quasinumerical(outcome) == FALSE)) {
        sentence_problem(_P_(C7NonDimensional),
            "multiplication rules can only be given between kinds of "
            "value which are known to be numerical",
            "and not all of these are. Saying something like 'Pressure is a "
            "kind of value.' is not enough - you may think 'pressure' ought "
            "to be numerical, but Inform doesn't know that yet. You need "
            "to add something like '100 Pa specifies a pressure.' before "
            "Inform will realise.");
        return;
    }
    int left_adt = kov_get_dimensional_data_type(left);
    int right_adt = kov_get_dimensional_data_type(right);
    int outcome_adt = kov_get_dimensional_data_type(outcome);
    record_multiplication_rule(left_adt, right_adt, outcome_adt);
    if ((left_adt == outcome_adt) && (right_adt == NUMBER_TY)) return;
    if ((right_adt == outcome_adt) && (left_adt == NUMBER_TY)) return;
    make_unit_derivation(left_adt, right_adt, outcome_adt);
}
```

The function dim_set_multiplication is called from 5/litp.

§5. **Unary operations.** All we need to know is which ones are unary, in fact, and:

```
int arithmetic_op_is_unary(int op) {
    switch (op) {
        case CUBEROOT_OPERATION:
        case ROOT_OPERATION:
        case UNARY_MINUS_OPERATION:
            return TRUE;
    }
    return FALSE;
}
```

The function arithmetic_op_is_unary is called from 7/tc and 10/eqns.

§**6. Euclid's algorithm.** In my entire life, I believe this is the only time I have ever actually used Euclid's algorithm for the GCD of two natural numbers. I've never quite understood why textbooks take this as somehow the typical algorithm. My maths students always find it a little oblique, despite the almost trivial proof that it works. I find it hard to visualise myself, for that matter. And then, consider that the average number of iterations τ_n, in effect its running time, is known to be

$$\tau_n = \frac{12 \log 2}{\pi^2} \log n + (4P + 5/2) + O(n^{-\frac{1}{6}+\epsilon})$$

for any $\epsilon > 0$, where P is defined in terms of an integral, Euler's constant, and an evaluation of the derivative of the Riemann zeta function – see D. E. Knuth, 'Evaluation of Porter's Constant', reprinted in *Selected Papers on Analysis of Algorithms* (Stanford: CSLI Lecture Notes 102, 2000). In practice, a shade under $\log n$ steps, then, which is nicely quick. But I don't look at the code and immediately see this, myself.

```
int gcd(int m, int n) {
    if ((m<1) || (n<1)) internal_error("applied gcd outside natural numbers");
    while (TRUE) {
        int rem = m%n;
        if (rem == 0) return n;
        m = n; n = rem;
    }
}
```

The function gcd is called from 5/litp.

§**7.** The sequence of operation here is to reduce the risk of integer overflows when multiplying `m` by `n`.

```
int lcm(int m, int n) {
    return (m/gcd(m, n))*n;
}
```

§**8. Unit sequences.** Given a base type B, convert it to a unit sequence: $B = B^1$, so we get a sequence with a single pair: $((B, 1))$. Uniquely, `NUMBER_TY` is born derived and dimensionless, though, so that comes out as the empty sequence.

```
unit_sequence data_spec_to_unit_sequence(int t) {
    unit_sequence us;
    if (t == NUMBER_TY) {
        us.no_unit_pairs = 0;
        us.unit_pairs[0].base_unit = 0; us.unit_pairs[0].power = 0;     redundant, but appeases gcc -O2
    } else {
        us.no_unit_pairs = 1;
        us.unit_pairs[0].base_unit = t; us.unit_pairs[0].power = 1;
    }
    return us;
}
```

The function data_spec_to_unit_sequence is called from 7/data.

§**9.** As noted above, two units represent dimensionally equivalent physical quantities if and only if they are identical, which makes comparison easy:

```
int compare_unit_sequences(unit_sequence *ik1, unit_sequence *ik2) {
    int i;
    if (ik1 == ik2) return TRUE;
    if ((ik1 == NULL) || (ik2 == NULL)) return FALSE;
    if (ik1->no_unit_pairs != ik2->no_unit_pairs) return FALSE;
    for (i=0; i<ik1->no_unit_pairs; i++)
        if ((ik1->unit_pairs[i].base_unit != ik2->unit_pairs[i].base_unit) ||
            (ik1->unit_pairs[i].power != ik2->unit_pairs[i].power))
                return FALSE;
    return TRUE;
}
```

The function compare_unit_sequences is called from 7/kov.

§**10.** We now have three fundamental operations we can perform on unit sequences. First, we can multiply them: that is, we store in `result` the unit sequence representing $X_1^{s_1} X_2^{s_2}$, where X_1 and X_2 are represented by unit sequences `us1` and `us2`.

So the case $s_1 = s_2 = 1$ represents multiplying X_1 by X_2, while $s_1 = 1, s_2 = -1$ represents dividing X_1 by X_2. But we can also raise to higher powers.

Our method relies on noting that

$$X_1 = T_{11}^{p_{11}} \cdot T_{12}^{p_{12}} \cdots T_{1n}^{p_{1n}}, \qquad X_2 = T_{21}^{p_{21}} \cdot T_{22}^{p_{22}} \cdots T_{2m}^{p_{2m}}$$

where $T_{11} < T_{12} < ... < T_{1n}$ and $T_{21} < T_{22} < ... < T_{2m}$. We can therefore merge the two in a single pass.

On each iteration of the loop the variables `i1` and `i2` are our current read positions in each sequence, while we are currently looking at the unit pairs (`t1`, `m1`) and (`t2`, `m2`). The following symmetrical algorithm holds on to each pair until the one from the other sequence has had a chance to catch up with it, because we always deal with the pair with the numerically lower `t` first. This also proves that the `results` sequence comes out in numerical order.

```
void multiply_unit_sequences(unit_sequence *us1, int s1, unit_sequence *us2, int s2,
    unit_sequence *result) {
    if ((result == us1) || (result == us2)) internal_error("result must be different structure");
    result->no_unit_pairs = 0;
    int i1 = 0, i2 = 0;                                          read position in sequences 1, 2
    int t1 = UNKNOWN, p1 = 0;                                    start with no current term from sequence 1
    int t2 = UNKNOWN, p2 = 0;                                    start with no current term from sequence 2
    while (TRUE) {
        ⟨If we have no current term from sequence 1, and it hasn't run out, fetch a new one 11⟩;
        ⟨If we have no current term from sequence 2, and it hasn't run out, fetch a new one 12⟩;
        if (t1 == t2) {
            if (t1 == UNKNOWN) return;                           both sequences have now run out
            ⟨Both terms refer to the same base unit, so combine these into the result 13⟩;
        } else {
            ⟨The terms refer to different base units, so copy the numerically lower one into the result 14⟩;
        }
    }
}
```

§**11.**

⟨If we have no current term from sequence 1, and it hasn't run out, fetch a new one 11⟩ ≡

```
if ((t1 == UNKNOWN) && (us1) && (i1 < us1->no_unit_pairs)) {
    t1 = us1->unit_pairs[i1].base_unit; p1 = us1->unit_pairs[i1].power; i1++;
}
```

This code is used in §10.

§**12.**

⟨If we have no current term from sequence 2, and it hasn't run out, fetch a new one 12⟩ ≡

```
if ((t2 == UNKNOWN) && (us2) && (i2 < us2->no_unit_pairs)) {
    t2 = us2->unit_pairs[i2].base_unit; p2 = us2->unit_pairs[i2].power; i2++;
}
```

This code is used in §10.

§**13.** So here the head of one sequence is T^{p_1} and the head of the other is T^{p_2}, so in the product we ought to see $(T^{p_1})^{s_1} \cdot (T^{p_2})^{s_2} = T^{p_1 s_1 + p_2 s_2}$. But we don't enter terms that have cancelled out, that is, where $p_1 s_1 + p_2 s_2 = 0$.

⟨Both terms refer to the same base unit, so combine these into the result 13⟩ ≡

```
int p = p1*s1 + p2*s2;                                   combined power of t1 = t2
if (p != 0) {
    if (result->no_unit_pairs == MAX_BASE_UNITS_IN_SEQUENCE)
        ⟨Trip a unit sequence overflow 15⟩;
    result->unit_pairs[result->no_unit_pairs].base_unit = t1;
    result->unit_pairs[result->no_unit_pairs++].power = p;
}
t1 = UNKNOWN; t2 = UNKNOWN;                              dispose of both terms as dealt with
```

This code is used in §10.

§**14.** Otherwise we copy. By copying the numerically lower term, we can be sure that it will never occur again in either sequence. So we can copy it straight into the results.

The code is slightly warped by the fact that `UNKNOWN`, representing the end of the sequence, happens to be numerically lower than all the valid data types. We don't want to make use of facts like that, so we write code to deal with `UNKNOWN` explicitly.

⟨The terms refer to different base units, so copy the numerically lower one into the result 14⟩ ≡

```
if ((t2 == UNKNOWN) || ((t1 != UNKNOWN) && (t1 < t2))) {
    if (result->no_unit_pairs == MAX_BASE_UNITS_IN_SEQUENCE)
        ⟨Trip a unit sequence overflow 15⟩;
    result->unit_pairs[result->no_unit_pairs].base_unit = t1;
    result->unit_pairs[result->no_unit_pairs++].power = p1*s1;
    t1 = UNKNOWN;                                  dispose of the head of sequence 1 as dealt with
} else if ((t1 == UNKNOWN) || ((t2 != UNKNOWN) && (t2 < t1))) {
    if (result->no_unit_pairs == MAX_BASE_UNITS_IN_SEQUENCE)
        ⟨Trip a unit sequence overflow 15⟩;
    result->unit_pairs[result->no_unit_pairs].base_unit = t2;
    result->unit_pairs[result->no_unit_pairs++].power = p2*s2;
    t2 = UNKNOWN;                                  dispose of the head of sequence 1 as dealt with
} else internal_error("unit pairs disarrayed");
```

This code is used in §10.

§**15.** For reasons explained above, this is really never going to happen by accident, but we'll be careful:

⟨Trip a unit sequence overflow 15⟩ ≡
```
sentence_problem(_P_(C7UnitSequenceOverflow),
    "reading that sentence led me into calculating such a complicated "
    "kind of value that I ran out of memory",
    "which my programmer really didn't expect to happen. I think you "
    "must have made an awful lot of numerical kinds of value, and "
    "then specified how they multiply so that one of them became "
    "weirdly tricky. Can you simplify?");
return;
```

This code is used in §13,14,13,14,13,14.

§**16.** The second operation is taking roots.

Surprisingly, perhaps, it's much easier to compute $\sqrt{X}$ or $\sqrt[3]{X}$ for any unit X – it's just that it can't always be done. Inform does not permit non-integer powers of units, so for instance $\sqrt{\text{time}}$ does not exist, whereas $\sqrt{\text{length}^2 \cdot \text{mass}^{-2}}$ does. Square roots exist if each power in the sequence is even, cube roots exist if each is divisible by 3. We return `TRUE` or `FALSE` according to whether the root could be taken, and if `FALSE` then the contents of `result` are undefined.

```
int root_unit_sequence(unit_sequence *us, int pow, unit_sequence *result) {
    *result = *us;
    int i;
    for (i=0; i<result->no_unit_pairs; i++) {
        if ((result->unit_pairs[i].power) % pow != 0) return FALSE;
        result->unit_pairs[i].power = (result->unit_pairs[i].power)/pow;
    }
    return TRUE;
}
```

§**17.** The final operation on unit sequences is substitution. Given a base type B, we substitute $B = K_D$ into an existing unit sequence K_E. (This is used when B is becoming a derived type – once we discover that $B = K_D$, we are no longer allowed to keep B in any unit sequence.)

We simply search for B^p, and if we find it, we remove it and then multiply by K_D^p.

```
void dim_substitute(unit_sequence *existing, int base, unit_sequence *derived) {
    int i, j, p = 0, found = FALSE;
    if (existing == NULL) return;
    for (i=0; i<existing->no_unit_pairs; i++)
        if (existing->unit_pairs[i].base_unit == base) {
            p = existing->unit_pairs[i].power;
            found = TRUE;
            ⟨Remove the B term from the existing sequence 18⟩;
        }
    if (found)
        ⟨Multiply the existing sequence by a suitable power of B's derivation 19⟩;
}
```

§**18.** We shuffle the remaining terms in the sequence down by one, overwriting B:

⟨Remove the B term from the existing sequence 18⟩ ≡
```
for (j = --(existing->no_unit_pairs); j>=i; j--)
    existing->unit_pairs[j] = existing->unit_pairs[j+1];
```
This code is used in §17.

§**19.** We now multiply by K_D^p.

⟨Multiply the existing sequence by a suitable power of B's derivation 19⟩ ≡
```
unit_sequence result;
multiply_unit_sequences(existing, 1, derived, p, &result);
*existing = result;
```
This code is used in §17.

§**20.** For reasons which will be explained below, a unit sequence also has a scale factor associated with it:

```
int us_get_scaling_factor(unit_sequence *us) {
    if (us == NULL) return 1;
    return us->scaling_factor;
}
```
The function us_get_scaling_factor is called from 7/kov.

§**21.** That just leaves, as usual, indexing...

```
void index_unit_sequence(unit_sequence *deriv) {
    if (deriv == NULL) internal_error("indexed null unit sequence");
    if (deriv->no_unit_pairs == 0) { INDEX("dimensionless"); return; }
    int j;
    for (j=0; j<deriv->no_unit_pairs; j++) {
        if (j>0) INDEX(".");
        INDEX("(");
        index_data_type_name(deriv->unit_pairs[j].base_unit, FALSE);
        INDEX(")");
        if (deriv->unit_pairs[j].power != 1)
            INDEX("<sup>%d</sup>", deriv->unit_pairs[j].power);
    }
}
```

§**22.** ...and logging.

```
void log_unit_sequence(unit_sequence *deriv) {
    if (deriv == NULL) { LOG("<null-us>"); return; }
    if (deriv->no_unit_pairs == 0) { LOG("dimensionless"); return; }
    int j;
    for (j=0; j<deriv->no_unit_pairs; j++) {
        if (j>0) LOG(".");
        LOG("("); log_data_type_name(deriv->unit_pairs[j].base_unit); LOG(")");
        if (deriv->unit_pairs[j].power != 1)
            LOG("%d", deriv->unit_pairs[j].power);
    }
}
```
The function log_unit_sequence is called from 7/kov.

§**23. Performing derivations.** The following is called when the user specifies that L times R specifies an O. These are required all to be quasinumerical: any of the three might be either a base unit (so far) or a derived unit (already).

If two or more are base units, we have a choice. That is, suppose we have created three KOVs already: mass, acceleration, force. Then we read:

Mass times acceleration specifies a force.

We could make this true in any of three ways: keep M and A as base units and derive F from them, keep A and F as base units and derive M from those, or keep M and F while deriving A. Inform always chooses the most recently created unit as the one to derive, on the grounds that the source text has probably set things out with what the user thinks are the most fundamental units first.

```
void make_unit_derivation(int left, int right, int outcome) {
    int types[3];
    types[0] = left; types[1] = right; types[2] = outcome;
    int newest_base = -1;
    ⟨Find which (if any) of the three units is the newest-made base unit 24⟩;
    if (newest_base >= 0) {
        unit_sequence *derivation = NULL;
        ⟨Derive the newest one by rearranging the equation in terms of the other two 25⟩;
        ⟨Substitute this new derivation to eliminate this base unit from other sequences 26⟩;
    } else
        ⟨Check this derivation to make sure it is redundant, not contradictory 27⟩;
}
```

§**24.** Data type IDs are allocated in creation order, so "newest" means largest ID.

⟨Find which (if any) of the three units is the newest-made base unit 24⟩ ≡

```
    int i, max = 0;
    for (i=0; i<3; i++)
        if ((types[i] > max) && (data_type_test_if_derived(types[i]) == FALSE)) {
            newest_base = i; max = types[i];
        }
```

This code is used in §23.

§**25.** We need to ensure that the user's multiplication rule is henceforth true, and we do that by fixing the newest unit to make it so.

⟨Derive the newest one by rearranging the equation in terms of the other two 25⟩ ≡

```
    unit_sequence *kx = NULL, *ky = NULL; int sx = 0, sy = 0;
    switch (newest_base) {
        case 0:                                          here L is newest and we derive L = R^-1 O
            kx = data_type_get_unit_sequence(types[1]); sx = -1;
            ky = data_type_get_unit_sequence(types[2]); sy = 1;
            break;
        case 1:                                          here R is newest and we derive R = L^-1 O
            kx = data_type_get_unit_sequence(types[0]); sx = -1;
            ky = data_type_get_unit_sequence(types[2]); sy = 1;
            break;
        case 2:                                          here O is newest and we derive O = LR
            kx = data_type_get_unit_sequence(types[0]); sx = 1;
            ky = data_type_get_unit_sequence(types[1]); sy = 1;
```

```
        break;
}
derivation = data_type_get_unit_sequence(types[newest_base]);
unit_sequence result;
multiply_unit_sequences(kx, sx, ky, sy, &result);
*derivation = result;
data_type_now_derived(types[newest_base]);
```

This code is used in §23.

§**26.** Later in Inform's run, when we start compiling code, many more unit sequences will exist on a temporary basis – as part of the KOVs for intermediate results in calculations – but early on, when we're here, the only unit sequences made are the derivations of the units. So it is easy to cover all of them.

⟨Substitute this new derivation to eliminate this base unit from other sequences 26⟩ ≡

```
int r;
LOOP_OVER_POSSIBLE_TYPE_IDS(r)
    if (data_type_is_quasinumerical(r)) {
        unit_sequence *existing = data_type_get_unit_sequence(r);
        dim_substitute(existing, types[newest_base], derivation);
    }
```

This code is used in §23.

§**27.** If we have $AB = C$ but all three of A, B, C are already derived, that puts us in a bind. Their definitions are fixed already, so we can't simply force the equation to come true by fixing one of them. That means either the derivation is redundant – because it's already true that $AB = C$ – or contradictory – because we know $AB \neq C$. We silently allow a redundancy, as it may have been put in for clarity, or so that the user can check the consistency of his own definitions, or to make the Kinds index page more helpful. But we must reject a contradiction.

⟨Check this derivation to make sure it is redundant, not contradictory 27⟩ ≡

```
unit_sequence product;
multiply_unit_sequences(
    data_type_get_unit_sequence(types[0]), 1,
    data_type_get_unit_sequence(types[1]), 1, &product);
if (compare_unit_sequences(&product, data_type_get_unit_sequence(types[2])) == FALSE)
    sentence_problem(_P_(C7DimensionsInconsistent),
        "this is inconsistent with what is already known about those kinds of value",
        "all three of which already have well-established relationships - see the "
        "Kinds index for more.");
```

This code is used in §23.

§**28. Classifying the units.** As noted above, we have base units and we have derived units.

```
int data_type_base(int t) {
    if ((data_type_is_quasinumerical(t)) &&
        (data_type_test_if_derived(t) == FALSE) &&
        (data_type_dimensionless(t) == FALSE) &&
        (t != INTERMEDIATE_TY)) return TRUE;
    return FALSE;
}
int data_type_derived(int t) {
    if (t == INTERMEDIATE_TY) return TRUE;
    if ((data_type_is_quasinumerical(t)) &&
        (data_type_test_if_derived(t) == TRUE) &&
        (data_type_dimensionless(t) == FALSE)) return TRUE;
    return FALSE;
}
```

§**29.** Some of the derived units are dimensionless, others not. `NUMBER_TY` is always dimensionless; and any unit whose derivation is the empty unit sequence must be dimensionless.

```
int data_type_dimensionless(int t) {
    if (t == NUMBER_TY) return TRUE;
    if (data_type_is_quasinumerical(t) == FALSE) return FALSE;
    unit_sequence *deriv = data_type_get_unit_sequence(t);
    if ((deriv) && (deriv->no_unit_pairs == 0)) return TRUE;
    return FALSE;
}
```

The function data_type_dimensionless is called from 7/kov.

§**30.** Using these definitions, we can now print analyses of the units into the index and the debugging log.

```
void index_dimensional_rules(void) {
    INDEX("<p><hr><p>");
    ⟨Index the rubric about quasinumerical KOVs 31⟩;
    ⟨Index the table of quasinumerical KOVs 32⟩;
    ⟨Index the table of multiplication rules 35⟩;
}
```

The function index_dimensional_rules is called from 9/inpw.

§31.

⟨Index the rubric about quasinumerical KOVs 31⟩ ≡

```
INDEX("<a name=calculator>");
begin_plain_html_table(ifl);
first_html_column(ifl, 0);
INDEX("<img border=0 src=inform:/doc_images/calc2.png>");
next_html_column(ifl, 0);
INDEX("Kinds of value marked with the <b>calculator symbol</b> are numerical - "
    "these are values we can add, multiply and so on. The range of these "
    "numbers depends on the Format setting for the project (Glulx format "
    "supports much higher numbers than Z-code).");
end_html_row(ifl);
end_html_table(ifl);
INDEX("<p>");
```

This code is used in §30.

§32.

⟨Index the table of quasinumerical KOVs 32⟩ ≡

```
int r;
begin_plain_html_table(ifl);
first_html_column(ifl, 0);
INDEX("<b>kind of value</b>");
next_html_column(ifl, 0);
INDEX("<b>minimum</b>");
next_html_column(ifl, 0);
INDEX("<b>maximum</b>");
next_html_column(ifl, 0);
INDEX("<b>dimensions</b>");
end_html_row(ifl);
LOOP_OVER_POSSIBLE_TYPE_IDS(r)
    if (data_type_is_quasinumerical(r)) {
        if (r == INTERMEDIATE_TY) continue;
        first_html_column(ifl, 0);
        index_data_type_name(r, FALSE);
        next_html_column(ifl, 0);
        ⟨Index the minimum positive value for a quasinumerical KOV 33⟩;
        next_html_column(ifl, 0);
        ⟨Index the maximum positive value for a quasinumerical KOV 34⟩;
        next_html_column(ifl, 0);
        if (data_type_dimensionless(r)) INDEX("<i>dimensionless</i>");
        else {
            unit_sequence *deriv = data_type_get_unit_sequence(r);
            index_unit_sequence(deriv);
        }
        end_html_row(ifl);
    }
end_html_table(ifl); INDEX("<p>");
```

This code is used in §30.

§**33.** At run-time, the minimum positive value is of course 1, but because of scaling this can appear to be much lower.

⟨Index the minimum positive value for a quasinumerical KOV 33⟩ ≡

```
switch (r) {
    case NUMBER_TY: INDEX("1"); break;
    case TIME_TY: INDEX("1 minute"); break;
    default: lp_index_value(data_type_list_of_literal_forms(r), 1); break;
}
```

This code is used in §32.

§**34.**

⟨Index the maximum positive value for a quasinumerical KOV 34⟩ ≡

```
switch (r) {
    case NUMBER_TY:
        if (target_VM_is_16_bit()) INDEX("32767");
        else INDEX("2147483647");
        break;
    case TIME_TY: INDEX("23 hours 59 minutes"); break;
    default:
        if (target_VM_is_16_bit())
            lp_index_value(data_type_list_of_literal_forms(r), 32767);
        else
            lp_index_value(data_type_list_of_literal_forms(r), 2147483647);
        break;
}
```

This code is used in §32.

§**35.** This is simply a table of all the multiplications declared in the source text, sorted into data type ID order of left and then right operand.

⟨Index the table of multiplication rules 35⟩ ≡

```
int L, R, O, NP = 0, wn;
LOOP_OVER_MULTIPLICATIONS(L, R, O, wn) {
    if (NP++ == 0) {
        INDEX("<p>This is how multiplication changes kinds:<p>\n");
        begin_plain_html_table(ifl);
    }
    first_html_column(ifl, 0);
    if (wn >= 0) index_link(lw_array[wn].lw_source);
    next_html_column(ifl, 0);
    index_data_type_name(L, FALSE);
    INDEX("<font color=\"#808080\"> x </font>");
    index_data_type_name(R, FALSE);
    INDEX("<font color=\"#808080\"> = </font>");
    index_data_type_name(O, FALSE);
    INDEX("    ");
    next_html_column(ifl, 0);
    data_type_index_benchmark_value(L);
    INDEX("<font color=\"#808080\"> x </font>");
    data_type_index_benchmark_value(R);
    INDEX("<font color=\"#808080\"> = </font>");
```

```
        data_type_index_benchmark_value(0);
        end_html_row(ifl);
    }
    if (NP > 0) { end_html_table(ifl); INDEX("<p>"); }
```

This code is used in §30.

§**36.** A simpler format for the debugging log, which is printed when we ask for the internal "dimensions" test:

```
void log_unit_analysis(void) {
    int c, r;
    LOG("Dimensionless: ");
    c = 0;
    LOOP_OVER_POSSIBLE_TYPE_IDS(r)
        if (data_type_dimensionless(r)) { if (c++ > 0) LOG(", "); log_data_type_name(r); }
    LOG("\nBase units: ");
    c = 0;
    LOOP_OVER_POSSIBLE_TYPE_IDS(r)
        if (data_type_base(r)) { if (c++ > 0) LOG(", "); log_data_type_name(r); }
    LOG("\nDerived units:\n");
    LOOP_OVER_POSSIBLE_TYPE_IDS(r)
        if ((data_type_derived(r)) && (r != INTERMEDIATE_TY)) {
            unit_sequence *deriv = data_type_get_unit_sequence(r);
            log_data_type_name(r); LOG(" = "); log_unit_sequence(deriv); LOG("\n");
        }
}
```

The function log_unit_analysis is called from 13/test.

§**37. Scaling.** Recall that every quasinumerical KOV U has a scale factor k_U, by default 1 and always ≥ 1, such that the true value v is represented at runtime as the I6 integer $k_U v$. For instance, if length is measured in metres but has a scale factor of 1000 then the I6 integer 1 means the true value 1mm, 10 means 1cm, 1000 means 1m. This I6 integer is called the scaled value s, and to reiterate, $s = k_U v$.

Scaled values are convenient, and they have no effect on how we add, subtract, approximate (that is, round off) or take remainder after division. If we have true values v_1 and v_2 with scaled values s_1 and s_2, and s_o is the scaled value for true value $v_1 + v_2$, then

$$s_1 + s_2 = k_U v_1 + k_U v_2 = k_U(v_1 + v_2) = s_o.$$

So ordinary I6 `+` at run-time correctly adds scaled values. But that's not true for all operations, and this is where we deal with that.

§**38.** First, multiplication. This time the values v_1 and v_2 may have different KOVs, which we'll call X and Y, and the result in general will be a third KOV, which we'll call O (for outcome). Then:

$$s_1 s_2 = k_X v_1 \cdot k_Y v_2 = k_O v_1 v_2 \cdot \left(\frac{k_X k_Y}{k_O}\right) = s_o \cdot \left(\frac{k_X k_Y}{k_O}\right)$$

so that simply multiplying the scaled values produces an answer which is too large by a factor of $k_X k_Y / k_O$. We need to correct for that, which we do either by dividing by this factor or multiplying by its reciprocal.

This is all a little delicate since rounding errors may be an issue and since $k_X k_Y / k_O$ is itself evaluated in integer arithmetic. In an ideal world we might use the same k for many units (e.g., $k = 1000$ throughout) and then of course this cancels to just 1000. But in practice people won't always do this – they may use some Babylonian, base 60, units, such as minutes and degrees, for instance, where $k = 3600$ would be more natural.

```
void kov_rescale_multiplication(OUTPUT_STREAM, kind_of_value *kovx, kind_of_value *kovy) {
    if ((kovx == NULL) || (kovy == NULL)) return;
    kind_of_value *kovo = arithmetic_on_kovs(kovx, kovy, TIMES_OPERATION);
    if (kovo == NULL) return;
    int k_X = kov_scale_factor(kovx);
    int k_Y = kov_scale_factor(kovy);
    int k_O = kov_scale_factor(kovo);
    if (k_X*k_Y > k_O) WRITE("/%d", (k_X*k_Y/k_O));
    if (k_X*k_Y < k_O) WRITE("*%d", (k_O/k_X/k_Y));
}
```

The function kov_rescale_multiplication is called from 10/eqns and 12/cinv.

§**39.** Second, division, which is similar.

$$\frac{s_1}{s_2} = \frac{k_X v_1}{k_Y v_2} = k_O \frac{v_1}{v_2} \cdot \left(\frac{k_X}{k_O k_Y}\right) = s_o \cdot \left(\frac{k_X}{k_O k_Y}\right)$$

so this time the excess to correct is a factor of $k_X / k_O k_Y$.

```
void kov_rescale_division(OUTPUT_STREAM, kind_of_value *kovx, kind_of_value *kovy) {
    if ((kovx == NULL) || (kovy == NULL)) return;
    kind_of_value *kovo = arithmetic_on_kovs(kovx, kovy, DIVIDE_OPERATION);
    if (kovo == NULL) return;
    int k_X = kov_scale_factor(kovx);
    int k_Y = kov_scale_factor(kovy);
    int k_O = kov_scale_factor(kovo);
    if (k_O*k_Y > k_X) WRITE("*%d", (k_O*k_Y/k_X));
    if (k_O*k_Y < k_X) WRITE("/%d", (k_X/k_O/k_Y));
}
```

The function kov_rescale_division is called from 10/eqns and 12/cinv.

§**40.** Third, the taking of pth roots, at any rate for $p = 2$ or $p = 3$.

```
void kov_rescale_root(OUTPUT_STREAM, kind_of_value *kovx, int power) {
    if (kovx == NULL) return;
    kind_of_value *kovo = NULL;
    if (power == 2) kovo = arithmetic_on_kovs(kovx, NULL, ROOT_OPERATION);
    if (power == 3) kovo = arithmetic_on_kovs(kovx, NULL, CUBEROOT_OPERATION);
    if (kovo == NULL) return;
    int k_X = kov_scale_factor(kovx);
    int k_O = kov_scale_factor(kovo);
    if (power == 2) ⟨Apply a scaling correction for square roots 41⟩
    else if (power == 3) ⟨Apply a scaling correction for cube roots 42⟩
    else internal_error("can only scale square and cube roots");
}
```

The function kov_rescale_root is called from 10/eqns and 12/cinv.

§**41.** For square roots,

$$\sqrt{s} = \sqrt{k_X v} = \sqrt{k_X}\sqrt{v} = k_O\sqrt{v} \cdot \left(\frac{\sqrt{k_X}}{k_O}\right) = s_o \cdot \left(\frac{\sqrt{k_X}}{k_O}\right)$$

and now the overestimate is a factor of $k = \sqrt{k_X}/k_O$. However, rather than calculating $k\sqrt{x}$ we calculate $\sqrt{k^2 x}$, since this way accuracy losses in taking the square root are much reduced. Therefore this scaling operating is to be performed inside the root function, not outside, and it scales by k^2 not k:

⟨Apply a scaling correction for square roots 41⟩ ≡

```
    if (k_O*k_O > k_X) WRITE("*%d", (k_O*k_O/k_X));
    if (k_O*k_O < k_X) WRITE("/%d", (k_X/k_O/k_O));
```

This code is used in §40.

§**42.** For cube roots,

$$\sqrt[3]{s} = \sqrt[3]{k_X v} = \sqrt[3]{k_X}\sqrt[3]{v} = k_O\sqrt[3]{v} \cdot \left(\frac{\sqrt[3]{k_X}}{k_O}\right) = s_o \cdot \left(\frac{\sqrt[3]{k_X}}{k_O}\right)$$

and the overestimate is $k = \sqrt[3]{k_X}/k_O$. Scaling once again within the rooting function, we scale by k^3:

⟨Apply a scaling correction for cube roots 42⟩ ≡

```
    if (k_O*k_O*k_O > k_X) WRITE("*%d", (k_O*k_O*k_O/k_X));
    if (k_O*k_O*k_O < k_X) WRITE("/%d", (k_X/k_O/k_O/k_O));
```

This code is used in §40.

§**43. Arithmetic on KOVs.** We are finally able to provide our central routine, the one providing a service for the rest of Inform. Given `kov1` and `kov2`, we return the KOV resulting from applying arithmetic operation `op`, or `NULL` if the operation cannot meaningfully be applied. In the case where `op` is a unary operation, `kov2` has no significance and should be `NULL`.

The KOVs here are called operands, because they are what will be operated on.

```
kind_of_value *arithmetic_on_kovs(kind_of_value *kov1, kind_of_value *kov2, int op) {
    if (kov1 == NULL) return NULL;
    if ((arithmetic_op_is_unary(op) == FALSE) && (kov2 == NULL)) return NULL;
    unit_sequence *operand1 = kov_get_dimensional_form(kov1);
    unit_sequence *operand2 = kov_get_dimensional_form(kov2);
    unit_sequence result;
    switch (op) {
        case PLUS_OPERATION:
        case MINUS_OPERATION:
        case EQUALS_OPERATION:
        case APPROXIMATION_OPERATION:
            if (compare_unit_sequences(operand1, operand2)) return kov1;
            return NULL;
        case UNARY_MINUS_OPERATION:
        case REMAINDER_OPERATION:
            return kov1;
        case ROOT_OPERATION:
            root_unit_sequence(operand1, 2, &result);
            break;
        case CUBEROOT_OPERATION:
            root_unit_sequence(operand1, 3, &result);
            break;
        case TIMES_OPERATION:
            multiply_unit_sequences(operand1, 1, operand2, 1, &result);
            break;
        case DIVIDE_OPERATION:
            multiply_unit_sequences(operand1, 1, operand2, -1, &result);
            break;
        case POWER_OPERATION: internal_error("can't dimension-check powers");
        default: internal_error("unimplemented arithmetic operation");
    }
    ⟨Handle calculations entirely between dimensionless units more delicately 44⟩;
    ⟨Identify the result as a known KOV, if possible 45⟩;
    ⟨And otherwise create a KOV as the intermediate result of a calculation 46⟩;
}
```

The function arithmetic_on_kovs is called from 7/tc and 10/eqns.

§**44.** If `result` is the empty unit sequence, we'll identify it as a number, because `NUMBER_TY` is the lowest type ID representing a dimensionless unit. Usually that's good: for instance, it says that a frequency times a time is a number, and not some more exotic dimensionless quantity like an angle.

But it's not so good when the calculation is not really physical at all, but purely mathematical, and all we are doing is working on dimensionless units. For instance, if take an angle θ and double it to 2θ, we don't want Inform to say the result is `NUMBER_TY` – we want 2θ to be another angle. So we make an exception.

⟨Handle calculations entirely between dimensionless units more delicately 44⟩ ≡

```
if (arithmetic_op_is_unary(op)) {
    if (kov_is_dimensionless(kov1)) return kov1;
} else {
    if ((kov_is_dimensionless(kov1)) && (kov_is_dimensionless(kov2))) {
        if (is_kova(kov2, NUMBER_TY)) return kov1;
        if (is_kova(kov1, NUMBER_TY)) return kov2;
        if (kov_compare(kov1, kov2)) return kov1;
    }
}
```

This code is used in §43.

§**45.** If we've produced the right combination of base units to make one of the named units, then we return that as an atomic KOV. For instance, maybe we divided a velocity by a time, and now we find that we have $\mathrm{m} \cdot \mathrm{s}^{-2}$, which turns out to have a name: acceleration.

⟨Identify the result as a known KOV, if possible 45⟩ ≡

```
int r;
LOOP_OVER_POSSIBLE_TYPE_IDS(r)
    if (compare_unit_sequences(&result, data_type_get_unit_sequence(r)))
        return kova(r);
```

This code is used in §43.

§**46.** Otherwise the `result` is a unit sequence which doesn't have a name, so we store it as an `INTERMEDIATE_TY` KOV, representing a temporary value living only for the duration of a calculation.

A last little wrinkle is: how we should scale this? For results like an acceleration, something defined in the source text, we know how accurate the author wants us to be. But these intermediate KOVs are not defined, and we don't know for sure what the author would want. It seems wise to set $k \geq k_X$ and $k \geq k_Y$, so that we have at least as much detail as the calculation would have had within each operand KOV. So perhaps we should put $k = \max(k_X, k_Y)$. But in fact we will choose $k = \mathrm{lcm}(k_X, k_Y)$, the least common multiple, so that any subsequent divisions will cancel correctly and we won't lose too much information through integer rounding. (In practice this will probably either be the same as $\max(k_X, k_Y)$ or will multiply by 6, since $\mathrm{lcm}(60, 1000) = 6000$ and so on.)

The same unit sequence can have different scalings each time it appears as an intermediate calculation. We could get to $\mathrm{m}^2 \cdot \mathrm{kg}$ either as $\mathrm{m} \cdot \mathrm{kg}$ times m, or as m^2 times kg, or many other ways, and we'll get different scalings depending on the route. This is why the `unit_sequence` structure has a `scaling_factor` field; the choice of scale factor does *not* depend on the physics but on the arithmetic method being used.

⟨And otherwise create a KOV as the intermediate result of a calculation 46⟩ ≡

```
result.scaling_factor = lcm(kov_scale_factor(kov1), kov_scale_factor(kov2));
return koviv(&result);
```

This code is used in §43.

Runtime Support for Data Types 9/rsdt

Purpose

To compile I6 material needed at runtime to enable data types to function as they should.

9/rsdt.§1-8 The heap; §9-14 Run-time support for units and enumerations; §15-20 Further runtime support

---

Template interpreter commands

```
2   {-callv:compile_heap_allocator}
9   {-callv:compile_data_type_support_routines}
```

---

§**1. The heap.** Some but not all works compiled by Inform make use of a "heap" of memory at runtime, to hold dynamic strings, lists and other flexibly-sized structures. This is needed only if the source text makes use of data types storing pointer values, which the Standard Rules do not, so on a minimal compilation no heap will be created.

Management of the heap is delegated to runtime code in the template file "Flex.i6t", so Inform itself needs to know surprisingly little about how the job is done. Our main task is to work out how large a heap is likely to be needed. This is important because although some virtual machines (recent versions of Glulx) can dynamically allocate memory and so can grow their heaps, others cannot, so that the initial size needs to be enough to get through the whole run.

We need to keep an approximate count of how much memory will be needed to store the blocks of data pointed to by values of kind "indexed text" and the like: this is of necessity approximate since we cannot know how large such texts might grow, or how much function call recursion will cause more and more texts to proliferate during calculations. Still, we can guess, and in one important case we can be certain: if nobody ever uses any of these pointer-value data types, the memory needed will be exactly 0, and this is worth knowing since it enables us to avoid using even the tiniest amount of the Z-machine's precious byte-addressable memory.

We keep track by spotting the compilation of all calls to create pointer values.

```
int total_heap_allocation = 0, heap_needed = FALSE;

void ensure_basic_heap_present(void) {
    heap_needed = TRUE;
    total_heap_allocation += 256;                    enough for the initial free-space block
}
```

The function ensure_basic_heap_present is called from 7/cmsp and 10/list.

§**2.** We need to provide a start-up routine which creates initial blocks of data on the heap for any permanent storage objects (global variables, property values, table entries, list items) of pointer-value data types:

```
void compile_heap_allocator(OUTPUT_STREAM) {
    WRITE("[ InitialHeapAllocation obj pv;\n"); INDENT;
    ⟨Allocate heap space for global variables with pointer-value KOVs 3⟩;
    ⟨Allocate heap space for properties with pointer-value KOVs 4⟩;
    ⟨Allocate heap space for table entries with pointer-value KOVs 6⟩;
    compile_list_constant_allocations(OUT);
    OUTDENT; WRITE("];\n");
    WRITE("[ DistributeBlockConstants obj pv;\n"); INDENT;
    ⟨Make multiple copies of constant values for properties 5⟩;
    OUTDENT; WRITE("];\n");
    if (heap_needed == FALSE) return;
    ⟨Compile a constant for the heap size needed 7⟩;
}
```

The function compile_heap_allocator is invoked by a command in a `.i6t` template file.

§**3.** For each global variable `Q` we compile code like this:

```
pv = Q;
Q = BlkValueCreate(K, pv, 0);
```

where `K` is the I6 identifier for the KOV, and `pv` represents the initial value – which is usually being cast from a word value; for instance, if we are working from this:

```
Gadget is an indexed text that varies. Gadget is "sonic screwdriver".
```

then we would compile, in effect:

```
pv = "sonic screwdriver";
Q = BlkValueCreate(INDEXED_TEXT_TY, pv, 0);
```

Thus allocating a new block on the heap and casting the word value "sonic screwdriver" (a pointer to an unindexed packed string in the virtual machine) into the new heap block.

However, a few pointer-value KOVs have constants which don't arise from casting: notably, lists. Blocks on the heap for constants like those are made separately, so we only create a new block if the initial value is 0, since this means "not allocated yet". For instance, if `Q` is a list of numbers, we would do this:

```
pv = Q;
if (pv == 0) Q = BlkValueCreate(LIST_OF_TY, pv, NUMBER_TY);
```

⟨Allocate heap space for global variables with pointer-value KOVs 3⟩ ≡

```
quantity *q;
LOOP_OVER(q, quantity)
    if (qty_is_a_global_variable(q)) {
        kind_of_value *kov = qty_kind_of_value(q);
        if (kov_uses_pointer_values(kov)) {
            WRITE("pv = "); compile_quantity_name(OUT, q); WRITE("; ");
            if (kov_has_pointer_value_constants(kov)) WRITE("if (pv==0) ");
            compile_quantity_name(OUT, q); WRITE(" = ");
            compile_heap_allocation(OUT, qty_kind_of_value(q), 1, TRUE);
            WRITE(";\n");
        }
    }
```

This code is used in §2.

§**4.** The same, but for property values of objects and other values.

⟨Allocate heap space for properties with pointer-value KOVs 4⟩ ≡

```
property_name *prn;
LOOP_OVER(prn, property_name) {
    if (prn_is_either_or(prn) == FALSE) {
        kind_of_value *kov = prn_get_kind_of_value(prn);
        if (kov_uses_pointer_values(kov)) {
            WRITE("if (%s) objectloop (obj provides %s) { pv = obj.%s; ",
                prn_get_i6_identifier(prn),
                prn_get_i6_identifier(prn),
                prn_get_i6_identifier(prn));
            if (kov_has_pointer_value_constants(kov)) WRITE("if (pv==0) ");
            WRITE("obj.%s = ", prn_get_i6_identifier(prn));
            compile_heap_allocation(OUT, prn_get_kind_of_value(prn), 1, TRUE);
            WRITE("; }\n", prn_get_i6_identifier(prn));
        }
    }
}
int t;
LOOP_OVER_DATA_TYPE_IDS(t)
    allocate_heap_for_KOV_properties(OUT, kova(t));
```

This code is used in §2.

§**5.** The rationale for the following is: suppose we have

> A person has a stored action called the manoeuvre. The manoeuvre of a person is usually the jumping action.

Then a pointer to the a block which will one day hold the jumping action is the manoeuvre value in the `Class` declaration which the person kind becomes on compilation. That means all people for whom manoeuvre was never explicitly defined will inherit this same pointer as their copy. That in turn means they don't really have independent copies – they all have the same manoeuvre action forever afterwards, making it impossible to alter during play. We must therefore force them to have independent copies, by copying. But that can't be done in the Initial Heap Allocation stage because the values within the blocks don't exist yet; the heap space has to be allocated first, and the values get written later. So we take care of it in a special "distribute block constants" phase of initialisation:

⟨Make multiple copies of constant values for properties 5⟩ ≡

```
property_name *prn;
LOOP_OVER(prn, property_name) {
    if (prn_is_either_or(prn) == FALSE) {
        kind_of_value *kov = prn_get_kind_of_value(prn);
        if ((kov_uses_pointer_values(kov)) &&
            (kov_has_pointer_value_constants(kov))) {
            WRITE("if (%s) objectloop (obj provides %s) { pv = obj.%s; ",
                prn_get_i6_identifier(prn),
                prn_get_i6_identifier(prn),
                prn_get_i6_identifier(prn));
            WRITE("obj.%s = BlkValueCreate(%d); BlkValueCopy(obj.%s, pv); }",
                    prn_get_i6_identifier(prn), kov_I6_ID(kov), prn_get_i6_identifier(prn));
        }
    }
}
```

This code is used in §2.

§**6.** The same, but for table entries. (Note that property values of values which are stored as table entries are taken care of here, too.)

⟨Allocate heap space for table entries with pointer-value KOVs 6⟩ ≡

```
int i;
table *t;
LOOP_OVER(t, table)
    for (i=0; i<tab_no_columns(t); i++) {
        kind_of_value *kov = table_column_data_type(t, i);
        if (kov_uses_pointer_values(kov)) {
            WRITE("for (obj=1:obj<=TableRows(%s):obj++) { ",
                tab_get_I6_representation(t));
            WRITE("if (ExistsTableLookUpEntry(%s, %d, obj)==false) continue; ",
                tab_get_I6_representation(t), i+1);
            WRITE("pv = TableLookUpEntry(%s, %d, obj); ",
                tab_get_I6_representation(t), i+1);
            if (kov_has_pointer_value_constants(kov))
                WRITE("if (pv==0) ");
            WRITE("TableLookUpEntry(%s, %d, obj, true, ",
                tab_get_I6_representation(t), i+1);
            compile_heap_allocation(OUT, table_column_data_type(t, i),
                tab_no_rows(t), TRUE);
            WRITE("); }\n");
        }
    }
```

This code is used in §2.

§**7.** By now, we know that we need at least `total_heap_allocation` bytes on the heap, but the initial heap size has to be a power of 2, so we compute the smallest such which is big enough. On Glulx, we then multiply by 4: one factor of 2 is because the word size is twice as much – words are 4-byte, not 2-byte as on the Z-machine – while the other is, basically, because we can, and because we want to store indexed text in particular using 2-byte characters (capable of storing Unicode) rather than 1-byte characters as on the Z-machine. Glulx has essentially no memory constraints compared with the Z-machine.

⟨Compile a constant for the heap size needed 7⟩ ≡

```
int max_heap = 1;
if (total_heap_allocation < get_dynamic_memory_allocation())
    total_heap_allocation = get_dynamic_memory_allocation();
while (max_heap < total_heap_allocation) max_heap = max_heap*2;
WRITE("#ifdef TARGET_ZCODE;\n");
WRITE("Constant MEMORY_HEAP_SIZE = %d;\n", max_heap);
WRITE("#ifnot;\n");
WRITE("Constant MEMORY_HEAP_SIZE = %d;\n", 4*max_heap);
WRITE("#endif;\n");
LOG("Providing for an total heap of %d, given requirement of %d\n",
    max_heap, total_heap_allocation);
```

This code is used in §2.

§**8.** The following routine both compiles code to create a pointer value, and also increments the heap allocation suitably. Each pointer-value KOV comes with an estimate of its likely size needs – its exact size needs if it is fixed in size, and a reasonable overestimate of typical usage if it is flexible.

The `multiplier` is used when we need to calculate the size of, say, a list of 20 indexed texts. For the cases above, it's always 1.

```
void compile_heap_allocation(OUTPUT_STREAM, kind_of_value *kov,
    int multiplier, int casting) {
    heap_needed = TRUE;
    if (kov_uses_pointer_values(kov) == FALSE)
        internal_error("unable to allocate heap storage for this kind of value");
    if (kov_get_heap_size_estimate(kov) == 0)
        internal_error("no heap storage estimate for this kind of value");
    total_heap_allocation += (kov_get_heap_size_estimate(kov))*multiplier;
    WRITE("BlkValueCreate(%d,", kov_I6_ID(kov));
    if (casting) WRITE("pv,"); else WRITE("0,");
    WRITE("%d)", kov_I6_construct_ID(kov));
}
```

The function compile_heap_allocation is called from 9/prop, 10/list, 12/phsf and 12/stv.

§**9. Run-time support for units and enumerations.** The following generates a small suite of I6 routines associated with each such data type, and needed at run-time.

```
void compile_data_type_support_routines(OUTPUT_STREAM) {
    int t;
    LOOP_OVER_DESIGNED_TYPE_IDS(t) {
        kind_of_value *kov = kova(t);
        char *I6_name_of_kov = kov_get_name_of_printing_rule(kov);
        if (kov_is_an_enumeration(kov)) {
            ⟨Compile I6 printing routine for an enumerated KOV 12⟩;
            ⟨Compile the A and B routines for an enumerated KOV 13⟩;
            ⟨Compile random-ranger routine for this KOV 14⟩;
        } else if (kov_is_a_unit(kov)) {
            ⟨Compile I6 printing routine for a unit KOV 11⟩;
            ⟨Compile random-ranger routine for this KOV 14⟩;
        } else {
            ⟨Compile I6 printing routine for a vacant but named KOV 10⟩;
        }
    }
    ⟨Compile a suite of I6 routines taking KOV IDs as arguments 15⟩;
    LOOP_OVER_DATA_TYPE_IDS(t)
        compile_KOV_properties(OUT, kova(t));
}
```

The function compile_data_type_support_routines is invoked by a command in a `.i6t` template file.

§**10.** A slightly bogus case first. If the source text declares a KOV but never gives any enumerated values or literal patterns, then such values will never appear at run-time; but we need the printing routine to exist to avoid compilation errors.

⟨Compile I6 printing routine for a vacant but named KOV 10⟩ ≡

```
WRITE("[ %s value;\n", I6_name_of_kov); INDENT;
WRITE("print value;\n");
OUTDENT; WRITE("];\n\n");
```

This code is used in §9.

§**11.** A unit is printed back with its earliest-defined literal pattern used as notation. If it had no literal patterns, it would come out as decimal numbers, but at present this can't happen.

⟨Compile I6 printing routine for a unit KOV 11⟩ ≡

```
WRITE("[ %s value ", I6_name_of_kov);
if (kov_list_of_literal_forms(kov))
    printing_routine_for_lp(OUT, kov_list_of_literal_forms(kov));
else {
    WRITE(";\n"); INDENT; WRITE("print value;\n");
}
OUTDENT; WRITE("];\n\n");
```

This code is used in §9.

§**12.**

⟨Compile I6 printing routine for an enumerated KOV 12⟩ ≡

```
WRITE("[ %s value;\n", I6_name_of_kov); INDENT;
WRITE("switch(value) {\n"); INDENT;

quantity *q;
LOOP_OVER(q, quantity) {
    if ((kov_compare(qty_kind_of_value(q), kov)) && (qty_is_a_variable(q) == FALSE)) {
        int k;
        compile_quantity_name(OUT, q);
        WRITE(": print \"");
        for (k=q->word_ref1; k<=q->word_ref2; k++) {
            isn_compile_string(OUT, lw_array[k].lw_rawtext, 0);
            if (k < q->word_ref2) WRITE(" ");
        }
        WRITE("\";\n");
    }
}

this default case should never be needed, unless the user has blundered at the I6 level:
int w1, w2;
kov_get_name(kov, &w1, &w2, FALSE);
WRITE("default: print \"<illegal ");
if (w1 >= 0) print_text_to_file(w1, w2, OUT); else WRITE("value");
WRITE(">\";\n");

OUTDENT; WRITE("}\n");
OUTDENT; WRITE("];\n\n");
```

This code is used in §9.

§**13.** If the type is called, say, `T1_colour`, then we have:

(a) `A_T1_colour(v)` advances to the next valid value for the type, wrapping around to the first from the last;
(b) `B_T1_colour(v)` goes back to the previous valid value for the type, wrapping around to the last from the first, so that it is the inverse function to `A_T1_colour(v)`.

⟨Compile the A and B routines for an enumerated KOV 13⟩ ≡

```
int j = kov_get_highest_valid_value_as_integer(kov);
WRITE("[ A_%s value;\n", I6_name_of_kov); INDENT;
WRITE("return (value %% %d)+1;\n", j);
OUTDENT; WRITE("];\n");
WRITE("[ B_%s value;\n", I6_name_of_kov); INDENT;
WRITE("return ((value+%d) %% %d)+1;\n", j-2, j);
OUTDENT; WRITE("];\n");
```

This code is used in §9.

§**14.** And here we add:

(a) `R_T1_colour()` returns a uniformly random choice of the valid values of the given type. (For a unit, this will be a uniformly random positive value, which will probably not be useful.)
(b) `R_T1_colour(a, b)` returns a uniformly random choice in between `a` and `b` inclusive.

⟨Compile random-ranger routine for this KOV 14⟩ ≡

```
WRITE("[ R_%s a b;\n", I6_name_of_kov); INDENT;
if (kov_is_a_unit(kov))
    WRITE("if (a == 0 && b == 0) return (random(MAX_POSITIVE_NUMBER));\n");
else
    WRITE("if (a == 0 && b == 0) return (random(%d));\n",
        kov_get_highest_valid_value_as_integer(kov));
WRITE("if (a == b) return b;\n");
WRITE("if (a > b) return b+(random(MAX_POSITIVE_NUMBER) %% (a-b+1));\n");
WRITE("return a+(random(MAX_POSITIVE_NUMBER) %% (b-a+1));\n");
OUTDENT; WRITE("];\n");
```

This code is used in §9.

§**15. Further runtime support.** These last routines are synoptic: they take the ID number of the KOV as an argument, so there is only one of each routine.

⟨Compile a suite of I6 routines taking KOV IDs as arguments 15⟩ ≡

```
⟨Compile PrintKindValuePair 16⟩;
⟨Compile DefaultValueOfKOV 17⟩;
⟨Compile KOVComparisonFunction 18⟩;
⟨Compile KOVIsBlockValue 19⟩;
⟨Compile KOVSupportFunction 20⟩;
```

This code is used in §9.

§**16.** `PrintKindValuePair(K, V)` prints out the value `V`, declaring its KOV to be `K`. (Since I6 is typeless and in general the KOV of `V` cannot be deduced from its value alone, `K` must explicitly be supplied.)

⟨Compile PrintKindValuePair 16⟩ ≡
```
WRITE("[ PrintKindValuePair k v;\n"); INDENT;
WRITE("  switch(k) {\n"); INDENT;
LOOP_OVER_DATA_TYPE_IDS(t)
    WRITE("%d: print (%s) v;\n",
        t, data_type_get_name_of_printing_rule(t));
WRITE("default: print v;\n");
OUTDENT; WRITE("}\n");
OUTDENT; WRITE("];\n");
```

This code is used in §15.

§**17.** `DefaultValueOfKOV(K)` returns the default value for KOV `K`: it's needed, for instance, when increasing the size of a list of K to include new entries, which have to be given some type-safe value to start out at.

⟨Compile DefaultValueOfKOV 17⟩ ≡
```
WRITE("[ DefaultValueOfKOV k;\n"); INDENT;
WRITE("switch(k) {\n"); INDENT;
LOOP_OVER_DATA_TYPE_IDS(t)
    if (data_type_can_be_stored_in_variables(t)) {
        WRITE("%d: return ", t);
        compile_default_value(OUT, kova(t), -1, -1, "list entry");
        WRITE(";\n");
    }
WRITE("default: return 0;\n");
OUTDENT; WRITE("}\n");
OUTDENT; WRITE("];\n");
```

This code is used in §15.

§**18.** `KOVComparisonFunction(K)` returns either the address of a function to perform a comparison between two values, or else 0 to signal that no special sort of comparison is needed. (In which case signed numerical comparison will be used.) The function `F` may be used in a sorting algorithm, so it must have no side-effects. `F(x,y)` should return 1 if $x > y$, 0 if $x = y$ and -1 if $x < y$. Note that it is not permitted to return 0 unless the two values are genuinely equal.

⟨Compile KOVComparisonFunction 18⟩ ≡
```
WRITE("[ KOVComparisonFunction k;\n"); INDENT;
WRITE("switch(k) {\n"); INDENT;
LOOP_OVER_DATA_TYPE_IDS(t)
    if (data_type_can_be_stored_in_variables(t)) {
        if (data_type_uses_pointer_values(t))
            WRITE("%d: return BlkValueCompare;\n", t);
        else if (data_type_uses_signed_comparisons(t) == FALSE)
            WRITE("%d: return UnsignedCompare;\n", t);
    }
WRITE("default: return 0;\n");
OUTDENT; WRITE("}\n");
OUTDENT; WRITE("];\n");
```

This code is used in §15.

§**19.** `KOVIsBlockValue(K)` is true if and only if `K` is the I6 ID of a KOV storing pointers to blocks on the heap.

⟨Compile KOVIsBlockValue 19⟩ ≡
```
WRITE("[ KOVIsBlockValue k;\n"); INDENT;
WRITE("if (k == ");
int j = 0;
LOOP_OVER_DATA_TYPE_IDS(t)
    if (data_type_uses_pointer_values(t)) {
        if (j++ > 0) WRITE(" or ");
        WRITE("%d", t);
    }
WRITE(") rtrue;\n");
WRITE("rfalse;\n");
OUTDENT; WRITE("];\n");
```

This code is used in §15.

§**20.** `KOVSupportFunction(K)` returns the address of the specific support function for a pointer-value KOV `K`, or returns 0 if `K` is not such a KOV. For what such a function does, see "Flex.i6t" and the example in, for instance, "IndexedText.i6".

⟨Compile KOVSupportFunction 20⟩ ≡
```
WRITE("[ KOVSupportFunction k;\n"); INDENT;
WRITE("switch(k) {\n"); INDENT;
LOOP_OVER_DATA_TYPE_IDS(t)
    if (data_type_uses_pointer_values(t))
        WRITE("%d: return %s_Support;\n", t, type_ID_get_source_name(t));
OUTDENT; WRITE("}\n");
WRITE("rfalse;\n");
OUTDENT; WRITE("];\n");
```

This code is used in §15.

Kinds Index 7/kix

Purpose

To produce most of the Kinds page in the Index for a project: the chart at the top, and the detailed entries below.

§1. Indexing the KOVs. The Kinds page of the index opens with a table summarising the hierarchy of KOVs (and therefore of objects too), and then follows with details. This routine is called twice, once with `pass` equal to 1, when it has to fill in the hierarchy of KOVs listed under "value" in the key chart at the top of the Kinds index; and then again lower down, with `pass` equal to 2, when it gives more detail.

Not all of the built-in KOVs are indexed on the Kinds page. The ones omitted are of no help to end users, and would only clutter up the table with misleading entries. Remaining KOVs are grouped together in "priority" order, a device to enable the quasinumerical KOVs to stick together, the enumerative ones, and so on. (The priorities are set in the "Types.i6t" template file; a lower priority number puts you higher up.)

```
int tabulating_kinds_index = FALSE;
void index_kinds_of_value(int pass) {
    int priority;
    int t;
    if (pass == 1) {
        tabulating_kinds_index = TRUE;
        begin_plain_html_table(ifl);
        ⟨Add a titling row to the chart of KOVs 2⟩;
    }
    for (priority = 1; priority <= LOWEST_INDEX_PRIORITY; priority++) {
        LOOP_OVER_DATA_TYPE_IDS(t) {
            if (priority == data_type_get_index_priority(t)) {
                kind_of_value *kov = kova(t);
                if (kov_can_be_stored_in_variables(kov)) {
                    switch (pass) {
                        case 1: ⟨Write table row for this KOV 3⟩; break;
                        case 2:
                            ⟨Write heading for the detailed index entry for this KOV 5⟩;
                            ⟨Index explanatory text supplied for a KOV 8⟩;
                            ⟨Index literal patterns which can specify this KOV 6⟩;
                            ⟨Index possible values of an enumerated KOV 7⟩;
                            INDEX("<p>\n"); break;
                    }
                    if (t == OBJECT_TY) index_object_kinds(pass);
                }
            }
        }
    }
    if (pass == 1) {
        end_html_table(ifl);
        INDEX("<p>\n");
        tabulating_kinds_index = FALSE;
        ⟨Add the rubric below the chart of KOVs 4⟩;
    }
}
```

The function index_kinds_of_value is called from 9/inpw.

§**2.** An atypical row:

⟨Add a titling row to the chart of KOVs 2⟩ ≡

```
first_html_column(ifl, 0);
INDEX("value");
index_KOV_col_head("default value", "default");
index_KOV_col_head("repeat", "repeat");
index_KOV_col_head("props", "props");
index_KOV_col_head("under", "under");
end_html_row(ifl);
```

This code is used in §1.

§**3.** And then a typical row:

⟨Write table row for this KOV 3⟩ ≡

```
char *repeat = "cross", *props = "cross", *under = "cross";
int shaded = FALSE;
if ((data_type_get_highest_valid_value_as_integer(t) == 0) &&
    (data_type_indexed_grey_if_empty(t)))
        shaded = TRUE;
if (kov_compile_domain_possible(kov)) repeat = "tick";
if (kov_has_properties(kov)) props = "tick";
if (kov_offers_I6_GPR(kov)) under = "tick";
index_KOV_begin_chart_row();
index_KOV_name_cell(shaded, t);
index_KOV_end_chart_row(shaded, NULL, t, repeat, props, under);
```

This code is used in §1.

§**4.** Note the named anchors here, which must match those linked from the titling row (in the second argument of `index_KOV_col_head`).

⟨Add the rubric below the chart of KOVs 4⟩ ≡

```
INDEX("<p><a name=col_default>"
    "<small>The <b>default value</b> is used when we make something like "
    "a variable but don't tell Inform what its value is. For instance, if "
    "we write 'Zero hour is a time that varies', but don't tell Inform "
    "anything specific like 'Zero hour is 11:21 PM.', then Inform uses "
    "the value in the table above to decide what it will be. "
    "The same applies if we create a property (for instance, 'A person "
    "has a number called lucky number.'). Kinds of value not included "
    "in the table cannot be used in variables and properties.</small>"
    "<p><a name=col_repeat>"
    "<small>A tick for <b>repeat</b> means that it's possible to "
    "repeat through values of this kind. For instance, 'repeat with T "
    "running through times:' is allowed, but 'repeat with N running "
    "through numbers:' is not - there are too many numbers for this to "
    "make sense. A tick here also means it's possible to form lists such "
    "as 'list of rulebooks', or to count the 'number of scenes'.</small>"
    "<p><a name=col_props>"
    "<small>A tick for <b>props</b> means that values of this "
    "kind can have properties. For instance, 'A scene can be thrilling or "
```

```
        "dull.' makes an either/or property of a scene, but 'A number can be "
        "nice or nasty.' is not allowed because it would cost too much storage "
        "space. (Of course 'Definition:' can always be used to make adjectives "
        "applying to numbers; it's only properties which have storage "
        "worries.)</small>"
        "<p><a name=col_under>"
        "<small>A tick for <b>under</b> means that it's possible "
        "to understand values of this kind. For instance, 'Understand \"award "
        "[number]\" as awarding.' might be allowed, if awarding were an action "
        "applying to a number, but 'Understand \"run [rule]\" as rule-running.' "
        "is not allowed - there are so many rules with such long names that "
        "Inform doesn't add them to its vocabulary during play.</small><p>");
```

This code is used in §1.

§**5.** The detailed entry lower down the page begins with:

⟨Write heading for the detailed index entry for this KOV 5⟩ ≡

```
    index_anchor_numbered(t);                                   ...the anchor to which the grey icon in the table led
    INDEX("<b>"); index_data_type_name(t, TRUE); INDEX("</b>");
    INDEX(" (<i>plural</i> "); index_data_type_plural(t); INDEX(")");
    if (data_type_get_documentation_reference(t))
        index_doc_link(data_type_get_documentation_reference(t));                blue help icon, if any
    INDEX("<br>");
```

This code is used in §1.

§**6.**

⟨Index literal patterns which can specify this KOV 6⟩ ≡

```
    if ((kov_is_a_unit(kov)) && (data_type_list_of_literal_forms(t))) {
        index_literal_patterns(t);
        INDEX("<br>");
    }
```

This code is used in §1.

§**7.** Note that an enumerated KOV only becomes so when its first possible value is made, so that the following sentence can't have an empty list in it.

⟨Index possible values of an enumerated KOV 7⟩ ≡

```
    if (kov_is_an_enumeration(kov)) {
        int f = FALSE;
        quantity *q;
        INDEX("<i>One of the following:</i> ");
        LOOP_OVER(q, quantity) {
            if ((kov_compare(qty_kind_of_value(q), kov)) && (qty_is_a_variable(q) == FALSE)) {
                if (f) INDEX(", ");
                print_raw_text_to_file(q->word_ref1, q->word_ref2, ifl);
                f = TRUE;
            }
        }
        INDEX("<br>");
    }
```

This code is used in §1.

§**8.** Explanations:

⟨Index explanatory text supplied for a KOV 8⟩ ≡

```
    char *explanation = kov_get_specification_text(kov);
    if (explanation) INDEX("%s<br>\n", explanation);
```

This code is used in §1.

§**9. KOV table construction.** First, here's the table cell for the heading at the top of a column: the link is to the part of the rubric explaining what goes into the column.

```
void index_KOV_col_head(char *name, char *anchor) {
    next_html_column(ifl, 0);
    INDEX("<i>%s</i> ", name);
    INDEX("<a href=#col_%s><img border=0 src=inform:/doc_images/shelp.png></a>", anchor);
}
```

§**10.** Once we're past the heading row, each row is made in two parts: first this is called –

```
void index_KOV_begin_chart_row(void) {
    first_html_column(ifl, 0);
}
```

The function index_KOV_begin_chart_row is called from 9/inpw.

§**11.** That leads us into the cell for the name of the KOV. The following routine is used for the KOV rows, but not for the kinds-of-object rows; the cell for those is filled in a different way in "Index Physical World".

It's convenient to return the shadedness: a row is shaded if it's for a KOV which can have enumerated values but doesn't at the moment – for instance, the sound effects row is shaded if there are none.

define KINDS_INDEX_SHADE "#808080" *standard HTML grey colour*

```
int index_KOV_name_cell(int shaded, int t) {
    if (shaded) INDEX("<font color=\"%s\">", KINDS_INDEX_SHADE);
    INDEX("    ");
    index_data_type_name(t, TRUE);
    if (kov_quasinumerical(kova(t)))
        INDEX(" <a href=#calculator><img border=0 src=inform:/doc_images/calc1.png></a>");
    if (data_type_get_highest_valid_value_as_integer(t) >= 1)
        INDEX(" [%d]", data_type_get_highest_valid_value_as_integer(t));
    index_below_link_numbered(t);                    a grey see below icon leading to an anchor on pass 2
    if (data_type_get_documentation_reference(t))
        index_doc_link(data_type_get_documentation_reference(t));
    if (shaded) INDEX("</font>");
    return shaded;
}
```

§**12.** Finally we close the name cell, add the remaining cells, and close out the whole row.

```
void index_KOV_end_chart_row(int shaded, world_object *k, int t,
    char *tick1, char *tick2, char *tick3) {
    next_html_column(ifl, 0);
    if (shaded) INDEX("<font color=\"%s\">", KINDS_INDEX_SHADE);
    ⟨Index the default value entry in the KOV chart 13⟩;
    if (shaded) INDEX("</font>");
    next_html_column(ifl, 0);
    INDEX("<img border=0 alt=\"%s\" src=inform:/doc_images/%s%s.png>",
        tick1, shaded?"grey":"", tick1);
    next_html_column(ifl, 0);
    INDEX("<img border=0 alt=\"%s\" src=inform:/doc_images/%s%s.png>",
        tick2, shaded?"grey":"", tick2);
    next_html_column(ifl, 0);
    INDEX("<img border=0 alt=\"%s\" src=inform:/doc_images/%s%s.png>",
        tick3, shaded?"grey":"", tick3);
    end_html_row(ifl);
}
```

The function index_KOV_end_chart_row is called from 9/inpw.

§**13.** For a kind of object, we get the default value by looking for the earliest created instance; for a KOV, the default comes from the value specified in the "Types.i6t" declaration, except that two special values expand:

⟨Index the default value entry in the KOV chart 13⟩ ≡

```
    if (k) {
        int found = FALSE;
        world_object *inst;
        LOOP_OVER(inst, world_object)
            if ((inst->kind_flag == FALSE) && (wo_of_kind(inst, k))) {
                index_name_object(inst, "", "");
                found = TRUE;
                break;
            }
        if (found == FALSE) INDEX("--");
    } else {
        kind_of_value *kov = kova(t);
        char *p = kov_get_index_default_value(kov);
        if (strcmp(p, "<0-in-literal-pattern>") == 0)
            ⟨Index the constant 0 but use the default literal pattern 14⟩
        else if (strcmp(p, "<first-constant>") == 0)
            ⟨Index the earliest-created constant value for this KOV 15⟩
        else INDEX(p);
    }
```

This code is used in §12.

§**14.** For every quasinumeric KOV the default value is 0, but we don't want to index just "0" because that means 0-as-a-number: we want it to come out as "0 kg", "0 hectares", or whatever is appropriate.

⟨Index the constant 0 but use the default literal pattern 14⟩ ≡

```
    if (kov_list_of_literal_forms(kov))
        lp_index_value(kov_list_of_literal_forms(kov), 0);
    else
        INDEX("--");
```

This code is used in §13.

§**15.**

⟨Index the earliest-created constant value for this KOV 15⟩ ≡

```
    int found = FALSE;
    quantity *q;
    LOOP_OVER(q, quantity) {
        if ((kov_compare(qty_kind_of_value(q), kov)) &&
            (qty_is_a_variable(q) == FALSE)) {
            print_raw_text_to_file(q->word_ref1, q->word_ref2, ifl);
            found = TRUE;
            break;
        }
    }
    if (found == FALSE) INDEX("--");
```

This code is used in §13.

§**16. Indexing data type names.** We need this for other index pages, too, so it's abstracted into routines. Note that we add "any kind" for constructors, so that "list of" becomes "list of any kind".

```
void index_data_type_name(int t, int with_links) {
    if (type_IDs_table[t].word_ref1 >= 0) {
        index_lowercase_dtid(t);
        if (with_links) index_link(lw_array[type_IDs_table[t].word_ref1].lw_source);
    } else {
        if (type_IDs_table[t].parsing_name != NULL)
            INDEX("%s", vocab_get_exemplar(type_IDs_table[t].parsing_name, FALSE));
        else data_type_print_long_name(ifl, t);
    }
    if (data_type_is_constructor(t)) INDEX(" <i>any kind</i>");
}
void index_data_type_plural(int t) {
    if (type_IDs_table[t].plural_ref1 >= 0) {
        print_text_to_file(type_IDs_table[t].plural_ref1, type_IDs_table[t].plural_ref2,
            ifl);
        index_link(lw_array[type_IDs_table[t].plural_ref1].lw_source);
    } else {
        if (type_IDs_table[t].parsing_plural != NULL)
            INDEX("%s", vocab_get_exemplar(type_IDs_table[t].parsing_plural, FALSE));
        else data_type_print_long_plural(ifl, t);
    }
    if (data_type_is_constructor(t)) INDEX(" <i>any kind</i>");
}
```

The function index_data_type_name is called from 7/dim and 11/act.

Specifications 7/spec

Purpose

To create, manage and compare specifications.

Definitions

¶1. Conceptually, `specification` is the single most important data structure in Inform, unifying all of its concepts of data – specific or vague, singular or plural, now or in the past – in a single unambiguous representation. See the Introduction to this chapter for more on this.

The implementation is more down to earth, and looking at the structure definition is like looking at the side of a fancier Apple laptop – a long row of sockets, each different, some plugging into something specific, others general-purpose. As with laptop design, this `struct` has changed its sockets over the years, never being quite the same two builds running, yet always fulfilling the same basic function. And as with laptops, the pressure is always to get rid of what isn't needed. Very often the laptop is unplugged completely, or only has one or two cables connected, which makes all those unused sockets look wasteful. But all will be needed by someone, and we need to make sure there are enough sockets for any combination we will need to have fixed up at once.

Certainly the thinner this structure is, the better. Here is the current, sleek design, which ought to be branded Specification Pro. (Earlier models were nearly twice this size; sleek as the current form of SP may be, SPs still occupy around 8% of Inform's memory.) We have a single general-purpose socket, which like a USB port can fit a bewildering range of peripheral structures; a row of oddball proprietory sockets, fitting specific gadgets; and then a row of "arguments", which are like networking connections to other SPs.

```
define MAXIMUM_SPEC_ARGUMENTS 4          must be ≤ 7; TABLE_ENTRY_TY, the current record-holder, needs 4

typedef struct specification {
    int spec_packed_record;                                   see below: a packed bitmap
    int family_and_species;                                   UNKNOWN, or a family ID
    struct kind_of_value *associated_kov;                     NULL, or a kind of value
    int word_ref1, word_ref2;                                 the text thus interpreted
    general_pointer which_structure;                          can point to any Inform structure
    struct time_period *condition_tense;                      CONDITION_FMY: records the tense
    struct pcalc_prop *proposition;                           DESCRIPTION_SPC, TEST_PROPOSITION_SPC
    struct description_docket *docket;                        DESCRIPTION_SPC
    struct specification *arguments[MAXIMUM_SPEC_ARGUMENTS];
} specification;

int new_spec_count = 0, copied_spec_count = 0;                for the debugging log
```

The structure specification is private to this section.

¶2. We consolidate a large collection of flags, and small-integer values, into the "packed record" of the SP – a form of bitmap. A single SP can have any combination of the flags set or clear; can have any argument count; and can store a number n in the range $0 \le n < 65536$ for any *one* of several different purposes. When not stored, n reads as if it were -1.

define GENERIC_DATA_SPFLAG 0x00000001 *generic rather than actual data*
define ACTION_FOR_OTHER_SPFLAG 0x00000002 ACTION_SPC*: asking someone else to do this?*
define CONDITION_NEGATED_SPFLAG 0x00000004 CONDITION_FMY*: if set, negates the condition*
define RECORD_AS_SELF_SPFLAG 0x00000008 PROPERTY_VALUE_SPC*: record recipient as* self *when writing this*
define SELF_OBJECT_SPFLAG 0x00000010 CONSTANT_SPC-OBJECT_TY*: this represents* self *at run-time*
define NOTHING_OBJECT_SPFLAG 0x00000020 CONSTANT_SPC-OBJECT_TY *objects: this represents* nothing *at run-time*
define COERCED_SPFLAG 0x00000040 *warning: do not trust my value in a cache, I've been coerced!*

define BASE_OF_SPDATA_ID 0x00000100 *these bits hold a* *_SPDATA *constant showing how we use...*
define SPDATA_ID_MASK 0x00000f00
define BASE_OF_SPDATA 0x00001000 *...these bits*
define SPDATA_MASK 0x0ffff000

define NO_SPDATA 0 *the data portion of the bitmap isn't used*
define PHRASE_OPTION_SPDATA 1 TEST_PHRASE_OPTION_SPC*:* 2^i *where* i *is the option number,* $0 \le i < 16$
define QUANTITY_ENUMERATION_SPDATA 2 CONSTANT_SPC*: which one from an enumerated KOV*
define LOCAL_NUMBER_SPDATA 3 LOCAL_VARIABLE_SPC*: which of the locals on the current stack frame*
define GRAMMAR_TOKEN_SCORE_SPDATA 4 *score applied to this grammar token*

define BASE_OF_ARGC 0x10000000 *these bits hold the argument count, from 0 to* MAXIMUM_SPEC_ARGUMENTS
define ARGC_MASK 0x70000000

¶3. Species and their abilities. Different species use different combinations of the "sockets" in our structure, and we police that using the following flags:

define ASSOCIATED_KOV_SPCFLAG 0x00000001 *allowed to have an associated KOV*
define TENSE_SPCFLAG 0x00000002 *allowed to have a tense*
define PROPOSITION_SPCFLAG 0x00000004 *allowed to have a proposition*
define DOCKET_SPCFLAG 0x00000008 *allowed to have a docket*
define ARGUMENTS_SPCFLAG 0x00000010 *allowed to have one or more arguments*
define INVLIST_SPCFLAG 0x00000020 *allowed to have an invocation list*
define PHRASAL_SPCFLAG 0x00000040 *this species compiles to a function call*
define EVALUATING_SPCFLAG 0x00000080 *this species compiles to a value*

---

§**1. Creation.** There are two ways to create a SP structure – by allocating a new one in memory and initialising it, or by copying it from an existing one. Profiling shows that about 90% of SPs in memory at the end of Inform's run were made from new, and only 10% copied.

define SPECIES_MULTIPLIER 100

```
specification *spec_new(int family, int species) {
    specification *spec = CREATE(specification);
    spec->spec_packed_record = 0;
    spec->family_and_species = family + SPECIES_MULTIPLIER*species;
    spec->associated_kov = NULL;
    spec->word_ref1 = -1; spec->word_ref2 = -1;
    spec->which_structure = NULL_GENERAL_POINTER;
    spec->condition_tense = NULL;
```

```
    spec->proposition = NULL;
    spec->docket = NULL;
    new_spec_count++;
    return spec;
}
```

The function spec_new is called from 7/specv, 7/masp, 7/vasp, 7/stsp, 7/cosp and 7/cmsp.

§**2.** This routine could perhaps be defined as a macro, but I lack the steely nerves required.

```
specification *spec_copy(specification *spec) {
    specification *sts = CREATE(specification);
    *sts = *spec;
    copied_spec_count++;
    return sts;
}
```

The function spec_copy is called from 6/pform, 6/asp, 7/specv, 7/vasp, 7/cosp, 7/tc, 8/creat, 8/assem, 9/qty, 10/tab, 11/chron and 11/ap.

§**3. Coercion.** The process of transfiguring a given SP structure so that it belongs to a different species, or even a different family, is called "coercion". SPs which have been coerced are marked with a flag as a warning that what were previously duplicates of them may now not correspond with the original. Coercion is used as little as possible.

```
void spec_coerce_to(specification *spec, int new_family, int new_species) {
    spec->family_and_species = new_family + SPECIES_MULTIPLIER*new_species;
    spec_set_flag(spec, COERCED_SPFLAG);
}
```

The function spec_coerce_to is called from 7/vasp, 7/cosp and 7/cmsp.

§**4. Family and species.** Those are the *only* ways to set or alter the family or species – by creation, or subsequent coercion. Here are the ways to read them:

```
int species_is(specification *spec, int species) {
    if ((spec) && ((spec->family_and_species)/SPECIES_MULTIPLIER == species)) return TRUE;
    return FALSE;
}
int family_is(specification *spec, int family) {
    if ((spec) && ((spec->family_and_species)%SPECIES_MULTIPLIER == family)) return TRUE;
    return FALSE;
}
```

The function species_is is called from 2/prob2, 5/mlc, 6/term, 6/pform, 6/simp, 6/asp, 6/cind, 7/specv, 7/vasp, 7/stsp, 7/cosp, 7/tc, 8/refpt, 8/creat, 8/knowp, 9/prop, 9/pp, 9/inf, 10/list, 10/tab, 10/isin, 11/chron, 11/act, 11/ap, 11/av, 12/phud, 12/phtd, 12/cinv, 12/rb, 13/tfg, 13/gty and 13/gtok.
The function family_is is called from 5/aph, 6/simp, 6/defer, 7/stsp, 7/cosp, 7/tc, 8/refpt, 8/mass, 10/tab, 11/ap, 12/phrcd, 12/phtd, 12/cinv and 13/tfg.

§**5.** More explicitly:

```
int spec_get_species(specification *spec) {
    if (spec == NULL) return UNKNOWN;
    return (spec->family_and_species)/SPECIES_MULTIPLIER;
}
int spec_get_family(specification *spec) {
    if (spec == NULL) return UNKNOWN;
    return (spec->family_and_species)%SPECIES_MULTIPLIER;
}
```

The function spec_get_species is called from 7/tids, 7/vasp, 7/stsp, 7/cosp, 7/cmsp and 7/tc.
The function spec_get_family is called from 7/tids, 7/cfsp, 7/tc and 12/phtd.

§**6.** Different specifications have different abilities, some from their family membership, others their species membership.

```
int spec_test_ability(specification *spec, int f) {
    if (spec == NULL) return FALSE;
    if (type_ID_test_flag(spec_get_family(spec), f)) return TRUE;
    if (type_ID_test_flag(spec_get_species(spec), f)) return TRUE;
    return FALSE;
}
```

§**7. The unknown family and species.** Because they need to be given somewhere, and do not belong to any of the families, we might as well handle UNKNOWN specifications here:

```
int spec_is_UNKNOWN(specification *spec) {
    if (family_is(spec, UNKNOWN)) return TRUE;
    return FALSE;
}
specification *new_UNKNOWN_spec(void) {
    return spec_new(UNKNOWN, UNKNOWN);
}
void coerce_spec_to_UNKNOWN(specification *spec) {
    if (spec == NULL) internal_error("tried to invalidate null spec");
    spec_coerce_to(spec, UNKNOWN, UNKNOWN);
}
```

The function spec_is_UNKNOWN is called from 5/aph, 5/litp, 5/mlc, 6/tcpr, 7/ttts, 7/tc, 8/refpt, 9/qty, 9/scene, 9/prop, 9/pp, 10/tab, 10/eqns, 10/isin, 11/ap, 11/av, 12/phtd, 12/stv, 12/cinv and 13/gtok.
The function new_UNKNOWN_spec is called from 2/prob2, 5/mlc, 7/ttts, 7/specv, 8/refpt, 9/qty, 9/scene, 10/tab and 12/phtd.
The function coerce_spec_to_UNKNOWN is called from 5/mlc and 7/cosp.

§**8. Actual vs generic.** SPs are by default created as actual, since all SP-flags are initially clear, including `GENERIC_DATA_SPFLAG`. In practice, that's what most of them need to be. To create a generic specification, create an actual one and then use the routine below to make it generic instead.

```
void spec_make_actual(specification *spec) {
    spec_clear_flag(spec, GENERIC_DATA_SPFLAG);
}
void spec_make_generic(specification *spec) {
    spec_set_flag(spec, GENERIC_DATA_SPFLAG);
}
```

The function spec_make_actual is called from 7/cosp.
The function spec_make_generic is called from 7/specv, 7/masp, 7/vasp, 7/stsp and 7/cosp.

§**9.** And for testing:

```
int spec_is_actual(specification *spec) {
    return (spec_test_flag(spec, GENERIC_DATA_SPFLAG))?FALSE:TRUE;
}
int spec_is_generic(specification *spec) {
    return spec_test_flag(spec, GENERIC_DATA_SPFLAG);
}
```

The function spec_is_actual is called from 5/mlc, 7/vasp, 7/stsp and 7/tc.
The function spec_is_generic is called from 6/tcpr, 7/cfsp, 7/vasp, 7/stsp, 7/tc, 8/knowp, 10/tab and 11/ap.

§**10. Flags and data.** The following routines handle the flags crammed into the `spec_packed_record` field.

```
void spec_set_flag(specification *spec, int flag) {
    if (spec == NULL) internal_error("tried to set flag for null SP");
    spec->spec_packed_record = (spec->spec_packed_record) | flag;
}
void spec_clear_flag(specification *spec, int flag) {
    if (spec == NULL) internal_error("tried to clear flag for null SP");
    spec->spec_packed_record = (spec->spec_packed_record) & (~flag);
}
int spec_test_flag(specification *spec, int flag) {
    int f = (spec)?(spec->spec_packed_record):0;
    if (f & flag) return TRUE;
    return FALSE;
}
```

The function spec_set_flag is called from 5/mlc, 7/vasp, 7/cmsp, 7/tc and 11/chron.
The function spec_clear_flag is called from 5/mlc, 7/cmsp and 11/chron.
The function spec_test_flag is called from 5/mlc, 7/vasp, 7/stsp, 7/cosp, 7/cmsp, 7/tc and 11/chron.

§**11.** And similarly for the data portion of the packed record.

```
void spec_set_data(specification *spec, int purpose, int data) {
    if (spec == NULL) internal_error("tried to set data for null SP");
    if (data == -1) { purpose = NO_SPDATA; data = 0; }
    if ((data < 0) || (data >= 65536)) internal_error("tried to set data out of range");
    spec->spec_packed_record = ((spec->spec_packed_record) & (~(SPDATA_ID_MASK + SPDATA_MASK)))
        + BASE_OF_SPDATA_ID*purpose
        + BASE_OF_SPDATA*data;
}
int spec_get_data(specification *spec, int purpose) {
    if (spec == NULL) return -1;
    int there = ((spec->spec_packed_record) & SPDATA_ID_MASK)/BASE_OF_SPDATA_ID;
    if (there == NO_SPDATA) return -1;
    if (there == purpose) return ((spec->spec_packed_record) & SPDATA_MASK)/BASE_OF_SPDATA;
    internal_error("tried to fetch data with the wrong SPDATA ID");
    return -1;
}
```

The function spec_set_data is called from 7/stsp, 7/cosp, 9/qty and 13/gtok.
The function spec_get_data is called from 7/stsp, 7/cosp, 7/tc, 9/qty and 13/gty.

§**12. Arguments.** Recall that "arguments" are links out to other specification structures. The number of these currently attached is the "argc" (argument count) in the packed record.

```
int spec_get_argc(specification *spec) {
    if (spec == NULL) internal_error("tried to read argument of null SP");
    return ((spec->spec_packed_record) & ARGC_MASK)/BASE_OF_ARGC;
}
specification *spec_get_argument(specification *spec, int c) {
    if (spec == NULL) internal_error("tried to read argument of null SP");
    if ((c<0) || (c>=MAXIMUM_SPEC_ARGUMENTS)) internal_error("tried to read bad argument of SP");
    if (spec_test_ability(spec, ARGUMENTS_SPCFLAG) == FALSE)
        internal_error("tried to read argument of bad SP");
    if (c >= spec_get_argc(spec)) internal_error("tried to read missing argument of SP");
    return spec->arguments[c];
}
void spec_set_argument(specification *spec, int c, specification *arg) {
    if (spec == NULL) internal_error("tried to set argument of null SP");
    if ((c<0) || (c>=MAXIMUM_SPEC_ARGUMENTS)) internal_error("tried to set bad argument of SP");
    if (spec_test_ability(spec, ARGUMENTS_SPCFLAG) == FALSE)
        internal_error("tried to set argument of bad SP");
    if (c >= spec_get_argc(spec))
        spec->spec_packed_record = ((spec->spec_packed_record) & (~(ARGC_MASK))) + (c+1)*BASE_OF_ARGC;
    spec->arguments[c] = arg;
}
```

The function spec_get_argc is called from 6/simp, 6/defer, 6/cind, 7/stsp, 7/cosp, 7/cmsp, 7/tc and 11/ap.
The function spec_get_argument is called from 6/simp, 6/defer, 6/cind, 7/stsp, 7/cosp, 7/cmsp, 7/tc and 11/ap.
The function spec_set_argument is called from 5/mlc, 7/stsp, 7/cosp and 7/cmsp.

§**13. The associated KOV.** Some SPs have a kind of value associated with them: the following routines manage access to this.

```
kind_of_value *spec_get_kind_of_value(specification *spec) {
    if (spec_test_ability(spec, ASSOCIATED_KOV_SPCFLAG)) return spec->associated_kov;
    return NULL;
}
void spec_set_kind_of_value(specification *spec, kind_of_value *kov) {
    if (spec_test_ability(spec, ASSOCIATED_KOV_SPCFLAG)) spec->associated_kov = kov;
    else { LOG("$S\n", spec); internal_error("tried to set associated kov for wrong spec"); }
}
```

The function spec_get_kind_of_value is called from 2/prob2, 5/rel, 5/litp, 5/mlc, 6/term, 6/asp, 7/specv, 7/vasp, 7/stsp, 7/cosp, 7/tc, 8/refpt, 8/creat, 8/mass, 8/knowp, 9/prop, 9/pp, 9/provr, 9/inf, 10/tab, 10/eqns, 11/act, 11/av, 12/phtd, 12/cinv, 12/rb, 13/tfg, 13/gl, 13/gty and 13/gtok.
The function spec_set_kind_of_value is called from 7/masp, 7/vasp, 7/stsp, 7/tc, 9/qty, 10/bib, 11/ap, 12/phtd and 12/cinv.

§**14. The general structure field.**

```
general_pointer spec_get_structure_field(specification *spec) {
    if (spec == NULL) internal_error("tried to get structure for null spec");
    return spec->which_structure;
}
void spec_set_structure_field(specification *spec, general_pointer gp) {
    if (spec == NULL) internal_error("tried to set structure for null spec");
    spec->which_structure = gp;
}
```

The function spec_get_structure_field is called from 2/mem.
The function spec_set_structure_field is called from 2/mem.

§**15.** Perhaps the most interesting use of the general structure field is to point to an invocation list:

```
void spec_create_invocation_list(specification *spec) {
    if (spec_test_ability(spec, INVLIST_SPCFLAG) == FALSE) {
        LOG("SP: $X\n", spec); internal_error("can't create inv list for this spec");
    }
    ATTACH_TO_SPEC(spec, invocation_list, new_invocation_list());
}
void spec_set_invocation_list(specification *spec, invocation_list *invl) {
    if (spec_test_ability(spec, INVLIST_SPCFLAG) == FALSE) {
        LOG("SP: $X\n", spec); internal_error("can't create inv list for this spec");
    }
    ATTACH_TO_SPEC(spec, invocation_list, invl);
}
invocation_list *spec_invocation_list(specification *spec) {
    if (spec_test_ability(spec, INVLIST_SPCFLAG))
        return RETRIEVE_FROM_SPEC(spec, invocation_list);
    return NULL;
}
int spec_has_invocation_list(specification *spec) {
    if ((spec_test_ability(spec, INVLIST_SPCFLAG))
        && (VALID_POINTER_invocation_list(spec->which_structure))) return TRUE;
    return FALSE;
}
```

The function spec_create_invocation_list is called from 7/vasp, 7/cosp and 7/cmsp.
The function spec_set_invocation_list is called from 7/tc.
The function spec_invocation_list is called from 5/mlc, 6/cind, 7/vasp, 7/tc, 7/inv, 12/phtd, 12/cinv, 12/ambig and 12/cph.
The function spec_has_invocation_list is called from 7/tc, 7/inv and 12/cph.

§16. The attached proposition.

```
pcalc_prop *spec_get_proposition(specification *spec) {
    if (spec_test_ability(spec, PROPOSITION_SPCFLAG)) return spec->proposition;
    return NULL;
}
void spec_set_proposition(specification *spec, pcalc_prop *prop) {
    if (spec_test_ability(spec, PROPOSITION_SPCFLAG)) spec->proposition = prop;
    else { LOG("$S\n", spec); internal_error("tried to set proposition for wrong spec"); }
}
```

The function spec_get_proposition is called from 5/mlc, 6/pform, 6/treec, 6/defer, 7/vasp, 7/cosp, 7/tc, 8/refpt, 8/creat, 8/mass, 8/knowp, 10/tab, 11/chron, 11/ap, 12/phtd, 12/cinv, 13/gtok and 13/nft.
The function spec_set_proposition is called from 5/mlc, 7/vasp, 7/cosp and 11/chron.

§17. The attached docket.

```
description_docket *spec_get_docket(specification *spec) {
    if (spec_test_ability(spec, DOCKET_SPCFLAG)) return spec->docket;
/* if ((spec) && (spec->docket)) LOG("Surprising docket for $S\n", spec);
if (spec) return spec->docket; */
    return NULL;
}
void spec_set_docket(specification *spec, description_docket *dd) {
    if (spec_test_ability(spec, DOCKET_SPCFLAG)) spec->docket = dd;
    else { LOG("$S\n", spec); internal_error("tried to set docket for wrong spec"); }
}
```

The function spec_get_docket is called from 7/cosp.
The function spec_set_docket is called from 7/cosp.

§18. The attached tense.

```
time_period *spec_get_condition_tense(specification *spec) {
    if (spec_test_ability(spec, TENSE_SPCFLAG)) return spec->condition_tense;
    return NULL;
}
void spec_set_condition_tense(specification *spec, time_period *tp) {
    if (spec_test_ability(spec, TENSE_SPCFLAG)) spec->condition_tense = tp;
    else { LOG("$S\n", spec); internal_error("tried to set tense for wrong spec"); }
}
```

The function spec_get_condition_tense is called from 5/ml, 5/mlc and 7/cosp.
The function spec_set_condition_tense is called from 5/mlc, 7/cosp and 11/chron.

§**19. Evaluating and phrasal.** A specification is said to be "evaluating" if it compiles to an expression in I6 code.

```
int spec_is_evaluating(specification *spec) {
    return spec_test_ability(spec, EVALUATING_SPCFLAG);
}
kind_of_value *spec_evaluates_to(specification *spec) {
    if (spec == NULL) return NULL;
    switch(spec_get_family(spec)) {
        case COMMAND_FMY:   return kov_when_COMMAND_is_evaluated(spec);
        case CONDITION_FMY: return kov_when_CONDITION_is_evaluated(spec);
        case MATCHING_FMY:  return kov_when_MATCHING_is_evaluated(spec);
        case STORAGE_FMY:   return kov_when_STORAGE_is_evaluated(spec);
        case VALUE_FMY:     return kov_when_VALUE_is_evaluated(spec);
    }
    return NULL;
}
```

The function spec_is_evaluating is called from 7/tc, 8/knowp and 12/phtd.

The function spec_evaluates_to is called from 6/tcpr, 6/sch, 6/atoms, 6/defer, 6/cind, 7/cfsp, 7/vasp, 7/stsp, 7/cmsp, 7/tc, 8/creat, 8/knowp, 10/list, 10/tab, 10/eqns, 11/ap, 12/phtd, 12/cinv and 13/test.

§**20.** A specification is said to be "phrasal" if it compiles to a function call in I6 code (which may or may not return a value).

```
int spec_is_phrasal(specification *spec) {
    return spec_test_ability(spec, PHRASAL_SPCFLAG);
}
```

The function spec_is_phrasal is called from 6/cind, 8/refpt, 10/tab, 12/cinv and 13/tfg.

§**21. Pretty-printing specifications.** We need to be able to print legible forms of translations in order to produce good error messages, and also in order to describe phrases in the Index; those have to be English language forms.

```
void write_specification_out_in_English(specification *spec, char *text) {
    text[0] = 0;
    if (spec == NULL) { sprintf(text+strlen(text), "something unknown"); return; }
    switch(spec_get_family(spec)) {
        case COMMAND_FMY:   write_COMMAND_out_in_English(spec, text); return;
        case CONDITION_FMY: write_CONDITION_out_in_English(spec, text); return;
        case MATCHING_FMY:  write_MATCHING_out_in_English(spec, text); return;
        case STORAGE_FMY:   write_STORAGE_out_in_English(spec, text); return;
        case UNKNOWN:       sprintf(text+strlen(text), "something unrecognised"); return;
        case VALUE_FMY:     write_VALUE_out_in_English(spec, text); return;
    }
    internal_error("no such family");
}
```

The function write_specification_out_in_English is called from 2/prob2, 7/stsp and 12/phtd.

§**22.** The debugging log uses the concise shorthand for specifications described in the Introduction to this chapter. The concise form is a little formula for use in the middle of a line; the more flowery exhaustive form extends over and entirely occupies at least one and possibly more lines.

```
void log_specification_concisely(specification *spec) {
    log_specification_recursively(spec, FALSE);
    STREAM_FLUSH(dl);
}
void log_specification_exhaustively(specification *spec) {
    log_specification_recursively(spec, TRUE);
    STREAM_FLUSH(dl);
}
```

The function log_specification_concisely is called from 2/dl.
The function log_specification_exhaustively is called from 2/dl.

§**23.** And this is where it all happens:

```
void log_specification_recursively(specification *spec, int details) {
    if (spec == NULL) {
        LOG("<null type>");
        if (details) LOG("\n");
    } else if ((((pointer_sized_int) spec) >= -1000) && (((pointer_sized_int) spec) <= 1000)) {
        LOG("<corrupted? type %08x>", ((pointer_sized_int) spec));
        if (details) LOG("\n");
    } else {
        int family = spec_get_family(spec), species = spec_get_species(spec);
        if (type_ID_exists(family) == FALSE) {
            LOG("[Malformed:%d/%d]_TY", family, species);
            if (details) LOG("\n");
        } else {
            int w1 = spec->word_ref1, w2 = spec->word_ref2;
            if (spec_is_generic(spec)) LOG("(G)"); else LOG("(A)");
            if ((w1 >= 0) && (w2 >= w1) && (w2 < lexer_wordcount)) LOG("'$W'/", w1, w2);
            if ((spec_test_ability(spec, DOCKET_SPCFLAG)) && (spec_makes_callings(spec)))
                LOG("c/");
            LOG("$s/$s", family, species);
            if (spec_get_kind_of_value(spec)) LOG("-$u", spec_get_kind_of_value(spec));
            if (details) {
                switch(family) {
                    case COMMAND_FMY:   log_COMMAND_spec_details(spec); break;
                    case CONDITION_FMY: log_CONDITION_spec_details(spec); break;
                    case MATCHING_FMY:  log_MATCHING_spec_details(spec); break;
                    case STORAGE_FMY:   log_STORAGE_spec_details(spec); break;
                    case VALUE_FMY:     log_VALUE_spec_details(spec); break;
                }
                LOG("\n");
                int i;
                for (i=0; i<spec_get_argc(spec); i++) {
                    LOG("%d: ", i+1);
                    log_specification_recursively(spec_get_argument(spec, i), TRUE);
                }
                if (spec_has_invocation_list(spec))
                    log_invocation_list_in_detail(spec_invocation_list(spec));
```

```
            }
        }
    }
}
```

§24. Sorting. Some specifications are used to describe the applicability of rules and phrases, and since those must be sorted in order of how specific they are, we will need a way of telling when one specification is more specific than another. For instance, "Will Parker in the vineyard" beats "Will Parker" beats "a man" beats "a person" beats "an object" beats "a value".

The following is one of Inform's standardised comparison routines, which takes a pair of objects A, B and returns 1 if A makes a more specific description than B, 0 if they seem equally specific, or −1 if B makes a more specific description than A. This is transitive, and intended to be used in sorting algorithms.

```
int cco = 0;                                    comparison count: used to make the debugging log vaguely searchable
char *c_s_stage_law = "";                       name of the law being applied, which caused this to be called

int compare_specificity_of_spec(specification *spec1, specification *spec2) {
    LOGIF(SPECIFICITIES, "Law %s (test %d): comparing $S with $S\n",
        c_s_stage_law, cco++, spec1, spec2);
    ⟨Existence is itself something specific 25⟩;
    ⟨An actual specification is more specific than a generic one 26⟩;
    world_object *wo1 = spec_object_exactly_described_if_any(spec1);
    world_object *wo2 = spec_object_exactly_described_if_any(spec2);
    ⟨An exact object is more specific than a vague one 27⟩;
    if (wo1) ⟨Enclosing regions beat enclosed ones, and regions beat rooms 28⟩;
    int a = species_is(spec1, DESCRIPTION_SPC), b = species_is(spec2, DESCRIPTION_SPC);
    if ((a == TRUE) && (b == TRUE)) {                                   case 1: both are descriptions
        return compare_specificity_of_DESCRIPTIONs(spec1, spec2);
    } else if ((a == FALSE) && (b == FALSE)) {                          case 2: neither is a description
        ⟨Table entries are more specific than other non-descriptions 29⟩;
    } else {                                                            case 3: one is a description, the other isn't
        ⟨When is a description more specific than a non-description? 30⟩;
    }
    return 0;
}
```

The function compare_specificity_of_spec is called from 11/ap, 12/phrcd and 13/gty.

§25. Whether or not, as Bertrand Russell thought in 1894, existence is itself a good ("Great God in Boots! - the ontological argument is sound!"), a specification which exists is certainly more significant than one which does not; and there is nothing to choose between two specifications, neither of which exists.

A God who wears boots is an incongruous thought. Sandals, possibly. But maybe Russell meant the local Cambridge branch of Boots, the chemist's shop. At any rate he decided atheism was sounder still in 1896.

⟨Existence is itself something specific 25⟩ ≡

```
    if ((spec1 == NULL) && (spec2 != NULL)) return -1;
    if ((spec1 != NULL) && (spec2 == NULL)) return 1;
    if ((spec1 == NULL) && (spec2 == NULL)) return 0;
```

This code is used in §24.

§**26.** Hard to argue with this one: "34", for instance, is more specific than "a number". (We might quibble about whether or not "a number which equals 34" is really less specific than "34" – Inform says it is; but in fact it doesn't much matter either way.)

⟨An actual specification is more specific than a generic one 26⟩ ≡
```
if ((spec_is_actual(spec1)) && (spec_is_generic(spec2)))
    return 1;
if ((spec_is_generic(spec1)) && (spec_is_actual(spec2)))
    return -1;
```
This code is used in §24.

§**27.** For instance, "the open Marble Door" is more specific than "an open door" or even just "a door", even though the latter is linguistically simpler.

⟨An exact object is more specific than a vague one 27⟩ ≡
```
if ((wo1) && (wo2 == NULL)) return 1;
if ((wo1 == NULL) && (wo2)) return -1;
```
This code is used in §24.

§**28.** Suppose both specifications exactly describe objects, `wo1` and `wo2`, and all other considerations are equal. It's not quite true that one object is as good as another, because a region can be thought of as a set of rooms. (Without the following criterion, rules such as "After waiting in Russia" and "After waiting in Vladivostok Railway Station Waiting Room" would have equal status.)

⟨Enclosing regions beat enclosed ones, and regions beat rooms 28⟩ ≡
```
LOGIF(SPECIFICITIES, "Test %d: Comparing specificity of WOs $O and $O\n",
    cco, wo1, wo2);
int r1 = wo_of_kind(wo1, kind_room), r2 = wo_of_kind(wo2, kind_room);
int reg1 = wo_of_kind(wo1, kind_region), reg2 = wo_of_kind(wo2, kind_region);
if ((r1) && (reg2)) return 1;
if ((reg1) && (r2)) return -1;
if ((reg1) && (reg2)) {
    if (wo_in_wo(wo1, wo2)) return 1;
    if (wo_in_wo(wo2, wo1)) return -1;
}
```
This code is used in §24.

§**29.** Here neither specification is a `DESCRIPTION_SPC` or an actual object. For the most part, then, we're left with two specifications of about equal merit, and we don't choose between them. But in one case we do intervene: a table entry reference beats anything else left, so that "Instead of taking the magic key corresponding to Merlin in the Table of Arcana" is more specific than "Instead of taking the brass key". It's debateable whether this is a good convention, but users reported the previous absence of such a convention as a bug, which is usually telling.

⟨Table entries are more specific than other non-descriptions 29⟩ ≡
```
int t1 = species_is(spec1, TABLE_ENTRY_SPC);
int t2 = species_is(spec2, TABLE_ENTRY_SPC);
if ((t1 == TRUE) && (t2 == FALSE)) return 1;
if ((t1 == FALSE) && (t2 == TRUE)) return -1;
```
This code is used in §24.

§**30.** To explicate the following: a description of an exact object beats any non-description – thus "the open Marble Door" (a `DESCRIPTION_SPC`) beats "the Marble Door" (a `CONSTANT_SPC`). But any non-description beats a description which is vague about the object – thus "the Marble Door" beats "an open door", which is not news since rules above would enact that anyway, but also "the tallest door in the Castle" (a `PHRASE_TO_DECIDE_VALUE_SPC`) beats "an open door".

⟨When is a description more specific than a non-description? 30⟩ ≡

```
if (wo1) { if (a == TRUE) return 1; else return -1; }
else { if (a == TRUE) return -1; else return 1; }
```

This code is used in §24.

Text to Specifications 7/ttts

Purpose

Complete forms of parsing from excerpts of text to specifications.

7/ttts.§1-4 The cache; §5-7 The parser itself

Definitions

¶1. In this section we provide a routine, `parse_expression`, which turns a word range into a specification of its meaning.

This can parse six different forms of expression, though in fact three of these are variations of the same thing, so there are essentially only four different cases. The forms are identified by "expression context" or `*_EXPCON` numbers, as follows:

```
define VALUE_EXPCON 0
define CONDITION_EXPCON 1
define VOID_EXPCON 2
define TYPE_EXPCON 3
define DESCRIPTIVE_TYPE_EXPCON 4
define TYPE_OR_VALUE_EXPCON 5
define NUMBER_OF_EXPCONS 6
```

¶2. To explain this in brief:

(a) `VALUE_EXPCON` looks for source text which can be evaluated – a constant, a variable or other storage object, or a phrase to decide a value.

(b) `CONDITION_EXPCON` looks for a condition – anything legal after "if", in short. This includes sentence-like excerpts such as "six animals have been in the Stables".

(c) `VOID_EXPCON`, named after C's "void context", looks for a command – i.e., a "To..." phrase which does something rather than deciding something.

(d) `TYPE_EXPCON` is used where we expect to find the "type" of something – for instance, the kind of value to be stored in a variable, or the specification of a phrase argument.

(e) `DESCRIPTIVE_TYPE_EXPCON` is the same thing, with one difference: it allows nounless descriptions, such as "open opaque fixed in place", and to this end it treats bare adjective names as descriptions rather than values. If we have said "Colour is a kind of value. The colours are red, green and taupe. A thing has a colour.", then "green" is parsed by `DESCRIPTIVE_TYPE_EXPCON` as a description meaning "any thing which is green", but by `TYPE_EXPCON` and `VALUE_EXPCON` as a constant value of the KOV "colour".

(f) `TYPE_OR_VALUE_EXPCON` performs a `TYPE_EXPCON` first, and if that fails then it tries to parse a `VALUE_EXPCON` from the same text.

---

§**1. The cache.** The function `parse_expression` is called pretty frequently on overlapping or coinciding runs of text. Inform runs substantially faster if the results of parsing the most recent expressions are cached; so, for instance, if Inform parses the text in words 507 to 511 once, it need not do so again *in the same context.*

The cache takes the form of a modest ring buffer for each of the contexts:

```
define MAXIMUM_CACHE_SIZE 20          a Goldilocks value: too high slows us down, too low doesn't cache enough

typedef struct expression_cache {
```

```
    struct expression_cache_entry pe_cache[MAXIMUM_CACHE_SIZE];
    int pe_cache_size;                                  number of entries used, 0 to MAXIMUM_CACHE_SIZE
    int pe_cache_posn;                                  next write position, 0 to pe_cache_size minus 1
} expression_cache;

typedef struct expression_cache_entry {
    int word_ref1, word_ref2;                           the word range whose parsing this is
    struct specification *cached_pe_spec;               and the result (quite possibly UNKNOWN)
} expression_cache_entry;

expression_cache contextual_cache[NUMBER_OF_EXPCONS];

int parse_expression_called_before = FALSE;
```

The structure expression_cache is private to this section.
The structure expression_cache_entry is private to this section.

§**2.** The following seeks a previously cached answer:

⟨Check the expression cache to see if we already know the answer 2⟩ ≡

```
    expression_cache *ec = &(contextual_cache[context]);
    if (parse_expression_called_before == FALSE) {
        warn_expression_cache();                        this empties all the caches
        parse_expression_called_before = TRUE;
    }
    int i;
    for (i=0; i<ec->pe_cache_size; i++)
        if ((w1 == ec->pe_cache[i].word_ref1) && (w2 == ec->pe_cache[i].word_ref2))
            return ec->pe_cache[i].cached_pe_spec;
```

This code is used in §5.

§**3.** The cache expands until it reaches MAXIMUM_CACHE_SIZE; after that, entries are written in a position cycling through the ring. In either case it takes MAXIMUM_CACHE_SIZE further parses (not found in the cache) to overwrite the one we put down now.

⟨Write the newly discovered specification to the cache for future use 3⟩ ≡

```
    expression_cache *ec = &(contextual_cache[context]);
    ec->pe_cache[ec->pe_cache_posn].word_ref1 = w1;
    ec->pe_cache[ec->pe_cache_posn].word_ref2 = w2;
    ec->pe_cache[ec->pe_cache_posn].cached_pe_spec = spec;
    ec->pe_cache_posn++;
    if (ec->pe_cache_size < MAXIMUM_CACHE_SIZE) ec->pe_cache_size++;
    if (ec->pe_cache_posn == MAXIMUM_CACHE_SIZE) ec->pe_cache_posn = 0;
```

This code is used in §5.

§**4.** As with all caches, we have to be careful that the information does not fall out of date. There are two things which can go wrong: the specification pointed to by `cached_pe_spec` might be altered, perhaps as a result of the type-checker trying to force a round peg into a square hole; or the stock of Inform's defined names might change, so that the same text now has to be read differently.

The first problem can't be fixed here. It's tempting to try something like flagging specifications which have been altered, and then ensuring that the cache never serves up an altered result. But that fails for timing reasons – by the time the specification might be altered, pointers to it may exist in multiple data structures already, because the cache might have served it more than once by that time. (Not just a theoretical possibility – tests show that this does, albeit rarely, happen.) The brute force solution is to serve a copy of the cache entry, and thus never send out the same pointer twice. But this more than doubles the memory required to store specifications, which is unacceptable, and also slows Inform down, because allocating memory for all those copies is laborious. We therefore just have to be very careful about modifying specifications which have arisen from parsing.

The second problem is easier. We require other parts of Inform which make or unmake name definitions to warn us, by calling this routine. Definitions are made and unmade relatively rarely, so the performance hit is small.

```
void warn_expression_cache(void) {
    int i;
    for (i=0; i<NUMBER_OF_EXPCONS; i++) {
        contextual_cache[i].pe_cache_size = 0;
        contextual_cache[i].pe_cache_posn = 0;
    }
}
```

The function warn_expression_cache is called from 5/em and 12/phsf.

§**5. The parser itself.** With that out of the way, the actual routine is simple.

```
specification *parse_expression(int w1, int w2, int context) {
    if ((w1 < 0) || (w2 < w1)) return new_UNKNOWN_spec();
    if ((context < 0) || (context >= NUMBER_OF_EXPCONS))
        internal_error ("bad expression parsing context");
    ⟨Check the expression cache to see if we already know the answer 2⟩;
    meaning_list *ml = NULL;
    specification *spec = NULL;
    int pc = PARSED_SPCONTEXT;
    no_calls_to_SP_noun_phrase = 0;
    ⟨Ask the S-parser to produce a meaning list, then convert it 6⟩;
    ⟨Produce a problem message if the complexity threshold is reached 7⟩;
    if (spec->word_ref1 < 0) { spec->word_ref1 = w1; spec->word_ref2 = w2; }
    ⟨Write the newly discovered specification to the cache for future use 3⟩;
    if (!(spec_is_UNKNOWN(spec))) type_ID_observe(spec, pc);
    return spec;
}
```

The function parse_expression is called from 5/rel, 5/aph, 5/litp, 7/tc, 8/refpt, 9/scene, 9/prop, 9/pp, 10/list, 10/tab, 10/eqns, 10/bib, 10/isin, 11/act, 11/ap, 11/av, 12/phud, 12/phtd, 12/phsf, 12/stv, 12/cph, 12/def, 12/rb, 13/tfg, 13/gl, 13/gtok, 13/test and 14/i6t.

§**6.** The factors neatly into calls to the S-parser and the meaning-list converter, respectively, but note that TE-conversion requires `%9` to be set to the current word range in order for some of its problem messages to work.

⟨Ask the S-parser to produce a meaning list, then convert it 6⟩ ≡

```
CLEAR_MLC_BACKTRACE;
switch(context) {
    case VALUE_EXPCON:
        ml = SP_value(w1, w2);
        spec = VAL_subtree_to_spec(ml);
        break;
    case CONDITION_EXPCON:
        ml = SP_condition(w1, w2, TRUE);
        spec = COND_subtree_to_spec(ml);
        break;
    case VOID_EXPCON:
        ml = SP_command(w1, w2);
        spec = CMD_subtree_to_spec(ml);
        break;
    case TYPE_EXPCON:
        ml = SP_type_expression(w1, w2, FALSE);
        quote_words(9, w1, w2);
        spec = TE_subtree_to_spec(ml);
        pc = TYPE_PARSED_SPCONTEXT;
        break;
    case DESCRIPTIVE_TYPE_EXPCON:
        ml = SP_type_expression(w1, w2, TRUE);
        quote_words(9, w1, w2);
        spec = TE_subtree_to_spec(ml);
        pc = TYPE_PARSED_SPCONTEXT;
        break;
    case TYPE_OR_VALUE_EXPCON:
        ml = SP_type_expression(w1, w2, FALSE);
        quote_words(9, w1, w2);
        spec = TE_subtree_to_spec(ml);
        pc = TYPE_PARSED_SPCONTEXT;
        if (spec_is_UNKNOWN(spec)) {
            [[w1, w2 == the ... --> w1, w2]];
            ml = SP_value(w1, w2);
            spec = VAL_subtree_to_spec(ml);
            pc = PARSED_SPCONTEXT;
        }
        break;
}
CLEAR_MLC_BACKTRACE;
```

This code is used in §5.

§**7.** I don't believe the following problem message has been seen in the wild, but it was a useful precaution in the early days of testing the S-parser.

⟨Produce a problem message if the complexity threshold is reached 7⟩ ≡

```
if (no_calls_to_SP_noun_phrase >= S_PARSER_COMPLEXITY_THRESHOLD) {
    sentence_problem(_P_(C7OutOfControl),
        "this sentence seems to involve something too complicated to parse",
        "in that applying Inform's usual parsing techniques to it "
        "would take a very long time indeed to complete: since this "
        "tends to happen only with source text where some accident "
        "has occurred, for instance its punctuation having been "
        "stripped out, I'm reporting this as a Problem rather than "
        "wading blindly into it.");
}
```

This code is used in §5.

Specification Versions of Other Structures 7/specv

Purpose

To see how specifications correspond to other structures which they conceptually include.

Definitions

¶1. We have already seen in Chapter 5 that `meaning_list` structures are converted into specifications, but that isn't what we mean here: that's a permanent change (and the meaning lists are thrown away afterwards). Here we look at how specifications can represent the same data as other, more specific, structures: a type ID integer; a `kind_of_value`; a `quantity`; and a `world_object`.

---

§**1. Type ID numbers.** Here the S-parser supplies us with the ID of what might refer to something as broad as a family, say "condition", or as narrow as a constant value of a particular KOV, say "number". We have to create the appropriate specification to refer to anything falling under this reference. (This is the only place in Inform where specifications are created other than via constructor routines which explicitly know what species they are making.)

```
specification *new_generic_type_from_ID(int ID_number) {
    if (ID_number == UNKNOWN) return new_UNKNOWN_spec();
    if (type_ID_is_data_type(ID_number))
        return new_generic_CONSTANT_type(kova(ID_number));
    int family = UNKNOWN, species = UNKNOWN;
    if (type_ID_is_family_ID(ID_number)) {
        family = ID_number;
        species = UNKNOWN;
    } else {
        family = type_ID_get_family_of_species(ID_number);
        species = ID_number;
        if (family == UNKNOWN) {
            LOG("ID number is %d = $s\n", ID_number, ID_number);
            internal_error("no permitted family for this species");
        }
    }
    specification *spec = spec_new(family, species);
    spec_make_generic(spec);
    return spec;
}
```

The function new_generic_type_from_ID is called from 5/ml.

§**2. How KOVs are embedded in the specification hierarchy.** KOVs form a sort of sub-hierarchy of the larger hierarchy of specifications.

Every KOV represents the concept of being a value of that kind – for instance, `kova(TEXT_TY)` represents "being a piece of text". The equivalent idea among specifications is a generic constant value of that kind: for instance,

`(G)VALUE_FMY/CONSTANT_SPC-text-KOV`

The following routine makes the correspondence: given K, it returns a specification for a generic constant K-value.

```
specification *kov_as_spec(kind_of_value *kov) {
    if (kov == NULL) return NULL;
    return new_generic_CONSTANT_type(kov);
}
```

The function kov_as_spec is called from 7/cmsp, 9/pp and 10/tab.

§**3. Specifications from quantities.** The `quantity` structure indicates a named value, which might be either a variable or a constant – this means it can give rise to a specification in either of two different families, so we handle the conversion here.

```
specification *new_QUANTITY_spec(quantity *q) {
    specification *spec;
    if (q == NULL) internal_error("new_QUANTITY_spec on null quantity");
    if (qty_is_a_variable(q)) {
        spec = new_actual_NONLOCAL_VARIABLE_type(q);
    } else {
        if (is_kova(qty_kind_of_value(q), SCENE_TY))
            spec = spec_copy(qty_get_initial_value(q));
        else {
            spec = new_actual_CONSTANT_spec(qty_kind_of_value(q));
            ATTACH_TO_SPEC(spec, quantity, q);
        }
        spec->word_ref1 = q->word_ref1; spec->word_ref2 = q->word_ref2;
    }
    return spec;
}
quantity *spec_get_constant_quantity_if_any(specification *spec) {
    if (spec_is_actual_CONSTANT(spec)) {
        kind_of_value *kov = spec_get_kind_of_value(spec);
        if (is_kova(kov, SCENE_TY))
            return scene_quantity(SCENE_spec_to_scene(spec));
        if (kov_is_an_enumeration(kov))
            return RETRIEVE_FROM_SPEC(spec, quantity);
    }
    return NULL;
}
```

The function new_QUANTITY_spec is called from 5/tandv, 5/mlc, 6/term, 6/simp, 6/defer, 8/refpt, 9/qty, 9/pp, 10/tab, 11/ap, 13/tfg and 13/nft.

The function spec_get_constant_quantity_if_any is called from 6/pform, 6/equal, 7/vasp, 7/tc, 8/mass, 9/scene, 9/inf, 9/cot, 12/phud, 12/phrcd and 12/cinv.

§**4. Exact and vague object descriptions.** Specifications which talk about objects lie in two different families: "an animal" or "Lassie" are in the `CONSTANT_SPC` species of the `VALUE_FMY` family, with KOV "object"; but "an open door which is in a lighted room" (say) is in the `DESCRIPTION_SPC` species of the `CONDITION_FMY` family. It's convenient to have routines handling these two cases jointly:

```
int spec_describes_an_object_vaguely_or_exactly(specification *spec) {
    if (spec == NULL) return FALSE;
    if (species_is(spec, DESCRIPTION_SPC)) return TRUE;
    if (spec_is_CONSTANT_of_kova(spec, OBJECT_TY)) return TRUE;
    return FALSE;
}

world_object *spec_object_exactly_described_if_any(specification *spec) {
    if (spec == NULL) return NULL;
    if (species_is(spec, DESCRIPTION_SPC)) return spec_get_described_object(spec);
    if (spec_is_actual_CONSTANT_of_kova(spec, OBJECT_TY)) {
        if (spec_is_nothing_object_constant(spec)) return NULL;
        return OBJECT_spec_to_world_object(spec);
    }
    return NULL;
}
```

The function spec_describes_an_object_vaguely_or_exactly is called from 5/aph, 11/ap and 13/gtok.

The function spec_object_exactly_described_if_any is called from 6/simp, 7/spec, 10/tab, 11/ap, 13/tfg and 13/gtok.

Compiling from Specifications 7/cfsp

Purpose

To compile specifications into Inform 6 values, conditions or void expressions.

Definitions

¶1. In a more traditional compiler, the code-generator would be something of a landmark – one of the three or four most important stations. Here it's something of an anticlimax, partly because traditional "code" – values and statements – are only a small part of the I6 we have to generate, which also includes object and class definitions, grammar, and so on.

Still, this is the key point where the actual rather than generic specifications – phrases to do something, or to decide things; constants; variables; conditions – finally convert into I6 code.

¶2. For the most part this is "modeless" – that is, the I6 code generated by a specification does not depend on any context. But not entirely so, and we have a small set of "C-modes", each of which slightly alters the result to fit some particular need.

The rule is that any part of Inform needing to do something in a specific mode should place that operation within a pair of `BEGIN_COMPILATION_MODE` and `END_COMPILATION_MODE` macros, in such a way that execution always passes from one to the other. Within those bookends, it can use either the enter or exit macros to switch a particular mode on or off.

```
define BEGIN_COMPILATION_MODE
    int status_quo_ante = compilation_mode;
define COMPILATION_MODE_ENTER(mode)
    compilation_mode |= mode;
define COMPILATION_MODE_EXIT(mode)
    compilation_mode &= (~mode);
define END_COMPILATION_MODE
    compilation_mode = status_quo_ante;
define TEST_COMPILATION_MODE(mode)
    (compilation_mode & mode)

define DEREFERENCE_POINTERS_CMODE     0x00000001    make an independent copy of the result if on the heap
define INSIDE_RESOLVER_CMODE          0x00000002    compiling inside an I6 ambiguity-resolving routine
define BLOCK_CONSTANT_CMODE           0x00000004    compiling a pointer to a constant value on the heap
define DO_NOT_CREATE_LOCAL_VARS_CMODE 0x00000008    some phrases, like "let", create local variables
define IMPLY_NEWLINES_IN_SAY_CMODE    0x00000010    true except in text routines
define PERMIT_LOCALS_IN_TEXT_CMODE    0x00000020    unless casting to indexed text
define COMPILE_TEXT_TO_XML_CMODE      0x00000040    use XML escapes and UTF-8 encoding
define TRUNCATE_TEXT_CMODE            0x00000080    into a plausible filename length

int compilation_mode = DEREFERENCE_POINTERS_CMODE + IMPLY_NEWLINES_IN_SAY_CMODE;    default
```

¶3. These modes are all explained where they are used. The one used right here is `DEREFERENCE_POINTERS_CMODE`. This applies only when compiling a specification which generates a pointer value – an I6 value which is a pointer to a larger block of data on the heap, such as a list or indexed text.

Inform presents such values to the end user exactly as if they are non-pointer values. It must always be careful to ensure that there are never two different I7 values each holding pointers to the same block of data, because then changing one would also change the other. So we ordinarily need to make a copy of any block of data produced as a value; this is called "dereferencing".

But there are some circumstances – initialising entries in an Inform 6 array, for instance – where we don't want to do this, and indeed can't, because the code to handle dereferencing is invalid as an Inform 6 constant. The mode therefore exists as a way of temporarily turning off dereferencing – by default, it is always on.

---

§**1.** The outer shell here has two purposes. One is to copy the specification onto the local stack frame and then compile that copy – useful since compilation may alter its contents. The other purpose, and this is not to be dismissed lightly, is to ensure correct indentation in the log when `compile_TS_primitive` exits unexpectedly, for instance due to a problem. Correct indentation in the log is much more helpful than might be guessed.

```
void spec_compile(OUTPUT_STREAM, specification *spec) {
    LOGIF(EXPRESSIONS, "Compiling: $S\n", spec);
    if (spec_is_generic(spec)) internal_error("compiled a generic specification");
    STREAM_INDENT(dl);
    specification breakable_copy = *spec;
    spec_compile_primitive(OUT, &breakable_copy);
    STREAM_OUTDENT(dl);
}
```

The function spec_compile is called from 6/defer, 6/cind, 7/stsp, 7/cosp, 7/cmsp, 9/qty, 9/scene, 9/inf, 9/cot, 10/list, 10/tab, 10/eqns, 10/bib, 11/chron, 11/ap, 11/av, 12/phrcd, 12/phsf, 12/cinv, 12/cph, 12/def, 12/br, 13/gl, 13/gty, 13/gpr, 13/test and 14/i6t.

§**2.** So this is where the compilation is done, or rather, delegated:

```
void spec_compile_primitive(OUTPUT_STREAM, specification *spec) {
    type_ID_observe(spec, COMPILED_SPCONTEXT);
    char *suffix = "";
    if ((TEST_COMPILATION_MODE(DEREFERENCE_POINTERS_CMODE)) &&
        (TEST_COMPILATION_MODE(INSIDE_RESOLVER_CMODE) == FALSE)) {
        kind_of_value *kov = spec_evaluates_to(spec);
        if ((kov) && (kov_uses_pointer_values(kov))) {
            pointer_allocation *pall = phsf_add_allocation(kov, NULL);
            WRITE("BlkValueCopy(%s, ", pall_get_local_reference(pall));
            suffix = ")";
        }
    }
    switch(spec_get_family(spec)) {
        case COMMAND_FMY:   compile_COMMAND_spec(OUT, spec); break;
        case CONDITION_FMY: compile_CONDITION_spec(OUT, spec); break;
        case MATCHING_FMY:  compile_MATCHING_spec(OUT, spec); break;
        case STORAGE_FMY:   compile_STORAGE_spec(OUT, spec); break;
        case UNKNOWN:       break;                                  for error recovery
        case VALUE_FMY:     compile_VALUE_spec(OUT, spec); break;
    }
    WRITE("%s", suffix);
}
```

MATCHING Specifications 7/masp

Purpose

Documentation on and functions handling specifications which fall into the MATCHING family.

§1. Creation. The only MATCHING species is `NEW_LOCAL_VARIABLE_NAME_SPC`, used only for a generic SP which matches text which either has no meaning, or refers to a globally defined excerpt which a local variable name is allowed to temporarily override. For instance, this is the specification of the parameter in:

> To prognosticate about (V - nonexisting number variable): ...

As in this case, such a SP can also have an associated KOV (here it is `NUMBER_TY`, of course). On a successful match with this token in type-checking, a new local variable is created with the name matched, and given the KOV supplied.

```
void create_MATCHING_species(void) {
    type_ID_set_description(MATCHING_FMY, "a miscellaneous type used for matching only");
    type_ID_set_source_name(MATCHING_FMY, "MATCHING_FMY");
    type_ID_permit_pair(MATCHING_FMY, NEW_LOCAL_VARIABLE_NAME_SPC,
        PARSED_SPCONTEXT + TYPE_PARSED_SPCONTEXT + TYPE_EXPECTED_SPCONTEXT + TYPE_FOUND_SPCONTEXT);
    type_ID_set_description(NEW_LOCAL_VARIABLE_NAME_SPC, "a name not so far used");
    type_ID_set_source_name(NEW_LOCAL_VARIABLE_NAME_SPC, "NEW_LOCAL_VARIABLE_NAME_SPC");
    type_ID_set_flag(NEW_LOCAL_VARIABLE_NAME_SPC, ASSOCIATED_KOV_SPCFLAG);
}
specification *new_NEW_LOCAL_VARIABLE_NAME_spec(kind_of_value *kov) {
    specification *spec = spec_new(MATCHING_FMY, NEW_LOCAL_VARIABLE_NAME_SPC);
    if (kov) spec_set_kind_of_value(spec, kov);
    spec_make_generic(spec);
    return spec;
}
```

The function create_MATCHING_species is called from 7/tids.

The function new_NEW_LOCAL_VARIABLE_NAME_spec is called from 5/mlc.

§2. Pretty-printing. There is nothing very specific to say about them:

```
void write_MATCHING_out_in_English(specification *spec, char *text) {
    sprintf(text+strlen(text), "a name not so far used");
}
void log_MATCHING_spec_details(specification *spec) {
}
```

The function write_MATCHING_out_in_English is called from 7/spec.

The function log_MATCHING_spec_details is called from 7/spec.

§**3. Compilation.** Since `MATCHING` family species are purely for type-checking, they are never actual, always generic; so they do not evaluate, and cannot be compiled.

```
kind_of_value *kov_when_MATCHING_is_evaluated(specification *spec) {
    return NULL;
}
void compile_MATCHING_spec(OUTPUT_STREAM, specification *spec_found) {
    internal_error("MATCHING specifications cannot be compiled");
}
```

The function kov_when_MATCHING_is_evaluated is called from 7/spec.

The function compile_MATCHING_spec is called from 7/cfsp.

VALUE Specifications 7/vasp

Purpose

Documentation on and functions handling specifications which fall into the VALUE family.

7/vasp.§1-5 Creation; §6-9 Coercion; §10-15 Testing; §16-17 Pretty-printing; §18-28 Compilation

---

Template interpreter commands

```
28  {-callv:create_block_constants}
```

Definitions

¶1. The `VALUE` family represents what in other programming languages would be called "lvalues" rather than "rvalues": things which evaluate, but which cannot be the target of assignments. "Rvalues", which both evaluate and can also be assigned to, make up the `STORAGE` family.

¶2. "Block" (or sometimes "pointer") values are those which, at run-time, are stored in blocks of data on the heap, and represented by pointers to these blocks. That makes constants harder to compile: whereas compiling "34" means writing a single word, compiling "$\{1, 2, 4\}$" *means writing a pointer and also remembering to compile the list conte*

```
typedef struct stored_block_value {
    struct specification *stored_constant;          the block value we'll need to store
    int size_of_block;                              number of bytes on the heap to allocate
    MEMORY_MANAGEMENT
} stored_block_value;
```

The structure stored_block_value is private to this section.

---

§**1. Creation.** The `CONSTANT_SPC` species include boths literals – values lexically recognised at once as being constants of a given KOV, in the way that "103" is recognised as a `NUMBER_TY` and that "21 kg" might be recognised as a value of a unit of weight – and also names for constants which need to be parsed from the registered excerpts: for instance, "green" in an enumerated KOV whose values are "red", "green" and "blue". Such SPs use the associated KOV to indicate what kind of value they have (or are expected to match against).

§**2.** PHRASE_TO_DECIDE_VALUE_SPC specifications carry an attached invocation list, as all phrase specifications do.

```
void create_VALUE_species(void) {
    type_ID_set_description(VALUE_FMY, "a value quoted or calculated but not stored");
    type_ID_set_source_name(VALUE_FMY, "VALUE_FMY");
    type_ID_set_flag(VALUE_FMY, EVALUATING_SPCFLAG);
    type_ID_permit_pair(VALUE_FMY, CONSTANT_SPC,
        TYPE_PARSED_SPCONTEXT + PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT +
        TYPE_EXPECTED_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(CONSTANT_SPC, "a value");
    type_ID_set_source_name(CONSTANT_SPC, "CONSTANT_SPC");
    type_ID_set_flag(CONSTANT_SPC, ASSOCIATED_KOV_SPCFLAG);
    type_ID_set_flag(CONSTANT_SPC,
        DOCKET_SPCFLAG + PROPOSITION_SPCFLAG);                    for DESCRIPTION_TY constants only
    type_ID_permit_pair(VALUE_FMY, PHRASE_TO_DECIDE_VALUE_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + TYPE_EXPECTED_SPCONTEXT +
        COMPILED_SPCONTEXT);
    type_ID_set_description(PHRASE_TO_DECIDE_VALUE_SPC, "a phrase to decide a value");
    type_ID_set_source_name(PHRASE_TO_DECIDE_VALUE_SPC, "PHRASE_TO_DECIDE_VALUE_SPC");
    type_ID_set_flag(PHRASE_TO_DECIDE_VALUE_SPC, PHRASAL_SPCFLAG + INVLIST_SPCFLAG);
}
```

The function create_VALUE_species is called from 7/tids.

§**3.** In the case of actual CONSTANT_SPC specifications, a pointer to some structure may be attached, with the choice of structure depending on KOV. For instance, a constant of TABLE_TY carries a pointer to the relevant table structure, but a constant NUMBER_TY has nothing attached.

Constants where no pointer is attached can be straightforwardly be made with the following:

```
specification *new_actual_CONSTANT_spec(kind_of_value *kov) {
    specification *spec = spec_new(VALUE_FMY, CONSTANT_SPC);
    spec_set_kind_of_value(spec, kov);
    return spec;
}
specification *new_generic_CONSTANT_type(kind_of_value *kov) {
    specification *spec = new_actual_CONSTANT_spec(kov);
    spec_make_generic(spec);
    return spec;
}
```

The function new_actual_CONSTANT_spec is called from 2/mem, 5/lit, 7/specv, 8/sob, 8/knowp, 9/madj, 11/ap and 12/phtd.
The function new_generic_CONSTANT_type is called from 2/prob2, 5/ml, 6/asp, 7/specv, 7/stsp, 7/tc, 8/refpt, 8/creat, 8/mass, 8/knowp, 9/pp, 10/tab, 10/eqns, 11/act, 11/ap, 11/av, 12/rb and 13/gtok.

§**4.** Constants with pointers attached are made with the macro-defined routines defined way back in "Memory", with names like `table_to_TABLE_spec`, which take one argument – a pointer to the relevant data structure, which must be the right type and non-`NULL` – and returns a new actual constant of the corresponding KOV and with this pointer attached.

That works very well, except for one quirk. There are two pseudo-objects for which no pointers to `world_object` can exist: "self" and "nothing". These are marked out with special flags:

```
specification *new_self_object_constant(void) {
    specification *spec = new_actual_CONSTANT_spec(kova(OBJECT_TY));
    spec_set_flag(spec, SELF_OBJECT_SPFLAG);
    return spec;
}
specification *new_nothing_object_constant(void) {
    specification *spec = new_actual_CONSTANT_spec(kova(OBJECT_TY));
    spec_set_flag(spec, NOTHING_OBJECT_SPFLAG);
    return spec;
}
```

The function new_self_object_constant is called from 6/sconv and 7/tc.
The function new_nothing_object_constant is called from 5/mlc and 6/treec.

§**5.** The following makes a new phrase-to-decide-value, with an empty invocation list ready to be filled in.

```
specification *new_PHRASE_TO_DECIDE_VALUE_spec(void) {
    specification *spec = spec_new(VALUE_FMY, PHRASE_TO_DECIDE_VALUE_SPC);
    spec_create_invocation_list(spec);
    return spec;
}
```

The function new_PHRASE_TO_DECIDE_VALUE_spec is called from 5/mlc.

§**6. Coercion.** The need for coercions is usually a sign of some subtle ambiguity in how the same data might be specified, and the following demonstrate that. First, when a description such as "people holding animals" is treated as a constant value to be passed as a phrase parameter, it has to migrate from the `CONDITION` to the `VALUE` family. It carries its docket and/or proposition with it.

```
int coerce_DESCRIPTION_to_VALUE(specification *spec, int objects) {
    if (species_is(spec, DESCRIPTION_SPC) == FALSE)
        internal_error("can't coerce non-DESCRIPTION");
    pcalc_prop *prop = spec_get_proposition(spec);
    if (prop) {
        prop = prop_copy(prop);
        prop = prop_trim_universal_quantifier(prop);
        if (vars_number_free(prop) != 1) return FALSE;
        spec_set_proposition(spec, prop);
    }
    spec_coerce_to(spec, VALUE_FMY, CONSTANT_SPC);
    if (objects) spec_set_kind_of_value(spec, kova(OBJECT_DESCRIPTION_TY));
    else spec_set_kind_of_value(spec, kova(VALUE_DESCRIPTION_TY));
    return TRUE;
}
```

The function coerce_DESCRIPTION_to_VALUE is called from 7/tc and 12/phrcd.

§**7.** Much the same can be said of action patterns:

```
void coerce_TEST_ACTION_to_STORED_ACTION(specification *spec) {
    if (species_is(spec, TEST_ACTION_SPC) == FALSE)
        internal_error("can't coerce non-TEST_ACTION_SPC");
    spec_coerce_to(spec, VALUE_FMY, CONSTANT_SPC);
    spec_set_kind_of_value(spec, kova(STORED_ACTION_TY));
}
```

The function coerce_TEST_ACTION_to_STORED_ACTION is called from 8/refpt and 10/tab.

§**8.** The following is used in order to make it possible to test whether stored actions match a given description. For instance, the "is" in:

if the current action is taking something...

compares a stored action ("current action" is a phrase to decide a stored action) with "taking something", which needs to be a value in order for it to be a noun phrase. So it is converted to a constant `DESCRIPTION_OF_ACTION_TY` to make this viable.

```
void coerce_TEST_ACTION_to_DESCRIPTION_OF_ACTION(specification *spec) {
    if (species_is(spec, TEST_ACTION_SPC) == FALSE)
        internal_error("can't coerce non-TEST_ACTION_SPC");
    spec_coerce_to(spec, VALUE_FMY, CONSTANT_SPC);
    spec_set_kind_of_value(spec, kova(DESCRIPTION_OF_ACTION_TY));
}
```

The function coerce_TEST_ACTION_to_DESCRIPTION_OF_ACTION is called from 6/simp.

§**9.** Lastly, the notation in the source for text to be printed is the same as that to be matched against typed commands, but these have very different representations at run-time. The following is used when we've guessed wrongly which we're going to want:

```
void coerce_UNDERSTANDING_to_TEXT(specification *spec) {
    if (species_is(spec, CONSTANT_SPC) == FALSE)
        internal_error("can't coerce non-CONSTANT_SPC");
    spec_set_kind_of_value(spec, kova(TEXT_TY));
}
```

The function coerce_UNDERSTANDING_to_TEXT is called from 7/cmsp.

§**10. Testing.** The following provide convenient shorthands for testing whether a given specification is a constant.

```
int spec_is_actual_CONSTANT(specification *spec) {
    if ((species_is(spec, CONSTANT_SPC)) && (spec_is_actual(spec))) return TRUE;
    return FALSE;
}
int spec_is_generic_CONSTANT(specification *spec) {
    if ((species_is(spec, CONSTANT_SPC)) && (spec_is_generic(spec))) return TRUE;
    return FALSE;
}
```

The function spec_is_actual_CONSTANT is called from 5/aph, 5/litp, 5/mlc, 6/term, 7/specv, 7/tc, 8/mass, 8/knowp, 10/tab, 10/eqns, 12/phtd and 12/rb.

The function spec_is_generic_CONSTANT is called from 5/rel, 5/litp, 5/mlc, 7/cosp, 7/tc, 8/refpt, 8/creat, 8/mass, 8/knowp, 9/prop, 9/pp, 9/provr, 9/inf, 10/tab, 10/eqns, 11/act, 11/ap, 11/av, 12/phtd, 12/cinv, 13/tfg, 13/gl, 13/gty and 13/gtok.

§**11.** And now versions which require a particular KOV, too.

```
int spec_is_CONSTANT_of_kova(specification *spec, int bt) {
    if ((species_is(spec, CONSTANT_SPC)) && (is_kova(spec_get_kind_of_value(spec), bt)))
        return TRUE;
    return FALSE;
}
int spec_is_actual_CONSTANT_of_kova(specification *spec, int bt) {
    if ((species_is(spec, CONSTANT_SPC)) && (spec_is_actual(spec)) &&
        (is_kova(spec_get_kind_of_value(spec), bt))) return TRUE;
    return FALSE;
}
int spec_is_generic_CONSTANT_of_kova(specification *spec, int bt) {
    if ((species_is(spec, CONSTANT_SPC)) && (spec_is_generic(spec)) &&
        (is_kova(spec_get_kind_of_value(spec), bt))) return TRUE;
    return FALSE;
}
```

The function spec_is_CONSTANT_of_kova is called from 2/prob2, 6/pform, 6/sconv, 7/specv, 7/stsp, 7/cmsp, 7/tc, 8/mass, 10/tab, 11/ap, 12/phtd, 12/cinv and 13/gtok.
The function spec_is_actual_CONSTANT_of_kova is called from 2/mem, 6/simp, 7/specv, 7/tc, 9/prop, 9/pp and 10/isin.
The function spec_is_generic_CONSTANT_of_kova is called from 7/tc and 12/phtd.

§**12.** To go back to the self/nothing anomalies (that really could be an episode title from *The Big Bang Theory*),

```
int spec_is_nothing_object_constant(specification *spec) {
    if (spec_test_flag(spec, NOTHING_OBJECT_SPFLAG)) return TRUE;
    return FALSE;
}
int spec_is_self_object_constant(specification *spec) {
    if (spec_test_flag(spec, SELF_OBJECT_SPFLAG)) return TRUE;
    return FALSE;
}
world_object *wo_of_CONSTANT_OBJECT_if_any(specification *spec) {
    if (spec_is_actual_CONSTANT_of_kova(spec, OBJECT_TY)) {
        if (spec_test_flag(spec, SELF_OBJECT_SPFLAG)) return NULL;
        if (spec_test_flag(spec, NOTHING_OBJECT_SPFLAG)) return NULL;
        return OBJECT_spec_to_world_object(spec);
    }
    return NULL;
}
```

The function spec_is_nothing_object_constant is called from 6/vars, 6/simp, 7/specv, 7/tc and 8/knowp.
The function spec_is_self_object_constant is called from 7/stsp.
The function wo_of_CONSTANT_OBJECT_if_any is called from 5/aph, 5/mlc, 6/term, 6/asp, 7/tc, 8/refpt, 9/qty and 11/ap.

§**13.** A similar convenience for specifications which might or might not refer to property names:

```
property_name *spec_get_property_name_if_any(specification *spec) {
    if (spec_is_actual_CONSTANT_of_kova(spec, PROPERTY_TY))
        return PROPERTY_spec_to_property_name(spec);
    return NULL;
}
```

The function spec_get_property_name_if_any is called from 7/stsp, 7/tc, 9/provr and 12/cinv.

§**14.** And also for finding the integer value of a literal integer, something we surprisingly seldom need to do. Here there's no pointer attached to the specification, and indeed it doesn't even store the integer in question; it simply reparses the text when needed, since parsing a literal is quick.

```
int value_of_NUMBER_type(specification *spec) {
    if (spec->word_ref1 < 0)
        internal_error("Text no longer available for CONSTANT compilation");
    if (is_kova(is_a_literal(spec->word_ref1, spec->word_ref2),
        NUMBER_TY) == FALSE) {
        LOG("SP: $S\n", spec);
        internal_error("Type anomaly for CONSTANT compilation");
    }
    return last_literal_evaluated;
}
```

The function value_of_NUMBER_type is called from 10/eqns.

§**15.** Our most elaborate test is a finicky one, checking if two constant values are equal at compile time – which is needed when seeing how to mesh table continuations together, and in rare cases to help the typechecker. This doesn't need to be especially rapid.

```
define COMPARE_CONSTANTS(DATA, STRUCTURE)
    if (is_kova(kov1, DATA##_TY)) {
        STRUCTURE *x1 = DATA##_spec_to_##STRUCTURE(spec1);
        STRUCTURE *x2 = DATA##_spec_to_##STRUCTURE(spec2);
        if (x1 == x2) return TRUE;
        return FALSE;
    }

int spec_compare_CONSTANT(specification *spec1, specification *spec2) {
    if (spec_is_actual_CONSTANT(spec1) == FALSE) return FALSE;
    if (spec_is_actual_CONSTANT(spec2) == FALSE) return FALSE;
    kind_of_value *kov1 = spec_get_kind_of_value(spec1);
    kind_of_value *kov2 = spec_get_kind_of_value(spec2);
    if (kov_compare(kov1, kov2) == FALSE) return FALSE;
    switch (kov_get_constant_compilation_method(kov1)) {
        case LITERAL_CCM: {
            int v1, v2;
            is_a_literal(spec1->word_ref1, spec1->word_ref2);
            v1 = last_literal_evaluated;
            is_a_literal(spec2->word_ref1, spec2->word_ref2);
            v2 = last_literal_evaluated;
            if (v1 == v2) return TRUE;
            return FALSE;
        }
```

```
        case QUANTITATIVE_CCM: {
            quantity *q1, *q2;
            q1 = spec_get_constant_quantity_if_any(spec1);
            q2 = spec_get_constant_quantity_if_any(spec2);
            if (q1 == q2) return TRUE;
            return FALSE;
        }
        case SPECIAL_CCM: {
            COMPARE_CONSTANTS(ACTION_NAME, action_name)
            COMPARE_CONSTANTS(ACTIVITY, activity)
            COMPARE_CONSTANTS(EQUATION, equation)
            COMPARE_CONSTANTS(PROPERTY, property_name)
            COMPARE_CONSTANTS(RELATION, binary_predicate)
            COMPARE_CONSTANTS(RULE, booked_rule)
            COMPARE_CONSTANTS(RULEBOOK, rulebook)
            COMPARE_CONSTANTS(RULEBOOK_OUTCOME, named_rulebook_outcome)
            COMPARE_CONSTANTS(SCENE, scene)
            COMPARE_CONSTANTS(TABLE, table)
            COMPARE_CONSTANTS(TABLE_COLUMN, table_column)
            COMPARE_CONSTANTS(UNDERSTANDING, grammar_verb)
            COMPARE_CONSTANTS(USEOPTION, use_option)
            if (is_kova(kov1, OBJECT_TY)) {
                world_object *wo1 = OBJECT_spec_to_world_object(spec1);
                world_object *wo2 = OBJECT_spec_to_world_object(spec2);
                if (wo1 == wo2) return TRUE;
                return FALSE;
            }
        }
    }
    return FALSE;
}
```

The function spec_compare_CONSTANT is called from 7/tc and 10/tab.

§16. Pretty-printing.

```
void write_VALUE_out_in_English(specification *spec, char *text) {
    switch (spec_get_species(spec)) {
        case PHRASE_TO_DECIDE_VALUE_SPC: {
            kind_of_value *dtr = spec_evaluates_to(spec);
            sprintf(text+strlen(text), "an instruction to work out ");
            copy_kov_to_string(dtr, text, FALSE);
            break;
        }
        case CONSTANT_SPC: {
            int w1 = spec->word_ref1, w2 = spec->word_ref2;
            if (spec_is_generic_CONSTANT(spec)) {
                copy_kov_to_string(spec_get_kind_of_value(spec), text, FALSE);
                return;
            }
            if (spec_is_CONSTANT_of_kova(spec, PROPERTY_TY)) {
                if (w1 > 0) print_modest_sized_text_to_string(w1, w2, text+strlen(text));
                else sprintf(text+strlen(text), "the name of a property");
                return;
```

```
            }
            print_modest_sized_text_to_string(w1, w2, text+strlen(text));
            break;
        }
    }
}
```

The function write_VALUE_out_in_English is called from 7/spec.

§**17.** And for the log:

```
void log_VALUE_spec_details(specification *spec) {
    switch (spec_get_species(spec)) {
        case CONSTANT_SPC: {
            quantity *q = spec_get_constant_quantity_if_any(spec);
            if (q)
                if (qty_get_I6_representation(q))
                    LOG("(%s)", qty_get_I6_representation(q));
            if (spec_is_actual_CONSTANT_of_kova(spec, OBJECT_TY)) {
                if (spec_test_flag(spec, SELF_OBJECT_SPFLAG)) LOG("(-self-)");
                else if (spec_test_flag(spec, NOTHING_OBJECT_SPFLAG)) LOG("(-nothing-)");
                else LOG("($O)", OBJECT_spec_to_world_object(spec));
            }
        }
    }
}
```

The function log_VALUE_spec_details is called from 7/spec.

§**18. Compilation.** First: clearly everything in this family evaluates, but what KOV does the result have?

```
kind_of_value *kov_when_VALUE_is_evaluated(specification *spec) {
    if (spec == NULL) internal_error("kov_when_VALUE_is_evaluated on NULL");
    switch (spec_get_species(spec)) {
        case CONSTANT_SPC:
            if (spec_is_actual_CONSTANT_of_kova(spec, OBJECT_TY))
                ⟨Work out the KOV for a constant object 19⟩;
            return spec_get_kind_of_value(spec);
        case PHRASE_TO_DECIDE_VALUE_SPC:
            ⟨Work out the KOV returned by a phrase 20⟩;
    }
    internal_error("unknown evaluating VALUE type"); return NULL;
}
```

The function kov_when_VALUE_is_evaluated is called from 5/aph, 6/simp, 7/spec, 7/stsp, 7/tc, 8/mass and 12/phtd.

§**19.** This is trickier than it looks. What KOV shall we say that the constants "nothing", "Cobbled Crawl", or "animal" have? The answer is the narrowest we can: "nothing" comes out as "object", "Cobbled Crawl" as "room" (a KOVKO), "animal" as itself (ditto).

⟨Work out the KOV for a constant object 19⟩ ≡

```
if (spec_test_flag(spec, SELF_OBJECT_SPFLAG)) return kova(OBJECT_TY);
else if (spec_test_flag(spec, NOTHING_OBJECT_SPFLAG)) return kova(OBJECT_TY);
else {
    world_object *wo_if_any = OBJECT_spec_to_world_object(spec);
    if (wo_if_any->kind_flag) return kovko(wo_if_any);
    if (wo_if_any->kind) return kovko(wo_if_any->kind);
}
```

This code is used in §18.

§**20.** This too is tricky. Some phrases to decide values are unambiguous. If they say they are "To decide a rule: ...", then clearly the return value will be a `kova(RULE_TY)`. But others are "polymorphic" – Greek for many-shaped, but in this context, it means that the return value's kind depends on the kinds of its arguments; addition is like this, for instance.

Compounding the problem is that we don't actually know which phrase will be invoked – we only have a list of possibilities. All of them return values, but those may have different kinds, and some may be polymorphic.

So what are we to do? First, we find the "deciding invocation": the first entry in the list which has been passed by the type-checker, or if none of them has, then the first entry of all. If the deciding invocation is of a phrase with an unambiguous KOV, then that's of course the answer. If it is polymorphic, then we look to see if typechecking has already resolved the difficulty by showing the result, and if so, then that's the answer. The worst case, then, is when we have a polymorphic phrase and typechecking hasn't yet sorted matters out – in that event we return simply "value" as the KOV, an extremely weak if certainly true answer.

We resort to returning `NULL`, an unknown KOV, only when the invocation list is empty. This should never happen except possibly after recovering from some problem message.

⟨Work out the KOV returned by a phrase 20⟩ ≡

```
invocation *deciding_inv = NULL, *first_inv = NULL;
INVOCATION_VARIABLE(inv);
LOOP_THROUGH_INVOCATION_LIST(inv, spec_invocation_list(spec)) {
    if (first_inv == NULL) first_inv = inv;
    if (inv_test_flag(inv, PASSED_INVFLAG)) { deciding_inv = inv; break; }
}
if (deciding_inv == NULL) deciding_inv = first_inv;
if (deciding_inv) {
    phrase *ph = deciding_inv->phrase_invoked;
    if (ph) {
        switch(phtd_determine_return_modifier(&(ph->type_data))) {
            case NO_RETURN_RTY: return NULL;
            case POLYMORPHIC_RTY:
                if (deciding_inv->kov_resulting) return deciding_inv->kov_resulting;
                return kova(ANY_VALUE_TY);
            case STANDARD_RTY:
                return phtd_get_return_kov(&(ph->type_data));
        }
    }
}
return NULL;
```

This code is used in §18.

§**21.** And so to the code for compiling constants.

```
void compile_VALUE_spec(OUTPUT_STREAM, specification *spec_found) {
    switch(spec_get_species(spec_found)) {
        case PHRASE_TO_DECIDE_VALUE_SPC:
            compile_invocational_phrase(OUT, spec_found);
            return;
        case CONSTANT_SPC: {
            kind_of_value *kov_of_constant = spec_get_kind_of_value(spec_found);
            ⟨Worry about the case of block values stored on the heap 22⟩;
            int ccm = kov_get_constant_compilation_method(kov_of_constant);
            switch(ccm) {
                case NONE_CCM:                                  constant values of this kind cannot exist
                    LOG("SP: $S; kov: $u\n", spec_found, kov_of_constant);
                    internal_error("Tried to compile CONSTANT SP for a disallowed data type");
                    return;
                case LITERAL_CCM: ⟨Compile a literal-compilation-mode constant 23⟩; return;
                case QUANTITATIVE_CCM: ⟨Compile a quantitative-compilation-mode constant 24⟩; return;
                case SPECIAL_CCM: ⟨Compile a special-compilation-mode constant 25⟩; return;
            }
        }
    }
}
```

The function compile_VALUE_spec is called from 7/cfsp.

§**22.** Suppose we're asked to compile an indexed text. What we normally do is to compile the name of an I6 `Constant` such as `Blk_Const_17`, whose value will be a pointer to data which will be placed on the heap. We call `store_block_value` to remember what the actual text was, because we're going to need to compile it in the end, even if we're deferring that now. And when that day dawns, we will have the compilation mode `BLOCK_CONSTANT_CMODE` set, to avoid going into an infinite regression.

This applied to all block-value KOVs except for the KOVCONSs – at present, just the "list of..." constructor. This makes its own arrangements; see below.

⟨Worry about the case of block values stored on the heap 22⟩ ≡

```
    if ((kov_uses_pointer_values(kov_of_constant)) && (is_kovcon(kov_of_constant) == FALSE)) {
        if (TEST_COMPILATION_MODE(BLOCK_CONSTANT_CMODE) == FALSE) {
            WRITE("Blk_Const_%d", store_block_value(spec_copy(spec_found)));
            return;
        }
    }
```

This code is used in §21.

§**23.** With that out of the way, there are three basic compilation modes.

Here, the literal-parser is used to resolve the text of the SP to an integer. I6 is typeless, of course, so it doesn't matter that this is not necessarily a `NUMBER_TY`: all that matters is that the correct integer value is compiled.

⟨Compile a literal-compilation-mode constant 23⟩ ≡

```
if (spec_found->word_ref1 < 0) {
    LOG("SP: $S; kov: $u\n", spec_found, kov_of_constant);
    internal_error("Text no longer available for CONSTANT compilation");
}
if (kov_compare(is_a_literal(spec_found->word_ref1, spec_found->word_ref2),
    kov_of_constant) == FALSE) {
    LOG("SP: $S; kov: $u\n", spec_found, kov_of_constant);
    internal_error("Type anomaly for CONSTANT compilation");
}
WRITE("%d", last_literal_evaluated);
```

This code is used in §21.

§**24.** Whereas here, a quantity is attached.

⟨Compile a quantitative-compilation-mode constant 24⟩ ≡

```
if (is_kova(kov_of_constant, SCENE_TY)) {
    scene *sc = SCENE_spec_to_scene(spec_found);
    WRITE("%d", sc->allocation_id + 1);
    return;
}
quantity *q = RETRIEVE_FROM_SPEC(spec_found, quantity);
compile_quantity_name_to_string(OUT, q);
```

This code is used in §21.

§**25.** Otherwise there are just miscellaneous different things to do in different kinds of value:

⟨Compile a special-compilation-mode constant 25⟩ ≡

```
if (is_kova(kov_of_constant, ACTION_NAME_TY)) {
    action_name *an = ACTION_NAME_spec_to_action_name(spec_found);
    WRITE("##%s", an_get_I6_representation(an));
    return;
}
if (is_kova(kov_of_constant, ACTIVITY_TY)) {
    activity *act = ACTIVITY_spec_to_activity(spec_found);
    WRITE("%s", av_get_I6_representation(act));
    return;
}
if (is_kova(kov_of_constant, DESCRIPTION_OF_ACTION_TY)) {
    action_pattern *ap = RETRIEVE_FROM_SPEC(spec_found, action_pattern);
    compile_action_pattern_match(OUT, *ap, FALSE);
    return;
}
if (is_kova(kov_of_constant, EQUATION_TY)) {
    equation *eqn = EQUATION_spec_to_equation(spec_found);
    WRITE("%s", eqn_get_I6_representation(eqn));
    return;
}
```

```
if (is_kova(kov_of_constant, EXTERNALFILE_TY)) {
    kind_of_value *ial = is_a_literal(spec_found->word_ref1, spec_found->word_ref2);
    if (is_kova(ial, EXTERNALFILE_TY)) compile_exf_constant(OUT, last_literal_evaluated);
    return;
}
if ((is_kova(kov_of_constant, OBJECT_DESCRIPTION_TY)) ||
    (is_kova(kov_of_constant, VALUE_DESCRIPTION_TY))) {
    compile_multiple_use_proposition(OUT, spec_found);
    return;
}
if (is_kova(kov_of_constant, LIST_OF_TY)) {
    compile_literal_list(OUT, spec_found->word_ref1);
    return;
}
if (is_kova(kov_of_constant, OBJECT_TY)) {
    if (spec_test_flag(spec_found, SELF_OBJECT_SPFLAG)) WRITE("self");
    else if (spec_test_flag(spec_found, NOTHING_OBJECT_SPFLAG)) WRITE("nothing");
    else {
        world_object *wo = OBJECT_spec_to_world_object(spec_found);
        WRITE("%s", wo_get_I6_representation(wo));
    }
    return;
}
if (is_kova(kov_of_constant, PROPERTY_TY)) {
    ⟨Compile property constants 26⟩;
    return;
}
if (is_kova(kov_of_constant, QUOT_TY)) {
    if (spec_found->word_ref1 < 0)
        internal_error("Text no longer available for CONSTANT/QUOT");
    compile_quotation(OUT, spec_found->word_ref1);
    return;
}
if (is_kova(kov_of_constant, RELATION_TY)) {
    binary_predicate *bp = RELATION_spec_to_binary_predicate(spec_found);
    WRITE("%d", bp->allocation_id);
    return;
}
if (is_kova(kov_of_constant, RULE_TY)) {
    booked_rule *br = RULE_spec_to_booked_rule(spec_found);
    br_compile(OUT, br);
    return;
}
if (is_kova(kov_of_constant, RULEBOOK_TY)) {
    rulebook *rb = RULEBOOK_spec_to_rulebook(spec_found);
    WRITE("%d", rb->allocation_id);
    return;
}
if (is_kova(kov_of_constant, RULEBOOK_OUTCOME_TY)) {
    named_rulebook_outcome *rbno =
        RULEBOOK_OUTCOME_spec_to_named_rulebook_outcome(spec_found);
    WRITE("RBNO_%d", rbno->allocation_id);
    return;
```

```
    }
    if (is_kova(kov_of_constant, STORED_ACTION_TY)) {
        action_pattern *ap = RETRIEVE_FROM_SPEC(spec_found, action_pattern);
        compile_action_pattern_to_block(OUT, ap);
        return;
    }
    if (is_kova(kov_of_constant, TABLE_TY)) {
        table *t = TABLE_spec_to_table(spec_found);
        WRITE("%s", tab_get_I6_representation(t));
        return;
    }
    if (is_kova(kov_of_constant, TABLE_COLUMN_TY)) {
        table_column *tc = TABLE_COLUMN_spec_to_table_column(spec_found);
        WRITE("%d", tc_get_column_id(tc));
        return;
    }
    if (is_kova(kov_of_constant, TEXT_TY)) {
        if (spec_found->word_ref1 < 0)
            internal_error("Text no longer available for CONSTANT/TEXT");
        compile_literal_text(OUT, spec_found->word_ref1);
        return;
    }
    if (is_kova(kov_of_constant, TEXT_ROUTINE_TY)) {
        ph_stack_frame *phsf = NULL;
        if (spec_found->word_ref1 < 0)
            internal_error("Text no longer available for CONSTANT/TEXT_ROUTINE");
        if (TEST_COMPILATION_MODE(PERMIT_LOCALS_IN_TEXT_CMODE)) {
            WRITE("(");
            phsf = phsf_compile_local_storage(OUT, current_stack_frame());
        }
        WRITE("text_routine_%d", text_routine_cue(spec_found->word_ref1, phsf));
        if (TEST_COMPILATION_MODE(PERMIT_LOCALS_IN_TEXT_CMODE))
            WRITE(")");
        return;
    }
    if (is_kova(kov_of_constant, UNDERSTANDING_TY)) {
        if (spec_found->word_ref1 < 0)
            internal_error("Text no longer available for CONSTANT/UNDERSTANDING");
        compile_understanding(OUT,
            spec_found->word_ref1, spec_found->word_ref2, FALSE);
        return;
    }
    if (is_kova(kov_of_constant, USEOPTION_TY)) {
        use_option *uo = USEOPTION_spec_to_use_option(spec_found);
        WRITE("%d", uo->allocation_id);
        return;
    }
    LOG("Kov is $u\n", kov_of_constant);
    internal_error("no special ccm provided");
```

This code is used in §21.

§**26.** The interesting, read "unfortunate", case is that of constant property names. The curiosity here is that it's legal to store the nameless negation of an either/or property in a `PROPERTY_TY` constant. This is purely so that the following ungainly syntax works:

```
change X to not P;
```

Recall that in Inform 6 syntax, an attribute `attr` can be negated in sense in several contexts by using a tilde: `~attr`.

⟨Compile property constants 26⟩ ≡

```
int parity = 1;
if [[(spec_found) == not ***]] parity = -1;

property_name *prn = PROPERTY_spec_to_property_name(spec_found);
if (prn == NULL) internal_error("PROPERTY SP with null property");

if (prn_is_either_or(prn)) {
    if (prn_either_or_implemented_as_attribute(prn)) {
        if (prn_either_or_stored_in_negation(prn))
            WRITE("%s%s", (parity>0)?"~":"",
                prn_get_i6_identifier(prn_either_or_get_negation(prn)));
        else
            WRITE("%s%s", (parity<0)?"~":"",
                prn_get_i6_identifier(prn));
    } else {
        if (prn_either_or_stored_in_negation(prn))
            WRITE("%s",
                prn_get_i6_identifier(prn_either_or_get_negation(prn)));
        else WRITE("%s", prn_get_i6_identifier(prn));
    }
} else WRITE("%s", prn_get_i6_identifier(prn));
```

This code is used in §25.

§**27.** See above for the discussion of block-value constants. At present we always allocate 32 bytes for the initial block, whatever the KOV.

```
int store_block_value(specification *spec) {
    stored_block_value *sbv = CREATE(stored_block_value);
    sbv->stored_constant = spec;
    sbv->size_of_block = 32;
    return sbv->allocation_id;
}
```

§**28.** And this is where all of those block constants are redeemed. This is probably the winner of the all-Inform shouting function competition: all those macros, all those capitals.

```
void create_block_constants(OUTPUT_STREAM) {
    stored_block_value *sbv;
    LOOP_OVER (sbv, stored_block_value) {
        WRITE("Array Blk_Const_%d --> %d;\n",
            sbv->allocation_id, sbv->size_of_block);
    }
    WRITE("[ CreateBlockConstants;\n"); INDENT;
    BEGIN_COMPILATION_MODE;
    COMPILATION_MODE_ENTER(BLOCK_CONSTANT_CMODE);
    LOOP_OVER (sbv, stored_block_value) {
        WRITE("BlkValueInitialCopy(Blk_Const_%d, ", sbv->allocation_id);
        compile_VALUE_spec(OUT, sbv->stored_constant);
        WRITE(");\n");
    }
    END_COMPILATION_MODE;
    OUTDENT; WRITE("];\n");
}
```

The function create_block_constants is invoked by a command in a `.i6t` template file.

STORAGE Specifications 7/stsp

Purpose

Documentation on and functions handling specifications which fall into the STORAGE family.

§**1. Creation.** "Storage items" are what other programming languages might call "lvalues", since they can occur on the left of an assignment sign: they are values which can be written to. Values which can only be read ("rvalues") form the `VALUE` family. There are currently five `STORAGE` species:

`LOCAL_VARIABLE_SPC` refers to a specific local variable, so it has meaning only within the routine currently being compiled. In the compiled code, it will actually be an I6 local variable; in I7 terms it is a "let" variable. The data field of the SP holds the local variable number in the current routine. (These number upwards from 0, and are compiled to I6 local variables named in the form `t_0`, `t_1`, ...) There are no references or arguments.

`NONLOCAL_VARIABLE_SPC` refers to a variable of any other scope: that is, a global variable, or perhaps a rulebook, action or activity variable. The important distinction between these other scopes and local scope is essentially that local variables live on the I6 call-stack and have only a local namespace, whereas others correspond to array entries or global I6 variables and share a global namespace. (It is basically a matter of implementation convenience which makes us divide the stock of variables into two different species this way.) A `quantity *` pointer is attached, identifying the name of the variable in question. There are no arguments.

`PROPERTY_VALUE_SPC` represents a given (value-)property of a given object, *not* the name of a property in abstract. Thus "description of the Police Commissioner" qualifies, but "description" does not. There are two arguments: the property and the object which possesses it, respectively.

`TABLE_ENTRY_SPC` represents a given entry to a table, which can be referred to in several different ways. There are four different kinds of table reference, distinguished by the number of arguments found:

(1) 1 argument. By column name only, the table and row to be understood from context because we have selected a row in the surrounding source text.
(2) 2 arguments. Used as a condition to see if a value is listed in a given column of a given table. Argument 0 must be a constant of KOV `TABLE_COLUMN_TY`, argument 1 any value of KOV `TABLE_TY`. (Argument 0 has to be a constant because it is not type-safe to allow looping through columns, say: different columns have different KOVs, and the compiler would be unable to tell the KOV of the result of such a lookup. The same doesn't apply to argument 1, perhaps oddly, because Inform requires that every column name have the same KOV in every table using it. So the choice of table does not have to be a constant, and this allows for some interesting data structures to be built.)
(3) 3 arguments. An explicitly specified entry. The arguments are the table column, row number, and table respectively.
(4) 4 arguments. A reference to the X corresponding to a Y value of Z in table T. The arguments are X, Y, Z, T respectively.

`LIST_ENTRY_SPC` represents a given entry in a list, which is much simpler: there are two arguments, the list and the numerical index, which counts from 1.

Note that property names, table names, and lists themselves are not storage items as such – they are places where storage items are found. They are all in the `VALUE` family.

```
void create_STORAGE_species(void) {
    type_ID_set_description(STORAGE_FMY, "a stored value");
    type_ID_set_source_name(STORAGE_FMY, "STORAGE_FMY");
    type_ID_set_flag(STORAGE_FMY, EVALUATING_SPCFLAG);
    type_ID_permit_pair(STORAGE_FMY, LOCAL_VARIABLE_SPC,
        TYPE_PARSED_SPCONTEXT + PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT +
```

```
        TYPE_EXPECTED_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(LOCAL_VARIABLE_SPC, "a temporary 'let' or 'repeat' variable");
    type_ID_set_parsing_names(LOCAL_VARIABLE_SPC, variable_V, variables_V);
    type_ID_set_source_name(LOCAL_VARIABLE_SPC, "LOCAL_VARIABLE_SPC");
    type_ID_set_flag(LOCAL_VARIABLE_SPC, ASSOCIATED_KOV_SPCFLAG);
    type_ID_permit_pair(STORAGE_FMY, NONLOCAL_VARIABLE_SPC,
        TYPE_PARSED_SPCONTEXT + PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT +
        TYPE_EXPECTED_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(NONLOCAL_VARIABLE_SPC, "the name of a variable");
    type_ID_set_source_name(NONLOCAL_VARIABLE_SPC, "NONLOCAL_VARIABLE_SPC");
    type_ID_set_flag(NONLOCAL_VARIABLE_SPC, ASSOCIATED_KOV_SPCFLAG);
    type_ID_permit_pair(STORAGE_FMY, PROPERTY_VALUE_SPC,
        PARSED_SPCONTEXT + TYPE_PARSED_SPCONTEXT + TYPE_EXPECTED_SPCONTEXT +
        TYPE_FOUND_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(PROPERTY_VALUE_SPC, "a property whose value is ");
    type_ID_set_source_name(PROPERTY_VALUE_SPC, "PROPERTY_VALUE_SPC");
    type_ID_set_parsing_names(PROPERTY_VALUE_SPC, property_value_V, property_values_V);
    type_ID_set_flag(PROPERTY_VALUE_SPC, ARGUMENTS_SPCFLAG);
    type_ID_permit_pair(STORAGE_FMY, TABLE_ENTRY_SPC,
        TYPE_PARSED_SPCONTEXT + PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT +
        TYPE_EXPECTED_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(TABLE_ENTRY_SPC, "an entry in a table");
    type_ID_set_parsing_names(TABLE_ENTRY_SPC, table_reference_V, table_references_V);
    type_ID_set_source_name(TABLE_ENTRY_SPC, "TABLE_ENTRY_SPC");
    type_ID_set_flag(TABLE_ENTRY_SPC, ARGUMENTS_SPCFLAG);
    type_ID_permit_pair(STORAGE_FMY, LIST_ENTRY_SPC,
        TYPE_PARSED_SPCONTEXT + PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT +
        TYPE_EXPECTED_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(LIST_ENTRY_SPC, "an entry in a list");
    type_ID_set_parsing_names(LIST_ENTRY_SPC, list_entry_V, list_entries_V);
    type_ID_set_source_name(LIST_ENTRY_SPC, "LIST_ENTRY_SPC");
    type_ID_set_flag(LIST_ENTRY_SPC, ARGUMENTS_SPCFLAG);
}
```

The function create_STORAGE_species is called from 7/tids.

§**2.** And here are some convenient creators. Variables:

```
specification *new_LOCAL_VARIABLE_spec(int w1, int w2, int ix) {
    specification *spec = spec_new(STORAGE_FMY, LOCAL_VARIABLE_SPC);
    spec->word_ref1 = w1; spec->word_ref2 = w2;
    spec_set_data(spec, LOCAL_NUMBER_SPDATA, ix);
    return spec;
}
specification *new_generic_LOCAL_VARIABLE_type(kind_of_value *kov) {
    specification *spec = spec_new(STORAGE_FMY, LOCAL_VARIABLE_SPC);
    spec_make_generic(spec);
    spec_set_kind_of_value(spec, kov);
    return spec;
}
specification *new_actual_NONLOCAL_VARIABLE_type(quantity *q) {
    specification *spec = spec_new(STORAGE_FMY, NONLOCAL_VARIABLE_SPC);
```

```
    ATTACH_TO_SPEC(spec, quantity, q);
    spec->word_ref1 = q->word_ref1;
    spec->word_ref2 = q->word_ref2;
    return spec;
}
specification *new_generic_NONLOCAL_VARIABLE_type(kind_of_value *kov) {
    specification *spec = spec_new(STORAGE_FMY, NONLOCAL_VARIABLE_SPC);
    spec_make_generic(spec);
    spec_set_kind_of_value(spec, kov);
    return spec;
}
```

The function new_LOCAL_VARIABLE_spec is called from 5/aph, 5/tandv and 12/cinv.
The function new_generic_LOCAL_VARIABLE_type is called from 5/mlc.
The function new_actual_NONLOCAL_VARIABLE_type is called from 6/asp, 7/specv, 8/creat and 8/knowp.
The function new_generic_NONLOCAL_VARIABLE_type is called from 5/mlc.

§**3.** Table entries have their arguments filled in by the relevant routines in "Meaning List Conversion":

```
specification *new_TABLE_ENTRY_spec(void) {
    specification *spec = spec_new(STORAGE_FMY, TABLE_ENTRY_SPC);
    return spec;
}
```

The function new_TABLE_ENTRY_spec is called from 5/mlc.

§**4.** List entries:

```
specification *new_LIST_ENTRY_spec(specification *owner, specification *index) {
    specification *spec = spec_new(STORAGE_FMY, LIST_ENTRY_SPC);
    spec_set_argument(spec, 0, owner);
    spec_set_argument(spec, 1, index);
    return spec;
}
```

The function new_LIST_ENTRY_spec is called from 5/mlc.

§**5.** Property values are constructed out of what's often only implied text: for instance, "description" sometimes means "the description [of the `self` object]". We give them a word range which is minimal such that it must contain word ranges of both property and owner, if given. Thus "carrying capacity of the trunk" will result from "carrying capacity" and "trunk". This is not very scientific, perhaps, but it's done only to make problem messages more readable.

```
specification *new_PROPERTY_VALUE_spec(specification *prop, specification *owner) {
    specification *spec = spec_new(STORAGE_FMY, PROPERTY_VALUE_SPC);
    spec_set_argument(spec, 0, prop);
    spec_set_argument(spec, 1, owner);
    if ((prop->word_ref1 >= 0) && (owner->word_ref1 >= 0)) {
        spec->word_ref1 = prop->word_ref1;
        if (spec->word_ref1 > owner->word_ref1) spec->word_ref1 = owner->word_ref1;
        spec->word_ref2 = prop->word_ref2;
        if (spec->word_ref2 < owner->word_ref2) spec->word_ref2 = owner->word_ref2;
    }
    return spec;
}
```

The function new_PROPERTY_VALUE_spec is called from 5/mlc, 6/simp and 7/tc.

§6. Testing.

```
int spec_get_storage_form(specification *spec) {
    if (family_is(spec, STORAGE_FMY)) return spec_get_species(spec);
    return UNKNOWN;
}
```

The function spec_get_storage_form is called from 5/mlc, 6/simp, 6/equal, 6/defer, 6/cind, 7/tc, 8/mass, 9/qty, 11/ap, 12/phtd, 12/cinv and 12/ambig.

§7. More specifically:

```
int spec_is_actual_NONLOCAL_VARIABLE(specification *spec) {
    if ((species_is(spec, NONLOCAL_VARIABLE_SPC)) && (spec_is_actual(spec))) return TRUE;
    return FALSE;
}
int spec_is_generic_NONLOCAL_VARIABLE(specification *spec) {
    if ((species_is(spec, NONLOCAL_VARIABLE_SPC)) && (spec_is_generic(spec))) return TRUE;
    return FALSE;
}
```

The function spec_is_actual_NONLOCAL_VARIABLE is called from 6/equal and 8/mass.
The function spec_is_generic_NONLOCAL_VARIABLE is called from 8/creat, 8/mass and 9/pp.

§8. And more specifically still:

```
int spec_is_this_NONLOCAL_VARIABLE(specification *spec, quantity *q) {
    if ((spec_is_actual_NONLOCAL_VARIABLE(spec)) &&
        (q == (RETRIEVE_FROM_SPEC(spec, quantity)))) return TRUE;
    return FALSE;
}
```

The function spec_is_this_NONLOCAL_VARIABLE is called from 6/equal.

§9. Not all non-local variables are global – some have scope local to rulebooks, actions and the like:

```
int spec_is_global_variable(specification *spec) {
    if (spec_is_actual_NONLOCAL_VARIABLE(spec)) {
        quantity *q = RETRIEVE_FROM_SPEC(spec, quantity);
        if ((q) && (qty_is_a_global_variable(q))) return TRUE;
    }
    return FALSE;
}
```

The function spec_is_global_variable is called from 6/cind and 10/eqns.

§10. Pretty-printing.

```
void write_STORAGE_out_in_English(specification *spec, char *text) {
    switch(spec_get_species(spec)) {
        case LOCAL_VARIABLE_SPC:
            strcpy(text+strlen(text), "a temporary 'let' or 'repeat' variable");
            break;
        case NONLOCAL_VARIABLE_SPC:
            if (spec_get_kind_of_value(spec)) {
                copy_kov_to_string(spec_get_kind_of_value(spec), text, FALSE);
                strcpy(text+strlen(text), " that varies");
            } else strcpy(text+strlen(text), "a non-temporary variable");
            break;
        case TABLE_ENTRY_SPC:
            sprintf(text+strlen(text), "a table entry");
            break;
        case LIST_ENTRY_SPC:
            sprintf(text+strlen(text), "a list entry");
            break;
        case PROPERTY_VALUE_SPC:
            if ((spec_get_argc(spec) == 2) &&
                (spec_is_CONSTANT_of_kova(spec_get_argument(spec, 0), PROPERTY_TY))) {
                property_name *prn = spec_get_property_name_if_any(spec_get_argument(spec, 0));
                specification *pval = new_generic_CONSTANT_type(prn_get_kind_of_value(prn));
                sprintf(text+strlen(text), "a property whose value is ");
                write_specification_out_in_English(pval, text+strlen(text));
            } else sprintf(text+strlen(text), "a property belonging to something");
            break;
    }
}
```

The function write_STORAGE_out_in_English is called from 7/spec.

§11. And the log.

```
void log_STORAGE_spec_details(specification *spec) {
    switch (spec_get_species(spec)) {
        case LOCAL_VARIABLE_SPC:
            if (spec_get_data(spec, LOCAL_NUMBER_SPDATA) == -1) LOG("(existing)");
            else LOG("(t_%d;$u)",
                spec_get_data(spec, LOCAL_NUMBER_SPDATA),
                unproblematic_data_type_of_local(
                    spec_get_data(spec, LOCAL_NUMBER_SPDATA)));
            break;
        case NONLOCAL_VARIABLE_SPC:
            if (spec_is_actual(spec)) {
                quantity *q = RETRIEVE_FROM_SPEC(spec, quantity);
                if (qty_get_I6_representation(q))
                    LOG("(%s)", qty_get_I6_representation(q));
            }
            break;
    }
}
```

The function log_STORAGE_spec_details is called from 7/spec.

§12. Compilation.

```
kind_of_value *kov_when_STORAGE_is_evaluated(specification *spec) {
    if (spec == NULL) internal_error("kov_when_VALUE_is_evaluated on NULL");
    switch (spec_get_species(spec)) {
        case LOCAL_VARIABLE_SPC: ⟨Return the KOV of a local variable 13⟩;
        case NONLOCAL_VARIABLE_SPC: ⟨Return the KOV of a non-local variable 14⟩;
        case TABLE_ENTRY_SPC: ⟨Return the KOV of a table entry 15⟩;
        case LIST_ENTRY_SPC: ⟨Return the KOV of a list entry 16⟩;
        case PROPERTY_VALUE_SPC: ⟨Return the KOV of a property value 17⟩;
    }
    internal_error("unknown evaluating STORAGE species"); return NULL;
}
```

The function kov_when_STORAGE_is_evaluated is called from 7/spec.

§13.

⟨Return the KOV of a local variable 13⟩ ≡

```
    int k = spec_get_data(spec, LOCAL_NUMBER_SPDATA);
    if (k == -1) return kova(ANY_VALUE_TY);                    for "existing"
    return unproblematic_data_type_of_local(k);
```

This code is used in §12.

§14.

⟨Return the KOV of a non-local variable 14⟩ ≡

```
    if (spec_is_generic(spec)) return kova(ANY_VALUE_TY);
    quantity *q = RETRIEVE_FROM_SPEC(spec, quantity);
    return qty_kind_of_value(q);
```

This code is used in §12.

§15. In every form of table entry, argument 0 is the column, and the column is enough to determine the KOV:

⟨Return the KOV of a table entry 15⟩ ≡

```
    if (spec_get_argc(spec) > 0) {                  i.e., always, for actual table entry specifications
        specification *fts = spec_get_argument(spec, 0);
        table_column *tc = TABLE_COLUMN_spec_to_table_column(fts);
        return get_data_type_stored(tc);
    }
    return NULL;                                    can happen when scanning phrase arguments, which are generic
```

This code is used in §12.

§16.

⟨Return the KOV of a list entry 16⟩ ≡
```
if (spec_get_argc(spec) == 2) {                    i.e., always, for actual list entry specifications
    kind_of_value *kov1 = spec_evaluates_to(spec_get_argument(spec, 0));
    if (kovcon_get_base(kov1)) return kovcon_get_base(kov1);
    return kova(ANY_VALUE_TY);                     to help the type-checker produce better problem messages
}
return NULL;                                       can happen when scanning phrase arguments, which are generic
```

This code is used in §12.

§17.

⟨Return the KOV of a property value 17⟩ ≡
```
if (spec_get_argc(spec) == 2) {
    property_name *prn = spec_get_property_name_if_any(spec_get_argument(spec, 0));
    if ((prn) && (prn_is_either_or(prn) == FALSE)) return prn_get_kind_of_value(prn);
    return kova(ANY_VALUE_TY);                     to help the type-checker produce better problem messages
}
return NULL;                                       can happen when scanning phrase arguments, which are generic
```

This code is used in §12.

§18. **Rvalue compilation.** We finally reach the compilation routine which produces an I6 expression evaluating to the contents of the storage item specified.

```
void compile_STORAGE_spec(OUTPUT_STREAM, specification *spec_found) {
    switch (spec_get_species(spec_found)) {
        case LOCAL_VARIABLE_SPC: ⟨Compile a local variable specification 19⟩;
        case NONLOCAL_VARIABLE_SPC: ⟨Compile a non-local variable specification 20⟩;
        case PROPERTY_VALUE_SPC: ⟨Compile a property value specification 21⟩;
        case LIST_ENTRY_SPC: ⟨Compile a list entry specification 26⟩;
        case TABLE_ENTRY_SPC: ⟨Compile a table entry specification 27⟩;
        default: internal_error("unable to compile this STORAGE species");
    }
}
```

The function compile_STORAGE_spec is called from 7/cfsp.

§19.

⟨Compile a local variable specification 19⟩ ≡
```
if (spec_get_data(spec_found, LOCAL_NUMBER_SPDATA) == -1)
    internal_error("Compiled never-specified LOCAL VARIABLE SP");
WRITE("t_%d", spec_get_data(spec_found, LOCAL_NUMBER_SPDATA));
return;
```

This code is used in §18.

§**20.**

⟨Compile a non-local variable specification 20⟩ ≡

```
quantity *q = RETRIEVE_FROM_SPEC(spec_found, quantity);
compile_quantity_name_to_string(OUT, q);
return;
```

This code is used in §18.

§**21.**

⟨Compile a property value specification 21⟩ ≡

```
if (spec_get_argc(spec_found) != 2) internal_error("malformed PROPERTY_OF SP");
if (spec_get_argument(spec_found, 0) == NULL) internal_error("PROPERTY_OF with null arg 0");
if (spec_get_argument(spec_found, 1) == NULL) internal_error("PROPERTY_OF with null arg 1");
property_name *prn = spec_get_property_name_if_any(spec_get_argument(spec_found, 0));
if (prn == NULL) internal_error("PROPERTY_OF with null property");
specification *prop_spec = spec_get_argument(spec_found, 0);
specification *owner = spec_get_argument(spec_found, 1);
kind_of_value *owner_kov = spec_evaluates_to(owner);
⟨Reinterpret the "self" for what are unambiguously conditions of single things 22⟩;
if (is_kova(owner_kov, OBJECT_TY)) ⟨Compile a property value of an object 23⟩
else ⟨Compile a property value of another value 24⟩;
return;
```

This code is used in §18.

§**22.** When Inform reads a text with a substitution like so:

> if the signpost is visible, say "The signpost is still [signpost condition]."

...it has to decide which object is meant as the owner of the property "signpost condition". Ordinarily, missing property owners are the self object, which works nicely because `self` always has the right value at run-time when we're, e.g., printing names of things. But what if, as here, there is no formal indication of the owner? If we compile with the self object as owner, the code may fail at run-time, complaining about using a property of nothing.

The author who wrote the source text above, though, felt able to write "[signpost condition]" without any indication of its owner because there could only be one possible owner: the signpost. And so that's the convention we use here. We replace "self" as a default owner by the only possible owner.

⟨Reinterpret the "self" for what are unambiguously conditions of single things 22⟩ ≡

```
if (spec_is_self_object_constant(owner)) {
    world_object *wo = prn_condition_of_which_object(prn);
    if ((wo) && (wo->kind_flag == FALSE))
        owner = world_object_to_OBJECT_spec(wo);
}
```

This code is used in §21.

§**23.** For details of the routines called here, see "RTP.i6t" in the template.

⟨Compile a property value of an object 23⟩ ≡
```
WRITE("ValueProperty(");
⟨Compile the property's owner 25⟩;
WRITE(",");
spec_compile(OUT, prop_spec);
WRITE(")");
```
This code is used in §21.

§**24.**

⟨Compile a property value of another value 24⟩ ≡
```
int range = pp_get_prn_offset_in_kov(NULL, owner_kov);
int offset = pp_get_prn_offset_in_kov(prn, owner_kov);
if (offset < range) {
    WRITE("PropertyKOV(KOVP_%d, %d, ", kov_I6_ID(owner_kov), offset);
    ⟨Compile the property's owner 25⟩;
    WRITE(")");
}  else {
    WRITE("ValueProperty(");
    ⟨Compile the property's owner 25⟩;
    WRITE(",");
    spec_compile(OUT, prop_spec);
    WRITE(")");
}
```
This code is used in §21.

§**25.** During type-checking, a small number of `PROPERTY_VALUE_SPC` SPs are marked with the `RECORD_AS_SELF_SPFLAG` flag. Such a SP compiles not only to code performing the property lookup, but also setting the `self` I6 variable at run-time to the object whose property is being looked up. The point of this is to change the context used for implicit property lookups involved in the actual property: e.g., if the value of this property turns out to be text which contains a substitution referring vaguely to another property, then we need to make sure that this other property is looked up from the same object as produced the original text containing the substitution.

⟨Compile the property's owner 25⟩ ≡
```
if (spec_test_flag(spec_found, RECORD_AS_SELF_SPFLAG)) WRITE("self=");
spec_compile(OUT, owner);
```
This code is used in §23,24,23,24,23,24.

§**26.** List entries are blessedly simpler.

⟨Compile a list entry specification 26⟩ ≡
```
if (spec_get_argc(spec_found) != 2) internal_error("malformed LIST_OF SP");
if (spec_get_argument(spec_found, 0) == NULL) internal_error("LIST_OF with null arg 0");
if (spec_get_argument(spec_found, 1) == NULL) internal_error("LIST_OF with null arg 1");
WRITE("LIST_OF_TY_GetItem(");
spec_compile(OUT, spec_get_argument(spec_found, 0));
WRITE(",");
spec_compile(OUT, spec_get_argument(spec_found, 1));
WRITE(")");
return;
```

This code is used in §18.

§**27.** Table entries are simple too, but come in four variant forms:

⟨Compile a table entry specification 27⟩ ≡
```
switch(spec_get_argc(spec_found)) {
    case 1:
        we_need_ct_locals();
        WRITE("TableLookUpEntry(ct_0,");
        spec_compile(OUT, spec_get_argument(spec_found, 0));
        WRITE(",ct_1)");
        break;
    case 2:                                          never here except when printing debugging code
        WRITE("(false)");
        break;
    case 3:
        WRITE("TableLookUpEntry(");
        spec_compile(OUT, spec_get_argument(spec_found, 2));
        WRITE(",");
        spec_compile(OUT, spec_get_argument(spec_found, 0));
        WRITE(",");
        spec_compile(OUT, spec_get_argument(spec_found, 1));
        WRITE(")");
        break;
    case 4:
        WRITE("TableLookUpCorr(");
        spec_compile(OUT, spec_get_argument(spec_found, 3));
        WRITE(",");
        spec_compile(OUT, spec_get_argument(spec_found, 0));
        WRITE(",");
        spec_compile(OUT, spec_get_argument(spec_found, 1));
        WRITE(",");
        spec_compile(OUT, spec_get_argument(spec_found, 2));
        WRITE(")");
        break;
    default: internal_error("TABLE REFERENCE with bad number of args");
}
return;
```

This code is used in §18.

§**28. Lvalue compilation.** To recap, if an assignment takes the form "now X is Y" then X is the lvalue, Y is the rvalue. We only need to read Y, but we need to write X. `spec_compile` applied to a STORAGE item produces code suitable for reading it, i.e., suitable for use in position Y – but not in general suitable for X.

To compile the lvalue form of a storage item, we use the following schemas. These in effect take the rvalue form and modify it. There are three versions, according to the nature of the data being moved. In these schemas, `*1` expands to the storage item's rvalue form, and `*2` to the value being assigned to it.

```
define STORE_WORD_TO_WORD 1
define STORE_WORD_TO_POINTER 2
define STORE_POINTER_TO_POINTER 3

char *storage_class_schema(int storage_class, int kind_of_store) {
    switch(kind_of_store) {
        case STORE_WORD_TO_WORD:
            switch(storage_class) {
                case LOCAL_VARIABLE_SPC: return "*=-*1 = *2";
                case NONLOCAL_VARIABLE_SPC: return "*=-*1 = *2";
                case TABLE_ENTRY_SPC: return "*=-*1*-,1,*2)";
                case PROPERTY_VALUE_SPC: return "*=-Write*1*-,*2)";
                case LIST_ENTRY_SPC: return "*=-Write*1*-,*2)";
            }
            return "";
        case STORE_WORD_TO_POINTER:
            switch(storage_class) {
                case LOCAL_VARIABLE_SPC: return "*=-BlkValueCast(*1, *#1, *#2, *!2)";
                case NONLOCAL_VARIABLE_SPC: return "*=-BlkValueCast(*1, *#1, *#2, *!2)";
                case TABLE_ENTRY_SPC: return "*=-BlkValueCast(*1*-,5), *#1, *#2, *!2)";
                case PROPERTY_VALUE_SPC: return "*=-BlkValueCast(*1, *#1, *#2, *!2)";
                case LIST_ENTRY_SPC: return "*=-BlkValueCast(*1, *#1, *#2, *!2)";
            }
            return "";
        case STORE_POINTER_TO_POINTER:
            switch(storage_class) {
                case LOCAL_VARIABLE_SPC: return "*=-BlkValueCopy(*1, *2)";
                case NONLOCAL_VARIABLE_SPC: return "*=-BlkValueCopy(*1, *2)";
                case TABLE_ENTRY_SPC: return "*=-BlkValueCopy(*1*-,5), *2)";
                case PROPERTY_VALUE_SPC: return "*=-BlkValueCopy(*1, *2)";
                case LIST_ENTRY_SPC: return "*=-BlkValueCopy(*1, *2)";
            }
            return "";
    }
    return "";
}
```

The function storage_class_schema is called from 7/data.

CONDITION Specifications 7/cosp

Purpose

Documentation on and functions handling specifications which fall into the CONDITION family.

Definitions

¶1. One species of condition, `DESCRIPTION_SPC`, is complex enough to need its own packet of additional data, the "docket":

```
typedef struct description_docket {
    struct quantifier *quant;                                  such as "at most..." or "all"
    int quant_parameter;                                       such as 3 in "at most 3"
    int no_adjectives_applied;
    struct adjective_list_entry *adjectives_applied;           see below
    struct kind_of_value *specific_kov;                        everything of this KOV...
    struct world_object *specific_kind;                        ...or everything of this kind...
    struct world_object *specific_object;                      ...or just this specific object
    int calling_w1, calling_w2;                                "black box" if "...(called the black box)"
} description_docket;
```

The structure description_docket is private to this section.

¶2. An adjective list is a linked list of entry structures used with `DESCRIPTION_SPC` types, as follows:

```
typedef struct adjective_list_entry {
    adjectival_phrase *ref_to;
    int ref_positive;                                          one of the above forms
    struct adjective_list_entry *next;                         in the list in this docket
} adjective_list_entry;
```

The structure adjective_list_entry is private to this section.

---

§**1. Creation.** In Inform, conditions are not values, nor can values be used directly as conditions: we therefore need to provide the logical operations of AND, OR, and NOT structurally via the SP rather than implementing them as phrases like the arithmetic operators. (Conditions and values are kept separate even though it does complicate the type system because this provides cleaner resolution of ambiguities, and I don't repent of this, because the mixture of the two is sometimes unclear enough even in C: in natural language, it tends to look very odd indeed.)

So `LOGICAL_AND_SPC` and `LOGICAL_OR_SPC` imitate the effect of logical operators. They have two arguments which must themselves be CONDITIONs.

For a set of logical operators which is adequate in the Boolean algebra sense we also need negation, of course, but no `LOGICAL_NOT_SPC` type exists because instead we can simply mark a `CONDITION_FMY` type as negated by setting its `CONDITION_NEGATED_SPFLAG` flag. As a double-negative is a positive, a flag is all that is required.

§**2.** TEST_PROPOSITION_SPC and NOW_PROPOSITION_SPC each contain a predicate calculus sentence in the proposition fields of their SPs: there are no arguments.

When parsed, any such SP is always of TEST_PROPOSITION_SPC. It can only convert to the other form by successfully passing type-checking against a generic NOW_PROPOSITION_SPC token, such as the one which is the type of the argument of the phrase:

To now (P - now-condition): ...

Such checking can easily fail, because there are many conditions which can be tested but not asserted. (For instance, conditions in the past tense – you can ask whether it rained yesterday, but you can't demand that henceforth it didn't rain yesterday.)

§**3.** TEST_PHRASE_OPTION_SPC tests the use of a phrase option, and is the actual SP parsed for the second usage of the word "thoroughly" in the following example:

To prognosticate, swiftly or thoroughly: ...; if thoroughly, ...

It uses the SP's data field to refer to the option in question: its value is the bitmap value of the option, which will usually be 2^n where the option is the n-th in the list for this phrase, counting upwards from 0.

§**4.** TEST_ACTION_SPC and TEST_PAST_ACTION_SPC test either that the current action matches a given pattern, or that a given action has succeeded in the past, respectively. Each has an attached action_pattern * pointer.

§**5.** PHRASE_TO_DECIDE_IF_SPC is a usage of a phrase defined as "To decide if ...": as with other phrasal SPs, it has an attached invocation list.

§**6.** DESCRIPTION_SPC is the most complex species of all, and represents the most slippery idea: a noun phrase which describes certain values. It does not represent any given object or kind, even though it can in simple cases be a description so specific that it could only apply to a single object or kind, which may in practical terms be the same thing. We should probably regard a DESCRIPTION_SPC as defining a set of objects. But this is tricky for two reasons: first, the concept is broad enough to overlap in some cases with other types, which makes it ambiguous how we should parse certain text; and secondly, there are at least two different ways to manifest the idea of a set of objects in I6 at run-time, and we want to use both.

First, the parsing ambiguity. Given that a noun phrase consists of a noun as head-word together perhaps with 0 or more adjectives, and then with relative clauses added, there are clearly a wide range of possible outcomes. (And in fact in a few cases the head-word is itself optional: we can say "fixed in place scenery" even though this concatenates two either/or properties, which are in theory adjectival, though it doesn't look it here.) The various cases are:

(a) A single object unqualified by adjectives is always parsed to a constant value of KOV kova(OBJECT_TY), and so becomes a CONSTANT_SPC and not a DESCRIPTION_SPC. Thus, parsing treats the name of a specific object as a single value, not as a singleton set of that value.

(b) A single either/or property name is similarly parsed as a constant value of KOV kova(PROPERTY_TY).

(c) Otherwise, if there are no relative clauses attached, then the text is parsed as a DESCRIPTION_SPC with an attached docket, and the adjectives applied – which may be either/or properties, or adjectival enumerated values, or adjectival defined phrases – are accumulated in that docket.

(d) Otherwise, in the most complex cases, the text is parsed into a DESCRIPTION_SPC which has been "forced into propositional form", that is, we regard it as a set

$$\{x \mid P(x)\}$$

where P is a predicate calculus proposition with one free variable, and we record P in the proposition field of the SP.

We can convert format (c) to (d) easily, but vice versa is generally speaking impossible, and we never try. It might therefore seem easier to use format

(d) all of the time, but this would be a nuisance when trying to decide whether one description D_1 logically contains another, D_2 – something we need to do a lot of when sorting rules in specificity order. So in practice we prefer format (c) given the option.

Secondly, the compilation ambiguity. Used as a condition, a `DESCRIPTION_SPC` type compiles to a test of whether the currently understood object (the I6 value `self`, generally speaking) belongs to its set – i.e., matches the noun phrase – or not. But in order to make definitions like this one work, we also need to treat descriptions as values:

```
To decide which number is total (P - property) of (D - description):
```

The way this is done is that type-checking an actual `DESCRIPTION_SPC` type (say, "closed doors in lighted rooms") against the expectation of finding the data type `OBJECT_DESCRIPTION_TY` forces Inform to convert it to a constant of that type, compiling it as an iterator routine in I6 capable of (among other things) supplying the members in turn, and then using the address of this routine as the actual I6 value. (Constants with the KOV `OBJECT_DESCRIPTION_TY` cannot arise in any other way.)

```
void create_CONDITION_species(void) {
    type_ID_set_source_name(CONDITION_FMY, "CONDITION_FMY");
    type_ID_set_description(CONDITION_FMY, "yes or no");
    type_ID_set_parsing_names(CONDITION_FMY, condition_V, conditions_V);
    type_ID_permit_pair(CONDITION_FMY, UNKNOWN,
        TYPE_PARSED_SPCONTEXT + TYPE_EXPECTED_SPCONTEXT);
    type_ID_set_flag(CONDITION_FMY, TENSE_SPCFLAG);
    type_ID_permit_pair(CONDITION_FMY, LOGICAL_AND_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(LOGICAL_AND_SPC, "'and'");
    type_ID_set_source_name(LOGICAL_AND_SPC, "LOGICAL_AND_SPC");
    type_ID_set_flag(LOGICAL_AND_SPC, ARGUMENTS_SPCFLAG);
    type_ID_permit_pair(CONDITION_FMY, LOGICAL_OR_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(LOGICAL_OR_SPC, "'or'");
    type_ID_set_source_name(LOGICAL_OR_SPC, "LOGICAL_OR_SPC");
    type_ID_set_flag(LOGICAL_OR_SPC, ARGUMENTS_SPCFLAG);
    type_ID_permit_pair(CONDITION_FMY, TEST_PROPOSITION_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(TEST_PROPOSITION_SPC, "'is'");
    type_ID_set_source_name(TEST_PROPOSITION_SPC, "TEST_PROPOSITION_SPC");
    type_ID_permit_pair(CONDITION_FMY, NOW_PROPOSITION_SPC,
        PARSED_SPCONTEXT + TYPE_PARSED_SPCONTEXT + TYPE_EXPECTED_SPCONTEXT +
        TYPE_FOUND_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(NOW_PROPOSITION_SPC, "a sentence to make come true");
    type_ID_set_source_name(NOW_PROPOSITION_SPC, "NOW_PROPOSITION_SPC");
    type_ID_set_parsing_names(NOW_PROPOSITION_SPC, now_condition_V, now_conditions_V);
    type_ID_set_flag(CONDITION_FMY, PROPOSITION_SPCFLAG);
    type_ID_permit_pair(CONDITION_FMY, TEST_PHRASE_OPTION_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(TEST_PHRASE_OPTION_SPC, "an optional way to use the current phrase");
    type_ID_set_source_name(TEST_PHRASE_OPTION_SPC, "TEST_PHRASE_OPTION_SPC");
    type_ID_permit_pair(CONDITION_FMY, TEST_ACTION_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + TYPE_EXPECTED_SPCONTEXT + COMPILED_SPCONTEXT);
```

```
    type_ID_set_description(TEST_ACTION_SPC, "an action used as a condition");
    type_ID_set_source_name(TEST_ACTION_SPC, "TEST_ACTION_SPC");
    type_ID_permit_pair(CONDITION_FMY, TEST_PAST_ACTION_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + TYPE_EXPECTED_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(TEST_PAST_ACTION_SPC, "a past action used as a condition");
    type_ID_set_source_name(TEST_PAST_ACTION_SPC, "TEST_PAST_ACTION_SPC");
    type_ID_permit_pair(CONDITION_FMY, PHRASE_TO_DECIDE_IF_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + TYPE_EXPECTED_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(PHRASE_TO_DECIDE_IF_SPC, "a phrase to make a decision");
    type_ID_set_source_name(PHRASE_TO_DECIDE_IF_SPC, "PHRASE_TO_DECIDE_IF_SPC");
    type_ID_set_flag(PHRASE_TO_DECIDE_IF_SPC, PHRASAL_SPCFLAG + INVLIST_SPCFLAG);
    type_ID_permit_pair(CONDITION_FMY, DESCRIPTION_SPC,
        TYPE_PARSED_SPCONTEXT + PARSED_SPCONTEXT + TYPE_EXPECTED_SPCONTEXT +
        TYPE_FOUND_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(DESCRIPTION_SPC, "a description of an object");
    type_ID_set_source_name(DESCRIPTION_SPC, "DESCRIPTION_SPC");
    type_ID_set_flag(CONDITION_FMY, PROPOSITION_SPCFLAG + DOCKET_SPCFLAG);
}
```

The function create_CONDITION_species is called from 7/tids.

§**7.** Now some creator routines. `new_CONDITION_spec` is actually used only as a way to park time periods during meaning list construction; it doesn't generate SPs which persist later on.

```
specification *new_CONDITION_spec(time_period *tp) {
    specification *spec = spec_new(CONDITION_FMY, UNKNOWN);
    spec_set_condition_tense(spec, tp);
    return spec;
}
```

The function new_CONDITION_spec is called from 5/ml.

§**8.** Whereas this is used for type-checking:

```
specification *new_generic_CONDITION_type(void) {
    specification *spec = new_CONDITION_spec(NULL);
    spec_make_generic(spec);
    return spec;
}
```

The function new_generic_CONDITION_type is called from 7/tc, 9/scene and 11/ap.

§**9.** So, some more specific creators. The operators are easy:

```
specification *new_LOGICAL_AND_spec(specification *spec1, specification *spec2) {
    specification *spec = spec_new(CONDITION_FMY, LOGICAL_AND_SPC);
    spec_set_argument(spec, 0, spec1);
    spec_set_argument(spec, 1, spec2);
    return spec;
}
specification *new_LOGICAL_OR_spec(specification *spec1, specification *spec2) {
    specification *spec = spec_new(CONDITION_FMY, LOGICAL_OR_SPC);
    spec_set_argument(spec, 0, spec1);
    spec_set_argument(spec, 1, spec2);
    return spec;
}
```

The function new_LOGICAL_AND_spec is called from 5/mlc.
The function new_LOGICAL_OR_spec is called from 5/mlc.

§**10.** Testing propositions is also straightforward. There's no creator for `NOW_PREPOSITION_SPC`, since this is formed only by coercion of one of these.

```
specification *new_TEST_PROPOSITION_spec(pcalc_prop *prop) {
    specification *spec = spec_new(CONDITION_FMY, TEST_PROPOSITION_SPC);
    spec_set_proposition(spec, prop);
    return spec;
}
```

The function new_TEST_PROPOSITION_spec is called from 5/mlc.

§**11.** The option number here is actually 2^i, where $0 \leq i < 15$. This is just exactly feasible, since the specification data field is 16 bits wide.

```
specification *new_TEST_PHRASE_OPTION_spec(int opt_num) {
    specification *spec = spec_new(CONDITION_FMY, TEST_PHRASE_OPTION_SPC);
    spec_set_data(spec, PHRASE_OPTION_SPDATA, opt_num);
    return spec;
}
```

The function new_TEST_PHRASE_OPTION_spec is called from 5/ml.

§**12.** Past and present action tests have the pattern as an attachment:

```
specification *new_TEST_ACTION_spec(action_pattern *ap) {
    specification *spec = spec_new(CONDITION_FMY, TEST_ACTION_SPC);
    if (ap) ATTACH_TO_SPEC(spec, action_pattern, ap);
    return spec;
}
specification *new_TEST_PAST_ACTION_spec(action_pattern *ap) {
    specification *spec = spec_new(CONDITION_FMY, TEST_PAST_ACTION_SPC);
    if (ap) ATTACH_TO_SPEC(spec, action_pattern, ap);
    return spec;
}
```

The function new_TEST_ACTION_spec is called from 5/ml, 5/tandv and 12/phtd.
The function new_TEST_PAST_ACTION_spec is called from 5/ml.

§**13.** All phrasal specifications need to start with an attached empty invocation list, and this is no exception:

```
specification *new_PHRASE_TO_DECIDE_IF_spec(void) {
    specification *spec = spec_new(CONDITION_FMY, PHRASE_TO_DECIDE_IF_SPC);
    spec_create_invocation_list(spec);
    return spec;
}
```

The function new_PHRASE_TO_DECIDE_IF_spec is called from 5/mlc.

§**14.** And that leaves descriptions. Recall that these *either* have a docket *or* a proposition, with just one exception: a description meaning "any object at all" is allowed to have neither. This is because the empty proposition is universally true (see Chapter 6).

It follows that the bare result of calling the following routine is indeed a legal description, but not a very useful one. What generally happens is that the caller subsequently tacks on a docket or a preposition to narrow it down.

```
specification *new_DESCRIPTION_spec(void) {
    specification *spec = spec_new(CONDITION_FMY, DESCRIPTION_SPC);
    return spec;
}
```

The function new_DESCRIPTION_spec is called from 5/mlc and 11/ap.

§**15.** Since, in principle, any condition might also have a time period attached to it, we need a follow-up routine to attach this as necessary to a newly created condition:

```
void spec_attach_tense(specification *spec, int t) {
    if (spec_get_condition_tense(spec) == NULL)
        spec_set_condition_tense(spec, store_tp(new_time_period()));
    tp_set_tense(spec_get_condition_tense(spec), t);
}
```

The function spec_attach_tense is called from 5/mlc.

§**16. Coercions.** Recall that the following is the only way `NOW_PROPOSITION_SPC` can come into being:

```
void coerce_TEST_PROPOSITION_to_NOW_PROPOSITION(specification *spec) {
    if (spec == NULL) internal_error("can't coerce null type");
    if (species_is(spec, TEST_PROPOSITION_SPC))
        spec_coerce_to(spec, CONDITION_FMY, NOW_PROPOSITION_SPC);
    else internal_error("tried coerce_TEST_PROPOSITION_to_NOW_PROPOSITION inapplicably");
}
```

The function coerce_TEST_PROPOSITION_to_NOW_PROPOSITION is called from 7/tc.

§**17.** In general we represent "a number" as a generic constant value with KOV "number", but once we want to apply an adjective to it, say "even", we make something complex enough to become a description. So the following routine makes the DESCRIPTION_SPC specification, all ready for "even" to be added to its docket.

```
void coerce_generic_CONSTANT_to_DESCRIPTION(specification *spec) {
    if (spec == NULL) internal_error("can't coerce null type");
    if (spec_is_generic_CONSTANT(spec)) {
        kind_of_value *kov = spec_get_kind_of_value(spec);
        spec_coerce_to(spec, CONDITION_FMY, DESCRIPTION_SPC);
        spec_set_described_kov(spec, kov);
        /* empty_adjective_list(spec);
        spec_set_proposition(spec, NULL); */
        spec_make_actual(spec);
        return;
    }
    internal_error("tried coerce_generic_CONSTANT_to_DESCRIPTION inapplicably");
}
```

The function coerce_generic_CONSTANT_to_DESCRIPTION is called from 7/tc.

§**18. Testing.** A tricky test of a DESCRIPTION: does this consist *exactly* of "all objects of a given kind"? Qualification by adjective or proposition is not permitted.

```
int spec_is_DESCRIPTION_of_kind(specification *spec) {
    if (species_is(spec, DESCRIPTION_SPC) == FALSE) return FALSE;
    if (spec_is_qualified_DESCRIPTION(spec)) return FALSE;
    if (spec_get_described_object(spec)) return FALSE;
    return TRUE;
}
```

The function spec_is_DESCRIPTION_of_kind is called from 12/phtd.

§**19.** DESCRIPTIONs are "qualified" when adjectives, or relative clauses, make them dependent on context. For instance, "a vehicle" is unqualified, but "an even number" or "a vehicle in Trafalgar Square" are qualified – a given value V might satisfy the description at some times but not others during play.

```
int spec_is_qualified_DESCRIPTION(specification *spec) {
    if ((species_is(spec, DESCRIPTION_SPC)) &&
        ((spec_get_proposition(spec)) || (number_of_adjectives_applied_to(spec) > 0)))
        return TRUE;
    return FALSE;
}
```

The function spec_is_qualified_DESCRIPTION is called from 5/aph, 7/tc and 13/tfg.

§**20.** Given a DESCRIPTION specification, constrain it further by making it belong to the new domain supplied, which might be a kind or a specific object. If this is impossible because its existing domain is incompatible with its new one, cancel the specification out to UNKNOWN.

```
void restrict_DESCRIPTION_domain(specification *spec, world_object *new_domain) {
    if (new_domain == NULL) return;
    world_object *current_domain = spec_get_described_object(spec);
    if (current_domain == NULL) current_domain = spec_get_described_kind(spec);
    if (current_domain == NULL) current_domain = new_domain;
    else if ((current_domain == new_domain) || (wo_of_kind(current_domain, new_domain)))
        ;
    else if (wo_of_kind(new_domain, current_domain))
        current_domain = new_domain;
    else {
        if (new_domain != kind_room) coerce_spec_to_UNKNOWN(spec);
        return;
    }
    spec_set_described_object(spec, NULL);
    spec_set_described_kind(spec, NULL);
    if (current_domain) {
        if (current_domain->kind_flag) spec_set_described_kind(spec, current_domain);
        else spec_set_described_object(spec, current_domain);
    }
}
```

The function restrict_DESCRIPTION_domain is called from 5/mlc.

§21. Pretty-printing.

```
void write_CONDITION_out_in_English(specification *spec, char *text) {
    switch (spec_get_species(spec)) {
        case PHRASE_TO_DECIDE_IF_SPC:
            sprintf(text+strlen(text), "a yes or no question");
            break;
        case DESCRIPTION_SPC:
            sprintf(text+strlen(text), "an object matching a description");
            break;
        default:
            sprintf(text+strlen(text), "a condition");
    }
}
```

The function write_CONDITION_out_in_English is called from 7/spec.

§**22.** And the log:

```
void log_CONDITION_spec_details(specification *spec) {
    if (spec_test_flag(spec, CONDITION_NEGATED_SPFLAG)) LOG("(negated)");
    if (spec_get_condition_tense(spec)) log_time_period(spec_get_condition_tense(spec));
    if (species_is(spec, DESCRIPTION_SPC)) {
        adjective_list_entry *ale;
        int i = 0;
        LOOP_THROUGH_ADJECTIVE_LIST(ale, spec) {
            if (i == 0) LOG("(adjectives:"); else LOG(",");
            LOG("$r", ale);
            i++;
        }
        if (i > 0) LOG(")");
        if (spec_get_described_object(spec)) LOG("(object $O)", spec_get_described_object(spec));
        if (spec_get_described_kind(spec)) LOG("(kind $O)", spec_get_described_kind(spec));
        if (spec_get_described_kov(spec)) LOG("(KOV $u)", spec_get_described_kov(spec));
        if (spec_get_proposition(spec)) LOG("(such that $D)", spec_get_proposition(spec));
    }
}
```

The function log_CONDITION_spec_details is called from 7/spec.

§**23. Comparing.** Descriptions are used to set out the domain of rules, and rules have to be sorted in order of narrowness, which means we need a way to tell which is narrower of two descriptions.

```
int compare_specificity_of_DESCRIPTIONs(specification *spec1, specification *spec2) {
    ⟨If either of these two descriptions has a proposition, then propositional size decides it 24⟩;
    ⟨If the typechecker recognises one as properly within the other, then that's the more specific 25⟩;
    ⟨If either of these two descriptions have adjectives in their dockets, adjective count decides it 26⟩;
    return 0;
}
```

The function compare_specificity_of_DESCRIPTIONs is called from 7/spec.

§**24.** We have to be careful here because some descriptions use a docket, some a proposition, to represent the same conceptual information. However, since propositional form is only used when docket form can't hold so complex a condition, we will follow the rule that *a proposition beats a docket*. Thus any noun phrase containing a relative clause is deemed more specific than a noun phrase without.

That leaves us deciding which of two propositions is more specific. To perform that test perfectly, we would need a way to determine, given two proposition A and B, whether one implied the other or not. Since predicate calculus is complete, and our domains are mostly finite, there do in fact exist (slow and difficult) algorithms which could determine this. But there would be real problems with larger domains not amenable to model-checking, such as "number", and anyway – why not cheat?

Proposition length is admittedly a crude measure, but because of the fact that both propositions have come from the same generative algorithm, it does indeed turn out that $A \Rightarrow B$ means $\ell(A) \geq \ell(B)$, where ℓ is the number of atoms in the proposition. So we simply use it as a sorting key.

⟨If either of these two descriptions has a proposition, then propositional size decides it 24⟩ ≡

```
    int len1 = prop_length(spec_get_proposition(spec1));
    int len2 = prop_length(spec_get_proposition(spec2));
    if ((len1 > 0) || (len2 > 0)) {
        LOGIF(SPECIFICITIES,
            "Test %d: Comparing specificity of props:\n(%d) $D\n(%d) $D\n",
```

```
            cco, len1, spec_get_proposition(spec1), len2, spec_get_proposition(spec2));
    }
    if (len1 > len2) return 1;
    if (len1 < len2) return -1;
```

This code is used in §23.

§**25.**

⟨If the typechecker recognises one as properly within the other, then that's the more specific 25⟩ ≡

```
    if (can_we_match_value_descriptions(spec1, spec2) == ALWAYS_MATCH) {
        if (can_we_match_value_descriptions(spec2, spec1) != ALWAYS_MATCH) return 1;
    } else {
        if (can_we_match_value_descriptions(spec2, spec1) == ALWAYS_MATCH) return -1;
    }
```

This code is used in §23.

§**26.**

⟨If either of these two descriptions have adjectives in their dockets, adjective count decides it 26⟩ ≡

```
    int count1 = number_of_adjectives_applied_to(spec1);
    int count2 = number_of_adjectives_applied_to(spec2);
    if (count1 > count2) return 1;
    if (count1 < count2) return -1;
```

This code is used in §23.

§**27.** We also need a way of determining which of two conditions is more complex, so that action-based rules with "when..." clauses tacked on can be sorted: the following is used to compare such "when..." conditions. Again it is essentially a counting argument. Long conditions, with many clauses, beat shorter ones.

```
int compare_specificity_of_CONDITIONs(specification *spec1, specification *spec2) {
    int count1 = count_CONDITION_coverage(spec1);
    int count2 = count_CONDITION_coverage(spec2);
    if (count1 > count2) return 1;
    if (count1 < count2) return -1;
    return 0;
}
```

The function compare_specificity_of_CONDITIONs is called from 11/ap.

§**28.** The only justification for the following complexity score is that it does seem to accord well with what people expect. (There's clearly no theoretically perfect way to define complexity of conditions in a language as complex as Inform; this bit of scruffy, rather than neat, logic will have to do.)

```
int count_CONDITION_coverage(specification *spec) {
    if (spec == NULL) return 0;
    if (family_is(spec, CONDITION_FMY) == FALSE) return 1;
    switch (spec_get_species(spec)) {
        case LOGICAL_AND_SPC:
            return count_CONDITION_coverage(spec_get_argument(spec, 0))
                + count_CONDITION_coverage(spec_get_argument(spec, 1));
        case LOGICAL_OR_SPC:
            return -1;
        case TEST_PROPOSITION_SPC:
            return prop_length(spec_get_proposition(spec));
        case PHRASE_TO_DECIDE_IF_SPC:
        case TEST_PHRASE_OPTION_SPC:
        case TEST_ACTION_SPC:
        case TEST_PAST_ACTION_SPC:
            return 1;
    }
    return 0;
}
```

§**29. Compiling.** Conditions of course cannot be evaluated, so:

```
kind_of_value *kov_when_CONDITION_is_evaluated(specification *spec) {
    return NULL;
}
```

The function kov_when_CONDITION_is_evaluated is called from 7/spec.

§**30.** We clear two oddball cases out of the way, and then for the most part we delegate to potent routines elsewhere. Note that in some situations a valid I6 condition must be enclosed in round brackets, and that a redundant pair of brackets never does any harm; so we always compile one.

```
void compile_CONDITION_spec(OUTPUT_STREAM, specification *spec_found) {
    if (species_is(spec_found, NOW_PROPOSITION_SPC))
        ⟨Compile the not-really-conditional "now" propositions 31⟩;
    if (spec_get_condition_tense(spec_found))
        ⟨The use of a past tense pre-empts condition compilation 32⟩;
    WRITE("(");
    if (spec_test_flag(spec_found, CONDITION_NEGATED_SPFLAG)) WRITE("~~(");
    switch (spec_get_species(spec_found)) {
        case DESCRIPTION_SPC: compile_test_of_subst__v(OUT, spec_found); break;
        case TEST_PROPOSITION_SPC:
            compile_test_of_proposition(OUT, NULL, spec_get_proposition(spec_found));
            break;
        case LOGICAL_AND_SPC: case LOGICAL_OR_SPC: ⟨Compile a logical operator 33⟩; break;
        case TEST_ACTION_SPC:
            compile_action_pattern_match(OUT,
                *(RETRIEVE_FROM_SPEC(spec_found, action_pattern)), FALSE);
            break;
```

```
        case TEST_PAST_ACTION_SPC:
            compile_past_tense_action(OUT, RETRIEVE_FROM_SPEC(spec_found, action_pattern));
            break;
        case TEST_PHRASE_OPTION_SPC: ⟨Compile a phrase option test 34⟩; break;
        case PHRASE_TO_DECIDE_IF_SPC:
            compile_invocational_phrase(OUT, spec_found);
            break;
    }
    if (spec_test_flag(spec_found, CONDITION_NEGATED_SPFLAG)) WRITE(")");
    WRITE(")");
}
```

The function compile_CONDITION_spec is called from 7/cfsp.

§**31.** In fact, "now" propositions are never empty, but there's nothing in principle wrong with asserting that the universally true proposition is henceforth to be true, so we simply compile empty code in that case.

⟨Compile the not-really-conditional "now" propositions 31⟩ ≡

```
    if (spec_get_proposition(spec_found))
        compile_now_proposition(OUT, spec_get_proposition(spec_found));
    return;
```

This code is used in §30.

§**32.** A condition SP has a valid `condition_tense` time period field only if the tense in question is not the present. What happens is basically that we defer evaluating the condition to another part of the I6 source text, and here write something different. Tenses therefore pre-empt the rest of condition compilation.

⟨The use of a past tense pre-empts condition compilation 32⟩ ≡

```
    compile_past_tense_condition(OUT, *(spec_get_condition_tense(spec_found)),
        spec_copy(spec_found));
    return;
```

This code is used in §30.

§**33.** An easy case, running straight out to I6 operators:

⟨Compile a logical operator 33⟩ ≡

```
    if (spec_get_argc(spec_found) != 2)
        internal_error("Compiled malformed CONDITION/AND SP");
    specification *left_operand = spec_get_argument(spec_found, 0);
    specification *right_operand = spec_get_argument(spec_found, 1);
    if ((left_operand == NULL) || (right_operand == NULL))
        internal_error("Compiled CONDITION/AND with LHS operands");
    spec_compile(OUT, left_operand);
    if (species_is(spec_found, LOGICAL_AND_SPC)) WRITE(" && ");
    if (species_is(spec_found, LOGICAL_OR_SPC)) WRITE(" || ");
    spec_compile(OUT, right_operand);
```

This code is used in §30.

§**34.** Phrase options are stored as bits in a 16-bit map, so that each individual option is a power of two from 2^0 to 2^15. We test if this is valid by performing logical-and against the I6 local variable `phrase_options`, which exists if and only if the enclosing I6 routine takes phrase options. The type-checker won't allow these specifications to be compiled anywhere else.

⟨Compile a phrase option test 34⟩ ≡

```
WRITE("phrase_options & %d", spec_get_data(spec_found, PHRASE_OPTION_SPDATA));
```

This code is used in §30.

§**35. Dockets attached to DESCRIPTION types.** A docket takes a modest amount of storage, so is allocated only when needed; of course most specifications are not DESCRIPTIONs, and will certainly never need one. The docket is created only when there's a need to store some non-trivial data in one.

```
void spec_clear_docket(specification *spec) {
    empty_adjective_list(spec);
    spec_set_described_kind(spec, NULL);
    spec_set_described_kov(spec, NULL);
    spec_set_described_object(spec, NULL);
    description_docket *dd = spec_get_docket(spec);
    dd->calling_w1 = -1;
    dd->calling_w2 = -1;
}
description_docket *spec_find_docket(specification *spec, int create_if_necessary) {
    if (spec == NULL) {
        if (create_if_necessary) internal_error("creating docket for null type");
        return NULL;
    }
    description_docket *dd = spec_get_docket(spec);
    if (dd == NULL) {
        if (create_if_necessary == FALSE) return NULL;
        dd = CREATE(description_docket);
        spec_set_docket(spec, dd);
        spec_clear_docket(spec);
        dd->quant = NULL;
        dd->quant_parameter = 0;
    }
    return dd;
}
```

§**36.** Because a specification structure contains a pointer to a docket, copying the specification bitwise leaves two different specifications pointing to the same docket. That very rarely matters, but in cases where it might, we must copy thus:

```
specification *spec_copy_and_copy_docket(specification *spec) {
    if (spec == NULL) return NULL;
    specification *nts = spec_copy(spec);
    if (spec_get_docket(spec)) {
        description_docket *ndd = CREATE(description_docket);
        spec_set_docket(nts, ndd);
        *ndd = *(spec_get_docket(spec));
    }
    return nts;
}
```

The function spec_copy_and_copy_docket is called from 11/los.

§**37.** The crucial moment in the life of any interesting DESCRIPTION comes when its docket is converted to a proposition – as it must be in order for it to be asserted true about the model world, or to be compiled in a "now".

```
void spec_convert_docket_to_proposition(specification *spec) {
    pcalc_prop *prop = prop_from_spec(spec, FALSE);
    spec_set_proposition(spec, prop);
    spec_clear_docket(spec);
}
```

The function spec_convert_docket_to_proposition is called from 6/defer, 8/refpt, 11/ap and 12/cinv.

§**38.** And now, many dull but straightforward routines for handling the docket's contents.

```
int number_of_adjectives_applied_to(specification *spec) {
    description_docket *dd = spec_find_docket(spec, FALSE);
    if (dd == NULL) return 0;
    return dd->no_adjectives_applied;
}
adjective_list_entry *first_adjective_list_entry(specification *spec) {
    description_docket *dd = spec_find_docket(spec, FALSE);
    if ((dd == NULL) || (dd->no_adjectives_applied == 0))
        internal_error("adjective list has no entries");
    return dd->adjectives_applied;
}
void empty_adjective_list(specification *spec) {
    description_docket *dd = spec_find_docket(spec, FALSE);
    if (dd) {
        dd->adjectives_applied = NULL;
        dd->no_adjectives_applied = 0;
    }
}
void add_to_adjective_list(adjective_list_entry *ale, specification *spec) {
    description_docket *dd = spec_find_docket(spec, TRUE);
    dd->no_adjectives_applied++;
    ale->next = NULL;
```

```
    adjective_list_entry *last = dd->adjectives_applied;
    if (last == NULL) { dd->adjectives_applied = ale; return; }
    while (last->next) last = last->next;
    last->next = ale;
}

int types_have_same_adjective_list(specification *spec1, specification *spec2) {
    description_docket *dd1 = spec_find_docket(spec1, FALSE);
    description_docket *dd2 = spec_find_docket(spec2, FALSE);
    if ((dd1 == NULL) && (dd2 == NULL)) return TRUE;
    if ((dd1) && (dd2) && (dd1->adjectives_applied == dd2->adjectives_applied)) return TRUE;
    return FALSE;
}
```

The function number_of_adjectives_applied_to is called from 5/mlc, 7/tc, 8/refpt, 8/assem, 8/knowp, 9/pp, 10/tab, 11/act, 11/av, 12/phtd, 12/rb and 13/tfg.
The function first_adjective_list_entry is called from 7/tc, 8/refpt, 10/tab and 13/tfg.
The function add_to_adjective_list is called from 5/mlc.

§39. Quantification.

```
void spec_attach_quantifier(specification *spec, quantifier *q, int par) {
    description_docket *dd = spec_find_docket(spec, TRUE);
    dd->quant = q;
    dd->quant_parameter = par;
}

quantifier *spec_get_quantifier(specification *spec) {
    description_docket *dd = spec_find_docket(spec, FALSE);
    if (dd == NULL) return NULL;
    return dd->quant;
}

int spec_get_quantification_parameter(specification *spec) {
    description_docket *dd = spec_find_docket(spec, FALSE);
    if (dd == NULL) return 0;
    return dd->quant_parameter;
}
```

The function spec_attach_quantifier is called from 5/mlc.
The function spec_get_quantifier is called from 6/pform and 8/refpt.
The function spec_get_quantification_parameter is called from 6/pform, 8/refpt, 8/creat and 8/mass.

§40. Callings.

```
void spec_attach_calling(specification *spec, int c1, int c2) {
    description_docket *dd = spec_find_docket(spec, TRUE);
    dd->calling_w1 = c1;
    dd->calling_w2 = c2;
}
void spec_get_calling(specification *spec, int *c1, int *c2) {
    description_docket *dd = spec_find_docket(spec, FALSE);
    if (dd == NULL) { *c1 = -1; *c2 = -1; }
    else { *c1 = dd->calling_w1; *c2 = dd->calling_w2; }
}
void spec_clear_calling(specification *spec) {
    description_docket *dd = spec_find_docket(spec, FALSE);
    if (dd) { dd->calling_w1 = -1; dd->calling_w2 = -1; }
}
int spec_makes_callings(specification *spec) {
    if (spec == NULL) return FALSE;
    int c1, c2;
    spec_get_calling(spec, &c1, &c2);
    if (c1 >= 0) return TRUE;
    return FALSE;
}
```

The function spec_attach_calling is called from 5/mlc.
The function spec_get_calling is called from 6/pform and 11/ap.
The function spec_clear_calling is called from 11/los.
The function spec_makes_callings is called from 7/spec and 11/ap.

§41. The kind.

```
void spec_set_described_kov(specification *spec, kind_of_value *kov) {
    description_docket *dd = spec_find_docket(spec, TRUE);
    dd->specific_kov = kov;
}
kind_of_value *spec_get_described_kov(specification *spec) {
    description_docket *dd = spec_find_docket(spec, FALSE);
    if (dd == NULL) return NULL;
    return dd->specific_kov;
}
void spec_set_described_kind(specification *spec, world_object *k) {
    description_docket *dd = spec_find_docket(spec, TRUE);
    dd->specific_kind = k;
}
world_object *spec_get_described_kind(specification *spec) {
    description_docket *dd = spec_find_docket(spec, FALSE);
    if (dd == NULL) return NULL;
    return dd->specific_kind;
}
void spec_set_described_object(specification *spec, world_object *wo) {
    description_docket *dd = spec_find_docket(spec, TRUE);
    dd->specific_object = wo;
```

```
}
world_object *spec_get_described_object(specification *spec) {
    description_docket *dd = spec_find_docket(spec, FALSE);
    if (dd == NULL) return NULL;
    return dd->specific_object;
}
```

The function spec_set_described_kov is called from 5/mlc.

The function spec_get_described_kov is called from 5/aph, 5/mlc, 6/pform, 7/tc, 8/refpt, 8/creat, 11/ap, 12/phud, 12/phtd, 12/cinv and 13/gtok.

The function spec_set_described_kind is called from 5/mlc and 11/ap.

The function spec_get_described_kind is called from 2/prob2, 5/aph, 5/mlc, 6/pform, 6/asp, 7/tc, 8/refpt, 8/creat, 8/assem, 9/pp, 10/tab, 10/isin, 11/act, 11/av, 12/phtd, 12/cinv, 12/rb and 13/tfg.

The function spec_set_described_object is called from 5/mlc.

The function spec_get_described_object is called from 2/prob2, 5/mlc, 6/pform, 6/asp, 7/specv, 8/refpt, 10/tab, 10/isin, 11/ap, 12/phtd and 13/tfg.

§**42.** Adjective lists are attached to descriptions as part of the docket: thus "lockable open door" generates a SP including the adjective list (lockable, open). It's often convenient to loop through the adjectives for a given SP:

```
define LOOP_THROUGH_ADJECTIVE_LIST(ale, spec)
    for (ale = (spec_find_docket(spec, FALSE))?((spec_find_docket(spec, FALSE))->adjectives_applied):NULL;
        ale; ale=ale->next)
```

```
adjective_list_entry *new_adjective_list_entry(adjectival_phrase *aph, int pos) {
    adjective_list_entry *ale = CREATE(adjective_list_entry);
    ale->ref_to = aph;
    ale->ref_positive = pos;
    ale->next = NULL;
    return ale;
}
void log_adjective_list_entry(adjective_list_entry *ale) {
    adjectival_phrase *aph = get_adjective_from_list_entry(ale);
    if (ale->ref_positive == FALSE) LOG("~");
    LOG("<adj:$W>", aph->word_ref1, aph->word_ref2);
}
adjective_list_entry *copy_adjective_list_entry(adjective_list_entry *ale_from) {
    return new_adjective_list_entry(ale_from->ref_to, ale_from->ref_positive);
}
adjectival_phrase *get_adjective_from_list_entry(adjective_list_entry *ale) {
    if (ale == NULL) return NULL;
    return ale->ref_to;
}
int adjective_used_positively(adjective_list_entry *ale) {
    if (ale == NULL) internal_error("null adjective tested for positivity");
    return ale->ref_positive;
}
```

The function new_adjective_list_entry is called from 5/mlc, 6/term and 6/aprop.

The function log_adjective_list_entry is called from 2/dl.

The function copy_adjective_list_entry is called from 5/mlc.

The function get_adjective_from_list_entry is called from 5/mlc, 6/term, 6/aprop, 6/prop, 6/pform, 6/tcpr, 6/asp, 6/atoms, 7/tc, 8/refpt, 8/imp, 10/tab and 13/tfg.

The function adjective_used_positively is called from 6/aprop, 6/pform, 6/asp, 6/atoms, 8/refpt and 13/tfg.

COMMAND Specifications 7/cmsp

Purpose

Documentation on and functions handling specifications which fall into the COMMAND family.

7/cmsp.§1-8 Creation; §9 Coercion; §10 Pretty-printing; §11-19 Compilation

---

Definitions

---

§**1. Creation.** There are a variety of specific types here, but most are variations on a theme: the phrase used as part of the definition of a rule or another phrase. For instance, in the rule

```
Before looking: say "You blink in surprise."
```

the test

```
say "You blink in surprise."
```

is parsed to an actual `TO_PHRASE_SPC` specification.

§**2.** `TO_PHRASE_SPC` has an invocation list attached, as all phrasal specifications do.

§**3.** `END_BLOCK_SPC` signifies an "end if", "end repeat", etc., phrase ending a block of code. Such phrases are unambiguous and do not in any case correspond to formal definitions (i.e., there is no line in the Standard Rules reading "To end if: ..."), so they are represented by this species rather than `TO_PHRASE_SPC`. In order to distinguish a use of "end if" from an "end while", or some other, a pointer is attached to the `phrase` which is ending ("if", "while" or whatever).

§**4.** `OTHERWISE_SPC` signifies an "otherwise", and since this applies only to "if" constructions, there is no need for any pointer this time.

§**5.** `CASE_SPC` signifies a case in a switch statement. This can either have a single argument, the value for this case, or no arguments, marking it out as the default case.

§**6.** `RULEBOOK_OUTCOME_PHRASE_SPC` is a rulebook outcome – such as "persuasion succeeds" in the persuasion rules – used as if it were a phrase: it means, of course, stop the rulebook here and use this as the outcome. It can only be parsed in rules belonging to that rulebook, so there is no danger of it being used in a phrase to decide a value or an outcome. A pointer to the named outcome in question is attached.

§**7.** TRY_ACTION_SPC is the cuckoo in the nest, and there are arguments for it being a form of CONDITION rather than COMMAND. Strictly speaking, it represents not a phrase like

try eating the buttercup;

but rather the action described, viz., "eating the buttercup". This is parsed as a TEST_ACTION_SPC – where it is indeed a condition – but is converted into a TRY_ACTION_SPC on being type-checked against a generic TRY_ACTION_SPC specification. For this to succeed, the action must be exact in every way, so relatively few TRY_ACTION_SPC types chosen at random would be eligible. All the same, that makes it sound like a species of CONDITION, so why isn't it? The answer really is that it compiles into I6 code which executes in a void context to perform the action in question, not into anything which is syntactically an I6 condition.

```
void create_COMMAND_species(void) {
    type_ID_set_description(COMMAND_FMY, "a phrase");
    type_ID_set_source_name(COMMAND_FMY, "COMMAND_FMY");
    type_ID_set_parsing_names(COMMAND_FMY, phrase_V, phrases_V);
    type_ID_permit_pair(COMMAND_FMY, UNKNOWN,
        TYPE_PARSED_SPCONTEXT + TYPE_EXPECTED_SPCONTEXT);
    type_ID_permit_pair(COMMAND_FMY, TO_PHRASE_SPC,
        PARSED_SPCONTEXT + TYPE_PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT +
        TYPE_EXPECTED_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(TO_PHRASE_SPC, "no value resulting");
    type_ID_set_source_name(TO_PHRASE_SPC, "TO_PHRASE_SPC");
    type_ID_set_flag(TO_PHRASE_SPC, PHRASAL_SPCFLAG + INVLIST_SPCFLAG);
    type_ID_permit_pair(COMMAND_FMY, OTHERWISE_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(OTHERWISE_SPC, "a phrase between blocks of phrases");
    type_ID_set_source_name(OTHERWISE_SPC, "OTHERWISE_SPC");
    type_ID_set_flag(OTHERWISE_SPC, PHRASAL_SPCFLAG);
    type_ID_permit_pair(COMMAND_FMY, CASE_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(CASE_SPC, "a case in an 'if X is...' switch");
    type_ID_set_source_name(CASE_SPC, "CASE_SPC");
    type_ID_set_flag(CASE_SPC, PHRASAL_SPCFLAG + ARGUMENTS_SPCFLAG);
    type_ID_permit_pair(COMMAND_FMY, END_BLOCK_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(END_BLOCK_SPC, "a phrase ending a block of phrases");
    type_ID_set_source_name(END_BLOCK_SPC, "END_BLOCK_SPC");
    type_ID_set_flag(END_BLOCK_SPC, PHRASAL_SPCFLAG);
    type_ID_permit_pair(COMMAND_FMY, RULEBOOK_OUTCOME_PHRASE_SPC,
        PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT + TYPE_EXPECTED_SPCONTEXT +
        COMPILED_SPCONTEXT);
    type_ID_set_description(RULEBOOK_OUTCOME_PHRASE_SPC,
        "a phrase providing an outcome to a rulebook");
    type_ID_set_source_name(RULEBOOK_OUTCOME_PHRASE_SPC,
        "RULEBOOK_OUTCOME_PHRASE_SPC");
    type_ID_set_flag(RULEBOOK_OUTCOME_PHRASE_SPC, PHRASAL_SPCFLAG);
    type_ID_permit_pair(COMMAND_FMY, TRY_ACTION_SPC,
        PARSED_SPCONTEXT + TYPE_PARSED_SPCONTEXT + TYPE_FOUND_SPCONTEXT +
        TYPE_EXPECTED_SPCONTEXT + COMPILED_SPCONTEXT);
    type_ID_set_description(TRY_ACTION_SPC, "an action");
    type_ID_set_parsing_names(TRY_ACTION_SPC, action_V, actions_V);
    type_ID_set_source_name(TRY_ACTION_SPC, "TRY_ACTION_SPC");
```

```
    type_ID_set_flag(TRY_ACTION_SPC, ARGUMENTS_SPCFLAG);
}
```

The function create_COMMAND_species is called from 7/tids.

§**8.** And the usual creator routines:

```
specification *new_COMMAND_spec(void) {
    specification *spec = spec_new(COMMAND_FMY, UNKNOWN);
    return spec;
}
specification *new_TO_PHRASE_spec(void) {
    specification *spec = spec_new(COMMAND_FMY, TO_PHRASE_SPC);
    spec_create_invocation_list(spec);
    return spec;
}
specification *new_OTHERWISE_spec(void) {
    specification *spec = spec_new(COMMAND_FMY, OTHERWISE_SPC);
    return spec;
}
specification *new_CASE_spec(specification *case_val) {
    specification *spec = spec_new(COMMAND_FMY, CASE_SPC);
    spec_set_argument(spec, 0, case_val);
    return spec;
}
specification *new_default_CASE_type(void) {
    specification *spec = spec_new(COMMAND_FMY, CASE_SPC);
    return spec;
}
specification *new_RULEBOOK_OUTCOME_COMMAND_spec(named_rulebook_outcome *nro) {
    specification *spec = spec_new(COMMAND_FMY, RULEBOOK_OUTCOME_PHRASE_SPC);
    ATTACH_TO_SPEC(spec, named_rulebook_outcome, nro);
    return spec;
}
specification *new_END_BLOCK_spec(phrase *ph) {
    specification *spec = spec_new(COMMAND_FMY, END_BLOCK_SPC);
    ATTACH_TO_SPEC(spec, phrase, ph);
    return spec;
}
```

The function new_COMMAND_spec is called from 12/cph.
The function new_TO_PHRASE_spec is called from 5/mlc.
The function new_OTHERWISE_spec is called from 5/mlc.
The function new_CASE_spec is called from 5/mlc.
The function new_default_CASE_type is called from 5/mlc.
The function new_RULEBOOK_OUTCOME_COMMAND_spec is called from 5/mlc.
The function new_END_BLOCK_spec is called from 5/mlc.

§**9. Coercion.** As noted above, TRY_ACTION_SPC specifications come into existence only through coercion:

```
void coerce_TEST_ACTION_to_TRY_ACTION(specification *spec,
    specification *nspec, specification *sspec, specification *aspec,
    int other_flag, action_name_list *anl) {
    if (spec == NULL) internal_error("coerce_TEST_ACTION_to_TRY_ACTION on NULL");
    spec_coerce_to(spec, COMMAND_FMY, TRY_ACTION_SPC);
    spec_set_argument(spec, 0, nspec);
    spec_set_argument(spec, 1, sspec);
    spec_set_argument(spec, 2, aspec);
    if (other_flag) spec_set_flag(spec, ACTION_FOR_OTHER_SPFLAG);
    else spec_clear_flag(spec, ACTION_FOR_OTHER_SPFLAG);
    ATTACH_TO_SPEC(spec, action_name_list, anl);
}
```

The function coerce_TEST_ACTION_to_TRY_ACTION is called from 11/ap.

§**10. Pretty-printing.** Was never so plain:

```
void write_COMMAND_out_in_English(specification *spec, char *text) {
    sprintf(text+strlen(text), "an instruction to do something");
}
void log_COMMAND_spec_details(specification *spec) {
}
```

The function write_COMMAND_out_in_English is called from 7/spec.
The function log_COMMAND_spec_details is called from 7/spec.

§**11. Compilation.** Some phrases do decide values, but those can't be compiled in a void context: they live in the VALUE family, not here. So:

```
kind_of_value *kov_when_COMMAND_is_evaluated(specification *spec) {
    return NULL;
}
```

The function kov_when_COMMAND_is_evaluated is called from 7/spec.

§**12.** So, then:

```
int compile_To_phrase_recursion_depth = 0;
void compile_COMMAND_spec(OUTPUT_STREAM, specification *spec_found) {
    switch (spec_get_species(spec_found)) {
        case CASE_SPC: ⟨Compile a switch case 15⟩;
        case END_BLOCK_SPC: ⟨Compile an "end..." 14⟩;
        case OTHERWISE_SPC: ⟨Compile an "otherwise" 13⟩;
        case RULEBOOK_OUTCOME_PHRASE_SPC: ⟨Compile a rulebook outcome 17⟩;
        case TO_PHRASE_SPC: ⟨Compile a usage of a "To..." phrase 16⟩;
        case TRY_ACTION_SPC: ⟨Compile a try of an action 18⟩;
    }
}
```

The function compile_COMMAND_spec is called from 7/cfsp.

§**13.** Because the type-checker works only locally, not at a level as large as an entire rule definition, there's no good way to issue this problem message at the type-checking stage. But it does no harm to leave it until now, anyway.

⟨Compile an "otherwise" 13⟩ ≡
```
    if (increment_otherwise_count() == 1) {
        sentence_problem(_P_(C7DoubleOtherwise),
            "that makes two unconditional 'otherwise' or 'else' clauses "
            "for this 'if'",
            "which is forbidden since 'otherwise' is meant to be a single "
            "(optional) catch-all clause at the end.");
    }
    WRITE("} else {\n");
    return;
```

This code is used in §12.

§**14.**

⟨Compile an "end..." 14⟩ ≡
```
    WRITE("}\n");
    if (ct_variables_saved_in_this_block() >= 0) WRITE("@pull ct_1; @pull ct_0;");
    close_code_block();
    return;
```

This code is used in §12.

§**15.** Switch statements in I6 look much like those in C, but are written without the ceaseless repetition of the keyword "case". Thus, `15:` does what `case 15:` would do in C. But `default:` is the same in both.

⟨Compile a switch case 15⟩ ≡
```
    if ((spec_get_argc(spec_found) != 0) && (spec_get_argc(spec_found) != 1))
        internal_error("Compiled malformed CASE SP");
    if (increment_otherwise_count() != 0) WRITE(";\n");
    if (spec_get_argc(spec_found) == 0) WRITE("default");
    else spec_compile(OUT, spec_get_argument(spec_found, 0));
    WRITE(":\n");
    return;
```

This code is used in §12.

§**16.** The following problem messages are issued here rather than in typechecking because, in fact, the constructions in question "ought" to be legal – from a formal point of view they are reasonable. But they lead to horrible ambiguities which end up being resolved in ways the author doesn't expect, or so experience showed. Our mechanism for blocking them is admittedly ugly, but works nicely in practice.

⟨Compile a usage of a "To..." phrase 16⟩ ≡

```
compile_To_phrase_recursion_depth++;
if ((compile_To_phrase_recursion_depth > 1) &&
    (detect_control_structure(spec_found->word_ref1, spec_found->word_ref2))) {
    sentence_problem(_P_(C7BlockWithinNonblock),
        "structural phrases like 'if', 'repeat', 'while' or 'otherwise' "
        "can't be used as part of the abbreviated form of other structural "
        "phrases",
        "so for instance 'if in darkness, repeat with X running through rooms' "
        "is not allowed because the 'repeat' is too important a phrase to be "
        "put at the end of a simple 'if' like this. Instead, the 'if' must be "
        "written out in full, so for instance 'if in darkness begin; "
        "repeat ... begin; ...; end repeat; end if;' would be fine.");
}
if ((is_otherwise_if(spec_found->word_ref1, spec_found->word_ref2)) &&
    (read_otherwise_count() > 0)) {
    sentence_problem(_P_(C7OtherwiseIfAfterOtherwise),
        "this seems to be misplaced since it is out of sequence within its 'if'",
        "with an 'otherwise if...' coming after the more general 'otherwise' "
        "rather than before. (If there's an 'otherwise' clause, it has to be "
        "the last clause of the 'if'.)");
}
compile_invocational_phrase(OUT, spec_found);
compile_To_phrase_recursion_depth--;
return;
```

This code is used in §12.

§**17.**

⟨Compile a rulebook outcome 17⟩ ≡

```
rb_compile_outcome(OUT, RETRIEVE_FROM_SPEC(spec_found, named_rulebook_outcome));
return;
```

This code is used in §12.

§**18.** `TryAction` is a template routine with four arguments: a bitmap of flags about the nouns and actor, the actor object, and the noun and second noun values.

⟨Compile a try of an action 18⟩ ≡

```
    if (spec_get_argc(spec_found) != 3) internal_error("Compiled malformed ACTION SP");
    specification *spec0 = spec_get_argument(spec_found, 0);                    the noun
    specification *spec1 = spec_get_argument(spec_found, 1);                    the second noun
    specification *spec2 = spec_get_argument(spec_found, 2);                    the actor
    if ((spec_is_CONSTANT_of_kova(spec0, UNDERSTANDING_TY)) && ([[spec0 == it]] == FALSE))
        coerce_UNDERSTANDING_to_TEXT(spec0);
    if ((spec_is_CONSTANT_of_kova(spec1, UNDERSTANDING_TY)) && ([[spec1 == it]] == FALSE))
        coerce_UNDERSTANDING_to_TEXT(spec1);
    action_name_list *anl = RETRIEVE_FROM_SPEC(spec_found, action_name_list);
    action_name *an = anl_get_singleton_action(anl);
    LOGIF(EXPRESSIONS, "Compiling from action name list:\n$L\n", anl);
    int flag_bits = 0;
    if (is_kova(spec_evaluates_to(spec0), TEXT_TY)) flag_bits += 16;
    if (is_kova(spec_evaluates_to(spec1), TEXT_TY)) flag_bits += 32;
    if (flag_bits > 0) ensure_basic_heap_present();
    if (spec_test_flag(spec_found, ACTION_FOR_OTHER_SPFLAG)) flag_bits += 1;
    WRITE("TryAction(%d, ", flag_bits);
    if (spec2) compile_try_action_parameter(OUT, spec2, kova(OBJECT_TY));
    else WRITE("player");
    WRITE(", ##%s, ", an_get_I6_representation(an));
    if (spec0) compile_try_action_parameter(OUT, spec0, act_get_data_type_of_noun(an));
    else WRITE("0");
    WRITE(", ");
    if (spec1) compile_try_action_parameter(OUT, spec1, act_get_data_type_of_second_noun(an));
    else WRITE("0");
    WRITE(");");
    return;
```

This code is used in §12.

§**19.** Which requires the following. As ever, there have to be hacks to ensure that text as an action parameter is correctly read as parsing grammar rather than text when the action expects that.

```
void compile_try_action_parameter(OUTPUT_STREAM, specification *spec, kind_of_value *required_kov) {
    specification *spec_expected = kov_as_spec(required_kov);
    if (is_kova(required_kov, UNDERSTANDING_TY)) {
        kind_of_value *kov = spec_evaluates_to(spec);
        if ((can_we_cast_kovs(kov, kova(UNDERSTANDING_TY))) ||
            (can_we_cast_kovs(kov, kova(INDEXED_TEXT_TY)))) {
            spec_expected = spec;
        }
    }
    if (typecheck(spec, spec_expected)) spec_compile(OUT, spec);
}
```

Type Checking 7/tc

Purpose

To decide whether a given specification, representing a value or construct found in the source text, can be used in a given context: the possible answers being yes, no, or "possibly, contingent on run-time type checking".

7/tc.§1-2 The middle level; §3 The outermost level; §4-8 Global state information; §9-13 Capping the recursion; §14-25 Rule (a); §26-28 Rule (b); §29-33 Rule (c); §34-37 Rule (d); §38-44 Rule (e); §45-52 Rule (f); §53-95 (g) Recurse down to check arguments; §96-103 Preserving successful invocations; §104-115 Problems, problems, problems

---

Definitions

```
typedef struct inv_token_problem_token {
    int word_ref1, word_ref2;
    struct specification *as_parsed;
    int already_described;
    MEMORY_MANAGEMENT
} inv_token_problem_token;
```

The structure inv_token_problem_token is private to this section.

---

§**1. The middle level.** Recall that the typechecker is built on three layers. The innermost layer, which is like typechecking in C, tests to see if one kind of value (KOV) can be used in place of another. This is provided by the `can_we_cast_kovs` routine, from "Kinds of Value.w".

The following modest little routine forms the entire middle layer of the typechecker, which expands outwards from single values to think about descriptions. Each of `from_spec` and `to_spec` is either an evaluating specification (an lvalue or rvalue), or a `DESCRIPTION_SPC`.

We find the narrowest possible KOVs for the two specifications and try to compare them. The wrinkle is that if the target `to_spec` is qualified, by adjectives or some proposition, then we can only know at run-time whether a match can be made, so we must return `SOMETIMES_MATCH` instead of `ALWAYS_MATCH`.

```
int can_we_match_value_descriptions(specification *from_spec, specification *to_spec) {
    LOGIF(TYPE_CASTS, "[Can we match from: $S to: $S?]\n", from_spec, to_spec);
    kind_of_value *from_kov = narrowest_KOV_of_value_or_description(from_spec);
    kind_of_value *to_kov = narrowest_KOV_of_value_or_description(to_spec);
    int result = NEVER_MATCH;
    if ((from_kov) && (to_kov)) result = can_we_cast_kovs(from_kov, to_kov);
    else if (to_kov) result = SOMETIMES_MATCH;
    if ((spec_is_qualified_DESCRIPTION(to_spec)) && (result == ALWAYS_MATCH))
        result = SOMETIMES_MATCH;
    switch(result) {
        case ALWAYS_MATCH:    LOGIF(TYPE_CASTS, "[Always]\n"); break;
        case SOMETIMES_MATCH: LOGIF(TYPE_CASTS, "[Sometimes]\n"); break;
        case NEVER_MATCH:     LOGIF(TYPE_CASTS, "[Never]\n"); break;
    }
    return result;
}
```

The function can_we_match_value_descriptions is called from 7/cosp, 11/ap, 12/phtd, 12/cinv and 13/gl.

§**2.** Either the docket or the proposition of a DESCRIPTION_SPC will tell us what it describes. If there's no other indication, it always describes objects.

```
kind_of_value *narrowest_KOV_of_value_or_description(specification *spec) {
    kind_of_value *kov = NULL;
    if (spec_is_evaluating(spec)) kov = spec_evaluates_to(spec);
    else if (species_is(spec, DESCRIPTION_SPC)) {
        if (spec_get_described_kov(spec))
            kov = spec_get_described_kov(spec);
        if (spec_get_described_kind(spec))
            kov = kovko(spec_get_described_kind(spec));
        if (spec_get_proposition(spec))
            kov = prop_describes_KOV(spec_get_proposition(spec));
        if (spec == NULL) kov = kova(OBJECT_TY);
    } else internal_error("tried to narrow KOV for inappropriate specification");
    return kov;
}
```

§**3. The outermost level.** We now have the two inner levels of the type-checker in place, and must write the outermost level: which handles the general case of any specification compared with any other, and which diagnoses failures in detail, producing Problem messages when things go wrong. The main reason that the outermost level is the most complicated is simply that, when checking fails, we need to distinguish between the many reasons why it failed, in order to produce worthwhile Problems. At the inner levels, a fail is a fail.

The outermost level of the typechecker sometimes performs coercion: it will sometimes modify the SP found to amend it in light of the circumstances indicated by the SP expected. It will also remove the one remaining ambiguity in the SP structure: for a phrasal SP with rival invocations, it will decide which invocation(s) will be compiled (or else will issue a Problem if none can be).

Partly because of the need to do this, the type-checker has a top-down approach. It aims to prove that the SP found can match the SP expected, making selections and coercions as needed to do this. For instance, when

> change the score to the score plus 10

is checked to see that it is valid in code, an actual COMMAND_FMY of species TO_PHRASE_SPC is checked against a generic COMMAND_FMY. The typechecker sees at once that this is provably correct, so long as the arguments to the phrase work. There are several valid interpretations of "change ... to ...", and these are all present in the SP found as alternative invocations, so the typechecker tries each in turn, accepting one (or more) if the arguments can be proved to be of the right type. This means proving that argument 0 ("the score") matches a generic NONLOCAL_VARIABLE_SPC and also that argument 1 ("the score plus 10") matches a generic VALUE_FMY. A further rule requires that the kind of value of argument 1 must match the kind of value stored in the variable, here a NUMBER_TY, so we must prove that too. Now "plus" is polymorphic and can produce different kinds of value depending on the kinds of value it acts upon, so again we must check all possible interpretations. But we finally succeed in showing that "score" is a number, "10" is a number, and that "plus" on two numbers gives a number, so we complete the proof and the phrase is accepted.

§**4. Global state information.** Throughout the recursive use of our typechecker, we need some global state data to keep track of two decidedly global side-effects of checking: the issuing of problem messages, and the creation of variables. These irrevocable acts can't be mere flickering possibilities on the stack, like other intermediate results of typechecking a complex expression.

First, we keep track of the problem messages issued, if any. Three grades of problem can appear: "ordinary", "gross" and "grosser than gross". We distinguish these in order to produce a Problem message which reflects the biggest thing wrong, rather than being so esoteric that it misses the main point. Changing a particular error condition from an ordinary to a gross problem, or vice versa, has no effect on the result of `typecheck()`: only on the Problem messages given to the user.

```
define THIS_IS_A_GROSSER_THAN_GROSS_PROBLEM
    no_gross_problems_thrown++;                          problems this gross cannot be suppressed
define THIS_IS_A_GROSS_PROBLEM
    no_gross_problems_thrown++;                          this increments even if the message is suppressed
    if (problem_threshold >= 1) return NEVER_MATCH;      meaning "suppress even gross problems"
define THIS_IS_AN_ORDINARY_PROBLEM
    if (problem_threshold >= 2) return NEVER_MATCH;      meaning "suppress ordinary problems"

int no_gross_problems_thrown = 0;
int problem_count_at_typecheck_start = 0;
```

§**5.**

⟨Zero the counts of problems produced in typechecking 5⟩ ≡

```
    no_gross_problems_thrown = 0;
    problem_count_at_typecheck_start = problem_count;
```

This code is used in §9.

§**6.** And subsequently we detect whether anything untoward has happened:

```
int problems_issued_during_typechecking(void) {
    if (problem_count_at_typecheck_start < problem_count) return TRUE;
    return FALSE;
}
```

§**7.** The other piece of state data is to do with the creation of local variables through the "let" phrase, which happens in the typechecker (since the kind of value for the variable can only be discovered by typechecking). We do not want to allow the recursive typechecker the power to create variables, because it might then use this power in a branch of checking which doesn't work out for some other reason – that would leave us with a bogus variable. Instead, the typechecker records a *request* to make a local variable in the following state variables. If the typechecker succeeds in proving a "let" phrase valid, only then will the variable be created.

```
typedef struct local_variable_request {
    int word_ref1, word_ref2;                                   name of the requested variable
    struct kind_of_value *KOV_from_prototype;                   the KOV it should have
    int depth_made_at;                  the depth of typechecking recursion at which the request was made
    struct specification *local_initial_value;
    struct kind_of_value *recently_discovered_KOV;          used when polymorphism makes this unexpected
} local_variable_request;

local_variable_request requested = { -1, -1, NULL, 0, NULL, NULL };
kind_of_value *KOV_of_recently_created_local = NULL;
```

The structure local_variable_request is private to this section.

§**8.**

⟨Empty any local variable request 8⟩ ≡

```
requested.word_ref1 = -1; requested.word_ref2 = -1;
requested.KOV_from_prototype = NULL;
requested.depth_made_at = 0;
requested.local_initial_value = NULL;
requested.recently_discovered_KOV = NULL;
```

This code is used in §9,97,9,97.

§**9. Capping the recursion.** At this top level, something checks out or it doesn't, so the two top-level routines return `TRUE` or `FALSE`. But beneath this head, the type-checker acts recursively, because the SP structure is itself recursive: some SPs include pointers to further SPs, and these must be type-checked in turn. In that recursion, each call to `typecheck_recursive` returns one of the three values `ALWAYS_MATCH`, `SOMETIMES_MATCH` or `NEVER_MATCH`.

Note that although `typecheck` is not *directly* recursive, it can sometimes be indirectly so, because it calls out to `prop_type_check` at rule (d), which can in turn start fresh rounds of `typecheck`. So we carefully preserve the local variable request on entry and exit – not because `prop_type_check` is likely to make requests (it's in fact not able to, I think), but because its calls to `typecheck` have the side-effect of emptying the request. This was at one time the source of some very subtle bugs occurring only very seldom.

```
int typecheck_without_expectations(specification *found) {
    return typecheck(found, found);
}
int typecheck(specification *found, specification *expected) {
    local_variable_request saved_request = requested;
    ⟨Empty any local variable request 8⟩;
    ⟨Zero the counts of problems produced in typechecking 5⟩;
    int rv = typecheck_recursive(found, expected, 0, 1);
    requested = saved_request;
    if (rv == NEVER_MATCH) return FALSE;
    return TRUE;
}
```

The function typecheck_without_expectations is called from 6/tcpr, 6/cind, 12/cinv, 13/test and 14/i6t.
The function typecheck is called from 7/cmsp, 9/scene, 11/ap and 12/cph.

§**10.** We still cannot quite begin the work, since a wrapper routine provides for careful debug logging of all that goes on. I have spent far too many hours staring at the log of what the typechecker has done. During Inform's infancy, it was not so much that this was the great habitat and breeding ground for bugs; more that those bugs which *were* here were by far the hardest to root out. So careful logging on demand is vital.

```
define LOG_STAGE_LEFT(stage)
    LOGIF(INVOCATIONS, "[%d%s] %s $S",
        unique_TR_call_identifier,
        (problem_threshold == 0)?"":
            ((problem_threshold == 1)?"-quiet":"-silent"), stage, found);
define LOG_STAGE(stage)
    { LOG_STAGE_LEFT(stage); LOGIF(INVOCATIONS, "\n"); }

int unique_TR_call_identifier = 0, TR_call_counter = 0;
int typecheck_recursive(specification *found, specification *expected,
    int problem_threshold, int depth) {
```

```
    int outer_id = unique_TR_call_identifier;
    unique_TR_call_identifier = TR_call_counter++;
    STREAM_INDENT(dl);
    LOG_STAGE_LEFT("try"); LOGIF(INVOCATIONS, " --?--> $S\n", expected);
    STREAM_INDENT(dl);
    int return_value = typecheck_recursive_inner(found, expected, problem_threshold, depth);
    STREAM_OUTDENT(dl);
    switch(return_value) {
        case ALWAYS_MATCH:    LOG_STAGE("always match"); break;
        case SOMETIMES_MATCH: LOG_STAGE("sometimes match"); break;
        case NEVER_MATCH:     LOG_STAGE("never match"); break;
        default: internal_error("impossible verdict from recursive typechecker");
    }
    STREAM_OUTDENT(dl);
    unique_TR_call_identifier = outer_id;
    return return_value;
}
```

§**11.** We first do a little fussing over `found` and `expected`.

```
int typecheck_recursive_inner(specification *found, specification *expected,
    int problem_threshold, int depth) {
    if ((expected) && (!(spec_is_UNKNOWN(expected))))
        type_ID_observe(expected, TYPE_EXPECTED_SPCONTEXT);
    if ((found) && (!(spec_is_UNKNOWN(found))))
        type_ID_observe(found, TYPE_FOUND_SPCONTEXT);
    if (spec_is_generic_CONSTANT(found))
        ⟨Coerce a found generic constant to a description 12⟩;
    if (spec_is_generic(found)) internal_error("type-check on a generic found specification");
    ⟨Perform the type-checking algorithm 13⟩;
}
```

§**12.** Inside the belly of the typechecker, `found` has to be an actual specification, because it's something which may be compiled some day. The only way we can see a generic `found` is as a result of an ambiguity in English. Consider:

> Colour is a kind of value.
>
> [1] Red is a colour.
>
> [2] let N be the number of colours;

In excerpt [1], "colour" is generic and will never be compiled as a value, but in [2] "colours" is something actual – it has to be compiled to a routine enumerating colours, which can be passed to the "number of ..." phrase. But both usages parse to generic `CONSTANT_SPC` specifications.

It's only when we typecheck that we realise we must mean this as an actual description, and we perform a coercion to turn the specification from a generic `CONSTANT_SPC` to an actual `DESCRIPTION_SPC`.

As can be seen, we can get rid of some easy typechecking cases straight away: a generic K_1 value matches a generic K_2 value if $K_1 \subseteq K_2$. In particular this ensures that a generic K_1 matches itself. But even if there's a match, the specification is still coerced. So even in this one permitted case when `found` is allowed into the typechecker while generic, it can't leave that way.

⟨Coerce a found generic constant to a description 12⟩ ≡

```
int x = NEVER_MATCH;
if (spec_is_generic_CONSTANT(expected))
    x = can_we_cast_kovs(spec_get_kind_of_value(found),
            spec_get_kind_of_value(expected));
coerce_generic_CONSTANT_to_DESCRIPTION(found);
if (x != NEVER_MATCH) return x;
```

This code is used in §11.

§**13.** From here on `found` is certainly an actual SP, and `expected` can be either actual or generic. It's time to perform some checking.

The best way to think about this algorithm is that it nearly does something simple: it compares the species of `found` and `expected` to see if they match or not. This is called the Main Rule, and even though it is partially buried under a great heap of exceptions and finicky special cases, it remains most of the story. Here is a less sketchy account:

(a) Handle the creation of new local variable names, using either text with no known meaning, or in some cases allowing the overloading of an existing meaning. Reject unknown meanings found in all other cases.

(b) Handle the case where a specific constant value expected.

(c) Make coercions, 1: expand bare property names into what they seem to be intended as properties of. For instance, "carrying capacity" might be interpreted as "carrying capacity of the item discussed". Reject bad property values.

(d) Make coercions, 2: typecheck any proposition attached to the SP found, and coerce to "now" rather than "test" form if that's expected.

(e) Make coercions, 3: miscellaneous other coercions to bring the SP found into line with what was expected.

(f) To match a generic constant value of KOV K, the SP found has to evaluate to a KOV which implicitly casts to K. To match a description, the SP found has to either specify an object which satisfies it, or has to be a description provably contained within it. In all other cases, the SP found matches if and only if it has the same species as the SP expected.

(g) Recurse downwards to typecheck any arguments.

(h) Make sure that structural commands are used in the right places: for instance, cases appear inside switches, and so on.

(i) Check through the list of invocations, if there is one, and select which possibilities remain for consideration at run-time. (Usually in fact there is just one possibility.)

⟨Perform the type-checking algorithm 13⟩ ≡

```
int best_result = ALWAYS_MATCH;      drops to SOMETIMES_MATCH if a need for run-time checking is realised
⟨Step (a) Deal with the UNKNOWN 14⟩;
⟨Step (b) Deal with expected actual CONSTANT types 26⟩;
⟨Step (c) Deal with bare property names 29⟩;
⟨Step (d) Deal with any attached proposition 34⟩;
⟨Step (e) Apply miscellaneous other coercions 38⟩;
⟨Step (f) The Main Rule of Type-Checking 45⟩;
if (spec_get_argc(found) > 0) ⟨Step (g) Recurse down to check arguments 53⟩;
if (family_is(found, COMMAND_FMY)) ⟨Step (h) Police the global structure of phrase bodies 62⟩;
if (spec_has_invocation_list(found)) ⟨Step (i) Typecheck the invocations, choosing among them 67⟩;
return best_result;
```

This code is used in §11.

§14. Rule (a). It might seem rather odd that the typechecker should ever have to deal with `found` being unknown text. Surely that's a sign that the parsing went wrong, so how did things get to this stage?

In a C-like language, where variables are predeclared, that would be true. But in Inform, a phrase like:

```
let the monster be a random pterodactyl;
```

can be valid even where "the monster" is text not known to the S-parser as yet – indeed, that's how local variables are made. It's the typechecker which sorts this out, because only the typechecker can decide which of the subtly different forms of "let" is being used. So, in short, we do need to cope with `found` containing an `UNKNOWN` specification. We get it out of the way first.

⟨Step (a) Deal with the UNKNOWN 14⟩ ≡
```
⟨Step (a.1) Deal with "kind of word value" and "kind of problem value" 15⟩;
⟨Step (a.2) Deal with text where a nonexisting variable name is expected 17⟩;
⟨Step (a.3) Let all other UNKNOWN text lead to a problem message 19⟩;
```

This code is used in §13.

§15. Rule (a.1) is a delicate manoeuvre, but luckily it takes action only very rarely and in very specific circumstances. The data type IDs `KIND_OF_WORD_VALUE_TY` and `KIND_OF_POINTER_VALUE_TY` are the way we match phrase definitions like

```
let (name - nonexisting variable) be (type - kind of word value);
```

where the "type" parameter has to be the name of any KOV which can be stored in a single word (rather than a pointer to data on the heap) – thus, we match [1] and [2] but not [3], [4] or [5] from:

```
[1] let X be a number;
[2] let X be a vehicle;
[3] let X be indexed text;
[4] let X be 21;
[5] let X be {1, 2, 3};
```

`KIND_OF_POINTER_VALUE_TY` represents "kind of pointer value" and would contrariwise allow only [3].

What all of this has to do with being `UNKNOWN` is that text parsed in the expectation of a value will usually not recognise something like "a list of numbers", so that would be here as `UNKNOWN`. We take the otherwise unheard-of measure of reparsing the text, but we only impose the result on `found` if the match can definitely be made successfully.

⟨Step (a.1) Deal with "kind of word value" and "kind of problem value" 15⟩ ≡
```
LOG_STAGE("(a.1)");
int pointer_status = NOT_APPLICABLE;
if (spec_is_generic_CONSTANT_of_kova(expected, KIND_OF_WORD_VALUE_TY))
    pointer_status = FALSE;
if (spec_is_generic_CONSTANT_of_kova(expected, KIND_OF_POINTER_VALUE_TY))
    pointer_status = TRUE;
if (pointer_status != NOT_APPLICABLE) {
    if ((spec_is_UNKNOWN(found)) || (species_is(found, DESCRIPTION_SPC))) {
        int w1 = found->word_ref1, w2 = found->word_ref2;
        specification *reparsed = parse_expression(w1, w2, TYPE_EXPCON);
        LOGIF(INVOCATIONS, "(a.1) reparsed as: $S\n", reparsed);
        if (spec_is_generic_CONSTANT(reparsed)) {
            kind_of_value *kov = spec_get_kind_of_value(reparsed);
            if (kov_uses_pointer_values(kov) == pointer_status) {
                *found = *reparsed;
```

```
                return ALWAYS_MATCH;
            }
            return NEVER_MATCH;
        }
        if (species_is(reparsed, DESCRIPTION_SPC))
            ⟨Disallow any description which cannot be proved or disproved at compile time 16⟩;
    }
}
```

This code is used in §14.

§**16.** Note that "a vehicle" parses as a DESCRIPTION_SPC, not a generic constant, and therefore falls into this category. (At compile time we can't always tell which object values will be vehicles.)

⟨Disallow any description which cannot be proved or disproved at compile time 16⟩ ≡

```
THIS_IS_A_GROSSER_THAN_GROSS_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
quote_spec(3, expected);
handmade_problem(_P_(C7LetDescription));
issue_problem_segment(
    "In the line %1, you seem to be using '%2' to spell out what kind of "
    "value to use. But this is too complicated a description of values - "
    "it needs to be something much simpler, just a literal name of a kind "
    "of value, such as 'a number' or 'an object'.");
issue_problem_end();
return NEVER_MATCH;
```

This code is used in §15.

§**17.** NEW_LOCAL_VARIABLE_NAME_SPC matches only against previously unknown names, or (to allow overloading of meanings and avoid needless namespace clashes) the names of objects, table columns or properties, and we can otherwise reject it.

⟨Step (a.2) Deal with text where a nonexisting variable name is expected 17⟩ ≡

```
LOG_STAGE("(a.2)");
if (species_is(expected, NEW_LOCAL_VARIABLE_NAME_SPC)) {
    if ((spec_is_UNKNOWN(found)) ||
        (spec_is_CONSTANT_of_kova(found, OBJECT_TY)) ||
        (spec_is_CONSTANT_of_kova(found, TABLE_COLUMN_TY)) ||
        (spec_is_CONSTANT_of_kova(found, PROPERTY_TY))) {
        if (!(TEST_COMPILATION_MODE(DO_NOT_CREATE_LOCAL_VARS_CMODE))) {
            requested.word_ref1 = found->word_ref1; requested.word_ref2 = found->word_ref2;
            if (spec_get_kind_of_value(expected) == NULL)
                requested.KOV_from_prototype = kova(ANY_VALUE_TY);
            else
                requested.KOV_from_prototype = spec_get_kind_of_value(expected);
            requested.depth_made_at = depth;
        }
        return ALWAYS_MATCH;
    }
    ⟨Issue a problem for an inappropriate variable name 18⟩;
}
```

This code is used in §14.

§**18.** This problem message is never normally seen using the definitions in the Standard Rules because the definitions made there are such that other problems appear first. So the only way to see this message is to declare an unambiguous phrase with one of its tokens requiring a variable of a species; and then to misuse that phrase.

⟨Issue a problem for an inappropriate variable name 18⟩ ≡

```
THIS_IS_AN_ORDINARY_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
if (spec_get_described_kov(found)) quote_text(3, "a kind of value");
else quote_type_of(3, found);
quote_type_of(4, expected);
handmade_problem(_P_(C7KindOfVariable));
issue_problem_segment(
    "In the sentence %1, I was expecting that '%2' would be a variable "
    "name (specifically %4), but it turned out to be %3.");
issue_problem_end();
return NEVER_MATCH;
```

This code is used in §17.

§**19.** In all other cases, unknown text in `found` is incorrect. We can produce any of more than twenty different problem messages here, in an attempt to be helpful about what exactly is wrong.

⟨Step (a.3) Let all other UNKNOWN text lead to a problem message 19⟩ ≡

```
LOG_STAGE("(a.3)");
if (spec_is_UNKNOWN(found)) {
    THIS_IS_A_GROSS_PROBLEM;
    LOG("(a.3) problem message:\nfound: $Xexpected: $X", found, expected);
    if (species_is(expected, TRY_ACTION_SPC)) ⟨Unknown found text occurs as an action to try 20⟩;
    if (family_is(expected, COMMAND_FMY)) ⟨Unknown found text occurs as a command 21⟩;
    quote_source(1, current_sentence);
    quote_words(2, found->word_ref1, found->word_ref2);
    quote_spec(3, expected);
    int preceding = found->word_ref1 - 1;
    if (((compiling_text_routines_mode) || [[current_sentence == say ...]])
        && ((preceding == current_sentence->word_ref1) || [[word preceding == COMMA]]))
        ⟨Unknown found text occurs as a text substitution 22⟩
    else if ((family_is(expected, CONDITION_FMY)) &&
             (is_list_divided(found->word_ref1, found->word_ref2,
                 LOOK_FOR_AND + LOOK_FOR_OR)))
        ⟨Issue a problem message for a compound condition which has gone bad 23⟩
    else
        ⟨Issue a problem message for miscellaneous suspicious wordings 24⟩;
    return NEVER_MATCH;
}
```

This code is used in §14.

§**20.**

⟨Unknown found text occurs as an action to try 20⟩ ≡

```
specification *spec = NULL;
kind_of_value *kov = NULL, *kov2 = NULL;
action_pattern ap;
clear_validation_case();
ap = parse_action_pattern(found->word_ref1, found->word_ref2, TRUE);
if (get_validation_case(&spec, &kov, &kov2)) {
    handmade_problem(_P_(C7UnknownTryAction1));
    quote_source(1, current_sentence);
    quote_words(2, found->word_ref1, found->word_ref2);
    quote_words(3, spec->word_ref1, spec->word_ref2);
    quote_kov(4, kov);
    quote_kov(5, kov2);
    issue_problem_segment(
        "You wrote %1, but '%2' is not an action I can try. This looks as "
        "if it might be because it contains something of the wrong kind. "
        "My best try involved seeing if '%3' could be %4, which might have "
        "made sense, but it turned out to be %5.");
    issue_problem_end();
} else {
    sentence_problem(_P_(C7UnknownTryAction2),
        "this is not an action I recognise",
        "or else is malformed in a way I can't see how to sort out.");
}
return NEVER_MATCH;
```

This code is used in §19.

§**21.**

⟨Unknown found text occurs as a command 21⟩ ≡

```
int k;
if ([[found == else/otherwise ...]]
    && ((compare_word(found->word_ref1+1, if_V)) == FALSE)
    && ((compare_word(found->word_ref1+1, unless_V)) == FALSE)) {
    sentence_problem(_P_(C7WrongOtherwise),
        "inside an 'if ... begin'/'end if' an 'otherwise' (or 'else') "
        "must be a single word phrase",
        "so it appears that the two ways to use 'otherwise' have been "
        "mixed up. It can either be used as 'if ... begin; ...; "
        "otherwise; ...; end if', or as 'if ..., ...; otherwise ...;'.");
} else if [[found == else/otherwise if/unless ... COMMA ... : k]] {
    sentence_problem_with_note(_P_(C7WrongOtherwise2),
        "an 'otherwise if' (or 'else if') which uses a comma to give "
        "a single consequence can only be used immediately following "
        "an equally simple 'if'",
        "and can't be used in the middle of a begin ... end if block.",
        "To provide alternatives within such a block, use 'otherwise "
        "if ... condition...;' (in the same way that one would use "
        "'otherwise;' without a condition).");
} else if [[found == else/otherwise if/unless ... begin]] {
    sentence_problem(_P_(C7WrongOtherwise3),
```

```
            "an 'otherwise if' (or 'else if') should not take a 'begin'",
            "but acts as a divider within a begin/end block all by itself.");
    } else if [[found == if/unless ... then]] {
        sentence_problem_with_note(_P_(C7WrongThen),
            "'then' should be followed by a phrase",
            "and should not be the end of the 'if'.",
            "There are two forms of 'if': either 'if ... then ...' (where a "
            "comma can be used in place of 'then'), or the more complex 'if "
            "... begin; ...; end if'. Perhaps you meant to write 'begin' "
            "instead of 'then'?");
    } else if [[found == if ...]] {
        sentence_problem_with_note(_P_(C7PossibleUnterminatedIf),
            "this is a phrase which I don't recognise",
            "even though it starts with 'if'. 'If' phrases normally have one "
            "of two forms: 'if (condition), (do something)' and 'if (condition) "
            "begin; ... ; end if'.",
            "The difference is that the begin/end version can encompass "
            "multiple-phrase consequences if the condition is true, whereas "
            "the simpler form allows only one phrase to count as the 'do "
            "something' part.");
    } else if [[found == unless ...]] {
        sentence_problem_with_note(_P_(C7PossibleUnterminatedUnless),
            "this is a phrase which I don't recognise",
            "even though it starts with 'unless'. 'Unless' phrases normally have "
            "one of two forms: 'unless (condition), (do something)' and 'unless "
            "(condition) begin; ... ; end unless'.",
            "The difference is that the begin/end version can encompass "
            "multiple-phrase consequences if the condition is true, whereas "
            "the simpler form allows only one phrase to count as the 'do "
            "something' part.");
    } else if [[found == continue]] {
        sentence_problem(_P_(C7WrongContinue),
            "this is a phrase which I don't recognise",
            "and which isn't defined. Perhaps you wanted the phrase which "
            "would skip to the next repetition of a loop, since that's "
            "written 'continue' in some programming languages (such as C "
            "and Inform 6)? If so, what you want is 'next'.");
    } else
        sentence_problem(_P_(C7UnknownPhrase),
        "this is a phrase which I don't recognise",
        "possibly because it is one you meant to define but never got round "
        "to, or because the wording is wrong (see the Phrasebook section of "
        "the Index to check). Alternatively, it may be that the text "
        "immediately previous to this was a definition whose ending, normally "
        "a full stop, is missing?");
    return NEVER_MATCH;
```

This code is used in §19.

§**22.**

⟨Unknown found text occurs as a text substitution 22⟩ ≡

```
if [[found == COMMA ...]] {
    handmade_problem(_P_(C7SayComma));
    issue_problem_segment(
        "In the line %1, I was expecting that '%2' would be something to "
        "'say', but unexpectedly it began with a comma. The usual form is "
        "just 'say \"text\"', perhaps with some substitutions in square "
        "brackets within the quoted text, but no commas.");
    issue_problem_end();
} else if [[found == unicode ...]] {
    handmade_problem(_P_(C7SayUnicode));
    issue_problem_segment(
        "In the line %1, I was expecting that '%2' would be something to "
        "'say', but it didn't look like any form of 'say' that I know. "
        "So I tried to read '%2' as a Unicode character, which seemed "
        "likely because of the word 'unicode', but that didn't work either. "
        "%PUnicode characters can be written either using their decimal "
        "numbers - for instance, 'Unicode 2041' - or with their standard "
        "names - 'Unicode Latin small ligature oe'. For efficiency reasons "
        "these names are only available if you ask for them; to make them "
        "available, you need to 'Include Unicode Character Names by Graham "
        "Nelson' or, if you really need more, 'Include Unicode Full "
        "Character Names by Graham Nelson'.");
    issue_problem_end();
} else {
    if [[found == ... condition]] handmade_problem(_P_(C7SayUnknownCondition));
    else handmade_problem(_P_(C7SayUnknown));
    issue_problem_segment(
        "In the line %1, I was expecting that '%2' would be something to "
        "'say', but it didn't look like any form of 'say' that I know. So "
        "I tried to read '%2' as a value of some kind (because it's legal "
        "to say values), but couldn't make sense of it that way either. "
        "%PSometimes this happens because punctuation has gone wrong - "
        "for instance, if you've omitted a semicolon or full stop at the "
        "end of the 'say' phrase.");
    if [[found == ... condition]] {
        issue_problem_segment(
            "%PNames which end in 'condition' often represent the current "
            "state of something which can be in any one of three or more "
            "states. This will only be the case if you have explicitly said "
            "so, with a line like 'The rocket is either dry, fuelled or launched.' - "
            "in which case the value 'rocket condition' will always be one "
            "of 'dry', 'fuelled' or 'launched'. Note that all of this only "
            "applies to a list of three or more possibilities - a thing can "
            "have any number of either/or properties. For instance, a "
            "container is open or closed, but it also transparent or opaque. "
            "Neither of these counts as its 'condition'.");
    }
    issue_problem_end();
}
```

This code is used in §19.

§**23.** It's a bit unenlightening when an entire condition is rejected as unknown if, in fact, only one of perhaps many clauses is broken. We therefore produce quite an elaborate problem message which goes through the clauses, summing up their status in turn:

⟨Issue a problem message for a compound condition which has gone bad 23⟩ ≡

```
handmade_problem(_P_(C7CompoundConditionFailed));
issue_problem_segment(
    "In the sentence %1, I was expecting that '%2' would be a condition. "
    "It didn't make sense as one long phrase, but because it was divided up by "
    "'and'/'or', I tried breaking it down into smaller conditions, but "
    "that didn't work either. ");
if (issue_condition_error_bit(found->word_ref1, found->word_ref2))
    issue_problem_segment(
        "but that combination of conditions isn't allowed to be joined "
        "together with 'and' or 'or', because that would just be too confusing. "
        "%PFor example, 'if the player is carrying a container or a "
        "supporter' has an obvious meaning in English, but Inform reads "
        "it as two different conditions glued together: 'if the player is "
        "carrying a container', and also 'a supporter'. The meaning of "
        "the first is obvious. The second part is true if the current "
        "item under discussion is a supporter - for instance, the noun of "
        "the current action, or the item to which a definition applies. "
        "Both of these conditions are useful in different circumstances, "
        "but combining them in one condition like this makes a very "
        "misleading line of text. So Inform disallows it.");
else
    issue_problem_segment(
        "so I ran out of ideas.");
issue_problem_end();
```

This code is used in §19.

§**24.** These are cases where the wording used in the source text suggests some common misunderstanding.

⟨Issue a problem message for miscellaneous suspicious wordings 24⟩ ≡

```
int w1 = found->word_ref1 - 2, w2 = found->word_ref2;
if [[w1, w2 == number of turns]] {
    handmade_problem(_P_(C7NumberOfTurns));
    ⟨Issue the generic unknown wording message 25⟩;
    issue_problem_segment(
        "%PGenerally speaking, 'number of ...' constructions are only "
        "allowed when counting up rooms or things: if by 'number of "
        "turns' you meant the number of turns of play to date, try "
        "'turn count' instead.");
    issue_problem_end();
} else if [[found == ... is/are out of play]] {
    handmade_problem(_P_(C7OutOfPlay));
    ⟨Issue the generic unknown wording message 25⟩;
    issue_problem_segment(
        "%PPeople sometimes say that things or people removed from all "
        "rooms are 'out of play', but Inform uses the adjective "
```

```
        "'off-stage' - for instance, 'if the ball is off-stage'. "
        "If you would like 'out of play' to work, you could always "
        "write 'Definition: A thing is out of play if it is off-stage.' "
        "Then the two would be equivalent.");
    issue_problem_end();
} else if ((is_kova(spec_get_kind_of_value(expected), USEOPTION_TY)) &&
    (![[found == ... option]])) {
    handmade_problem(_P_(C7OptionlessOption));
    ⟨Issue the generic unknown wording message 25⟩;
    issue_problem_segment(
        "%PThe names of use options, on the rare occasions when they "
        "appear as values, always end with the word 'option' - for "
        "instance, we have to write 'American dialect option' not "
        "'American dialect'. As your text here doesn't end with the "
        "word 'option', perhaps you've forgotten this arcane rule?");
    issue_problem_end();
} else if ((is_kova(spec_get_kind_of_value(expected), ACTIVITY_TY)) &&
    [[found == ... of]]) {
    handmade_problem(_P_(C7ActivityOf));
    ⟨Issue the generic unknown wording message 25⟩;
    issue_problem_segment(
        "%PActivity names rarely end with 'of': for instance, when we talk "
        "about 'printing the name of something', properly speaking "
        "the activity is called 'printing the name'. Maybe that's it?");
    issue_problem_end();
} else if [[found == unicode ...]] {
    handmade_problem(_P_(C7MidTextUnicode));
    ⟨Issue the generic unknown wording message 25⟩;
    issue_problem_segment(
        "%PMaybe you intended this to produce a Unicode character? "
        "Unicode characters can be written either using their decimal "
        "numbers - for instance, 'Unicode 2041' - or with their standard "
        "names - 'Unicode Latin small ligature oe'. For efficiency reasons "
        "these names are only available if you ask for them; to make them "
        "available, you need to 'Include Unicode Character Names by Graham "
        "Nelson' or, if you really need more, 'Include Unicode Full "
        "Character Names by Graham Nelson'.");
    issue_problem_end();
} else if [[found == ... condition]] {
    handmade_problem(_P_(C7UnknownCondition));
    ⟨Issue the generic unknown wording message 25⟩;
    issue_problem_segment(
        "%PNames which end in 'condition' often represent the current "
        "state of something which can be in any one of three or more "
        "states. Names like this only work if you've declared them, with "
        "a line like 'The rocket is either dry, fuelled or launched.' - "
        "in which case the value 'rocket condition' will always be one "
        "of 'dry', 'fuelled' or 'launched'. Maybe you forgot to declare "
        "something like this, or mis-spelled the name of the owner?");
    issue_problem_end();
} else if [[found == OPENBRACE ... CLOSEBRACE]] {
    handmade_problem(_P_(C7BadConstantList));
    ⟨Issue the generic unknown wording message 25⟩;
```

```
        issue_problem_segment(
            "%PNote that lists have to be written with spaces before commas, "
            "so I like '{2, 4}' but not '{2,4}', for instance.");
        issue_problem_end();
    } else {
        handmade_problem(_P_(C7Unknown));
        ⟨Issue the generic unknown wording message 25⟩;
        issue_problem_end();
    }
```

This code is used in §19.

§**25.**

⟨Issue the generic unknown wording message 25⟩ ≡

```
    issue_problem_segment(
        "In the sentence %1, I was expecting to read %3, but instead found some "
        "text that I couldn't understand - '%2'. ");
```

This code is used in §24.

§**26. Rule (b).** This is something else that wouldn't appear in a typical compiler's typechecker. Here we are dealing with a phrase specification such as:

To attract (N - 10) things: ...

where the "N" argument will be accepted if and only if it's the value 10. The fact that Inform allows this is further evidence of the slippery way that natural language doesn't distinguish values from types; early designs of Inform didn't allow it, but many people reported this as a bug.

⟨Step (b) Deal with expected actual CONSTANT types 26⟩ ≡

```
    LOG_STAGE("(b)");
    if ((spec_is_evaluating(found)) && (spec_is_actual_CONSTANT(expected)) &&
        (found != expected)) {
        kind_of_value *kov_found = spec_evaluates_to(found);
        kind_of_value *kov_expected = spec_get_kind_of_value(expected);
        int failed = FALSE;
        if (spec_is_actual_CONSTANT(found)) {
            if (!(spec_compare_CONSTANT(found, expected))) failed = TRUE;
        } else if (is_kova(kov_found, ANY_VALUE_TY)) {
            best_result = SOMETIMES_MATCH;
            LOGIF(INVOCATIONS, "dropping to sometimes level for polymorphism\n");       see
below
        } else if (can_we_cast_kovs(kov_found, kov_expected) != NEVER_MATCH) {
            best_result = SOMETIMES_MATCH;
            LOGIF(INVOCATIONS, "dropping to sometimes level for value matching\n");
        } else failed = TRUE;
        if (failed) ⟨Fail typechecking rule (b) 28⟩;
    }
```

This code is used in §13.

§**27.** To explain the cryptic use of `ANY_VALUE_TY`:

When `found` appears to have KOV "value" – the broadest KOV that could possibly exist – what this usually means is that the KOV isn't known yet. That will be the case if `found` invokes a polymorphic arithmetic phrase, so that only typechecking (which isn't finished yet) will tell its KOV. For instance, rule (b) applied to `found` being "10km plus 20km" in the following:

```
attract 10km plus 20km things;
```

...will result in `best_result` being lowered to `SOMETIMES_MATCH`. This does *not* have the effect of accepting `found` – which is good, because it needs to be rejected – but only of saying "If we ever accept this, it will have to be subject to run-time checking."

§**28.** We fail `found` in only two circumstances: (i) where we can demonstrate that `found` is a different constant from `expected`, or (ii) where we can demonstrate that `found` and `expected` have incomparable KOVs.

⟨Fail typechecking rule (b) 28⟩ ≡

```
LOGIF(INVOCATIONS, "(b) on $S rejected: found $u, expected $u\n",
    found, kov_found, kov_expected);
THIS_IS_AN_ORDINARY_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
quote_spec(3, found);
quote_spec(4, expected);
handmade_problem(_P_(C7NotExactValueWanted));
issue_problem_segment(
    "In the sentence %1, I was expecting that '%2' would be the specific "
    "value '%4'.");
issue_problem_end();
return NEVER_MATCH;
```

This code is used in §26.

§**29. Rule (c).** This is all concerned with a shorthand far more convenient to an Inform author than it is to us – where a property's name is used without any indication of its owner.

⟨Step (c) Deal with bare property names 29⟩ ≡

```
LOG_STAGE("(c)");
if (spec_is_CONSTANT_of_kova(found, PROPERTY_TY)) {
    ⟨Step (c.1) If a bare property name is used where we expect one, accept it 30⟩;
    ⟨Step (c.2) If a bare property name is used where we expect a property value, coerce it 31⟩;
    ⟨Step (c.3) If a bare property name is used where we expect a value, coerce it if the KOVs allow 32⟩;
}
```

This code is used in §13.

§**30.** Step (c.1) is needed only to avoid rejecting this case wrongly at (c.3).

⟨Step (c.1) If a bare property name is used where we expect one, accept it 30⟩ ≡

```
LOG_STAGE("(c.1)");
if (spec_is_generic_CONSTANT_of_kova(expected, PROPERTY_TY)) return ALWAYS_MATCH;
```

This code is used in §29.

§**31.** Sometimes we definitely want a property value, for instance in a phrase explicitly set up to manipulate those.

⟨Step (c.2) If a bare property name is used where we expect a property value, coerce it 31⟩ ≡

```
LOG_STAGE("(c.2)");
if (spec_get_storage_form(expected) == PROPERTY_VALUE_SPC) {
    property_name *prn = spec_get_property_name_if_any(found);
    if (prn_is_value_property(prn)) {
        ⟨Coerce into a property of the "self" object 33⟩;
        return ALWAYS_MATCH;
    }
}
```

This code is used in §29.

§**32.** But more often we want a value which just happens in this case to come from a property. For instance, in a text routine printing a description like "The cedarwood box could hold [carrying capacity in words] item[s].", we want "carrying capacity" to be a number value, and we treat it as if it read "carrying capacity of the cedarwood box".

⟨Step (c.3) If a bare property name is used where we expect a value, coerce it if the KOVs allow 32⟩ ≡

```
LOG_STAGE("(c.3)");
if (spec_is_generic_CONSTANT(expected)) {
    property_name *prn = spec_get_property_name_if_any(found);
    if (prn_is_value_property(prn)) {
        kind_of_value *kov_if_coerced = prn_get_kind_of_value(prn);
        kind_of_value *kov_expected = spec_get_kind_of_value(expected);
        int verdict = can_we_cast_kovs(kov_if_coerced, kov_expected);
        if (verdict != NEVER_MATCH) {
            ⟨Coerce into a property of the "self" object 33⟩;
        } else {
            LOGIF(INVOCATIONS, "(c.3) declining to cast into property value form\n");
        }
        return verdict;
    }
}
```

This code is used in §29.

§**33.** The tricky part is working out what the implicitly meant object is, a classic donkey anaphora-style problem in linguistics. We don't even begin to solve that here: indeed the decision is taken rather indirectly, because we simply compile code which uses Inform 6's `self` variable to refer to the owner. The I6 library and our own run-time code conspire to ensure that `self` is always equal to something sensible.

⟨Coerce into a property of the "self" object 33⟩ ≡

```
specification *become = new_PROPERTY_VALUE_spec(spec_copy(found), new_self_object_constant());
*found = *become;
spec_set_flag(found, COERCED_SPFLAG);
LOGIF(INVOCATIONS, "(c) coercing PROPERTY to PROPERTY VALUE: $S\n", found);
```

This code is used in §31,32,31,32,31,32.

§34. Rule (d).

⟨Step (d) Deal with any attached proposition 34⟩ ≡
```
⟨Step (d.1) Typecheck any proposition found 35⟩;
⟨Step (d.2) Coerce tested proposition to NOW form 36⟩;
```

This code is used in §13.

§35. An unchecked SP can contain a proposition which, though valid as a predicate calculus sentence, makes no sense for type reasons: for instance, "the Orange Room is 10" compiles to a valid sentence but one in which the binary predicate for equality is applied to incomparable SPs. To typecheck, we must prove that any proposition needed is valid on these grounds, and we delegate that to "Type Check Propositions.w".

⟨Step (d.1) Typecheck any proposition found 35⟩ ≡
```
LOG_STAGE("(d.1)");
char *desired_to = NULL;
if (species_is(found, TEST_PROPOSITION_SPC)) desired_to = "be a condition";
if (species_is(found, DESCRIPTION_SPC)) desired_to = "be a description";
if (desired_to) {
    if (prop_type_check(spec_get_proposition(found), tc_no_problem_reporting())
        == NEVER_MATCH) {
        LOGIF(INVOCATIONS, "(d.1) on $S failed proposition type-checking\n", found);
        THIS_IS_A_GROSS_PROBLEM;
        prop_type_check(spec_get_proposition(found),
            tc_problem_reporting(found->word_ref1, found->word_ref2, desired_to));
        return NEVER_MATCH;
    }
}
```

This code is used in §34.

§36. The text in a phrase such as

now the cat is on the mat;

is parsed as a condition, which means `found` for the single argument to "now" will belong to the `CONDITION_FMY` family. But the only valid sort of condition here is a `TEST_PROPOSITION_SPC`, such as "the cat is on the mat". We want to reject other sorts of condition, and we find it convenient to coerce even a test into `NOW_PROPOSITION_SPC` form, to mark it out.

⟨Step (d.2) Coerce tested proposition to NOW form 36⟩ ≡
```
LOG_STAGE("(d.2)");
if (species_is(expected, NOW_PROPOSITION_SPC)) {
    if (species_is(found, TEST_PROPOSITION_SPC))
        coerce_TEST_PROPOSITION_to_NOW_PROPOSITION(found);
    else if (family_is(found, CONDITION_FMY))
        ⟨Issue a problem message for the wrong sort of condition in a "now" 37⟩;
}
```

This code is used in §34.

§**37.** Another deluxe problem message.

⟨Issue a problem message for the wrong sort of condition in a "now" 37⟩ ≡

```
THIS_IS_A_GROSS_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
if (species_is(found, PHRASE_TO_DECIDE_IF_SPC)) {
    handmade_problem(_P_(C7BadNow1));
    issue_problem_segment(
        "You wrote %1, but although '%2' is a condition which it is legal "
        "to test with 'if', 'when', and so forth, it is not something I "
        "can arrange to happen on request. Whether it is true or not "
        "depends on current circumstances: so to make it true, you will "
        "need to adjust those circumstances.");
    issue_problem_end();
} else if (species_is(found, LOGICAL_AND_SPC)) {
    handmade_problem(_P_(C7BadNow2));
    issue_problem_segment(
        "You wrote %1, but 'now' does not work with the condition '%2' "
        "because it can only make one wish come true at a time: so it "
        "doesn't like the 'and'. Try rewriting as two 'now's in a row?");
    issue_problem_end();
} else {
    handmade_problem(_P_(C7BadNow3));
    issue_problem_segment(
        "You wrote %1, but '%2' isn't the sort of condition which can be "
        "made to be true, in the way that 'the box is on the table' can be "
        "made true with a straightforward movement of one object (the ball).");
    issue_problem_end();
}
return NEVER_MATCH;
```

This code is used in §36.

§**38. Rule (e).** Something of a grab-bag, this one. What these three situations have in common is that all use the typechecker to clarify ambiguities in syntax.

⟨Step (e) Apply miscellaneous other coercions 38⟩ ≡

```
⟨Step (e.1) Coerce TEST ACTION to TRY ACTION 39⟩;
⟨Step (e.2) Coerce constant TEXT and TEXT ROUTINE to UNDERSTANDING 40⟩;
⟨Step (e.3) Coerce a description to a value, if we expect a noun-like description 41⟩;
⟨Step (e.4) Reject plausible but wrong uses due to use of inline-only types in phrases 44⟩;
```

This code is used in §13.

§39. An action pattern can be an action if specific enough, and this is crucial: it enables phrases such as "try taking the box" to work. When such phrases are type-checked, they expect the argument to be an `TRY_ACTION_SPC`, which is a specific action: but what they find is an `TEST_ACTION_SPC`, which may be a vague pattern-matching definition applying to many possible actions.

⟨Step (e.1) Coerce TEST ACTION to TRY ACTION 39⟩ ≡

```
LOG_STAGE("(e.1)");
if ((species_is(found, TEST_ACTION_SPC)) && (species_is(expected, TRY_ACTION_SPC))) {
    action_pattern *ap = RETRIEVE_FROM_SPEC(found, action_pattern);
    if (ap_is_unspecific(ap)) {
        THIS_IS_A_GROSSER_THAN_GROSS_PROBLEM;
        quote_source(1, current_sentence);
        quote_words(2, found->word_ref1, found->word_ref2);
        handmade_problem(_P_(C7ActionNotSpecific));
        issue_problem_segment(
            "You wrote %1, but '%2' is too vague to describe a specific action. "
            "%PIt has to be an exact instruction about what is being done, and "
            "to what. For instance, 'taking the box' is fine, but 'dropping or "
            "taking something openable' is not.");
        issue_problem_end();
        return NEVER_MATCH;
    }
    if (ap_is_overspecific(ap)) {
        THIS_IS_A_GROSSER_THAN_GROSS_PROBLEM;
        quote_source(1, current_sentence);
        quote_words(2, found->word_ref1, found->word_ref2);
        handmade_problem(_P_(C7ActionTooSpecific));
        issue_problem_segment(
            "You wrote %1, but '%2' imposes too many restrictions on the "
            "action to be carried out, by saying something about the "
            "circumstances which you can't guarantee will be true. "
            "%PSometimes this problem appears because I've misread text like "
            "'in ...' as a clause saying that the action takes place in a "
            "particular room, when in fact it was part of the name of one of "
            "the items involved. If that's the problem, try using 'let' to "
            "create a simpler name for it, and then rewrite the 'try' to use "
            "that simpler name - the ambiguity should then vanish.");
        issue_problem_end();
        return NEVER_MATCH;
    }
    coerce_TEST_ACTION_to_TRY_ACTION_by_ap(found, ap);
    return ALWAYS_MATCH;
}
```

This code is used in §38.

§**40.** The following applies only to literal text in double-quotes, which might or might not include text substitutions in square brackets: if we check it against `UNDERSTANDING_TY`, then we are trying to interpret it as a grammar to parse rather than text to print. We need to coerce since these have very different representations at run-time.

⟨Step (e.2) Coerce constant TEXT and TEXT ROUTINE to UNDERSTANDING 40⟩ ≡

```
LOG_STAGE("(e.2)");
if (((spec_is_CONSTANT_of_kova(found, TEXT_TY)) ||
    (spec_is_CONSTANT_of_kova(found, TEXT_ROUTINE_TY))) &&
    (spec_is_CONSTANT_of_kova(expected, UNDERSTANDING_TY))) {
    spec_set_flag(found, COERCED_SPFLAG);
    spec_set_kind_of_value(found, kova(UNDERSTANDING_TY));
}
```

This code is used in §38.

§**41.** Another ambiguity is that the text "women who are in lighted rooms" in:

```
let N be the number of women who are in lighted rooms;
```

...is parsed as a `DESCRIPTION_SPC`, in the `CONDITION_FMY` family. But in fact it's a noun here – it has to be a value, in fact, which can go into the "number of..." phrase as an argument. We make this happen by coercing it to a constant value.

We distinguish between two forms of noun-like description:

```
[1] to scrutinise (D - description);

[2] to scrutinise (V - value description);
```

which coerce to `kova(OBJECT_DESCRIPTION_TY)` and `kova(VALUE_DESCRIPTION_TY)` respectively. This is done because the implementation of "scrutinise" generally has to be very different when ranging over values than when ranging over objects, so we can give two different definitions and the type-checker will sort out which is needed – "scrutinise open containers" will match [1], but "scrutinise even numbers" will match [2].

⟨Step (e.3) Coerce a description to a value, if we expect a noun-like description 41⟩ ≡

```
LOG_STAGE("(e.3)");
int D_not_V = NOT_APPLICABLE;
if (spec_is_CONSTANT_of_kova(expected, OBJECT_DESCRIPTION_TY)) D_not_V = TRUE;
else if (spec_is_CONSTANT_of_kova(expected, VALUE_DESCRIPTION_TY)) D_not_V = FALSE;
if ((D_not_V != NOT_APPLICABLE) && (species_is(found, DESCRIPTION_SPC))) {
    int describes_objects = TRUE;
    kind_of_value *kov = spec_get_described_kov(found);
    if ((kov) && (can_we_cast_kovs(kov, kova(OBJECT_TY)) != ALWAYS_MATCH))
        describes_objects = FALSE;
    LOGIF(INVOCATIONS, "(e.3) D_not_V is %d, describes_objects is %d\n",
        D_not_V, describes_objects);
    ⟨Throw out the wrong sort of description with a seldom-seen problem message 42⟩;
    if (coerce_DESCRIPTION_to_VALUE(found, describes_objects) == FALSE)
        ⟨Issue a problem message for a malformed proposition in the description 43⟩;
    return ALWAYS_MATCH;
}
```

This code is used in §38.

§**42.** The following message is seldom seen since most phrases using descriptions are set up with two parallel versions. As every description matches exactly one of these, there won't be a problem. But just in case the user has intentionally defined a phrase for only one case:

⟨Throw out the wrong sort of description with a seldom-seen problem message 42⟩ ≡

```
if (D_not_V != describes_objects) {
    THIS_IS_AN_ORDINARY_PROBLEM;
    quote_source(1, current_sentence);
    quote_words(2, found->word_ref1, found->word_ref2);
    if (D_not_V == TRUE) {
        handmade_problem(_P_(C7WrongDescriptionAsNoun1));
        issue_problem_segment(
            "In the line %1, the text '%2' is given where a description of "
            "a collection of objects was required. For instance, 'rooms which "
            "contain something' would be fine, or 'closed containers' - but "
            "not a description of values which aren't objects, like 'all "
            "scenes' or 'even numbers'.");
        issue_problem_end();
    } else {
        handmade_problem(_P_(C7WrongDescriptionAsNoun2));
        issue_problem_segment(
            "In the line %1, the text '%2' is given where a description of "
            "values was required. For instance, 'recurring scenes' or 'even "
            "numbers' would be fine, but not something like 'closed containers' "
            "or 'things which are in lighted rooms' - those describe objects.");
        issue_problem_end();
    }
    return NEVER_MATCH;
}
```

This code is used in §41.

§**43.** I can't see an easy proof that this can never occur, but nor can I make it happen. The problem message is just in case someone finds a way. It appears if the description has a proposition with other than one free variable, once any universal quantifier ("all", etc.) is removed.

⟨Issue a problem message for a malformed proposition in the description 43⟩ ≡

```
THIS_IS_AN_ORDINARY_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
handmade_problem(_P_(BelievedImpossible));
issue_problem_segment(
    "In the line %1, the text '%2' is given where a description of a collection "
    "of things or values was required. For instance, 'rooms which contain "
    "something', or 'closed containers' - note that there is no need to say "
    "'all' or 'every' in this context, as that is understood already.");
issue_problem_end();
return NEVER_MATCH;
```

This code is used in §41.

§**44.** It might look as if this ought to be checked when phrase definitions are made; the trouble is, "action", "condition" and so on *are* valid in phrase definitions, but only in inline-defined ones. We don't want to get into all that here, because the message is aimed more at Inform novices who have made an understandable confusion.

⟨Step (e.4) Reject plausible but wrong uses due to use of inline-only types in phrases 44⟩ ≡

```
if (family_is(found, STORAGE_FMY)) {
    kind_of_value *kov = spec_evaluates_to(found);
    if (kov == NULL) {
        THIS_IS_AN_ORDINARY_PROBLEM;
        quote_source(1, current_sentence);
        quote_words(2, found->word_ref1, found->word_ref2);
        handmade_problem(_P_(C7PhraseParameterInlineOnly));
        issue_problem_segment(
            "In the line %1, '%2' ought to be a value, but isn't - there must be "
            "something fishy about the way it was created. %P"
            "Usually this happens because it is one of the named items in "
            "a phrase definition, but stood for a chunk of text which can't "
            "be a value - for instance, 'To marvel at (feat - an action)' "
            "doesn't make 'feat' a value. (Calling it a 'stored action' "
            "would have been fine; and similarly, if you want something "
            "which is either true or false, use 'truth state' not 'condition'.)");
        issue_problem_end();
        return NEVER_MATCH;
    }
}
```

This code is used in §38.

§**45. Rule (f).** The "main rule" is, as we shall see, that `found` should have the same species as `expected`, or if `expected` give no species then at least it should have the same family. The two exceptional cases are when `expected` is a description such as "an even number", or the name of a kind of value such as "a scene", in which case we allow `found` if it's a value which meets these requirements.

⟨Step (f) The Main Rule of Type-Checking 45⟩ ≡

```
int exceptional_case = FALSE;
⟨Step (f.1) Exception: when expecting a DESCRIPTION 46⟩;
⟨Step (f.2) Exception: when expecting a generic or actual CONSTANT 49⟩;
if (exceptional_case == FALSE) ⟨Step (f.3) Main rule 51⟩;
```

This code is used in §13.

§**46.** Suppose we are matching the parameter of a phrase like this:

To inspect (D - an open door): ...

Then we might have `found` set to a constant value – "the Marble Portal", say – and `expected` would be a `DESCRIPTION_SPC` for open doors.

What we must do is to see whether `found` is a value that satisfies the description. Any match will be at best at the "sometimes" level, since by definition a `DESCRIPTION_SPC` is something we can't always prove that a value meets until run-time.

⟨Step (f.1) Exception: when expecting a DESCRIPTION 46⟩ ≡

```
LOG_STAGE("(f.1)");
if ((species_is(expected, DESCRIPTION_SPC)) &&
    (spec_is_actual(expected)) &&
    (found != expected)) {
    if ((spec_is_nothing_object_constant(found)) &&
        (spec_get_described_kind(expected)))
        ⟨Disallow "nothing" as a match for a description requiring a kind of object 47⟩;
    int rv = NEVER_MATCH;
    if (!(species_is(found, DESCRIPTION_SPC)))
        rv = can_we_match_value_descriptions(found, expected);
    if (rv == NEVER_MATCH)
        ⟨Fail on the grounds that the description has not been met 48⟩;
    best_result = rv;
    if (rv == SOMETIMES_MATCH)
        LOGIF(INVOCATIONS, "dropping to sometimes level for description\n");
    exceptional_case = TRUE;
}
```

This code is used in §45.

§**47.** "You can't have/ Something for nothing," as Canadian power-trio *Rush* tell us with the air of having just made a great discovery; well, you can't have "nothing" for something, either –

⟨Disallow "nothing" as a match for a description requiring a kind of object 47⟩ ≡

```
THIS_IS_AN_ORDINARY_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
quote_object(3, spec_get_described_kind(expected));
handmade_problem(_P_(C7NothingForSomething));
issue_problem_segment(
    "You wrote %1, but '%2' is literally no thing, and it consequently does "
    "not count as %3.");
issue_problem_end();
return NEVER_MATCH;
```

This code is used in §46.

§**48.** More orthodox failures:

⟨Fail on the grounds that the description has not been met 48⟩ ≡

```
THIS_IS_AN_ORDINARY_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
if (spec_get_described_kind(expected)) quote_object(3, spec_get_described_kind(expected));
else quote_words(3, expected->word_ref1, expected->word_ref2);
if (species_is(found, DESCRIPTION_SPC)) {
    LOG("Description found: $S\nDescription expected: $S\n", found, expected);
    if (spec_get_proposition(found)) {
        handmade_problem(_P_(C7GenericDescription));
        issue_problem_segment(
            "In the line %1, I read '%2' where I was expecting to find a "
            "(single) specific example of '%3'. Although what you wrote did "
            "make sense as a description, it could refer to many different "
            "values or to none, so it wasn't specific enough.");
        issue_problem_end();
    } else {
        handmade_problem(_P_(C7NotSpecificDescription2));
        issue_problem_segment(
            "In the line %1, I read '%2' where I was expecting to find a "
            "(single) specific example of '%3'.");
        issue_problem_end();
    }
} else {
    if (spec_is_actual_CONSTANT_of_kova(found, OBJECT_TY))
        quote_object(4, OBJECT_spec_to_world_object(found)->kind);
    else
        quote_spec(4, found);
    handmade_problem(_P_(C7WrongKind));
    issue_problem_segment(
        "You wrote %1, but '%2' has the wrong kind of value to go there: %4 "
        "instead of %3.");
    issue_problem_end();
}
return NEVER_MATCH;
```

This code is used in §46.

§**49.** Now for the related, but slightly simpler, case of matching the name of a KOV. Suppose we are parsing "award 5 points" against

> To award (N - a number) points: ...

Here `found` will be the actual constant value 5, and `expected` the generic constant value with KOV "number".

A phrase which returns a value must have its own return value's data type checked. Unfortunately we can't do that yet: we want to wait until recursive type-checking has removed incorrect invocations before drawing a conclusion about the return data type of the phrase.

⟨Step (f.2) Exception: when expecting a generic or actual CONSTANT 49⟩ ≡

```
LOG_STAGE("(f.2)");
if ((species_is(expected, CONSTANT_SPC)) &&
    (spec_is_evaluating(found)) &&
    (found != expected)) {
    if (species_is(found, PHRASE_TO_DECIDE_VALUE_SPC)) {
        LOGIF(INVOCATIONS, "(f.2) exempting phrase from return value checking for now\n");
    } else {
        switch (can_we_match_value_descriptions(found, expected)) {
            case NEVER_MATCH: ⟨Fail with a mismatched value problem message 50⟩;
            case SOMETIMES_MATCH:
                best_result = SOMETIMES_MATCH;
                LOGIF(INVOCATIONS, "dropping to sometimes level\n");
                break;
            case ALWAYS_MATCH: break;
        }
    }
    exceptional_case = TRUE;
}
```

This code is used in §45.

§**50.** This is the error message a typical C compiler's type-checker would issue; it says the value has the wrong kind.

⟨Fail with a mismatched value problem message 50⟩ ≡

```
THIS_IS_AN_ORDINARY_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
quote_type_of(3, expected);
if (species_is(found, LOCAL_VARIABLE_SPC)) {
    kind_of_value *kov = kov_of_local(spec_get_data(found, LOCAL_NUMBER_SPDATA));
    quote_kov(4, kov);
    handmade_problem(_P_(C7LocalMismatch));
    issue_problem_segment(
        "You wrote %1, but '%2' is a temporary name for %4 (created by 'let' "
        "or 'repeat'), whereas I was expecting to find %3 there.");
    issue_problem_end();
} else {
    quote_kov(4, spec_evaluates_to(found));
    handmade_problem(_P_(C7TypeMismatch));
    issue_problem_segment(
        "You wrote %1, but '%2' has the wrong kind of value: %4 rather than %3.");
```

```
        issue_problem_end();
    }
    return NEVER_MATCH;
```

This code is used in §49.

§**51.** We now apply the main rule, supposing that neither of the exceptional cases has intervened to stop us getting here. The found and expected specifications must have the same family and, unless the expected species is `UNKNOWN`, the same species as well.

⟨Step (f.3) Main rule 51⟩ ≡

```
    LOG_STAGE("(f.3)");
    int mtf = spec_get_family(found), stf = spec_get_species(found);
    int mte = spec_get_family(expected), ste = spec_get_species(expected);
    if ((mtf != mte) || ((ste != UNKNOWN) && (stf != ste)))
        ⟨Fail with a catch-all typechecking problem message 52⟩;
```

This code is used in §45.

§**52.** This is the general-purpose Problem message to which the type-checker resorts when it has nothing more specific to say.

⟨Fail with a catch-all typechecking problem message 52⟩ ≡

```
    THIS_IS_AN_ORDINARY_PROBLEM;
    quote_source(1, current_sentence);
    quote_words(2, found->word_ref1, found->word_ref2);
    quote_type_of(3, expected);
    quote_type_of(4, found);
    handmade_problem(_P_(BelievedImpossible));          well, at any rate I haven't seen it lately
    issue_problem_segment(
        "You wrote %1, but '%2' seems to be %4, whereas I was expecting to "
        "find %3 there.");
    issue_problem_end();
    return NEVER_MATCH;
```

This code is used in §51.

§**53. (g) Recurse down to check arguments.** We have now agreed that the current node in the type tree checks correctly. It remains only to see that any arguments also check, and this is where recursion comes in.

⟨Step (g) Recurse down to check arguments 53⟩ ≡

```
    LOG_STAGE("(g)");
    if ((species_is(found, LOGICAL_AND_SPC)) ||
        (species_is(found, LOGICAL_OR_SPC)))
        ⟨Step (g.1) Check individual conditions in a compound one 54⟩
    else {
        int i, an_arg_failed = FALSE;
        for (i=0; i<spec_get_argc(found); i++) {
            specification *ith_argument = spec_get_argument(found, i);
            if (ith_argument) {
                int rv = NEVER_MATCH;
                if (!(spec_is_UNKNOWN(ith_argument)))
                    rv = typecheck_recursive(ith_argument, ith_argument, 2, depth+1);
                if (rv != ALWAYS_MATCH) an_arg_failed = TRUE;
```

```
            }
        }
        if (species_is(found, LIST_ENTRY_SPC))
            ⟨Step (g.2) Check arguments of a list entry 55⟩;
        if (species_is(found, PROPERTY_VALUE_SPC))
            ⟨Step (g.3) Check arguments of a property value 56⟩;
        if (an_arg_failed) return NEVER_MATCH;
    }
```

This code is used in §13.

§**54.** LOGICAL_AND_SPC and LOGICAL_OR_SPC each take two arguments, both conditions, and we want to typecheck those against "condition".

⟨Step (g.1) Check individual conditions in a compound one 54⟩ ≡

```
    LOG_STAGE("(g.1)");
    specification *cond_spec = new_generic_CONDITION_type();
    if (typecheck_recursive(spec_get_argument(found, 0),
        cond_spec, problem_threshold, depth+1) != ALWAYS_MATCH)
            return NEVER_MATCH;
    if (typecheck_recursive(spec_get_argument(found, 1),
        cond_spec, problem_threshold, depth+1) != ALWAYS_MATCH)
            return NEVER_MATCH;
```

This code is used in §53.

§**55.** For a list entry, we have to have a list and an index.

⟨Step (g.2) Check arguments of a list entry 55⟩ ≡

```
    LOG_STAGE("(g.2)");
    kind_of_value *kov1 = spec_evaluates_to(spec_get_argument(found, 0));
    kind_of_value *kov2 = spec_evaluates_to(spec_get_argument(found, 1));
    if (kovcon_get_base(kov1) == NULL) {
        THIS_IS_A_GROSSER_THAN_GROSS_PROBLEM;
        sentence_problem(_P_(C7EntryOfNonList),
            "that doesn't make sense to me as a list entry",
            "since the entry is taken from something which isn't a list.");
        return NEVER_MATCH;
    }
    if (is_kova(kov2, NUMBER_TY) == FALSE) {
        THIS_IS_A_GROSSER_THAN_GROSS_PROBLEM;
        sentence_problem(_P_(C7NonNumericListEntry),
            "that doesn't make sense to me as a list entry",
            "because the indication of which entry is not a number. "
            "For instance, 'entry 3 of L' is allowed, but not 'entry "
            "\"six\" of L'. (List entries are numbered 1, 2, 3, ...)");
        return NEVER_MATCH;
    }
```

This code is used in §53.

§**56.** For a property value, we have to have a property and an owner (perhaps an object, perhaps a value).

⟨Step (g.3) Check arguments of a property value 56⟩ ≡
```
LOG_STAGE("(g.3)");
specification *the_property = spec_get_argument(found, 0);
kind_of_value *kov1 = spec_evaluates_to(the_property);
if (is_kova(kov1, PROPERTY_TY) == FALSE) ⟨Issue a "not a property" problem message 57⟩;
property_name *prn = PROPERTY_spec_to_property_name(the_property);
if (prn == NULL)
    internal_error("null property name in type checking");
if (prn_is_either_or(prn)) ⟨Issue a "not a value property" problem message 58⟩;
specification *the_owner = spec_get_argument(found, 1);
kind_of_value *kov2 = spec_evaluates_to(the_owner);
if (kov2) {
    int owner_is_object = is_kova(kov2, OBJECT_TY);
    int owner_should_be_value = prn_belongs_to_value(prn);
    if ((owner_should_be_value == FALSE) && (owner_is_object == FALSE))
        ⟨Issue a problem message for using a value's property of an object 59⟩;
    if ((owner_should_be_value) && (owner_is_object))
        ⟨Issue a problem message for using an object's property of a value 60⟩;
} else {
    ⟨Issue a problem message for being too vague about the owner 61⟩;
}
```

This code is used in §53.

§**57.** Inform constructs property-value specifications quite carefully, and I think it's only possible for the typechecker to see one where the property isn't a property when recovering from other problems.

⟨Issue a "not a property" problem message 57⟩ ≡
```
THIS_IS_A_GROSSER_THAN_GROSS_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
quote_words(3, the_property->word_ref1, the_property->word_ref2);
handmade_problem(_P_(C7PropertyOfNonProperty));
issue_problem_segment(
    "In the sentence %1, it looks as if you intend '%2' to be a property "
    "of something, but there is no such property as '%3'.");
issue_problem_end();
return NEVER_MATCH;
```

This code is used in §56.

§58.

⟨Issue a "not a value property" problem message 58⟩ ≡
```
THIS_IS_A_GROSSER_THAN_GROSS_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
handmade_problem(_P_(C7EitherOrAsValue));
issue_problem_segment(
    "In the sentence %1, it looks as if you intend '%2' to be the value "
    "of a property of something, but that property has no value: it's "
    "something which an object either is or is not.");
issue_problem_end();
return NEVER_MATCH;
```

This code is used in §56.

§59.

⟨Issue a problem message for using a value's property of an object 59⟩ ≡
```
THIS_IS_A_GROSSER_THAN_GROSS_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
if (the_owner->word_ref1 >= 0) {
    quote_words(3, the_owner->word_ref1, the_owner->word_ref2);
    handmade_problem(_P_(C7PropertyOfNonObject));
    issue_problem_segment(
        "In the sentence %1, it looks as if you intend '%2' to be a property, "
        "but '%3' is not an object.");
    issue_problem_end();
} else {
    handmade_problem(_P_(BelievedImpossible));
    issue_problem_segment(
        "In the sentence %1, it looks as if you intend '%2' to be a property, "
        "but if so it is unclear to what object it would belong.");
    issue_problem_end();
}
return NEVER_MATCH;
```

This code is used in §56.

§60.

⟨Issue a problem message for using an object's property of a value 60⟩ ≡
```
THIS_IS_A_GROSSER_THAN_GROSS_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, prn->word_ref1, prn->word_ref2);
handmade_problem(_P_(C7PropertyOfObject));
issue_problem_segment(
    "In the sentence %1, it looks as if you are applying '%2' "
    "to an object, but it is really a property of a value.");
issue_problem_end();
return NEVER_MATCH;
```

This code is used in §56.

§**61.**

⟨Issue a problem message for being too vague about the owner 61⟩ ≡

```
THIS_IS_A_GROSSER_THAN_GROSS_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, found->word_ref1, found->word_ref2);
if (spec_get_described_kind(the_owner))
    quote_object(3, spec_get_described_kind(the_owner));
else
    quote_words(3, the_owner->word_ref1, the_owner->word_ref2);
handmade_problem(_P_(C7PropertyOfKind));
if (found->word_ref1 >= 0)
    issue_problem_segment(
        "In the sentence %1, it looks as if you intend '%2' to be a property, "
        "but '%3' is not specific enough about who or what the owner is. ");
else
    issue_problem_segment(
        "In the sentence %1, it looks as if you intend to look up a property "
        "of something, but '%3' is not specific enough about who or what "
        "the owner is. ");
issue_problem_segment(
    "%PSometimes this mistake is made because Inform mostly doesn't understand "
    "the English language habit of referring to something indefinite by a "
    "common noun - for instance, writing 'change the carrying capacity of "
    "the container to 10' throws Inform because it doesn't understand "
    "that 'the container' means one which has been discussed recently.");
issue_problem_end();
return NEVER_MATCH;
```

This code is used in §56.

§**62.** Rule (h). It's a bit of a curiosity that this is done in the type-checker, but here we go, just the same. We will reject the use of "end if" outside of an "if", and apply similar constraints on other structural phrases.

⟨Step (h) Police the global structure of phrase bodies 62⟩ ≡

```
LOG_STAGE_LEFT("(h)");
char *block_name = name_of_current_block();
LOGIF(INVOCATIONS, "Current block: <%s>\n", (block_name)?(block_name):"none");
⟨Step (h.1) Rulebook outcomes are legal only in rules belonging to rulebooks having them 63⟩;
⟨Step (h.2) "Otherwise" can only be used in an "if" 64⟩;
⟨Step (h.3) A switch case must have a value of the right kind 65⟩;
⟨Step (h.4) An "end structure" phrase can only occur at the end of the right structure 66⟩;
```

This code is used in §13.

§63.

⟨Step (h.1) Rulebook outcomes are legal only in rules belonging to rulebooks having them 63⟩ ≡

```
if (species_is(found, RULEBOOK_OUTCOME_PHRASE_SPC)) {
    named_rulebook_outcome *nrbo = RETRIEVE_FROM_SPEC(found, named_rulebook_outcome);
    if (rb_allow_outcome(nrbo)) return ALWAYS_MATCH;
    THIS_IS_AN_ORDINARY_PROBLEM;
    sentence_problem(_P_(C7MisplacedRulebookOutcome),
        "this is a rulebook outcome which can only be used in rules "
        "belonging to rulebooks which recognise it",
        "and is not meaningful in general phrases or other rules. (For "
        "instance, the outcome 'persuasion succeeds' is allowed in a persuasion "
        "rule, but not anywhere else.)");
    return NEVER_MATCH;
}
```

This code is used in §62.

§64.

⟨Step (h.2) "Otherwise" can only be used in an "if" 64⟩ ≡

```
if (species_is(found, OTHERWISE_SPC)) {
    if (block_name == NULL) {
        THIS_IS_AN_ORDINARY_PROBLEM;
        sentence_problem(_P_(C7OtherwiseWithoutIf),
            "this is an 'else' or 'otherwise' with no matching 'if' (or 'unless')",
            "which must be wrong.");
        return NEVER_MATCH;
    }
    if ((strcmp(block_name, "if") != 0) && (strcmp(block_name, "unless") != 0)) {
        THIS_IS_AN_ORDINARY_PROBLEM;
        quote_source(1, current_sentence);
        quote_text(2, block_name);
        handmade_problem(_P_(C7OtherwiseInNonIf));
        issue_problem_segment(
            "The 'else' or 'otherwise' here did not make sense inside a "
            "'%2' structure: it's provided for 'if' (or 'unless').");
        issue_problem_end();
        return NEVER_MATCH;
    }
}
```

This code is used in §62.

§**65.**

⟨Step (h.3) A switch case must have a value of the right kind 65⟩ ≡

```
if ((species_is(found, CASE_SPC)) && (spec_get_argc(found) > 0)) {
    kind_of_value *switch_kov = kov_for_current_block();
    kind_of_value *case_kov = spec_get_kind_of_value(spec_get_argument(found, 0));
    LOGIF(INVOCATIONS, "(h.3) switch kov is $u, case kov is $u\n", switch_kov, case_kov);
    if (case_kov == NULL) {
        THIS_IS_AN_ORDINARY_PROBLEM;
        quote_source(1, current_sentence);
        quote_kov(2, switch_kov);
        handmade_problem(_P_(C7CaseValueNonConstant));
        issue_problem_segment(
            "The case %1 is required to be a constant value, rather than "
            "something which has different values at different times: "
            "specifically, it has to be %2.");
        issue_problem_end();
        return NEVER_MATCH;
    }
    if (can_we_cast_kovs(case_kov, switch_kov) != ALWAYS_MATCH) {
        THIS_IS_AN_ORDINARY_PROBLEM;
        quote_source(1, current_sentence);
        quote_kov(2, case_kov);
        quote_kov(3, switch_kov);
        handmade_problem(_P_(C7CaseValueMismatch));
        issue_problem_segment(
            "The case %1 has the wrong kind of value for the possibilities "
            "being chosen from: %2 instead of %3.");
        issue_problem_end();
        return NEVER_MATCH;
    }
}
```

This code is used in §62.

§**66.** This used to be a much-seen problem message, until Inform moved to Pythonesque structure-by-indentation. Nowadays "end if", "end while" and such are automatically inserted into the stream of commands, always in the right place, and always passing these checks. But the problem messages are kept for the sake of old-format source text, and for refuseniks.

⟨Step (h.4) An "end structure" phrase can only occur at the end of the right structure 66⟩ ≡

```
if (species_is(found, END_BLOCK_SPC)) {
    if (block_name == NULL) {
        THIS_IS_AN_ORDINARY_PROBLEM;
        if (problem_count == 0)
            sentence_problem_with_note(_P_(C7EndWithoutBegin),
                "this is an 'end' with no matching 'begin'",
                "which should not happen: every phrase like 'if ... begin;' "
                "should eventually be followed by its bookend 'end if'. "
                "It makes no sense to have an 'end ...' on its own.",
                "Perhaps the problem is actually that you opened several "
                "such begin... end 'blocks' and accidentally closed them "
                "once too many? This is very easily done.");
        else problem_count++;
```

```
            return NEVER_MATCH;
        } else {
            phrase *ended = RETRIEVE_FROM_SPEC(found, phrase);
            if (strcmp(lw_array[ended->type_data.words[0]].lw_text, block_name) != 0) {
                THIS_IS_AN_ORDINARY_PROBLEM;
                quote_source(1, current_sentence);
                quote_text(2, block_name);
                quote_source(3, start_of_current_block());
                handmade_problem(_P_(C7WrongEnd));
                issue_problem_segment(
                    "You wrote %1, but the end I was expecting next was 'end %2', "
                    "finishing the block you began with %3.");
                issue_problem_end();
                return NEVER_MATCH;
            }
        }
    }
```

This code is used in §62.

§**67.** Rule (i). This is the big one. It's applied whenever `found` has an invocation list.

```
⟨Step (i) Typecheck the invocations, choosing among them 67⟩ ≡
    invocation_list *found_invl = spec_invocation_list(found);
    invocation *list_of_invocations_to_test[MAX_INVOCATIONS_PER_PHRASE];
    int test_count = 0;
    int issue_interesting_problems = FALSE;
    int current_group = -1;
    int invocation_list_passes = FALSE;
    phrase *ph;
    ⟨Step (i.1) Set up the list of invocations to test 68⟩;
    ⟨Step (i.2) Type-check each invocation in turn 70⟩;
    if (problems_issued_during_typechecking()) return NEVER_MATCH;
    if (invocation_list_passes) ⟨Step (i.3) Preserve successful invocations 96⟩
    else ⟨Step (i.4) All invocations failed error 104⟩;
```

This code is used in §13.

§68. Phrase definitions are kept in a linked list with a total ordering which properly contains the partial ordering in which $P_1 \leq P_2$ if they are lexically identical and if each parameter of P_1 provably, at compile time, also satisfies the requirements for the corresponding parameter of P_2. They have already been lexically parsed in that order, so the list of invocations (which will have accumulated during parsing) is also in that same order. Now this is nearly the correct order for type-checking. But we make one last adjustment: the phrase being compiled is moved to the back of the list. This is to make recursion always the last thing checked, so that later rules can override earlier ones but still make use of them.

The following amounts to an $O(G^2)$ sorting algorithm, where G is the number of groups in the invocation list, but the coefficient is very low and G small: for everything except text, $G = 1$, and otherwise G is the number of pieces of text or substitutions in a double-quoted literal text. This seldom exceeds 10, in practice.

⟨Step (i.1) Set up the list of invocations to test 68⟩ ≡

```
LOG_STAGE("(i.1)");
INVOCATION_VARIABLE(inv);
int g = -1, g_max = 0;
for (g = 0; g <= g_max; g++) {
    LOOP_THROUGH_INVOCATION_LIST(inv, found_invl) {
        int n = inv_get_group(inv);
        if (n > g_max) g_max = n;
        if ((g == n) && (inv->phrase_invoked != phrase_being_compiled))
            ⟨Add this invocation to the list of test cases 69⟩;
    }
    LOOP_THROUGH_INVOCATION_LIST(inv, found_invl) {
        int n = inv_get_group(inv);
        if ((g == n) && (inv->phrase_invoked == phrase_being_compiled))
            ⟨Add this invocation to the list of test cases 69⟩;
    }
}
if (test_count != length_of_invocation_list(found_invl))
    internal_error("invocation list broken");
```

This code is used in §67.

§69.

⟨Add this invocation to the list of test cases 69⟩ ≡

```
if (test_count >= MAX_INVOCATIONS_PER_PHRASE) internal_error("overrun");
list_of_invocations_to_test[test_count++] = inv;
inv_clear_flag(inv, TESTED_INVFLAG);
inv_clear_flag(inv, PASSED_INVFLAG);
inv_clear_flag(inv, UNPROVEN_INVFLAG);
inv_clear_flag(inv, GROSSLY_FAILED_INVFLAG);
inv_clear_flag(inv, INTERESTING_PROBLEM_INVFLAG);
```

This code is used in §68.

§**70.** Now we work through the list of tests, which is in a sequence of groups numbered 0, 1, 2, ... Each group must produce at least one invocation passing at least at the "sometimes" level marked by the `UNPROVEN_INVFLAG`, or else the whole specification fails its match. Within each group, the first proven match stops our work, since we can never need lower-priority interpretations.

⟨Step (i.2) Type-check each invocation in turn 70⟩ ≡

```
    LOG_STAGE("(i.2)");
    STREAM_INDENT(dl);
    int ref, skip_rest_of_group = FALSE, passed_in_group = 0;
    for (ref = 0, current_group = -1; ref<test_count; ref++) {
        invocation *inv = list_of_invocations_to_test[ref];
        int first_in_group = FALSE, last_in_group = FALSE;
        ⟨Is this invocation the first or last in its group? 71⟩;
        if (first_in_group) {
            current_group = inv_get_group(inv);
            skip_rest_of_group = FALSE; passed_in_group = 0;
            LOGIF(INVOCATIONS, "Group G%d begins:\n", current_group);
        }
        if (skip_rest_of_group) continue;
        ⟨Test the current invocation and set its results flags accordingly 74⟩;
        ⟨Describe the result for this particular invocation 72⟩;
        if (inv_test_flag(inv, PASSED_INVFLAG)) {
            invocation_list_passes = TRUE;
            passed_in_group++;
            if ((last_in_group) || (inv_test_flag(inv, UNPROVEN_INVFLAG) == FALSE)) {
                LOGIF(INVOCATIONS, "Group G%d passed%s\n", current_group,
                    (inv_test_flag(inv, UNPROVEN_INVFLAG))?" subject to run-time checking":"");
                skip_rest_of_group = TRUE;
            }
        } else {
            if ((last_in_group) && (passed_in_group == 0)) {
                LOGIF(INVOCATIONS, "Group G%d failed\n", current_group);
                invocation_list_passes = FALSE;                          to force error
                break;
            }
        }
        if (problems_issued_during_typechecking()) break;       to prevent duplication of problem
messages
    }
    STREAM_OUTDENT(dl);
    ⟨Summarise the results in the debugging log 73⟩;
```

This code is used in §67.

§71.

⟨Is this invocation the first or last in its group? 71⟩ ≡
```
if ((ref+1 == test_count) ||
    (inv_get_group(list_of_invocations_to_test[ref+1]) != inv_get_group(inv)))
    last_in_group = TRUE;
if ((ref-1 < 0) ||
    (inv_get_group(list_of_invocations_to_test[ref-1]) != inv_get_group(inv)))
    first_in_group = TRUE;
```

This code is used in §70.

§72.

⟨Describe the result for this particular invocation 72⟩ ≡
```
char *verdict = "Failed";
if (inv_test_flag(inv, PASSED_INVFLAG)) verdict = "Passed";
if (inv_test_flag(inv, UNPROVEN_INVFLAG)) verdict = "Passed (with qualifications)";
if (inv_test_flag(inv, GROSSLY_FAILED_INVFLAG)) verdict = "Failed grossly";
LOGIF(INVOCATIONS, "%s: P%d: $e\n", verdict, ref, inv);
```

This code is used in §70.

§73.

⟨Summarise the results in the debugging log 73⟩ ≡
```
int g = -1, i;
for (i=0; i<test_count; i++) {
    invocation *inv = list_of_invocations_to_test[i];
    if (g != inv_get_group(inv)) { g = inv_get_group(inv); LOGIF(INVOCATIONS, "| "); }
    if (inv_test_flag(inv, TESTED_INVFLAG) == FALSE) { LOGIF(INVOCATIONS, "- "); }
    else if (inv_test_flag(inv, UNPROVEN_INVFLAG)) { LOGIF(INVOCATIONS, "? "); }
    else if (inv_test_flag(inv, GROSSLY_FAILED_INVFLAG)) { LOGIF(INVOCATIONS, "g "); }
    else if (inv_test_flag(inv, PASSED_INVFLAG)) { LOGIF(INVOCATIONS, "p "); }
    else { LOGIF(INVOCATIONS, "f "); }
}
LOGIF(INVOCATIONS, "|\n");
```

This code is used in §70.

§74.

⟨Test the current invocation and set its results flags accordingly 74⟩ ≡
```
int invocation_passed = TRUE, with_qualifications = FALSE;
int no_gross_problems_thrown_before = no_gross_problems_thrown;
LOGIF(INVOCATIONS, "(P%d) Trying invocation <$W>: $e\n",
    ref, inv->invocation_w1, inv->invocation_w2, inv);
⟨Type-check this particular invocation 75⟩;
inv_set_flag(inv, TESTED_INVFLAG);
if (invocation_passed) {
    inv_set_flag(inv, PASSED_INVFLAG);
    if (with_qualifications) inv_set_flag(inv, UNPROVEN_INVFLAG);
} else if (no_gross_problems_thrown > no_gross_problems_thrown_before)
    inv_set_flag(inv, GROSSLY_FAILED_INVFLAG);
```

This code is used in §70,107,70,107,70.

§**75.** We can now forget about groups, and the rest of the list, and concentrate only on the current invocation `inv`. To fail it, we have to clear the `invocation_passed` flag; the failure will be deemed gross if and only if gross problem messages are called for during the checking process. We accept matches of arguments at the "sometimes" level, but need to set the `with_qualifications` flag if any arise.

We are *not* allowed to return from the routine here, because no matter how bad this particular invocation looks, others might be better. So we issue problem messages using the following macro only.

```
define THIS_IS_AN_INTERESTING_PROBLEM
    inv_set_flag(inv, INTERESTING_PROBLEM_INVFLAG);
    invocation_passed = FALSE;
    if (issue_interesting_problems)
```

⟨Type-check this particular invocation 75⟩ ≡

```
    ph = inv->phrase_invoked;
    inv->kov_resulting = phtd_get_return_kov(&(ph->type_data));
    if (phtd_is_any_let_phrase(&(ph->type_data))) {
        requested.local_initial_value = inv_get_token_as_parsed(inv, 1);
        requested.recently_discovered_KOV = NULL;
        LOGIF(INVOCATIONS, "Setting let-value to $S\n", requested.local_initial_value);
    }
    int arithmetical_phrase = FALSE;
    if ((ph->type_data.arithmetical_operation >= 0) &&
        (ph->type_data.arithmetical_operation < NO_OPERATIONS)) arithmetical_phrase = TRUE;
    are the arguments of the right kind?
    if (invocation_passed) ⟨Step (i.2.1) Take care of polymorphic phrases 76⟩;
    if (invocation_passed) ⟨Step (i.2.2) Take care of monomorphic phrases 84⟩;
    if (invocation_passed) ⟨Step (i.2.3) Match type templates in the argument specifications 85⟩
    if (invocation_passed) ⟨Step (i.2.4) Match KOVs in assignment phrases 86⟩;
    if this evaluates something, is it a value of the right kind?
    if (invocation_passed) ⟨Step (i.2.5) Check kind of value returned 92⟩;
    are there any special rules about invoking this phrase?
    if (invocation_passed) ⟨Step (i.2.6) Check any phrase options 93⟩;
    if (invocation_passed) ⟨Step (i.2.7) Worry about self in say property of 94⟩;
    if (invocation_passed) ⟨Step (i.2.8) Worry about using a phrase outside of the control structure it belongs to 95⟩;
```

This code is used in §74.

§**76.** "Polymorphic" here means that the phrase (i) produces a value, and (ii) that the kind of this value depends on the kinds of its arguments. Inform supports only a few polymorphic phrases, all clearly declared as such in the Standard Rules, and they come in two sorts: those marked with a "polymorphism exception", and those marked as "arithmetic operations".

⟨Step (i.2.1) Take care of polymorphic phrases 76⟩ ≡

```
LOG_STAGE("(i.2.1)");
switch (ph->type_data.polymorphism_exception) {
    case -1: break;                          this is the default value, for unexceptional phrases
    case 0: ⟨Step (i.2.1.1) "List of D" has KOV list-of-K 77⟩; break;
    case 1: ⟨Step (i.2.1.2) "A random D" has KOV K 78⟩; break;
    default: internal_error("Unknown polymorphism exception"); break;
}
if (ph->type_data.arithmetical_operation == TOTAL_OPERATION)
    ⟨Step (i.2.1.3) "Total P of O" has KOV the KOV of P 79⟩
else if (arithmetical_phrase) ⟨Step (i.2.1.4) Dimension-check arithmetic phrases 81⟩;
```

This code is used in §75.

§**77.** Exception 0 arises for "list of D", where D describes some domain: for instance, "the list of colours", which constructs the list of all valid colours and returns it as a value. Clearly "list of D" returns a value which needs to be "list of K" where K is the KOV of the members of the domain D – hence the polymorphism.

⟨Step (i.2.1.1) "List of D" has KOV list-of-K 77⟩ ≡

```
LOG_STAGE("(i.2.1.1)");
specification *domain_description = inv_get_token_as_parsed(inv, 0);
kind_of_value *K;
if (spec_is_generic_CONSTANT(domain_description))
    K = kov_when_VALUE_is_evaluated(domain_description);
else K = spec_get_described_kov(domain_description);
inv->kov_resulting = kov_list_of(K);
requested.recently_discovered_KOV = inv->kov_resulting;
LOGIF(INVOCATIONS, "'List of D' polymorphism exception: $u\n", inv->kov_resulting);
```

This code is used in §76.

§**78.** Exception 1 is similar, but simpler: the KOV of "a random D" is the same as the KOV of the members of the domain D.

⟨Step (i.2.1.2) "A random D" has KOV K 78⟩ ≡

```
LOG_STAGE("(i.2.1.2)");
specification *domain_description = inv_get_token_as_parsed(inv, 0);
kind_of_value *K;
if (spec_is_generic_CONSTANT(domain_description))
    K = kov_when_VALUE_is_evaluated(domain_description);
else K = spec_get_described_kov(domain_description);
inv->kov_resulting = K;
requested.recently_discovered_KOV = inv->kov_resulting;
LOGIF(INVOCATIONS, "'A random D' polymorphism exception: $u\n", inv->kov_resulting);
```

This code is used in §76.

§**79.** For instance, the KOV of "total carrying capacity of people in the Dining Room" is a number, because the KOV of the property "carrying capacity" is "number".

⟨Step (i.2.1.3) "Total P of O" has KOV the KOV of P 79⟩ ≡
```
LOG_STAGE("(i.2.1.3)");
specification *P = inv_get_token_as_parsed(inv, 0);
if (spec_is_CONSTANT_of_kova(P, PROPERTY_TY)) {
    property_name *prn = PROPERTY_spec_to_property_name(P);
    if (prn_is_value_property(prn)) inv->kov_resulting = prn_get_kind_of_value(prn);
    else {
        THIS_IS_AN_INTERESTING_PROBLEM {
            sentence_problem(_P_(C7TotalEitherOr),
                "this seems to be an attempt to total up an either/or property",
                "and by definition such a property has nothing to total.");
        }
    }
} else ⟨Fail the invocation for totalling something other than a property 80⟩;
```

This code is used in §76.

§**80.** The problem message here is to help what turns out to be quite a popular mistake. (Perhaps we should simply implement column-totalling and be done with it.)

⟨Fail the invocation for totalling something other than a property 80⟩ ≡
```
if (is_kova(spec_get_kind_of_value(P), TABLE_COLUMN_TY)) {
    THIS_IS_AN_INTERESTING_PROBLEM {
        sentence_problem(_P_(C7TotalTableColumn),
            "this seems to be an attempt to total up the column of a table",
            "whereas it's only legal to use 'total' for properties.");
    }
}
invocation_passed = FALSE;
```

This code is used in §79.

§**81.** For instance, the following blocks an attempt to add a number to a text.

⟨Step (i.2.1.4) Dimension-check arithmetic phrases 81⟩ ≡
```
LOG_STAGE("(i.2.1.4)");
int op_number = ph->type_data.arithmetical_operation;
LOGIF(INVOCATIONS, "(P%d) Arithmetic operation <op-%d>\n", ref, op_number);
specification *L, *R;
kind_of_value *kov_wanted, *left_kov, *right_kov, *kov_produced;
⟨Work out the KOVs of the operands, and what we want, and what we get 82⟩;
if (kov_produced) inv->kov_resulting = kov_produced;
else ⟨Fail the invocation for a dimensional problem 83⟩;
```

This code is used in §76.

§**82.** For the way this is actually worked out, see the section on "Dimensions".

⟨Work out the KOVs of the operands, and what we want, and what we get 82⟩ ≡

```
L = inv_get_token_as_parsed(inv, 0);
left_kov = fix_arithmetic_operand(L, depth);
if (arithmetic_op_is_unary(op_number)) {
    R = NULL; right_kov = NULL;
} else {
    R = inv_get_token_as_parsed(inv, 1); right_kov = fix_arithmetic_operand(R, depth);
}
kov_wanted = spec_get_kind_of_value(expected);
kov_produced = arithmetic_on_kovs(left_kov, right_kov, op_number);
```

This code is used in §81.

§**83.** Note that "value" – the vaguest KOV of all – might come up here as a result of some problem evaluating one of the operands, which has already been reported in a problem message; so we only issue this problem message when L and R are more definite.

⟨Fail the invocation for a dimensional problem 83⟩ ≡

```
if ((left_kov) && (is_kova(left_kov, ANY_VALUE_TY) == FALSE) &&
    (right_kov) && (is_kova(right_kov, ANY_VALUE_TY) == FALSE)) {
    THIS_IS_AN_INTERESTING_PROBLEM {
        quote_source(1, current_sentence);
        quote_words(2, L->word_ref1, L->word_ref2);
        quote_words(3, R->word_ref1, R->word_ref2);
        quote_spec(4, new_generic_CONSTANT_type(left_kov));
        quote_spec(5, new_generic_CONSTANT_type(right_kov));
        switch(op_number) {
            case PLUS_OPERATION: quote_text(6, "adding"); quote_text(7, "to"); break;
            case MINUS_OPERATION: quote_text(6, "subtracting"); quote_text(7, "from"); break;
            case TIMES_OPERATION: quote_text(6, "multiplying"); quote_text(7, "by"); break;
            case DIVIDE_OPERATION:
            case REMAINDER_OPERATION: quote_text(6, "dividing"); quote_text(7, "by"); break;
            case APPROXIMATION_OPERATION: quote_text(6, "rounding"); quote_text(7, "to");
break;
        }
        handmade_problem(_P_(C7BadArithmetic));
        issue_problem_segment(
            "You wrote %1, but that seems to involve %6 %4 ('%2') %7 %5 ('%3'), "
            "which is not good arithmetic.");
        issue_problem_end();
    }
}
invocation_passed = FALSE;
```

This code is used in §81.

§84. I really have no idea if computer scientists ever use the word "monomorphic" for a function whose type is definitely known, but I suppose it's the opposite of being polymorphic. Anyway, this is the general case: almost all phrases fall into this category, including all phrases created outside the Standard Rules.

The deal is simply that every argument must match its specification. For instance, if `inv` is an invocation of this phrase:

> To truncate (L - a list of values) to (N - a number) entries: ...

...then token 0 must match "list of values", and token 1 must match "number".

⟨Step (i.2.2) Take care of monomorphic phrases 84⟩ ≡

```
if (!arithmetical_phrase) {
    LOG_STAGE("(i.2.2)");
    int i, exit_at_once = FALSE;
    for (i=0; i<inv_get_number_tokens(inv); i++) {
        specification ith_token = *(inv_get_token_as_parsed(inv, i));
        specification *ith_spec = ph->type_data.tokens_spec[i];
        inv_set_token_check_to_do(inv, i, NULL);
        switch(typecheck_recursive(&ith_token, ith_spec, 2, depth+1)) {
            case NEVER_MATCH:
                LOGIF(INVOCATIONS, "(i.2.2) on $S failed at token %d\n", found, i);
                invocation_passed = FALSE;
                if (problems_issued_during_typechecking()) exit_at_once = TRUE;
                break;
            case SOMETIMES_MATCH:
                LOGIF(INVOCATIONS, "(i.2.2) on $S qualified at token %d\n", found, i);
                inv_set_token_check_to_do(inv, i, ith_spec);
                with_qualifications = TRUE;
                break;
        }
        if (exit_at_once) break;
        if (spec_test_flag(&ith_token, COERCED_SPFLAG)) {
            LOGIF(INVOCATIONS, "(i.2.2) duplicates $S\n", &ith_token);
            inv_set_token_as_parsed(inv, i, spec_copy(&ith_token));
        } else {
            LOGIF(INVOCATIONS, "(i.2.2) declines to duplicate $S\n", &ith_token);
            *(inv_get_token_as_parsed(inv, i)) = ith_token;
        }
    }
    if (invocation_passed) LOGIF(INVOCATIONS, "(i.2.2) argument type matching passed\n");
}
```

This code is used in §75.

§**85.** For templates and the meaning of `prototype_checking_pass`, see the section on "Kinds of Value". But basically this handles the matching of an invocation against a definition like:

To remove (N - value = kov reference 1) from (L - list of values = list of kov marker 1): ...

⟨Step (i.2.3) Match type templates in the argument specifications 85⟩ ≡

```
    LOG_STAGE("(i.2.3)");
    int template_present = FALSE, i, exit_at_once = FALSE;
    for (i=0; i<inv_get_number_tokens(inv); i++)
        if (ph->type_data.tokens_template[i]) template_present = TRUE;
    if (template_present) {
        for (prototype_checking_pass = 1; prototype_checking_pass <= 2; prototype_checking_pass++)
{
            LOGIF(INVOCATIONS, "(i.2.3) prototype check pass %d\n", prototype_checking_pass);
            for (i=0; i<inv_get_number_tokens(inv); i++) {
                if (ph->type_data.tokens_template[i]) {
                    switch(can_we_cast_kovs(
                        spec_evaluates_to(inv_get_token_as_parsed(inv, i)),
                        ph->type_data.tokens_template[i])) {
                        case NEVER_MATCH:
                            LOGIF(INVOCATIONS, "(i.2.3) failed at token %d\n", i);
                            invocation_passed = FALSE;
                            if (problems_issued_during_typechecking()) exit_at_once = TRUE;
                            break;
                        case SOMETIMES_MATCH:
                            best_result = SOMETIMES_MATCH;
                            we won't use with_qualifications – we don't know exactly what they
are
                            LOGIF(INVOCATIONS, "(i.2.3) dropping to sometimes at token %d\n",
i);
                            break;
                        case ALWAYS_MATCH:
                            break;
                    }
                }
                if (exit_at_once) break;
            }
            if (exit_at_once) break;
        }
        if (invocation_passed) LOGIF(INVOCATIONS, "(i.2.3) prototype check passed\n");
    }
```

This code is used in §75.

§**86.** Although we don't implement it with prototypes as such, there's a similar constraint on the arguments of an assignment. If we are checking an invocation against:

To let (t - existing variable) be (u - assignable-value): ...

then we have so far checked that argument 0 is indeed the name of a variable which already exists, and that argument 1 does indeed evaluate a permissible value. But suppose the invocation is

let N be "there'll be no mutant enemy";

where N has already been created as a variable of KOV "number". This clearly has to be rejected, as it would violate type-safety. Step (i.2.4) therefore makes sure that all assignments match the KOV of the new value against the KOV of the storage item to which it is being written.

A reasonable question might be why we don't implement this using the prototype system of (i.2.3), thus removing a rule from this already-complex algorithm. The answer is that we would need much more elaborate syntax to express the specifications of the tokens (e.g., "value = kov reference 1 table-reference"), which would be icky, and that we would get less helpful problem messages when an invocation failed.

⟨Step (i.2.4) Match KOVs in assignment phrases 86⟩ ≡

```
LOG_STAGE("(i.2.4)");
if (ph->type_data.assignment_exception) {
    specification *target = inv_get_token_as_parsed(inv, 0);
    specification *new_value = inv_get_token_as_parsed(inv, 1);
    specification *target_spec = ph->type_data.tokens_spec[0];
    specification *new_value_spec = ph->type_data.tokens_spec[1];
    if (is_kova(spec_get_kind_of_value(target_spec), OBJECT_TY))
        ⟨Step (i.2.4.1) Police an assignment to an object 87⟩;
    if (family_is(target_spec, STORAGE_FMY))
        ⟨Step (i.2.4.2) Police an assignment to a storage item 91⟩;
}
```

This code is used in §75.

§**87.** It doesn't always look like an assignment, but a phrase such as:

change the Marble Door to open;

has similar type-checking needs.

⟨Step (i.2.4.1) Police an assignment to an object 87⟩ ≡

```
LOG_STAGE("(i.2.4.1)");
world_object *target_wo = wo_of_CONSTANT_OBJECT_if_any(target);
property_name *prn = NULL;
int make_check = FALSE;
if (is_kova(spec_get_kind_of_value(new_value_spec), ANY_VALUE_TY))
    ⟨Maybe we're changing an object to a value of a KOV coinciding with a property 88⟩;
if (spec_is_CONSTANT_of_kova(new_value_spec, PROPERTY_TY))
    ⟨Maybe we're changing an object to a named either/or property or condition state 89⟩;
if (make_check)
    ⟨Check that the property exists and that the object is allowed to have it 90⟩;
```

This code is used in §86.

§**88.** There are actually two definitions like this in the Standard Rules:

[1] To change (o - object) to (w - value): ...

[2] To change (o - object) to (p - property): ...

Here's the code for [1], the less obvious case. This is needed for something like

change the canvas to blue;

where "blue" is a constant colour, and "colour" is both a KOV and also a property. (This case really is an assignment – it assigns the value "blue" to the colour property of the canvas.)

⟨Maybe we're changing an object to a value of a KOV coinciding with a property 88⟩ ≡

```
LOG_STAGE("(i.2.4.1a)");
quantity *q = spec_get_constant_quantity_if_any(new_value);
if (q == NULL) invocation_passed = FALSE;
else {
    prn = kov_get_coinciding_property(qty_kind_of_value(q));
    if (prn == NULL) invocation_passed = FALSE;
    else make_check = TRUE;
}
```

This code is used in §87.

§**89.** And here's the simpler case, [2]. A small quirk here is that it will also pick up "change the Atrium to spiffy" in the following:

Atrium is a room. The Atrium can be spiffy, cool or lame.

When play begins: change the Atrium to spiffy.

...where "spiffy" is deemed a property rather than a constant value of a KOV because of the way the condition of the Atrium is declared. This is a little bit horrid, but works fine in practice. (If we try to accommodate this case within (1.2.4.1a), which might seem more logical, we run into trouble because the property name is cast to a property value of `self` when being typechecked against "value".)

⟨Maybe we're changing an object to a named either/or property or condition state 89⟩ ≡

```
LOG_STAGE("(i.2.4.1b)");
if (spec_is_CONSTANT_of_kova(new_value, PROPERTY_TY))
    prn = PROPERTY_spec_to_property_name(new_value);
else if (number_of_adjectives_applied_to(new_value) == 1) {
    adjectival_phrase *aph = get_adjective_from_list_entry(first_adjective_list_entry(new_value));
    if (aph_has_ENUMERATIVE_meaning(aph))
        prn = kov_get_coinciding_property(qty_kind_of_value(aph_has_ENUMERATIVE_meaning(aph)));
    else if (aph_has_EORP_meaning(aph))
        prn = aph_has_EORP_meaning(aph);
}
make_check = TRUE;
```

This code is used in §87.

§**90.** We do something quite interesting here, if the object is not explicitly named: we deliberately allow an assignment which may not be type-safe, and without even dropping to the "sometimes" level. This is for phrases like so:

change the item to closed;

Here the author seems to know what he's doing, and is pretty sure that the current contents of "item" will accept closure. All we can prove is that "item" contains an object (or perhaps `nothing`, the non-object). But we allow the assignment because it will compile to code which will issue a helpful run-time problem message if it goes wrong.

⟨Check that the property exists and that the object is allowed to have it 90⟩ ≡

```
LOGIF(INVOCATIONS, "Property appears to be $Y\n", prn);
if (prn == NULL) {
    THIS_IS_AN_INTERESTING_PROBLEM {
        quote_source(1, current_sentence);
        quote_words(2, target->word_ref1, target->word_ref2);
        quote_words(3, new_value->word_ref1, new_value->word_ref2);
        handmade_problem(_P_(BelievedImpossible));      the parser seems not to allow these
        issue_problem_segment(
            "You wrote %1, asking to change the object '%2'. This would "
            "make sense if '%3' were an either/or property like 'open' "
            "(or perhaps a named property value like 'blue') - but it "
            "isn't, so the change makes no sense.");
        issue_problem_end();
    }
}
if ((target_wo) && (prn) && (does_object_provide(target_wo, prn) == FALSE)) {
    THIS_IS_AN_INTERESTING_PROBLEM {
        quote_source(1, current_sentence);
        quote_words(2, target->word_ref1, target->word_ref2);
        quote_property(3, prn);
        handmade_problem(_P_(C7PropertyNotAllowed));
        issue_problem_segment(
            "You wrote %1, but '%2' is not allowed to have the property '%3'.");
        issue_problem_end();
    }
}
```

This code is used in §87.

§**91.** This is more straightforward, with just a tiny glitch to make the rules tougher on variables which hold text to be parsed. (Because the regular rules for exchanging the subtly different forms of text which a double-quoted literal can mean are too generous.)

⟨Step (i.2.4.2) Police an assignment to a storage item 91⟩ ≡

```
kind_of_value *kov_wanted, *kov_found;
LOGIF(INVOCATIONS, "Check assignment of $S to $S\n", new_value, target);
switch(spec_get_storage_form(target_spec)) {
    case LOCAL_VARIABLE_SPC: quote_text(6, "the name of"); break;
    case PROPERTY_VALUE_SPC: quote_text(6, "a property whose kind of value is"); break;
    case NONLOCAL_VARIABLE_SPC: quote_text(6, "a variable whose kind of value is"); break;
    case TABLE_ENTRY_SPC: quote_text(6, "a table entry whose kind of value is"); break;
    case LIST_ENTRY_SPC: quote_text(6, "an entry in a list whose kind of value is"); break;
}
kov_wanted = spec_evaluates_to(target);
kov_found = spec_evaluates_to(new_value);
if (requested.recently_discovered_KOV) kov_found = requested.recently_discovered_KOV;
if (((is_kova(kov_wanted, UNDERSTANDING_TY)) &&
    (is_kova(kov_found, UNDERSTANDING_TY) == FALSE))
    || (can_we_cast_kovs(kov_found, kov_wanted) == NEVER_MATCH)) {
    THIS_IS_AN_INTERESTING_PROBLEM {
        quote_source(1, current_sentence);
        quote_words(2, target->word_ref1, target->word_ref2);
        quote_kov(3, spec_evaluates_to(target));
        quote_words(4, new_value->word_ref1, new_value->word_ref2);
        quote_kov(5, spec_evaluates_to(new_value));
        handmade_problem(_P_(C7ChangeToWrongValue));
        issue_problem_segment(
            "You wrote %1, but '%2' is supposed to be "
            "%6 %3, so it cannot be set equal to %4, whose kind is %5.");
        issue_problem_end();
    }
}
```

This code is used in §86.

§**92.** Suppose we have something like this:

award the current action points;

and we are typechecking `found` as "the current action" (a phrase deciding a value) against `expected` as "number", the parameter expected in "award (N - a number) points".

No matter how peculiar this invocation of `found` was, we have now successfully worked out the kind of the value it would return if compiled, and this is stored in `inv->kov_resulting`. We now check to see if this matches the KOV expected – in this example, it won't, because a stored action does not cast to a number.

⟨Step (i.2.5) Check kind of value returned 92⟩ ≡

```
LOG_STAGE("(i.2.5)");
if (species_is(found, PHRASE_TO_DECIDE_VALUE_SPC)) {
    kind_of_value *kov_needed = spec_get_kind_of_value(expected);
    int outcome_test = ALWAYS_MATCH;
    if (kov_needed) {
        LOGIF(INVOCATIONS, "(P%d) Checking returned $u against desired $u\n",
            ref, inv->kov_resulting, kov_needed);
```

```
            outcome_test = can_we_cast_kovs(inv->kov_resulting, kov_needed);
        }
        switch (outcome_test) {
            case NEVER_MATCH: invocation_passed = FALSE; break;
            case SOMETIMES_MATCH: best_result = SOMETIMES_MATCH; break;
        }
    }
```

This code is used in §75.

§**93.** The final stage in type-checking a phrase is to ensure that any phrase options are properly used.

⟨Step (i.2.6) Check any phrase options 93⟩ ≡

```
    LOG_STAGE("(i.2.6)");
    if ((invocation_passed) && (inv->phrase_options_invoked)) {
        int ow1, ow2;
        inv_get_phrase_options(inv, &ow1, &ow2);
        int cso = ph_To_check_supplied_options(ph,
            (problem_threshold >= 2)?TRUE:FALSE, inv, ow1, ow2);
        if (cso == FALSE) invocation_passed = FALSE;
    }
```

This code is used in §75.

§**94.** A say phrase which involves a property of something implicitly changes the scope for any vaguely described properties within the text supplied as that property (if it is indeed text). We have to mark any such property, and any such say. For instance, suppose we are typechecking

[1] "Oh, look: [initial appearance of the escritoire]"

and the initial appearance in question is:

[2] "A small, portable writing desk holding up to [carrying capacity] letters."

Printing text [2], it's important for the `self` object to be the escritoire, which might not be the case otherwise; so during the printing of [1], we have to change `self` temporarily and restore it afterwards.

⟨Step (i.2.7) Worry about self in say property of 94⟩ ≡

```
    LOG_STAGE("(i.2.7)");
    if ((ph->type_data.say_phrase) &&
        (inv_get_number_tokens(inv) == 1) &&
        (spec_get_storage_form(inv_get_token_as_parsed(inv, 0)) == PROPERTY_VALUE_SPC)) {
        spec_set_flag(inv_get_token_as_parsed(inv, 0), RECORD_AS_SELF_SPFLAG);
        inv_set_flag(inv, SAVE_SELF_INVFLAG);
    }
```

This code is used in §75.

§**95.** Some phrases are defined with a notation making them allowable only inside loops, or other control structures; for instance,

To break – in loop: ...

And here is where we check that "break" is indeed used only in a loop.

⟨Step (i.2.8) Worry about using a phrase outside of the control structure it belongs to 95⟩ ≡

```
LOG_STAGE("(1.2.8)");
if ((ph) && (ph->type_data.only_in_loop)) {
    LOGIF(INVOCATIONS, "Required to be inside loop body\n");
    if (inside_a_loop_body() == FALSE) {
        THIS_IS_AN_INTERESTING_PROBLEM {
            quote_source(1, current_sentence);
            quote_text(2, lw_array[ph->type_data.only_in_block_wn].lw_text);
            handmade_problem(_P_(C7CantUseOutsideLoop));
            issue_problem_segment(
                "%1 makes sense only inside a 'while' or 'repeat' loop.");
            issue_problem_end();
        }
    }
}
if ((ph) && (ph->type_data.only_in_block_wn >= 0)) {
    char *required = lw_array[ph->type_data.only_in_block_wn].lw_text;
    char *actual = name_of_current_block();
    LOGIF(INVOCATIONS, "Required to be inside block '%s'\n", required);
    if ((actual) && (strcmp(actual, "unless") == 0)) actual = "if";
    if ((actual == NULL) || (strcmp(required, actual) != 0)) {
        THIS_IS_AN_INTERESTING_PROBLEM {
            quote_source(1, current_sentence);
            quote_text(2, lw_array[ph->type_data.only_in_block_wn].lw_text);
            handmade_problem(_P_(C7CantUseOutsideStructure));
            issue_problem_segment(
                "%1 makes sense only inside a '%2' block.");
            issue_problem_end();
        }
    }
}
```

This code is used in §75.

§**96. Preserving successful invocations.** This is the happy ending, in which the phrase can probably be passed, though there are still a handful of pitfalls. We know definitely which phrase(s) to invoke for each group, and there's no group where everything failed.

⟨Step (i.3) Preserve successful invocations 96⟩ ≡

```
LOG_STAGE("(i.3)");
⟨Step (i.3.1) Create any requested local variable 97⟩;
⟨Step (i.3.2) Winnow the invocation list down to the survivors 102⟩;
```

This code is used in §67.

§**97.** Here's the usual way a local variable is made. One invocation we matched is for the phrase whose prototype reads:

To let (T - nonexisting variable) be (V - value): ...

To be definite, let's suppose we are working on:

let the magic word be "Shazam [turn count] times!";

The checking code above filled out a request when it handled the T argument, further down in the recursion from the current call, to say that a new local should be made. It set `requested.word_ref1` and `requested.word_ref2` to the word range "magic word", and `requested.KOV_from_prototype` to "value" – meaning that T's prototype did not require any specific KOV. (It would have been legal to write "To let (T - nonexisting number variable) be...", for instance, and then `requested.KOV_from_prototype` would be "number".) So in this case, at least, the prototype didn't tell us what the KOV of this local should be – we must deduce it from the initial value,

"Shazam [turn count] times!"

which is stored in `requested.local_initial_value`. It occasionally happens that we have already worked out some ingenious KOV for this as a result of a polymorphism exception or arithmetic rules, in which case `requested.recently_discovered_KOV` will be set. But more often not, and certainly not in this example.

⟨Step (i.3.1) Create any requested local variable 97⟩ ≡

```
LOG_STAGE("(i.3.1)");
if ((requested.word_ref1 >= 0) && (depth <= requested.depth_made_at) &&
    (!(TEST_COMPILATION_MODE(DO_NOT_CREATE_LOCAL_VARS_CMODE)))) {
    int w1 = requested.word_ref1, w2 = requested.word_ref2;          name of new local
    kind_of_value *kov = requested.KOV_from_prototype;                  and its KOV
    specification *initial_value = requested.local_initial_value;
    kind_of_value *discovered_kov = requested.recently_discovered_KOV;
    ⟨Empty any local variable request 8⟩;
    if ((is_kova(kov, ANY_VALUE_TY)) && (initial_value))
        ⟨Where no KOV was explicitly stated, infer this from the supplied initial value 98⟩;
    phrase_made_local = make_new_local(w1, w2, kov);
    KOV_of_recently_created_local = kov;
}
```

This code is used in §96.

§**98.** Unusually, it's legal for the initial value to be a generic constant –

let the magic digraph be an indexed text;

This doesn't give us an initial value as such, but it explicitly tells us the KOV, which is good enough.

Otherwise, we either know the KOV already from polymorphism calculations, or we can work it out by seeing what the initial value evaluates to.

But there's a wrinkle. The KOV of the initial value is not always the same as the KOV of a variable to store it in – we sometimes want to widen this. For instance, the KOV of

"Shazam [turn count] times!"

is "text-routine", meaning, text with substitutions in. We want to widen this to "text"; the widening is done by `kov_for_storage_to_hold`.

⟨Where no KOV was explicitly stated, infer this from the supplied initial value 98⟩ ≡

```
kind_of_value *seems_to_be;
```

```
if (discovered_kov)
    seems_to_be = discovered_kov;
else if (spec_is_generic_CONSTANT(initial_value))
    seems_to_be = spec_get_kind_of_value(initial_value);
else
    seems_to_be = spec_evaluates_to(initial_value);
LOGIF(LOCAL_VARIABLES, "New variable $W from $S seems to be: $u\n",
    w1, w2, initial_value, seems_to_be);
if (seems_to_be == NULL) ⟨Fail: the initial value of the local is unknown 99⟩;
if ((is_kovcon(seems_to_be)) &&
    (is_kova(kovcon_get_base(seems_to_be), NO_ENTRIES_TY)))
    ⟨Fail: the initial value of the local is the empty list 100⟩;
kov = kov_for_storage_to_hold(seems_to_be);
if (kov == NULL) ⟨Fail: the initial value can't be stored 101⟩;
LOGIF(LOCAL_VARIABLES, "Inferred the KOV: $u\n", kov);
```

This code is used in §97.

§**99.** So there are various things that can go wrong:

⟨Fail: the initial value of the local is unknown 99⟩ ≡

```
THIS_IS_AN_ORDINARY_PROBLEM;
quote_source(1, current_sentence);
quote_words(2, initial_value->word_ref1, initial_value->word_ref2);
handmade_problem(_P_(BelievedImpossible));
issue_problem_segment(
    "The phrase %1 tries to use 'let' to give a temporary name to a value, "
    "but the value ('%2') is one that I can't understand.");
issue_problem_end();
return NEVER_MATCH;
```

This code is used in §98.

§**100.** Bet you didn't think of this one:

⟨Fail: the initial value of the local is the empty list 100⟩ ≡

```
THIS_IS_AN_ORDINARY_PROBLEM;
quote_source(1, current_sentence);
handmade_problem(_P_(C7CantLetEmptyList));
issue_problem_segment(
    "The phrase %1 tries to use 'let' to give a temporary name to the "
    "empty list '{ }', but because it's empty, I can't tell what kind of "
    "value the list should have. Try 'let X be a list of numbers' (or "
    "whatever) instead.");
issue_problem_end();
return NEVER_MATCH;
```

This code is used in §98.

§**101.** And some KOVs are just forbidden in storage, and can't even be widened to something legitimate ("property", for instance):

⟨Fail: the initial value can't be stored 101⟩ ≡
```
LOGIF(INVOCATIONS, "Unable to store the KOV: $u\n", kov);
THIS_IS_AN_ORDINARY_PROBLEM;
sentence_problem(_P_(C7BadLocalKOV),
    "this is a value which can't be given a temporary name",
    "using 'let'.");
return NEVER_MATCH;
```

This code is used in §98.

§**102.** To recap, each group in the list has been checked. Each group contains 1 or more invocations which passed; the last of these is either the end of the group, or an invocation which passed without qualification. Within each group, failed invocations are interleaved with passed ones, and there will also be unchecked invocations if they occurred after an unqualified pass. A typical group might produce:

```
f ? f g ? ? p - - -
```

We can now throw away the `f`, `g` and `-` invocations to leave just those which will be compiled:

```
? ? ? p
```

This will eventually result in a Resolver routine to check if the arguments allow the first invocation and run it if so; then the second; then the third; and, if those three fell through, run the fourth invocation without further checking. See "Compile Invocations" in Chapter 12.

⟨Step (i.3.2) Winnow the invocation list down to the survivors 102⟩ ≡
```
LOG_STAGE("(i.3.2)");
invocation_list *invl = new_invocation_list();
int nfi = 0, nfi_group = -1, number_ambiguity = FALSE;
INVOCATION_VARIABLE(inv);
LOOP_THROUGH_INVOCATION_LIST(inv, found_invl) {
    if (inv_test_flag(inv, PASSED_INVFLAG)) {
        int nti = inv_get_number_tokens(inv);
        if (nfi_group != inv_get_group(inv)) {
            nfi = nti;
            nfi_group = inv_get_group(inv);
        } else {
            if (nfi != nti) number_ambiguity = TRUE;
        }
        add_to_invocation_list(invl, inv);
    }
}
if (number_ambiguity) ⟨Issue the number ambiguity problem message 103⟩;
spec_set_invocation_list(found, invl);
found_invl = invl;
```

This code is used in §96.

§**103.** This is another sort of error which couldn't happen with a conventional programming language – in C, for instance, it's syntactically obvious how many arguments a function call has, because the brackets and commas are unambiguous. But in Inform, there are no reserved tokens of syntax acting like that. So we could easily have two accepted invocations in the list which have different numbers of arguments to each other, and there's no way safely to adjudicate that at run-time.

⟨Issue the number ambiguity problem message 103⟩ ≡

```
THIS_IS_AN_ORDINARY_PROBLEM;
quote_source(1, current_sentence);
handmade_problem(_P_(C7UnequalValueAmbiguity));
issue_problem_segment(
    "The phrase %1 is ambiguous in a way that I can't disentangle. "
    "It has more than one plausible interpretation, such that it "
    "would only be possible to tell which is valid at run-time: "
    "ordinarily that would be fine, but because the different "
    "interpretations are so different (and involve different "
    "numbers of values being used) there's no good way to cope. "
    "Try rewording one of the phrases which caused this clash: "
    "there's a good chance the problem will then go away.");
issue_problem_end();
it_is_not_worth_adding = TRUE;
int i = 0;
LOOP_THROUGH_INVOCATION_LIST(inv, invl) {
    i++;
    quote_number(1, &i);
    quote_invocation(2, inv);
    issue_problem_begin("*");
    issue_problem_segment("%1. %2");
    issue_problem_end();
}
describe_inv_token_problems();
it_is_not_worth_adding = FALSE;
return NEVER_MATCH;
```

This code is used in §102.

§**104. Problems, problems, problems.** Where (i.3) was the happy ending, this is the unhappy one. We are now in a situation where none of the invocations in group `current_group` passed, so that it's impossible compile legal code which would do anything at run-time. Thus `found` unequivocally fails to match, so it would be legitimate simply to perform `return NEVER_MATCH` here.

But of course we actually want to produce helpful problem messages. It's not entirely clear how best to do this. Often, when `found` fails, it fails for seven different reasons – each different possibility fails for a different cause. We want, somehow, to guess which was the most likely to have been intended and to report the problem with that one.

⟨Step (i.4) All invocations failed error 104⟩ ≡

```
LOG_STAGE("(i.4)");
THIS_IS_AN_ORDINARY_PROBLEM;
if (current_group < 0) return NEVER_MATCH;      just in case found had an empty invocation list
LOGIF(INVOCATIONS, "All invocations failed: issuing problem\n$E", found_invl);
invocation *first_inv_in_group = NULL;
invocation *first_failing_interestingly = NULL;
```

```
invocation *first_not_failing_grossly = NULL;
int say_w1 = -1, say_w2 = -1;
int list_includes_lets = FALSE;
int nongross_count = 0;
⟨Scan through the invocations in the problematic group, gathering information 105⟩;
invocation *most_likely_to_have_been_intended = NULL;
⟨Decide which invocation is the one most likely to have been intended 106⟩;
if (first_failing_interestingly)
    ⟨Re-typecheck the first interesting invocation, allowing interesting problems this time 107⟩
else if (most_likely_to_have_been_intended)
    ⟨Re-typecheck the tokens of the most likely invocation with silence off 108⟩
else {
    if (say_w1 >= 0) ⟨Issue a problem for a text substitution with multiple failed possibilities 109⟩
    else ⟨Issue a problem for a regular phrase with multiple failed possibilities 110⟩;
}
return NEVER_MATCH;
```

This code is used in §67.

§105.

⟨Scan through the invocations in the problematic group, gathering information 105⟩ ≡

```
int ref;
for (ref=0; ref<test_count; ref++) {
    invocation *inv = list_of_invocations_to_test[ref];
    if (inv_get_group(inv) == current_group) {
        if ((inv_test_flag(inv, INTERESTING_PROBLEM_INVFLAG)) &&
            (first_failing_interestingly == NULL)) first_failing_interestingly = inv;
        ph = inv->phrase_invoked;
        if (phtd_is_any_let_phrase(&(ph->type_data))) list_includes_lets = TRUE;
        if (first_inv_in_group == NULL) first_inv_in_group = inv;
        if (inv_test_flag(inv, GROSSLY_FAILED_INVFLAG) == FALSE) {
            first_not_failing_grossly = inv;
            if (phtd_is_a_say_X_phrase(&(ph->type_data))) {
                say_w1 = inv->invocation_w1; say_w2 = inv->invocation_w2;
            }
            nongross_count++;
        }
    }
}
```

This code is used in §104.

§106.

⟨Decide which invocation is the one most likely to have been intended 106⟩ ≡

```
if (first_failing_interestingly)
    most_likely_to_have_been_intended = first_failing_interestingly;
else if ((nongross_count > 1) && (list_includes_lets))
    most_likely_to_have_been_intended = first_not_failing_grossly;
else if (nongross_count == 1)
    most_likely_to_have_been_intended = first_not_failing_grossly;
else if ((nongross_count == 0) && (first_inv_in_group))
    most_likely_to_have_been_intended = first_inv_in_group;
```

This code is used in §104.

§107.

⟨Re-typecheck the first interesting invocation, allowing interesting problems this time 107⟩ ≡

```
issue_interesting_problems = TRUE;
invocation *inv = first_failing_interestingly;
int ref = 0;                                    will be shown in the debugging log as P0 now
⟨Test the current invocation and set its results flags accordingly 74⟩;
```

This code is used in §104.

§108.

⟨Re-typecheck the tokens of the most likely invocation with silence off 108⟩ ≡

```
int ec = problem_count;
ph = most_likely_to_have_been_intended->phrase_invoked;
int i;
for (i=0; i<inv_get_number_tokens(most_likely_to_have_been_intended); i++) {
    typecheck_recursive(inv_get_token_as_parsed(most_likely_to_have_been_intended, i),
        ph->type_data.tokens_spec[i], 0, depth+1);
    if (problem_count > ec) break;
}
if (problem_count == ec)
    sentence_problem(_P_(BelievedImpossible),
        "the ingredients in this phrase do not fit it",
        "but I am confused enough by this that I can't give a very helpful "
        "problem message. Sorry about that.");
```

This code is used in §104.

§**109.**

⟨Issue a problem for a text substitution with multiple failed possibilities 109⟩ ≡

```
if ([[say_w1, say_w2 == time]] || [[say_w1, say_w2 == the time]]) {
    sentence_in_detail_problem(_P_(C7Time2), say_w1, say_w2,
        "this asked to say 'time' itself",
        "rather than any specific time. (If you meant the current "
        "time, this is called 'time of day' in Inform to avoid "
        "confusing it with the various other meanings that the "
        "word 'time' can have.)");
    issue_problem_end();
    return NEVER_MATCH;
}
specification *say_text_as_value = parse_expression(say_w1, say_w2, TYPE_OR_VALUE_EXPCON);
kind_of_value *kov = spec_evaluates_to(say_text_as_value);
LOG("Say text $W; as value, $S; evaluating to $u\n",
    say_w1, say_w2, say_text_as_value, kov);
if (species_is(say_text_as_value, DESCRIPTION_SPC)) {
    sentence_in_detail_problem(_P_(C7SayDescription), say_w1, say_w2,
        "this asked to say something which describes items too vaguely",
        "and so does not (or not in an elementary enough way) tell "
        "me exactly which thing or room should have its name printed.");
    return NEVER_MATCH;
}
if (kov) {
    sentence_in_detail_problem(_P_(C7AllSayInvsFailed), say_w1, say_w2,
        "this asked to say something of a kind which can't be said",
        "or rather, printed. Although this problem can arise when you "
        "use complicated text substitutions which come in variant "
        "forms depending on the kinds of value used, far more often "
        "what this means is just that you tried to use a substituted "
        "value (e.g., in 'say \"The dial reads [V].\"') of a kind "
        "which could not be printed out. For instance, if V is a "
        "number or a piece of text, there is no problem: but if V "
        "is a parsing topic, say an entry in a 'topic' column of "
        "a table, then this problem will arise.");
    return NEVER_MATCH;
}
sentence_in_detail_problem(_P_(BelievedImpossible), say_w1, say_w2,
    "this asked to say something which I do not recognise",
    "either as a value or as one of the possible text substitutions.");
```

This code is used in §104.

§**110.**

⟨Issue a problem for a regular phrase with multiple failed possibilities 110⟩ ≡

```
handmade_problem(_P_(C7AllInvsFailed));
quote_source(1, current_sentence);
issue_problem_segment(
    "You wrote %1, which is a phrase that can be construed in more "
    "than one way. I tried the possibilities in the order numbered "
    "below, and would have accepted the first which successfully "
    "matched - but none of them did.");
issue_problem_end();
⟨Itemise the failed possibilities in a list 111⟩;
```

This code is used in §104.

§**111.** Where, at long, long last:

⟨Itemise the failed possibilities in a list 111⟩ ≡

```
it_is_not_worth_adding = TRUE;
int i;
for (i=1; i<=test_count; i++) {
    invocation *inv = list_of_invocations_to_test[i-1];
    if (inv_get_group(inv) == current_group) {
        quote_number(1, &i);
        quote_invocation(2, inv);
        issue_problem_begin("*");
        issue_problem_segment("%1. %2");
        issue_problem_end();
    }
}
describe_inv_token_problems();
it_is_not_worth_adding = FALSE;
```

This code is used in §110.

§**112.** In that final checklist of doomed possibility, the code to quote an invocation in a problem message will call `note_inv_token_text` for each parsed token. This remembers the token so that it can be explained in notes at the end of the big list; but each word range is remembered only once, for brevity. We don't gloss the meanings of literal constants like `26` or `"frog"` since these are glaringly obvious.

```
void note_inv_token_text(specification *found) {
    if ((spec_is_actual_CONSTANT_of_kova(found, NUMBER_TY)) ||
        (spec_is_actual_CONSTANT_of_kova(found, TEXT_TY))) return;
    inv_token_problem_token *itpt;
    LOOP_OVER(itpt, inv_token_problem_token)
        if ((itpt->word_ref1 == found->word_ref1) && (itpt->word_ref2 == found->word_ref2))
            return;
    itpt = CREATE(inv_token_problem_token);
    itpt->word_ref1 = found->word_ref1; itpt->word_ref2 = found->word_ref2;
    itpt->as_parsed = found; itpt->already_described = FALSE;
}
```

The function note_inv_token_text is called from 12/phtd.

§**113.** And this is the code which produces those glossary notes at the foot of the list.

```
void describe_inv_token_problems(void) {
    inv_token_problem_token *itpt;
    LOOP_OVER(itpt, inv_token_problem_token)
        if (itpt->already_described == FALSE) {
            itpt->already_described = TRUE;
            quote_words(1, itpt->word_ref1, itpt->word_ref2);
            quote_spec(2, itpt->as_parsed);
            issue_problem_begin("*");
            if (spec_is_evaluating(itpt->as_parsed)) {
                quote_kov(3, spec_evaluates_to(itpt->as_parsed));
                if (family_is(itpt->as_parsed, STORAGE_FMY))
                    issue_problem_segment("%1: %2, holding <i>%3</i>");
                else if (species_is(itpt->as_parsed, PHRASE_TO_DECIDE_VALUE_SPC))
                    issue_problem_segment("%1: %2, which results in <i>%3</i>");
                else
                    issue_problem_segment("%1: <i>%3</i>");
            } else {
                issue_problem_segment("%1: <i>%2</i>");
            }
            issue_problem_end();
        }
}
```

§**114.** That only leaves two little utility routines needed by the typechecker. The first of these does something substantive: it works out the KOV of an operand for an arithmetic operation.

```
kind_of_value *fix_arithmetic_operand(specification *operand, int depth) {
    if (spec_is_UNKNOWN(operand)) return NULL;
    specification *expected = operand;
    if (spec_is_CONSTANT_of_kova(operand, PROPERTY_TY)) {
        property_name *prn = spec_get_property_name_if_any(operand);
        if (prn_is_either_or(prn) == FALSE)
            expected = new_generic_CONSTANT_type(prn_get_kind_of_value(prn));
    }
    typecheck_recursive(operand, expected, 2, depth+1);
    return spec_evaluates_to(operand);
}
```

§**115.** The second utility fills in a note in a problem message, identifying which sub-parts of a compound condition were okay.

```
int issue_condition_error_bit(int w1, int w2) {
    if (is_list_divided(w1, w2, LOOK_FOR_AND + LOOK_FOR_OR)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        int ok1 = issue_condition_error_bit(lw1, lw2);
        int ok2 = issue_condition_error_bit(rw1, rw2);
        if ((ok1) && (ok2)) return TRUE;
        return FALSE;
    }
    specification *clause = parse_expression(w1, w2, CONDITION_EXPCON);
    LOG("Condition clause $W = $S\n", w1, w2, clause);
    quote_words(4, w1, w2);
    if (spec_is_UNKNOWN(clause)) {
        clause = parse_expression(w1, w2, VALUE_EXPCON);
        if (spec_is_UNKNOWN(clause))
            issue_problem_segment("'%4' did not make sense; ");
        else
            issue_problem_segment(
                "'%4' only made sense as a value, which can't be used as "
                "a condition; ");
        return FALSE;
    }
    issue_problem_segment("'%4' was okay; ");
    return TRUE;
}
```

Invocations 7/inv

Purpose

Specifications which ask to use a phrase (which are "phrasal") indicate which phrase they intend by means of a list of "invocations". This list goes on to record the outcome of type-checking and provides instructions for code generation, as we see here.

7/inv.§1-14 Invocations themselves; §15 The implied newlines rule; §16-29 Invocation lists

---

Definitions

¶1. An invocation is a request to perform a particular phrase with a particular set of parameters. For instance, to perform "award (N - a number) points" with N set to 100.

Phrasal specifications, the ones which request the use of phrases, are built up by parsing the source text. This is often so ambiguous that it is impossible, at first, to narrow down the meaning to a single invocation, and so a phrasal specification builds up a list of possibilities.

The process of type-checking then strikes out definitely incorrect invocations, reducing the size of the list. If it becomes empty, the type-checker produces a Problem message; and similarly if it is impossible to remove mutually exclusive possibilities, for that is an ambiguity which Inform cannot resolve. But a successful type-checking may still leave more than one invocation in the list. There are two reasons for this:

(a) The list is bunched up into groups, which are numbered within the list from 0 upwards. Type-checking proceeds on each group independently. For most phrases, there is only one group – group 0 – and this business therefore has no effect, but it comes into its own for "say" phrases:

```
say "The time is ", time of day in words, " and you yawn."
```

Here what looks like a single phrase is in fact a sequence of three phrases, to say each of the items required, and this gives rise to groups 0, 1 and 2 in the invocation list: each group is treated as, in effect, a separate phrase in its own right. (Except that compilation of the final group has a small difference: that's where Inform appends a line break to text which, from its punctuation, apparently ends a sentence.) Groups are not interleaved: the list starts with the whole of group 0, then the whole of group 1, and so on.

(b) More interestingly, type-checking of any individual invocation has three possible outcomes, not two: as in Scottish law, the verdict can be guilty, not guilty, or "not proven". An invocation which can be disproved is thrown out of the list; but of the remainder, some are marked as proven to be correct, some as unproven. What this means is that we cannot tell at compile-time whether the usage is valid, but that we will be able to tell at run-time: so the code-generator must compile suitable disambiguation code to perform this run-time checking automatically.

¶2. Invocation lists are a little more abstractly wrapped up than most linked lists inside Inform, for ease of sorting:

```
typedef struct invocation_list {
    struct invocation_list_entry *list_head;
    struct invocation_list_entry *list_tail;          storing this makes Inform 5% faster on large sources
} invocation_list;
typedef struct invocation_list_entry {
    struct invocation *listed_inv;
    struct invocation_list_entry *next;
} invocation_list_entry;
```

The structure invocation_list is private to this section.
The structure invocation_list_entry is private to this section.

¶3. Just the same, that amounts to a simple linked list of these items:

```
typedef struct invocation {
    int inv_packed_record;                                    see below: a packed bitmap
    int invocation_w1, invocation_w2;                         text which this invocation interprets
    struct phrase *phrase_invoked;                            the phrase believed to be invoked
    struct invocation_options *phrase_options_invoked;        details of any options used
    int ssp_segment_count;                                    number of subsequent complex-say phrases in stream
    int ssp_closing_segment_wn;                               identifier for the last of these, or -1
    struct invocation_token_list *tokens_invoked;             linked list of these
    struct kind_of_value *kov_resulting;                      what if anything is returned
} invocation;
```

The structure invocation is shared with 7/vasp, 7/tc, 12/phtd, 12/cinv and 12/ambig.

¶4. Once again (as with specifications) we combine a host of flags in a single packed integer. A text of about 50,000 words produces 60,000 or so invocations, so economy is again worth a little trouble.

define SUPPRESS_IMPLIED_NEWLINES_INVFLAG 0x00000001 *in saying of literal text ending with punctuation*
define INSTEAD_INVFLAG 0x00000002 *has the "instead" keyword been prefixed or suffixed?*
define SAVE_SELF_INVFLAG 0x00000004 *save the* `self` *value on I6 stack around this invocation*
define PASSED_INVFLAG 0x00000008 *once type-checked: did this pass type checking?*
define UNPROVEN_INVFLAG 0x00000010 *once type-checked: will this need run-time checking?*
define GROSSLY_FAILED_INVFLAG 0x00000020 *once type-checked: oh, this one failed big time*
define TESTED_INVFLAG 0x00000040 *has been type-checked*
define INTERESTING_PROBLEM_INVFLAG 0x00000080 *an interesting problem message could be produced about the way this failed*
define BASE_OF_GROUP 0x00000100 *to which group this belongs: 0, 1, 2, ...*
define GROUP_MASK 0x000FFF00 *up to a possible 4095*
define BASE_OF_USN 0x00100000 *unsorted position in list, 0 up to 2047*
define USN_MASK 0x7FF00000

¶5. When necessary, and it usually isn't, an invocation has a packet of details attached about any phrase options used; for instance, in

    list the contents of the Box, with newlines;

this packet records the text "with newlines" along with its translation as a run-time bitmap (with just one bit set, since only one option is used).

```
typedef struct invocation_options {
    int options;                                        bitmap of any phrase options appended
    int options_invoked_w1, options_invoked_w2;         text of any phrase options appended
} invocation_options;
```

The structure invocation_options is private to this section.

¶6. An invocation can have an arbitrary number of "tokens". These are the arguments for the phrase being invoked; so for instance in

    a random number between 2 and 7;

token 0 is "2" and token 1 is "7". For each token we record both what we've parsed – these will each be constant `VALUE_SPC` specifications of KOV "number" – and also what match the type-checker needs to make for them to be valid.

```
typedef struct invocation_token_list {
    struct specification *token_as_parsed;              what we find
    struct specification *token_check_to_do;            what we need
    struct invocation_token_list *next;
} invocation_token_list;
```

The structure invocation_token_list is private to this section.

¶7. At first sight, the `invocation` structure appears to contain redundant information. And two of the fields are, in a way, redundant: the word number fields holding the text will, of course, be common to every entry in its group, so it is wasteful to store them so many times. But not very wasteful, by our standards, and the information is not easily available elsewhere, and it enables the debugging log to be much more informative about the working of our most complicated algorithms.

Similarly, `token_as_parsed` values look as if they too must be the same for everything in the group. But this is not always true. If we are interpreting the text "award 11 points" against possible phrases "award (O - an object)" and "award (N - a number) points", then the 0th token will be "11 points" (probably parsing to `UNKNOWN`) for the first invocation, and "11" (`NUMBER`) for the second. We shall of course accept the second. But the word positions for the 0th token, and its parsing values, are different. So each invocation in the list records its own `token_as_parsed` values.

¶8. On the other hand, `token_check_to_do` and `kov_resulting` look as if they are purely a function of the `phrase_invoked`, and therefore need not be in this structure at all. This is not true:

(i) A SP in `token_check_to_do` means that Inform has not yet satisfied itself that the `token_as_parsed` value matches properly. Once the type-checker does convince itself, the relevant entry in `token_check_to_do` is cleared to `NULL`. If every token can be cleared, the invocation is declared proven. But if the type-checker accepts the invocation but marks it as "not proven", one or more values remain as a sort of to-do list, telling the code generator what remains to be checked at run-time. So `token_check_to_do` really is associated with a specific invocation, and is not redundant.

(ii) Because arithmetic phrases return different data depending on the data types fed in, `kov_resulting` will be different for different invocations of, say, "+".

```
define LOOP_THROUGH_TOKENS_PARSED_IN_INV(inv, spec)
    int lttpii_counter, lttpii_upto = inv_get_number_tokens(inv);
    for (lttpii_counter=0,
        spec = (lttpii_counter < lttpii_upto)?
            inv_get_token_as_parsed(inv, lttpii_counter):NULL;
        lttpii_counter < lttpii_upto;
        lttpii_counter++,
        spec = (lttpii_counter < lttpii_upto)?
            inv_get_token_as_parsed(inv, lttpii_counter):NULL)
```

¶9.

```
define MAX_INVOCATIONS_PER_PHRASE 4096
```

---

§**1. Invocations themselves.** Are created thus:

```
invocation *new_invocation(void) {
    invocation *inv = CREATE(invocation);
    inv->inv_packed_record = 0;
    set_phrase_invoked(inv, NULL);
    inv->kov_resulting = NULL;
    inv->phrase_options_invoked = NULL;
    inv->ssp_segment_count = 0;
    inv->ssp_closing_segment_wn = -1;
    inv->invocation_w1 = -1; inv->invocation_w2 = -1;
    inv->tokens_invoked = NULL;
    return inv;
}
```

The function new_invocation is called from 12/phtd.

§**2.** And logging thus:

```
void log_invocation(invocation *inv) {
    int i;
    if (inv == NULL) { LOG("<null invocation>"); return; }
    if (inv->phrase_invoked == NULL) { LOG("<invocation of null phrase>"); return; }
    char *verdict = "untested";
    if (inv_test_flag(inv, TESTED_INVFLAG))         verdict = "failed  ";
    if (inv_test_flag(inv, PASSED_INVFLAG))         verdict = "proven  ";
    if (inv_test_flag(inv, UNPROVEN_INVFLAG))       verdict = "unproven";
    if (inv_test_flag(inv, GROSSLY_FAILED_INVFLAG)) verdict = "gross   ";
    LOG("group:%d %c%c [%04d] %s ", inv_get_group(inv),
        (inv_test_flag(inv, INSTEAD_INVFLAG))?'I':' ',
        (inv_test_flag(inv, SAVE_SELF_INVFLAG))?'s':' ',
        ph_sequence_count(inv->phrase_invoked),
        verdict);
    log_phrase_briefly(inv->phrase_invoked);
    for (i=0; i<inv_get_number_tokens(inv); i++) {
        LOG(" ($S", inv_get_token_as_parsed(inv, i));
        if (inv_get_token_check_to_do(inv, i)) LOG(" =? $S", inv_get_token_check_to_do(inv,
i));
        LOG(")");
    }
    int ow1, ow2;
    inv_get_phrase_options(inv, &ow1, &ow2);
    if (ow1 >= 0) LOG(" [0x%x $W]", inv_get_phrase_options_bitmap(inv), ow1, ow2);
}
```

The function log_invocation is called from 2/dl.

§**3.** The packed record of flags:

```
void inv_set_flag(invocation *inv, int flag) {
    if (inv == NULL) internal_error("tried to set flag for null inv");
    inv->inv_packed_record = (inv->inv_packed_record) | flag;
}
void inv_clear_flag(invocation *inv, int flag) {
    if (inv == NULL) internal_error("tried to clear flag for null inv");
    inv->inv_packed_record = (inv->inv_packed_record) & (~flag);
}
int inv_test_flag(invocation *inv, int flag) {
    int f = (inv)?(inv->inv_packed_record):0;
    if (f & flag) return TRUE;
    return FALSE;
}
```

The function inv_set_flag is called from 7/tc, 12/phtd and 12/cinv.
The function inv_clear_flag is called from 7/tc and 12/phtd.
The function inv_test_flag is called from 7/vasp, 7/tc and 12/cinv.

§**4.** And the group number, a non-negative integer:

```
int inv_get_group(invocation *inv) {
    if (inv == NULL) internal_error("tried to read group of null inv");
    return ((inv->inv_packed_record) & GROUP_MASK)/BASE_OF_GROUP;
}
void inv_set_group(invocation *inv, int g) {
    if (inv == NULL) internal_error("tried to set group of null inv");
    if ((g<0) || (g>4095)) internal_error("tried to set group out of range");
    inv->inv_packed_record = ((inv->inv_packed_record) & (~GROUP_MASK))
        + BASE_OF_GROUP*g;
}
```

The function inv_get_group is called from 5/mlc, 7/tc and 12/cinv.
The function inv_set_group is called from 5/mlc and 12/phtd.

§**5.** Likewise the unsorted position:

```
int inv_get_unsorted_position(invocation *inv) {
    if (inv == NULL) internal_error("tried to read USN of null inv");
    return ((inv->inv_packed_record) & USN_MASK)/BASE_OF_USN;
}
void inv_set_unsorted_position(invocation *inv, int n) {
    if (inv == NULL) internal_error("tried to set USN of null inv");
    if ((n<0) || (n>2047)) internal_error("tried to set USN out of range");
    inv->inv_packed_record = ((inv->inv_packed_record) & (~USN_MASK))
        + BASE_OF_USN*n;
}
```

§**6.** The word range:

```
void inv_set_word_range(invocation *inv, int w1, int w2) {
    if (inv == NULL) internal_error("tried to set word range of null inv");
    inv->invocation_w1 = w1; inv->invocation_w2 = w2;
}
```

The function inv_set_word_range is called from 12/phtd.

§**7.** The tokens. Recall that these are stored as a linked list; the following creates a new entry structure.

```
invocation_token_list *itl_new(void) {
    invocation_token_list *itl = CREATE(invocation_token_list);
    itl->token_as_parsed = NULL;
    itl->token_check_to_do = NULL;
    itl->next = NULL;
    return itl;
}
```

§**8.** However, we want to access it as if it were an array. (Speed is not too vital here, and the list sizes are almost always lower than 3.) So:

```
void inv_set_token_check_to_do(invocation *inv, int i, specification *spec) {
    int k;
    invocation_token_list *itl;
    if (i<0) internal_error("tried to set token out of range");
    if (inv->tokens_invoked == NULL) inv->tokens_invoked = itl_new();
    for (itl = inv->tokens_invoked, k = 0; itl; itl = itl->next, k++) {
        if (k == i) { itl->token_check_to_do = spec; return; }
        if (itl->next == NULL) itl->next = itl_new();
    }
}
void inv_set_token_as_parsed(invocation *inv, int i, specification *spec) {
    int k;
    invocation_token_list *itl;
    if (i<0) internal_error("tried to set token out of range");
    if (inv->tokens_invoked == NULL) inv->tokens_invoked = itl_new();
    for (itl = inv->tokens_invoked, k = 0; itl; itl = itl->next, k++) {
        if (k == i) { itl->token_as_parsed = spec; return; }
        if (itl->next == NULL) itl->next = itl_new();
    }
}
```

The function inv_set_token_check_to_do is called from 7/tc.

The function inv_set_token_as_parsed is called from 7/tc and 12/phtd.

§**9.** And similarly for reading them:

```
specification *inv_get_token_check_to_do(invocation *inv, int i) {
    int k;
    invocation_token_list *itl;
    for (itl = inv->tokens_invoked, k = 0; itl; itl = itl->next, k++)
        if (k == i) return itl->token_check_to_do;
    return NULL;
}
specification *inv_get_token_as_parsed(invocation *inv, int i) {
    int k;
    invocation_token_list *itl;
    for (itl = inv->tokens_invoked, k = 0; itl; itl = itl->next, k++)
        if (k == i) return itl->token_as_parsed;
    return NULL;
}
```

The function inv_get_token_check_to_do is called from 12/cinv and 12/ambig.

The function inv_get_token_as_parsed is called from 7/tc, 12/phtd and 12/cinv.

§**10.** The phrase, and the "instead" flag:

```
void set_phrase_invoked(invocation *inv, phrase *ph) {
    inv->phrase_invoked = ph;
}
void inv_set_instead_flag(invocation *inv, int t) {
    if (t) inv_set_flag(inv, INSTEAD_INVFLAG);
    else inv_clear_flag(inv, INSTEAD_INVFLAG);
}
```

The function set_phrase_invoked is called from 12/phtd.
The function inv_set_instead_flag is called from 5/mlc and 12/phtd.

§**11.** The following routine might become more interesting if we ever allowed variable-argument phrases like C's `printf`.

```
int inv_get_number_tokens(invocation *inv) {
    if (inv == NULL) internal_error("tried to read NTI of null inv");
    if (inv->phrase_invoked) return phtd_get_no_tokens(&(inv->phrase_invoked->type_data));
    return 0;
}
```

The function inv_get_number_tokens is called from 7/tc, 12/cinv and 12/ambig.

§**12.** The phrase options invoked:

```
void inv_set_phrase_options(invocation *inv, int w1, int w2) {
    if (inv->phrase_options_invoked == NULL) {
        inv->phrase_options_invoked = CREATE(invocation_options);
        inv->phrase_options_invoked->options = 0;
    }
    inv->phrase_options_invoked->options_invoked_w1 = w1;
    inv->phrase_options_invoked->options_invoked_w2 = w2;
}
```

The function inv_set_phrase_options is called from 12/phtd.

§**13.** Reading the word range:

```
void inv_get_phrase_options(invocation *inv, int *w1, int *w2) {
    if (inv->phrase_options_invoked == NULL) { *w1 = -1; *w2 = -1; }
    else {
        *w1 = inv->phrase_options_invoked->options_invoked_w1;
        *w2 = inv->phrase_options_invoked->options_invoked_w2;
    }
}
```

The function inv_get_phrase_options is called from 7/tc.

§**14.** The functional part is of course the bitmap, which we read and write thus. When no options are set, the bitmap is always 0.

```
int inv_get_phrase_options_bitmap(invocation *inv) {
    if (inv->phrase_options_invoked == NULL) return 0;
    return inv->phrase_options_invoked->options;
}
void inv_set_phrase_options_bitmap(invocation *inv, int further_bits) {
    if (further_bits == 0) return;
    if (inv->phrase_options_invoked == NULL) {
        inv->phrase_options_invoked = CREATE(invocation_options);
        inv->phrase_options_invoked->options_invoked_w1 = -1;
        inv->phrase_options_invoked->options_invoked_w2 = -1;
        inv->phrase_options_invoked->options = 0;
    }
    inv->phrase_options_invoked->options |= further_bits;
}
```

The function inv_get_phrase_options_bitmap is called from 12/cinv and 12/ambig.
The function inv_set_phrase_options_bitmap is called from 12/phod.

§**15. The implied newlines rule.** This is applied only when the invocation passes the following stringent test:

```
int inv_implies_newline(invocation *inv) {
    if (!(inv->phrase_invoked->type_data.say_phrase)) return FALSE;
    if (!(TEST_COMPILATION_MODE(IMPLY_NEWLINES_IN_SAY_CMODE))) return FALSE;
    if (inv_test_flag(inv, SUPPRESS_IMPLIED_NEWLINES_INVFLAG)) return FALSE;
    return TRUE;
}
```

The function inv_implies_newline is called from 12/cinv.

§**16. Invocation lists.** Specifications for phrases parsed in the source text often contain lists of invocations.

```
invocation_list *new_invocation_list(void) {
    invocation_list *invl = CREATE(invocation_list);
    invl->list_head = NULL; invl->list_tail = NULL;
    return invl;
}
```

The function new_invocation_list is called from 7/spec and 7/tc.

§**17.** Appending:

```
void append_invocation_list(invocation_list *to, invocation_list *from) {
    invocation_list_entry *ent;
    for (ent=from->list_head; ent; ent=ent->next)
        add_to_invocation_list(to, ent->listed_inv);
}
```

The function append_invocation_list is called from 5/mlc.

§**18.** New entries are always added to the ends.

```
void add_to_invocation_list(invocation_list *invl, invocation *inv) {
    invocation_list_entry *ent = CREATE(invocation_list_entry);
    ent->listed_inv = inv; ent->next = NULL;
    if (invl->list_head) invl->list_tail->next = ent;
    else invl->list_head = ent;
    invl->list_tail = ent;
}
```

The function add_to_invocation_list is called from 7/tc and 12/phtd.

§**19.** That completes the construction routines. Now, reading the length and first item:

```
int length_of_invocation_list(invocation_list *invl) {
    if (invl == NULL) return 0;
    int L = 0;
    invocation_list_entry *inve;
    for (inve = invl->list_head; inve; inve = inve->next) L++;
    return L;
}
invocation *first_in_invocation_list(invocation_list *invl) {
    if ((invl == NULL) || (invl->list_head == NULL)) return NULL;
    return invl->list_head->listed_inv;
}
```

The function length_of_invocation_list is called from 5/mlc and 7/tc.

§**20.** Sorting the invocations in a list is much more important than it may at first appear, since the sorted order is a precedence order for parsing purposes: that is, the earliest type-checked match is the one accepted, so that being sorted up front gives a possible interpretation of a phrase priority over those sorted to the back.

```
invocation **pigeon_holes = NULL;
int number_of_pigeon_holes = 0;
void sort_spec_invocation_list(invocation_list *invl) {
    int L = length_of_invocation_list(invl);
    ⟨Make sure there are at least L pigeonholes available for sorting into 21⟩;
    int i;
    invocation_list_entry *ent;
    for (i=0, ent=invl->list_head; (i<L) && (ent); i++, ent=ent->next) {
        pigeon_holes[i] = ent->listed_inv;
        inv_set_unsorted_position(pigeon_holes[i], (i>2047)?2047:i);
    }
    qsort(pigeon_holes, L, sizeof(invocation *), inv_comparison);
    for (i=0, ent=invl->list_head; i<L; i++, ent=ent->next)
        ent->listed_inv = pigeon_holes[i];
}
```

The function sort_spec_invocation_list is called from 5/mlc.

§**21.** We allocate 1000 pigeonholes in the first instance, then double each time we run out. (We will quite likely never run out, as 1000 is plenty. But we want to avoid all possible arbitrary limits.)

⟨Make sure there are at least L pigeonholes available for sorting into 21⟩ ≡

```
if (L > number_of_pigeon_holes) {
    number_of_pigeon_holes = 2*L;
    if (number_of_pigeon_holes < 1000)
        number_of_pigeon_holes = 1000;
    pigeon_holes =
        I7_calloc(number_of_pigeon_holes, sizeof(invocation *), INV_LIST_MREASON);
}
```

This code is used in §20.

§**22.** So much for the mechanism. The sorting order is specified by the following.

(a) We first group invocations by group number, which is simply a matter of the sequence in which they are to take effect.

(b) Then, within each group, we sort by logical priority. This is quite an expensive test, and therefore we effectively cache it by referring to the "sequence count" of the phrases instead: that is, to the numbered position in the logical priority ordering of the phrases. It's very important that each phrase has a unique sequence count value, because the C library sorting function `qsort` is unstable, so that it might arbitrarily rearrange invocations which happened to have equal group numbers and sequence counts – a problem because Inform's behaviour in terms of logical priority must be predictable.

(c) Our final sorting is on the original parsing sequence order, which we preserve in cases where two different invocations in the same group both invoke the same phrase (as for instance where "two minutes before two minutes before midnight" can be invoked either as "two minutes before (two minutes before midnight)" or "(two minutes before two minutes) before midnight"). The result is that the comparison function can return 0 only when the invocations are actually equal, so that the instability of the sorting algorithm does not produce any ambiguity in how Inform prioritises phrases.

```
int inv_comparison(const void *i1, const void *i2) {
    invocation **inv1s = (invocation **) i1; invocation *inv1 = *inv1s;
    invocation **inv2s = (invocation **) i2; invocation *inv2 = *inv2s;
    (a) sort by group
    int delta = inv_get_group(inv1) - inv_get_group(inv2);
    if (delta != 0) return delta;
    (b) sort by logical priority
    delta = ph_sequence_count(inv1->phrase_invoked) - ph_sequence_count(inv2->phrase_invoked);
    if (delta != 0) return delta;
    (c) sort by creation sequence
    return inv_get_unsorted_position(inv1) - inv_get_unsorted_position(inv2);
}
```

§**23.** The following macros abstract the process of looping through the invocations in a list. First, declaring the loop variable (actually three variables):

```
define INVOCATION_VARIABLE(inv)
    invocation *inv;
    invocation_list_entry *inv##_ent;
    int inv##_pos;
```

§**24.** The basic loop:

```
define LOOP_THROUGH_INVOCATION_LIST(inv, invl)
    for (inv##_ent = invl->list_head, inv##_pos = 0,
            inv = (inv##_ent)?inv##_ent->listed_inv:NULL;
        inv##_ent;
        inv##_ent = inv##_ent->next, inv##_pos++,
            inv = (inv##_ent)?inv##_ent->listed_inv:NULL)
```

§**25.** A loop which starts at position `from` (which must be another invocation variable) and runs for `size` items:

```
define LOOP_THROUGH_PART_OF_INVOCATION_LIST(inv, invl, p, from, size)
    for (inv##_pos = 0, inv##_ent = invl->list_head;
        ((inv##_pos<from) && (inv##_ent));
        inv##_pos++, inv##_ent = inv##_ent->next) ;
    for (inv##_pos = 0, p = 0, inv = (inv##_ent)?inv##_ent->listed_inv:NULL;
        ((inv##_pos<size) && (inv##_ent));
        inv##_pos++, p++, inv##_ent = inv##_ent->next,
            inv = (inv##_ent)?inv##_ent->listed_inv:NULL)
```

§**26.** A loop starting at `from` and running to the end of the list:

```
define LOOK_AHEAD_THROUGH_INVOCATION_LIST(inv, from, invl)
    for (inv##_ent = from##_ent, inv##_pos = from##_pos,
            inv = (inv##_ent)?inv##_ent->listed_inv:NULL;
        inv##_ent;
        inv##_ent = inv##_ent->next, inv##_pos++,
            inv = (inv##_ent)?inv##_ent->listed_inv:NULL)
```

§**27.** The log routines demonstrate the use of the basic macros:

```
void log_invocation_list(invocation_list *invl) {
    INVOCATION_VARIABLE(inv);
    LOG("Invocation list (%d):\n", length_of_invocation_list(invl));
    LOOP_THROUGH_INVOCATION_LIST(inv, invl)
        LOG("P%d: $e\n", inv_pos, inv);
}
```

The function log_invocation_list is called from 2/dl and 5/mlc.

§**28.** A more detailed version, which hierarchically shows the lists inside the invocations listed:

```
void log_invocation_list_in_detail(invocation_list *invl) {
    INVOCATION_VARIABLE(inv);
    LOG("Invocation list in detail (%d):\n", length_of_invocation_list(invl));
    LOOP_THROUGH_INVOCATION_LIST(inv, invl) {
        int j;
        LOG("P%d: $e\n", inv_pos, inv);
        for (j=0; j<inv_get_number_tokens(inv); j++) {
            specification *tok = inv_get_token_as_parsed(inv, j);
            LOG("  %d: $S\n", j, tok);
            if (spec_has_invocation_list(tok)) {
                STREAM_INDENT(dl);
                log_invocation_list_in_detail(spec_invocation_list(tok));
                STREAM_OUTDENT(dl);
            }
        }
    }
}
```

The function log_invocation_list_in_detail is called from 7/spec.

8 Assertions

`8/tass`: *Traverse for Assertions.w* To manage the overall process of traversing the parse tree for assertion sentences.

`8/sob`: *Subject and Object.w* To keep track of the implicit subject and object in assertion sentences, for use in understanding words like "it" and "here" in subsequent sentences.

`8/refpt`: *Refine Parse Tree.w* To determine which world objects are referred to by noun phrases such as "the table" and "a paper cup" found in assertion sentences being parsed.

`8/creat`: *The Creator.w* This is where all objects, kinds of object, and newly named values are made.

`8/mass`: *Make Assertions.w* To read assertion sentences, sort them according to a detailed grammatical classification and take the primary actions necessary either to generate problem messages, or to infer information.

`8/assem`: *Assemblies.w* To build the complex multi-object assemblies which result from allowing the source text to say things like "in every room is a vehicle".

`8/knowc`: *Character Knowledge.w* Character in the sense of the nature of things – both the kind of an object, but also more generally that it satisfies some proposition.

`8/knowp`: *Property Knowledge.w* This section draws inferences from assertions which seem to be about the properties of things, independent of their location.

`8/relv`: *Relation Knowledge.w* This section draws inferences about the relationships between objects or values.

`8/imp`: *Implications.w* To keep track of a dangerous form of super-assertion called an implication, which is allowed to generalise about properties.

Traverse for Assertions 8/tass

Purpose

To manage the overall process of traversing the parse tree for assertion sentences.

8/tass.¶3-0 The sentence handler tables; §3-10 Performing the main traverse

Definitions

¶1. Inferences are collected during two traverses of the parse tree, which are numbered in the following global variable.

```
int traverse;                                                    always 1 or 2
```

¶2. Sentence handlers provide an abstraction for the choice of what to do with each kind of sentence, on each traverse. This is really only a disguise for a couple of `switch` statements and a lot of function calls. The only real purpose is to make it easier to encapsulate code for one sort of sentence away from the others: if we were writing `C++` or Python, it would just be a method in the class for sentences.

Properly speaking, there can be sentence handlers for any node type at the children-of-root level of the parse tree, although we mostly use this for `SENTENCE_NT` nodes (and then we look further at the verb type in the `VERB_NT` first child). The main traverse is a two-pass operation, and we can supply a routine to do something with the node on either of the passes (or neither, or even both).

```
typedef struct sentence_handler {
    int sentence_node_type;                          usually but not always SENTENCE_NT
    int verb_type;                                   for those which are indeed SENTENCE_NT
    int handle_on_traverse;                          1 or 2 to restrict to that pass, or 0 for both
    void (*handling_routine)(struct parse_node *PN);                          or NULL not to handle
} sentence_handler;
```

The structure sentence_handler is private to this section.

¶3. The sentence handler tables.

define `MAX_OF_NTS_AND_VBS 64`

```
sentence_handler *how_to_handle_nodes[MAX_OF_NTS_AND_VBS];
sentence_handler *how_to_handle_sentences[MAX_OF_NTS_AND_VBS];
```

¶4. We recognise either node types `*_NT`, or node type `SENTENCE_NT` plus an associated verb number `*_VB`. The following macro registers a sentence handler by entering a pointer to it into one of the above tables:

```
define REGISTER_SENTENCE_HANDLER(sh_name) {
    sentence_handler *the_sh = &sh_name##_handler;
    if ((the_sh->sentence_node_type == SENTENCE_NT) && (the_sh->verb_type != -1))
        how_to_handle_sentences[the_sh->verb_type] = the_sh;
    else
        how_to_handle_nodes[the_sh->sentence_node_type] = the_sh;
}
```

---

§**1.** The actual handlers are mostly not declared here (indeed, that's the point of the whole exercise – to allow them to be decentralised). But we do need to know their names, so every `*_SH` constant below must correspond to a sentence handler structure called `*_SH_handler` defined somewhere else in NI.

```
int sentence_handlers_initialised = FALSE;
void initialise_sentence_handlers(void) {
    if (sentence_handlers_initialised) return;
    sentence_handlers_initialised = TRUE;
    ⟨Empty the sentence handler tables 2⟩;
    register_sentence_handlers();
}
```

§**2.** At this stage, all we do is empty the tables. The reason we have to delay before entering the valid handlers is that some of them will be defined in sections of NI appearing after this one in the code: since C requires all identifiers used to be predeclared, this means we can't enter the valid handlers until right at the end of NI. The routine which does so, `register_sentence_handlers`, consists only of a run of `REGISTER_SENTENCE_HANDLER` macro expansions and can be found in Chapter 14.

⟨Empty the sentence handler tables 2⟩ ≡

```
    int i;
    for (i=0; i<MAX_OF_NTS_AND_VBS; i++) {
        how_to_handle_nodes[i] = NULL;
        how_to_handle_sentences[i] = NULL;
    }
```

This code is used in §1.

§3. Performing the main traverse. The following routine is called twice, once with `pass` equal to 1, then with `pass` equal to 2.

`trace_sentences` is true between each pair of `TRACE_NT` nodes, if there are any: these arise from the special debugging sentence consisting only of an asterisk. When tracing, we print an account of what is being read to the debugging log (both here, and in more detail elsewhere), except that we don't bother to print details of the closing `TRACE_NT` node.

```
void traverse_for_assertions(int pass) {
    parse_node *p;
    new_discussion();                    clear memory of what the subject and object of discussion are
    traverse = pass;
    trace_sentences = FALSE;
    log_asterisked_parts_of_parse_tree("start of assertion traverse");
    initialise_sentence_handlers();                in case the sentence handler tables not made yet
    for (TREE_START(p); p; TREE_NEXT(p)) {
        finish_this_session_of_parsing();
        if ((trace_sentences) && (pn_get_node_type(p) != TRACE_NT))
            LOG("\n[$W]\n", p->word_ref1, p->word_ref2);
        ⟨If this sentence can be handled, then do so and continue 4⟩;
        log_subtree(p, 1);
        internal_error_on_node_type(p->down);
    }
}
```

The function traverse_for_assertions is called from 14/i6t.

§4. The outer `if` here should always be true: we verify that the node type is a valid index in the sentence handler tables, then that a sentence handler has indeed been registered for that type. If that should fail, there must be a bug somewhere, and we will throw an internal error.

However, it's entirely open for the sentence handler to choose to do nothing on either or both traverses, so the inner `if` can happily fail.

⟨If this sentence can be handled, then do so and continue 4⟩ ≡

```
    if (((pn_get_node_type(p) >= 0) && (pn_get_node_type(p) < MAX_OF_NTS_AND_VBS))
        && (how_to_handle_nodes[pn_get_node_type(p)])) {
        int desired = how_to_handle_nodes[pn_get_node_type(p)]->handle_on_traverse;
        if (((traverse == desired) || (desired == 0)) &&
            (how_to_handle_nodes[pn_get_node_type(p)]->handling_routine))
            (*(how_to_handle_nodes[pn_get_node_type(p)]->handling_routine))(p);
        continue;
    }
```

This code is used in §3.

§**5.** While most of the sentence handlers are scattered across the rest of NI, two will be given here. The first is the one which acts on `TRACE_NT` asterisks and switches tracing off and on:

```
sentence_handler TRACE_SH_handler =
    { TRACE_NT, -1, 0, switch_sentence_trace };
void switch_sentence_trace(parse_node *PN) {
    if (PN->word_ref2 > PN->word_ref1) {
        int tr = telemetry_recording;
        telemetry_recording = TRUE;
        write_telemetry_note(lw_array[PN->word_ref2].lw_text);
        telemetry_recording = FALSE;
        sentence_problem(_P_(C8TelemetryAccepted),
            "that's a message for the Author, not me",
            "so I'll note it down in the Telemetry file (if you're keeping "
            "one.)");
        telemetry_recording = tr;
    } else {
        trace_sentences = 1 - trace_sentences;
        if (traverse == 1) debug_log_tracing(trace_sentences, "Pass 1");
        else debug_log_tracing(trace_sentences, "Pass 2");
    }
}
```

§**6.** And the other is the sentence handler for `SENTENCE_NT` itself, which is special because it looks at the `VERB_NT` child of the sentence and then refers on to other sentence handlers accordingly:

```
sentence_handler SENTENCE_SH_handler =
    { SENTENCE_NT, -1, 0, handle_sentence_with_primary_verb };
void handle_sentence_with_primary_verb(parse_node *p) {
    prevailing_mood = UNKNOWN_CE;
    if (p->down == NULL) ⟨Handle a sentence with no primary verb 7⟩;
    internal_error_if_node_type_wrong(p->down, VERB_NT);
    prevailing_mood = pn_int_annotation(p->down, verbal_certainty_ANNOT);
    ⟨Issue problem message if either subject or object contains mismatched brackets 10⟩;
    ⟨Act on the primary verb in the sentence 8⟩;
}
```

§**7.** A sentence node with no children indicates that we couldn't find any verb earlier. This might just be a piece of quoted matter which is intended as the description or initial appearance of the most recent object, but in all other eventualities we must produce a "no such sentence" problem.

⟨Handle a sentence with no primary verb 7⟩ ≡

```
    if ((p->word_ref1 == p->word_ref2) &&
        (vocab_test_flags(p->word_ref1, TEXT_MC+TEXTWITHSUBS_MC))) {
        if (traverse == 2) set_appearance(p->word_ref1);
        return;
    }
    ⟨Issue problem message about not being able to find a verb 9⟩;
    return;
```

This code is used in §6.

§**8.** We now use the other sentence-handler table, with almost the same code as for the first (above). A small point of difference is that it's allowed for a valid verb number to have no handler: if so, we handle the verb by doing nothing on either traverse, of course.

⟨Act on the primary verb in the sentence 8⟩ ≡

```
int vn = pn_int_annotation(p->down, verb_id_ANNOT);
if ((vn < 0) || (vn >= MAX_OF_NTS_AND_VBS)) {
    LOG("Unimplemented verb %d\n", pn_int_annotation(p->down, verb_id_ANNOT));
    internal_error_on_node_type(p->down);
}
if (how_to_handle_sentences[vn]) {
    int desired = how_to_handle_sentences[vn]->handle_on_traverse;
    if (((traverse == desired) || (desired == 0)) &&
        (how_to_handle_sentences[vn]->handling_routine))
        (*(how_to_handle_sentences[vn]->handling_routine))(p);
}
```

This code is used in §6.

§**9.** The number of different cases this splits into is a typical example of how beta-testing puts emphasis on unexpected parts of the code: it never occurred to me in the original design that NI would be found so sensitive to punctuation errors as between comma, paragraph break and semicolon, nor that this would be the place where that sensitivity would begin to bite.

⟨Issue problem message about not being able to find a verb 9⟩ ≡

```
int i;
if [[p == before/every/after/when/instead/check/carry/report ...]]
    sentence_problem(_P_(C8RuleWithoutColon),
        "I can't find a verb that I know how to deal with, "
        "so can't do anything with this sentence. It looks "
        "as if it might be a rule definition",
        "but if so then it is lacking the necessary colon (or comma). "
        "The punctuation style for rules is 'Rule conditions: do this; "
        "do that; do some more.' Perhaps you used a full stop instead "
        "of the colon?");
else if [[p == if ...]]
    sentence_problem(_P_(C8IfOutsidePhrase),
        "I can't find a verb that I know how to deal with. This looks "
        "like an 'if' phrase which has slipped its moorings",
        "so I am ignoring it. ('If' phrases, like all other such "
        "instructions, belong inside definitions of rules or phrases - "
        "not as sentences which have no context. Maybe a full stop was "
        "accidentally used instead of semicolon, so that you "
        "inadvertently ended the last rule early?)");
else if [[p == ... COMMA ... : i]] sentence_problem(_P_(C8NoSuchVerbComma),
    "I can't find a verb here that I know how to deal with",
    "so I am ignoring this sentence altogether. (I notice there's a "
    "comma here, which is sometimes used to abbreviate rules which "
    "would normally be written with a colon - for instance, 'Before "
    "taking: say \"You draw breath.\"' can be abbreviated to 'Before "
    "taking, say...' - but that's only allowed for Before, Instead and "
    "After rules. I mention all this in case you meant this sentence "
    "as a rule in some rulebook, but used a comma where there should "
    "have been a colon ':'?");
```

```
    else sentence_problem(_P_(C8NoSuchVerb),
        "I can't find a verb here that I know how to deal with",
        "so I am ignoring this sentence altogether.");
```

This code is used in §7.

§**10.** Inform source text does not make much use of parentheses to group subexpressions, but the ability does exist, and we defend it a little here:

⟨Issue problem message if either subject or object contains mismatched brackets 10⟩ ≡

```
    if ((p->down->next) && (p->down->next->next) &&
        ((brackets_unbalanced(p->down->next->word_ref1,
        p->down->next->word_ref2)) ||
        (brackets_unbalanced(p->down->next->next->word_ref1,
        p->down->next->next->word_ref2)))) {
        quote_source(1, current_sentence);
        quote_words(2, p->down->next->word_ref2+1, p->down->next->word_ref2+1);
        quote_words(3, p->down->next->word_ref1, p->down->next->word_ref2);
        quote_words(4, p->down->next->next->word_ref1, p->down->next->next->word_ref2);
        handmade_problem(_P_(BelievedImpossible));
        if (p->down->next->next->word_ref1 >= 0)
            issue_problem_segment(
                "I must be misreading the sentence %1. The verb "
                "looks to me like '%2', but then the brackets don't "
                "match in what I have left: '%3' and '%4'.");
        else
            issue_problem_segment(
                "I must be misreading the sentence %1. The verb "
                "looks to me like '%2', but then the brackets don't "
                "match in what I have left: '%3'.");
        issue_problem_end();
        return;
    }
```

This code is used in §6.

Subject and Object 8/sob

Purpose

To keep track of the implicit subject and object in assertion sentences, for use in understanding words like "it" and "here" in subsequent sentences.

Definitions

¶1. To define the terms of reference for this chapter, an "assertion" is a sentence such as

> In the middle drawer is a sugared almond.

An "inference" is a basic fact deduced from such a sentence. The above sentence generates three:

O82⟨middle drawer⟩ - `CONTAINSTHINGS_INF` - Certain - 1:O84⟨sugared almond⟩

O84⟨sugared almond⟩ - `PARENTAGE_INF` - Certain - 1:O82⟨middle drawer⟩

O84⟨sugared almond⟩ - `ISAROOM_INF` - Impossible

---

§**1.** "It" and "here", and implicit forms of them such as statements which omit to specify the object they apply to, are surprisingly difficult to parse. Consider:

> The Pavilion is a room. East is the Cricket Square.

East of where? Clearly of the current subject, the Pavilion (*not* the room kind). On the other hand,

> On the desk is a pencil. It has description "2B."

"It" here is the pencil, not the desk. To disentangle such things, we keep track of two different running references: the current subject and the current object. English is an SVO language, so that in assertions of the form "X is Y", X is the subject and Y the object. But it will turn out to be more complicated than that, because we disregard all references which are not to tangible things and kinds.

§**2.** The routine `new_discussion` is called when we reach a heading or other barrier in the source text, to make clear that there has been a change of the topic discussed. `change_discussion_topic` is the only place where the subject and object are changed: it is called once at the end of processing each assertion during each pass.

```
world_object *object_of_sentences, *subject_of_sentences;
int subject_seems_to_be_plural = FALSE;

world_object *get_current_subject(void) {
    return subject_of_sentences;
}

void set_current_subject(world_object *wo) {
    subject_of_sentences = wo;
}

sentence_handler HEADING_SH_handler =
    { HEADING_NT, -1, 0, handle_heading };

void handle_heading(parse_node *PN) {
    new_discussion();
}

void new_discussion(void) {
    subject_of_sentences = NULL; object_of_sentences = NULL;
```

```
}
void change_discussion_topic(world_object *ox, world_object *oy) {
    world_object *old_sub = subject_of_sentences,
        *old_obj = object_of_sentences;
    subject_seems_to_be_plural = FALSE;
    if (current_sentence->word_ref2 - current_sentence->word_ref1 > 0)
        near_start_of_extension = 0;
    pn_set_interpretation_of_subject(current_sentence, subject_of_sentences);
    if (pn_has_annotation(current_sentence, implicit_in_creation_of_ANNOT))
        return;
    if (is_a_direction_object(ox)) ox = NULL;
    if (ox != NULL) subject_of_sentences = ox;
    if ((oy != NULL) && (oy->kind_flag == FALSE)) object_of_sentences = oy;
    else {
        if (ox != NULL) object_of_sentences = ox;
    }
    if (subject_of_sentences != old_sub)
        LOGIF(PRONOUNS, "[Changed subject of sentences to $O]\n",
            subject_of_sentences);
    if (object_of_sentences != old_obj)
        LOGIF(PRONOUNS, "[Changed object of sentences to $O]\n",
            object_of_sentences);
}
void subject_of_discussion_a_list(void) {
    subject_seems_to_be_plural = TRUE;
}
```

The function get_current_subject is called from 6/asp and 8/mass.
The function set_current_subject is called from 8/mass.
The function new_discussion is called from 4/iext and 8/tass.
The function change_discussion_topic is called from 8/mass.
The function subject_of_discussion_a_list is called from 8/mass.

§**3.** The "appearance" is not a property as such. When a quoted piece of text is given without explanation, NI takes it to be either a `description` (in the case of a room) or an `initial` (otherwise). Unfortunately we have no means of knowing yet whether an object is to be a room. We therefore record the text in a special pseudo-property called the "appearance". This will ultimately lead to inferences being drawn about genuine properties in the orthodox way, but not yet.

```
void set_appearance(int w) {
    world_object *wo = get_current_subject();
    specification *spec;
    if (near_start_of_extension >= 1) {
        extension_file *ef = NULL;
        if (lw_array[w].lw_source.file_of_origin)
            ef = sf_get_extension_corresponding(lw_array[w].lw_source.file_of_origin);
        if (ef) {
            dequote_word(w);
            switch (near_start_of_extension++) {
                case 1: ef_set_rubric(ef, lw_array[w].lw_text); break;
                case 2: ef_set_extra_credit(ef, lw_array[w].lw_text);
                    near_start_of_extension = 0; break;
            }
```

```
        }
        return;
    }
    spec = new_actual_CONSTANT_spec(is_a_literal(w, w));
    spec->word_ref1 = w; spec->word_ref2 = w;
    if (wo == NULL) {
        sentence_problem(_P_(C8TextWithoutSubject),
            "I'm not sure what you're referring to",
            "that is, I can't decide to what room or thing you intend "
            "that text to belong. Perhaps you could rephrase this more "
            "explicitly? ('The description of the Inner Sanctum is...')");
        return;
    }
    if (wo->quoted_appearance != NULL) {
        parse_node *P1, *P2;
        P1 = new_nounphrase_raw(wo->quoted_appearance->word_ref1,
            wo->quoted_appearance->word_ref1);
        P2 = new_nounphrase_raw(w, w);
        contradiction_problem(_P_(C8TwoAppearances),
            P1, P2, wo,
            "seems to have two different descriptions",
            "perhaps because you intended the second description "
            "to apply to something mentioned in between, but "
            "declared it in such a way that it was never the "
            "subject of an assertion. For instance, 'The Forest "
            "Clearing is northeast of the Woods.' makes the Forest "
            "Clearing the current room being discussed, but "
            "'Northeast of the Woods is the Forest Clearing.' "
            "leaves the room under discussion unchanged, because "
            "the Forest Clearing is not the subject of the sentence.");
        return;
    }
    wo->quoted_appearance = spec;
    wo->quoted_appearance_sentence = current_sentence;
}
```

The function set_appearance is called from 8/tass.

Refine Parse Tree 8/refpt

Purpose

To determine which world objects are referred to by noun phrases such as “the table” and “a paper cup” found in assertion sentences being parsed.

8/refpt.§16 And surgery; §17 With surgery; §18 Location surgery

§**1.**

```
void resolve_references(parse_node *p) {
    if (p == NULL) internal_error("Resolve references on null pn");
    refine_parse_tree(p, FORBID_CREATION);
}
```

The function resolve_references is called from 6/treec and 9/pp.

§**2.** Resolution of assertions is what happens when an assertion like “On the table is a paper cup” is being read, during the first pass. The noun phrases “the table” and “a paper cup” need to be parsed to see if they refer to things already known to NI, and suitable objects must be created if not.

What happens is that the parse tree structures representing the nouns in any such sentence are “resolved”, using the routine below, which annotates the parse tree with object (and other value) references as necessary.

The flag `permit_creation` can have three values: `FALSE`, which forbids us from creating an object that the text can represent; `TRUE`, which allows us to; or the following special value which positively obliges us to (except in the one special case of the noun phrase “it”):

```
define FORBID_CREATION 0
define ALLOW_CREATION 1
define MANDATE_CREATION 2

int forbid_nowhere = FALSE;
void refine_parse_tree(parse_node *p, int permit_creation) {
    if (p == NULL) internal_error("Refine parse tree on null pn");
    if (pn_int_annotation(p, resolved_ANNOT)) return;
    pn_annotate_int(p, resolved_ANNOT, TRUE);
    LOGIF(NP_RESOLUTION, "Refine parse tree (%s permit creation):\n$T",
        (permit_creation?"do":"do not"), p);
    switch(pn_get_node_type(p)) {
        case X_OF_Y_NT:
            refine_parse_tree(p->down, permit_creation);
            refine_parse_tree(p->down->next, FORBID_CREATION);
            return;
        case WITH_NT:
            refine_parse_tree(p->down, permit_creation);
            perform_with_surgery(p);
            perform_with_action_surgery(p);
            if (pn_int_annotation(p, resolved_ANNOT) == FALSE) {
                refine_parse_tree(p, permit_creation);
                return;
```

```
            }
            return;
        case AND_NT:
            refine_parse_tree(p->down, permit_creation);
            refine_parse_tree(p->down->next, permit_creation);
            perform_and_surgery(p);
            return;
        case RELATIONSHIP_NT:
            perform_location_surgery(p);
            if (pn_get_node_type(p) == AND_NT) {
                pn_annotate_int(p, resolved_ANNOT, FALSE);
                refine_parse_tree(p, permit_creation);
                return;
            }
            ⟨RR - Resolve RELATIONSHIP nodes 4⟩; return;
        case CALLED_NT: ⟨RR - Resolve CALLED nodes 3⟩; return;
        case KIND_NT: ⟨RR - Resolve KIND nodes 5⟩; return;
        case NOUNPHRASE_NT: ⟨RR - Resolve NOUNPHRASE nodes 6⟩; return;
    }
}
```

The function refine_parse_tree is called from 8/creat.

§**3.** A `CALLED_NT` node has two children: in the phrase "an X called Y", they will represent X and Y respectively. Y must be created afresh whatever its name, since the whole point of "called" is that it enables the designer to use names which would otherwise be interpreted as meaning something significant: it is a sort of literal escape, like the backslash character in C strings. X is never something new: it is expected to be a kind. We convert the whole node into a simple `NOUNPHRASE_NT` with the name of Y and the kind of X. In this way, all `CALLED_NT` nodes are removed from the tree.

⟨RR - Resolve CALLED nodes 3⟩ ≡

```
    LOGIF(NP_RESOLUTION, "Called_NT node:\n$T", p);
    switch(pn_get_node_type(p->down)) {
        case RELATIONSHIP_NT: {
                LOGIF(NP_RESOLUTION, "Called_NT as relationship:\n$T", p);
            parse_node *x_pn = p->down->down->next;
            parse_node *name_pn = p->down->next;
            pn_set_node_type(p, RELATIONSHIP_NT);
            pn_annotate_int(p, relationship_node_type_ANNOT,
                pn_int_annotation(p->down, relationship_node_type_ANNOT));
            pn_set_node_type(p->down, CALLED_NT);
            p->down->next = x_pn;
            p->down->down->next = name_pn;
            pn_annotate_int(p, resolved_ANNOT, FALSE);
            refine_parse_tree(p, permit_creation);
            return;
        }
    }
    refine_parse_tree(p->down, FORBID_CREATION);
    if (pn_int_annotation(p->down, nounphrase_multiplicity_ANNOT) > 1) {
        sentence_problem(_P_(C8MultipleCalled),
            "I can only make a single 'called' thing at a time",
            "or rather, the 'called' is only allowed to apply to one thing "
```

```
            "at a time. For instance, 'A thing called a vodka and tonic is "
            "on the table.' is allowed, but 'Two things called vodka and tonic' "
            "is not.");
    }
    forbid_nowhere = TRUE;
    if (permit_creation == FORBID_CREATION)
        sentence_problem(_P_(BelievedImpossible),
            "'called' can't be used in this context",
            "and is best reserved for full sentences.");
    else refine_parse_tree(p->down->next, MANDATE_CREATION);
    forbid_nowhere = FALSE;
    LOGIF(NP_RESOLUTION, "Non-location called:\n$T", p);
    LOGIF(NP_RESOLUTION, "Becomes:\n$T\n - - -\n", p);
```

This code is used in §2.

§**4.** A `RELATIONSHIP_NT` node may have no children, and if so it represents "here". Otherwise we resolve its children, allowing creations: thus "a green marble in a blue box" may cause both marble and box to be created as new.

⟨RR - Resolve RELATIONSHIP nodes 4⟩ ≡ {

```
    binary_predicate *bp;
    if (p->down == NULL) return;
    refine_parse_tree(p->down, permit_creation);
    bp = pn_get_relationship(p);
    if (bp) {
        world_object *dir = bp_get_mapping_direction(bp_get_reversal(bp));
        if (dir) {
            LOGIF(NP_RESOLUTION, "Directional predicate with BP %s ($O, $O)\n",
                bp_get_log_name(bp), bp_get_mapping_direction(bp), dir);
            pn_annotate_int(p, relationship_node_type_ANNOT, DIRECTION_RELN);
            p->down->next =
                new_nounphrase_raw(dir->word_ref1, dir->word_ref2);
            pn_set_node_type(p->down->next, NOUNPHRASE_NT);
            pn_set_refers_to(p->down->next, dir);
        }
    }
    if (p->down->next != NULL) refine_parse_tree(p->down->next, permit_creation);
}
```

This code is used in §2.

§**5.** A `KIND_NT` node may have no children, and if so it represents the bare word "kind": the reference must be to the kind "kind" itself. Otherwise it has one child. Usually this will be the name of an object, e.g. "container", as in "a kind of container", but it might be something like the name of a type, as in "a kind of number".

If this code completes without error, then any `KIND_NT` node `p` either (a) means the kind object held in `pn_get_refers_to(p)`, or (b) has `pn_get_refers_to(p)` null and means "kind of value".

⟨RR - Resolve KIND nodes 5⟩ ≡

```
if (p->down == NULL) { pn_set_refers_to(p, kind_kind); return; }
refine_parse_tree(p->down, FORBID_CREATION);
pn_set_refers_to(p, pn_get_refers_to(p->down));
if (pn_get_refers_to(p) == NULL) {
    if (pn_get_node_type(p->down) != VALUE_NT) goto UnrecognisedKind;
    if (spec_is_generic_CONSTANT(pn_get_evaluation(p->down))) {
        kind_of_value *kov = spec_get_kind_of_value(pn_get_evaluation(p->down));
        if (is_kova(kov, OBJECT_TY)) {
            kind_of_object_problem(_P_(C8KindOfObject));
            return;
        }
        if (is_kova(kov, ANY_VALUE_TY))
            return;
        LOG("$X", pn_get_evaluation(p->down));
        sentence_problem(_P_(C8KindOfExotica),
            "you are only allowed to create kinds of objects "
            "(things, rooms, and so on) and kinds of 'value'",
            "so for example 'colour is a kind of value' is "
            "allowed but 'prime is a kind of number' is not.");
        break;
        return;
    }
    if (spec_is_UNKNOWN(pn_get_evaluation(p->down))) {
        UnrecognisedKind:
        LOG("p-tree:\n$T", p);
        sentence_problem(_P_(C8NoSuchKind),
            "I don't recognise that as a kind",
            "such as 'room' or 'door'.");
        return;
    }
    sentence_problem(_P_(C8KindOfActualValue),
        "I don't recognise that as a kind",
        "such as 'room' or 'door': it would need to be "
        "straightforwardly the name of a kind, and not "
        "be qualified with adjectives like 'open'.");
    return;
} else {
    if (pn_get_refers_to(p)->kind_flag == FALSE) {
        sentence_problem(_P_(C8KindOfInstance),
            "kinds can only be made from other kinds",
            "so 'a kind of container' is allowed but 'a kind of "
            "Mona Lisa' (where Mona Lisa is a specific thing "
            "you've already made), wouldn't be allowed. There is "
            "only one Mona Lisa.");
```

```
            return;
        }
    }
```

This code is used in §2.

§**6.** The simple description of what happens to a `NOUNPHRASE_NT` node is that if it's an existing object, then `refers` is set to that; and if not, a new object of that name is created, and `refers` set to the new thing. But "it" is an exception and we also look out for action patterns, property names and values like "121" or "number of turns". And if `permit_creation` is `FALSE`, then we give up and set the `refers` to null rather than create something new. This is important for cases where the noun phrase is known to be not an object – for instance, for the Y in "an X with Y", which is expected to be a property list. The more complicated description is as follows:

⟨RR - Resolve NOUNPHRASE nodes 6⟩ ≡

```
    if ((p->word_ref1 == -2) && (p->word_ref1 == -2)) {
        pn_make_player_noun_phrase(p);
        return;
    }
    ⟨RRN - Resolve new-property of 7⟩;
    ⟨RRN - Resolve IT 8⟩;
    if ([[p == nowhere]] &&
        (unexpectedly_upper_case(p->word_ref1) == FALSE) &&
        (forbid_nowhere == FALSE)) {
        pn_set_refers_to(p, NULL);
        return;
    }
    if (permit_creation != MANDATE_CREATION) {
        if [[p == description]] {
            ⟨RRN - Resolve property names 13⟩;
        } else {
            ⟨RRN - Resolve values 9⟩;
            ⟨RRN - Resolve property names 13⟩;
        }
        ⟨RRN - Resolve action patterns 14⟩;
    }
    Unresolved:
    LOGIF(NP_RESOLUTION, "Unresolved\n");
    if (permit_creation) pn_set_node_type(p, CREATED_NT);
    else pn_set_refers_to(p, NULL);
```

This code is used in §2.

§**7.** The following is needed to handle something like "colour of the box", where "colour" is a newly designed type which has coincided with a property, but which was not known to be a property when the parse tree was grown.

⟨RRN - Resolve new-property of 7⟩ ≡ {

```
    int i = is_word_intermediate(of_V, p->word_ref1, p->word_ref2);
    if (i >= 0) {
        property_name *prn = parse_property_name(p->word_ref1, i-1);
        if (prn) {
            if (prn_coincides_with_kind_of_value(prn)) {
                LOGIF(NP_RESOLUTION, "Resolving new-property of: $Y\n", prn);
                pn_set_node_type(p, X_OF_Y_NT);
                p->down = new_nounphrase_articled(i+1, p->word_ref2);
                p->down->next = new_nounphrase_worldly(p->word_ref1, i-1, FALSE);
                pn_annotate_int(p, resolved_ANNOT, FALSE);
                LOGIF(NP_RESOLUTION, "Resolved new-property to:\n$T\n", p);
                refine_parse_tree(p, permit_creation);
                return;
            }
        }
    }
}
```

This code is used in §6.

§**8.** A noun phrase consisting of a pronoun has `refers` set to the relevant thing. (If we had more and better pronouns, they would go here.)

⟨RRN - Resolve IT 8⟩ ≡

```
    if (pn_int_annotation(p, nounphrase_article_ANNOT) == IT_ART) {
        if ([[p == they]] && (subject_seems_to_be_plural)) {
            sentence_problem(_P_(C8EnigmaticThey),
                "I'm unable to handle 'they' here",
                "since it looks as if it needs to refer to more than one "
                "object here, and that's something I can't manage.");
            return;
        }
        if (object_of_sentences == NULL) {
            sentence_problem(_P_(C8EnigmaticPronoun),
                "I'm not sure what to make of the pronoun here",
                "since it is unclear what previously mentioned thing "
                "is being referred to. In general, it's best only to use "
                "'it' where it's unambiguous, and it may be worth noting "
                "that 'they' is not allowed to stand for more than one "
                "object at a time.");
            return;
        }
        pn_set_refers_to(p, object_of_sentences);
        LOGIF(PRONOUNS, "Interpreting 'it' as $O\n$P", object_of_sentences, current_sentence);
        return;
    }
```

This code is used in §6.

§**9.** The hard parsing is left to the routine `parse_expression`, which converts textual matter into a specification. It can return a number of type IDs other than the ones checked below, and indeed can return other forms of the type `DESCRIPTION_SPC`: but these other forms are never valid in an assertion, so we need not worry about them here.

⟨RRN - Resolve values 9⟩ ≡

```
{
    specification *spec;
    LOGIF(NP_RESOLUTION, "Resolve hunts through %d, %d, $W\n",
        p->word_ref1, p->word_ref2, p->word_ref1, p->word_ref2);
    if ((p->word_ref1 < 0) || (p->word_ref1 > p->word_ref2)) {
        log_subtree(current_sentence, 1);
        internal_error("Tried to resolve malformed noun-phrase");
    }
    if ((p->word_ref1 == p->word_ref2) &&
        (vocab_test_flags(p->word_ref1, BUILTINTYPE_MC)) &&
        ([[p == action]] == FALSE)) {                    "action" left to error handling elsewhere
        spec = new_generic_CONSTANT_type(kova(
            vocab_get_literal_number_value(lw_array[p->word_ref1].lw_identity)));
    } else {
        int t1 = p->word_ref1, t2 = p->word_ref2;
        if [[t1, t2 == action of ... --> t1, t2]] {
            spec = parse_expression(t1, t2, CONDITION_EXPCON);
            if (species_is(spec, TEST_ACTION_SPC)) {
                coerce_TEST_ACTION_to_STORED_ACTION(spec);
                spec->word_ref1 = p->word_ref1; spec->word_ref2 = p->word_ref2;
            } else spec = new_UNKNOWN_spec();
        } else {
            spec = parse_expression(t1, t2, DESCRIPTIVE_TYPE_EXPCON);
            if (spec_is_UNKNOWN(spec)) {
                meaning_list *ml = SP_excerpt(QUANTITY_MC, t1, t2);
                if (ml) {
                    quantity *q = RETRIEVE_POINTER_quantity(em_data(ml_meaning(ml)));
                    if ((q) && (qty_is_a_variable(q))) spec = new_QUANTITY_spec(q);
                }
            }
        }
    }
    if (spec_get_quantifier(spec)) {
        if (quant_can_be_used_in_assertions(spec_get_quantifier(spec)) == FALSE) {
            sentence_problem(_P_(C8ComplexDeterminer),
                "complicated determiners are not allowed in assertions",
                "so for instance 'More than three people are in the Dining Room' "
                "or 'None of the containers is open' will be rejected. Only "
                "simple numbers will be allowed, as in examples like 'Three "
                "people are in the Dining Room.'");
            return;
        }
        if (spec_get_quantifier(spec) == for_all_quantifier) {
            if ((spec_get_described_kind(spec)) && (spec_get_described_object(spec) == NULL)
&&
                (number_of_adjectives_applied_to(spec) == 0)) {
                pn_set_refers_to(p, spec_get_described_kind(spec));
```

```
            pn_set_node_type(p, EVERY_NT);
            return;
        }
        sentence_problem(_P_(C8ComplexEvery),
            "in an assertion 'every' or 'all' can only be used with a kind",
            "so for instance 'A coin is in every container' is all right, "
            "because 'container' is a kind, but not 'A coin is in every "
            "open container', because 'open container' is now a kind "
            "qualified by a property which may come or go during play. "
            "(This problem sometimes happens because a thing has been "
            "called something like an 'all in one survival kit' - if you "
            "need that sort of name, try using 'called' to set it.)");
        return;
    }
}
if (spec_get_quantifier(spec) == at_least_quantifier) {
    pn_annotate_int(p, nounphrase_multiplicity_ANNOT, spec_get_quantification_parameter(spec));
}
if ([[p == below]] && (wo_down_direction)) {
    pn_set_refers_to(p, wo_down_direction);
    pn_set_node_type(p, NOUNPHRASE_NT);
    return;
}
if ([[p == above]] && (wo_up_direction)) {
    pn_set_refers_to(p, wo_up_direction);
    pn_set_node_type(p, NOUNPHRASE_NT);
    return;
}
LOGIF(NP_RESOLUTION, "Resolved as value: $S\n", spec);
if (spec_is_generic_CONSTANT(spec)) {
    pn_set_evaluation(p, spec);
    pn_set_node_type(p, VALUE_NT);
    return;
}

if (p->word_ref2 > p->word_ref1) {
    if (vocab_test_flags(p->word_ref2, BUILTINTYPE_MC)) {
        kind_of_value *ckov =
            kova(vocab_get_literal_number_value(lw_array[p->word_ref2].lw_identity));
        int X;
        LOGIF(NP_RESOLUTION,
            "Possibly $W is a modifier of kind of value $u\n",
                p->word_ref1, p->word_ref2-1, ckov);
        X = kov_parse_modifying_adjectives(ckov, p->word_ref1, p->word_ref2-1);
        if (X >= 0) {
            LOGIF(NP_RESOLUTION, "Modifier accepted: bitmap X = %d\n", X);
            pn_set_evaluation(p, new_generic_CONSTANT_type(ckov));
            pn_set_node_type(p, VALUE_NT);
            pn_annotate_int(p, creation_modifier_ANNOT, X);
            return;
        }
    }
}

if (species_is(spec, DESCRIPTION_SPC)) {
```

```
    adjective_list_entry *tr;
    ⟨RRN1 - Try adjectives plus KOV 12⟩;
    ⟨RRN1 - Try single adjective 10⟩;
    ⟨RRN1 - Try adjectives plus kind 11⟩;
    int original_adjectives = number_of_adjectives_applied_to(spec);
    if (spec_get_proposition(spec) == NULL)
        LOOP_THROUGH_ADJECTIVE_LIST(tr, spec) {
            if (adjective_used_positively(tr)) {
                if (spec_get_described_object(spec)) goto Unresolved;
                LOGIF(NP_RESOLUTION,
                    "Resolve forcing SP to propositional form\n");
                spec_convert_docket_to_proposition(spec);
                break;
            }
        }
    pn_set_refers_to(p, NULL);
    if (spec_get_described_object(spec) != NULL) {
        if (spec_get_proposition(spec) == NULL) pn_set_refers_to(p, spec_get_described_object(spec));
        pn_set_node_type(p, NOUNPHRASE_NT);
    } else if (spec_get_described_kind(spec) != NULL) {
        pn_set_refers_to(p, spec_get_described_kind(spec));
        pn_set_evaluation(p, spec);
        pn_set_node_type(p, INSTANCE_NT);
    } else if (spec_get_proposition(spec)) {
        world_object *kv = vars_kind_of_variable_0(spec_get_proposition(spec));
        if (kv) pn_set_refers_to(p, kv);
        pn_set_evaluation(p, spec);
        pn_set_node_type(p, INSTANCE_NT);
        return;
    }
    if ((pn_int_annotation(p, nounphrase_article_ANNOT) == DEF_ART)
        && (original_adjectives == 0))
        return;
    if ((spec_get_described_object(spec) != NULL) && (original_adjectives > 0))
        goto Unresolved;
    if (pn_get_refers_to(p) != NULL) return;
}
if (species_is(spec, NONLOCAL_VARIABLE_SPC)) {
    pn_set_evaluation(p, spec);
    pn_set_node_type(p, VALUE_NT);
    return;
}
if (family_is(spec, VALUE_FMY)) {
    kind_of_value *kov;
    if (spec_is_phrasal(spec)) break;
    pn_set_evaluation(p, spec);
    kov = spec_get_kind_of_value(spec);
    if (is_kova(kov, OBJECT_TY)) {
        world_object *wo_reference = wo_of_CONSTANT_OBJECT_if_any(spec);
        if (wo_reference) {
            if (wo_reference->kind_flag) pn_set_node_type(p, INSTANCE_NT);
            else pn_set_node_type(p, NOUNPHRASE_NT);
```

```
                pn_set_refers_to(p, wo_reference);
                return;
            }
        }
        pn_set_node_type(p, VALUE_NT);
        return;
    }
}
```

This code is used in §6.

§10.

⟨RRN1 - Try single adjective 10⟩ ≡

```
    if ((number_of_adjectives_applied_to(spec) == 1)
        && (spec_get_described_kov(spec) == NULL)
        && (spec_get_described_object(spec) == NULL)
        && (spec_get_described_kind(spec) == NULL)
        && (spec_get_proposition(spec) == NULL)
        && (pn_get_refers_to(p) == NULL)) {
        adjectival_phrase *aph = get_adjective_from_list_entry(first_adjective_list_entry(spec));
        if (adjective_used_positively(first_adjective_list_entry(spec))) {
            quantity *q = aph_has_ENUMERATIVE_meaning(aph);
            if (q) {
                property_name *prn = kov_get_coinciding_property(qty_kind_of_value(q));
                LOGIF(TABLE_CONSTRUCTION, "Adjectival usage at table entry\n");
                pn_set_property(p, prn);
                pn_set_aph(p, aph);
                pn_set_node_type(p, ADJECTIVE_NT);
                pn_set_evaluation(p, new_QUANTITY_spec(q));
                pn_set_full_phrase_evaluation(p, spec);
                return;
            }
        }
        property_name *prn = aph_has_EORP_meaning(aph);
        if (prn) {
            pn_set_property(p, prn);
            pn_set_aph(p, aph);
            if (adjective_used_positively(first_adjective_list_entry(spec)))
                pn_annotate_int(p, negated_boolean_ANNOT, FALSE);
            else
                pn_annotate_int(p, negated_boolean_ANNOT, TRUE);
            pn_set_node_type(p, ADJECTIVE_NT);
            pn_set_evaluation(p, NULL);
            pn_set_full_phrase_evaluation(p, spec);
            return;
        }
    }
```

This code is used in §9.

§**11.**

⟨RRN1 - Try adjectives plus kind 11⟩ ≡

```
    if ((number_of_adjectives_applied_to(spec) > 0) && (spec_get_proposition(spec) == NULL))
{
        if (spec_get_described_kind(spec) != NULL) {
            if (grow_as_NP_with_adjectives(p, spec)) {
                LOGIF(NP_RESOLUTION, "grow_as_NP_with_adjectives:\n$T", p);
                return;
            }
        }
        if (spec_get_described_kov(spec) != NULL) {
            if (grow_as_NP_with_adjectives(p, spec)) {
                LOGIF(NP_RESOLUTION, "grow_as_NP_with_adjectives:\n$T", p);
                return;
            }
        }
    }
```

This code is used in §9.

§**12.**

⟨RRN1 - Try adjectives plus KOV 12⟩ ≡

```
    if ((spec_get_described_kov(spec) != NULL) && (number_of_adjectives_applied_to(spec) > 0))
{
        kind_of_value *kov = spec_get_described_kov(spec);
        pn_set_evaluation(p, new_generic_CONSTANT_type(kov));
        pn_set_node_type(p, VALUE_NT);
        spec_convert_docket_to_proposition(spec);
        pn_set_creation_proposition(p, spec_get_proposition(spec));
        return;
    }
```

This code is used in §9.

§**13.** Perhaps it is the name of a valued property? (If it is the name of an either/or property, this will already have been caught in the value case above.) This might look as if it ought to be converted to a `PROPERTYLIST_NT` or `ADJECTIVE_NT` node, but that would imply that it has some specific implication for an object: specifying that a given either/or property holds ("open") or that a given valued property ought to have a given value ("score for finding 10"). This is just the name of the property ("score for finding", say) used as a noun: we convert it to a `PROPERTYNOUN_NT` node.

⟨RRN - Resolve property names 13⟩ ≡

```
    property_name *pprn = parse_property_name(p->word_ref1, p->word_ref2);
    pn_set_property(p, pprn);
    if (pprn) {
        pn_set_node_type(p, PROPERTYNOUN_NT);
        return;
    }
```

This code is used in §6.

§**14.** If the noun phrase is a valid action pattern, such as "taking something", we change it to a new node type to mark this. We don't keep the pattern: it will be reparsed much later on.

⟨RRN - Resolve action patterns 14⟩ ≡

```
{ if (pn_int_annotation(p, nounphrase_article_ANNOT) == NO_ART) {
        action_pattern ap =
            parse_action_pattern(p->word_ref1, p->word_ref2, IS_TENSE);
        if (ap_is_valid(&ap)) {
            pn_set_node_type(p, ACTION_NT);
            return;
        }
    }
}
```

This code is used in §6.

§**15.** This routine is also convenient for assemblies, so has been divided from the main trunk above.

```
int grow_as_NP_with_adjectives(parse_node *p, specification *spec) {
    int i, and_needed = FALSE, allow = TRUE;
    adjective_list_entry *ale;
    parse_node *p2, *p3;
    LOOP_THROUGH_ADJECTIVE_LIST(ale, spec) {
        adjectival_phrase *aph = get_adjective_from_list_entry(ale);
        property_name *prn = aph_has_EORP_meaning(aph);
        if (prn == NULL) allow = FALSE;
    }
    if (allow == FALSE) return allow;
    p2 = new_node(NOUNPHRASE_NT);
    pn_copy(p2, p);
    p->down = p2; pn_set_node_type(p, WITH_NT);
    pn_set_node_type(p2, INSTANCE_NT);
    pn_set_evaluation(p2, spec);
    pn_set_evaluation(p, spec);
    pn_set_full_phrase_evaluation(p, pn_get_evaluation(p));
    if (spec_get_described_kind(spec)) {
        pn_set_refers_to(p, spec_get_described_kind(spec));
        pn_set_refers_to(p2, spec_get_described_kind(spec));
    }
    int n = number_of_adjectives_applied_to(spec);
    if (n > 1) and_needed = TRUE;
    i = 0;
    LOOP_THROUGH_ADJECTIVE_LIST(ale, spec) {
        i++;
        adjectival_phrase *aph = get_adjective_from_list_entry(ale);
        property_name *prn = aph_has_EORP_meaning(aph);
        p3 = new_node(ADJECTIVE_NT);
        pn_set_property(p3, prn);
        pn_set_aph(p3, aph);
        pn_set_evaluation(p3, NULL);
        pn_set_full_phrase_evaluation(p3, pn_get_evaluation(p));
        if (adjective_used_positively(ale))
            pn_annotate_int(p3, negated_boolean_ANNOT, FALSE);
```

```
        else
            pn_annotate_int(p3, negated_boolean_ANNOT, TRUE);
        if (and_needed && (i < n)) {
            p2->next = new_node(AND_NT);
            p2->next->down = p3;
            and_needed = FALSE;
        } else { p2->next = p3; and_needed = TRUE; }
        p2 = p3;
    }
    return TRUE;
}
```

The function grow_as_NP_with_adjectives is called from 8/assem.

§**16. And surgery.** "And surgery" is a fiddly operation to correct the parse tree after resolution of all the nouns in a phrase which involves both "and" and "with" in a particular way. There's no problem with either of these:

> In the Pitch are a bat and ball with score for finding 10. In the Pitch is a sweater with score for finding 5 and description "White wool."

neither of which is altered by and surgery. The difficulty arises with

> In the Pitch is an openable and open door with description "The Hut door."

which, we notice, has exactly the same grammatical structure as the first of the two sentences above, yet a very different meaning, since "openable" is a property whereas "bat" was an object. We perform surgery on:

```
AND_NT
    ADJECTIVE_NT prop:p46_openable
    WITH_NT
        INSTANCE_NT K4_door
        ADJECTIVE_NT prop:p44_open
```

to restructure the nodes as:

```
WITH_NT
    INSTANCE_NT K4_door
    AND_NT
        ADJECTIVE_NT prop:p46_openable
        ADJECTIVE_NT prop:p44_open
```

This innocent-looking little routine involved drawing a *lot* of diagrams on the back of an envelope. Change at your peril.

```
void perform_and_surgery(parse_node *p) {
    parse_node *x, *a_p, *w_p, *p1_p, *p2_p, *i_p;
    if ((pn_get_node_type(p->down) == ADJECTIVE_NT)
        && (pn_get_node_type(p->down->next) == WITH_NT)) {
        a_p = p; p1_p = p->down; w_p = p->down->next;
        i_p = w_p->down; p2_p = i_p->next;
        pn_set_node_type(a_p, WITH_NT);
        pn_set_node_type(w_p, AND_NT);
        pn_set_refers_to(a_p, pn_get_refers_to(w_p));
        pn_set_refers_to(w_p, NULL);
        x = a_p; a_p = w_p; w_p = x;
        w_p->down = i_p;
```

```
        i_p->next = a_p;
        a_p->down = p1_p;
        a_p->down->next = p2_p;
    }
}
```

§**17. With surgery.** This is a less traumatic operation, motivated by sentences like:

> In the Pitch is an open container with description "The box of stumps and bails."

The initial parse tree for such a sentence will have two nested `WITH_NT` clauses, which is arguably correct – "a (container with property open) with description ..." – but which is inconvenient for our implementation of `WITH_NT` later on. So we construe the sentence instead with a single "with", as "a container with properties open and description ..." In terms of the tree,

```
    WITH_NT
        WITH_NT
            INSTANCE_NT K4_container
            ADJECTIVE_NT prop:p44_open
        PROPERTYLIST_NT "The box..."
```

is reconstructed as:

```
    WITH_NT
        INSTANCE_NT K4_container
        AND_NT
            ADJECTIVE_NT prop:p44_open
            PROPERTYLIST_NT "The box..."
```

```
void perform_with_surgery(parse_node *p) {
    parse_node *inst, *prop_1, *prop_2;
    if ((pn_get_node_type(p) == WITH_NT) && (pn_get_node_type(p->down) == WITH_NT)) {
        inst = p->down->down;
        prop_1 = p->down->down->next;
        prop_2 = p->down->next;
        p->down = inst;
        p->down->next = new_node(AND_NT);
        p->down->next->down = prop_1;
        p->down->next->down->next = prop_2;
    }
}
void perform_with_action_surgery(parse_node *p) {
    if (pn_get_node_type(p) == WITH_NT) {
        int a1 = p->down->word_ref1, a2 = p->down->next->word_ref2;
        if ((a1>=0) && (a2>=a1)) {
            action_pattern ap = parse_action_pattern(a1, a2, IS_TENSE);
            if (ap_is_valid(&ap)) {
                pn_set_node_type(p, ACTION_NT);
                p->word_ref1 = a1; p->word_ref2 = a2; p->down = NULL;
            }
        }
    }
}
```

§**18. Location surgery.** This is needed to make sentences like the second one here work:

> The escalator is a door. It is below the Kudamm and above the U-Bahn.

```
    RELATIONSHIP_NT <below> (CONTAINSTHINGS_INF)
        AND_NT
            NOUNPHRASE_NT <kudamm> (definite)
            RELATIONSHIP_NT <above> (CONTAINSTHINGS_INF)
                NOUNPHRASE_NT <u-bahn> (definite)
```

into:

```
    AND_NT
        RELATIONSHIP_NT <below> (CONTAINSTHINGS_INF)
            NOUNPHRASE_NT <kudamm> (definite)
        RELATIONSHIP_NT <above> (CONTAINSTHINGS_INF)
            NOUNPHRASE_NT <u-bahn> (definite)
```

```
void perform_location_surgery(parse_node *p) {
    parse_node *old_and, *old_np1, *old_loc2;
    if ((pn_get_node_type(p) == RELATIONSHIP_NT) &&
        (p->down) && (pn_get_node_type(p->down) == AND_NT) &&
        (p->down->down) && (p->down->down->next) &&
        (pn_get_node_type(p->down->down->next) == RELATIONSHIP_NT)) {
        pn_annotate_int(p, resolved_ANNOT, FALSE);          otherwise this will be wrongly copied
        old_and = p->down;
        old_np1 = old_and->down;
        old_loc2 = old_and->down->next;
        pn_copy(old_and, p);                                making this the new first location node
        pn_set_node_type(p, AND_NT);                                          and this is new AND
        p->down = old_and;
        old_and->down = old_np1;
        old_and->next = old_loc2;
        old_np1->next = NULL;
    }
}
```

The Creator 8/creat

Purpose

This is where all objects, kinds of object, and newly named values are made.

8/creat.§9 Instantiation

Template interpreter commands

```
1   {-callv:create_kind_kind}
```

---

§**1.** The following routine is used early on to creates the special kind "kind". There are other kinds which need to exist for NI to work, but those will all be created in the Standard Rules: "kind" cannot be created that way because there is nothing for it to be a kind of.

Note that the kind "kind" has no name. This is because we needed the word to be a reserved word, so to speak, when resolving assertions: and we would not want people to create "a kind of kind", and such.

```
void create_kind_kind(void) {
    pcalc_prop *prop = prop_to_create_something(kova(OBJECT_TY), -1, -1);
    prop = prop_concatenate(prop, prop_to_make_a_kind());
    prop_true_in_world_model(prop);
}
```

The function create_kind_kind is invoked by a command in a `.i6t` template file.

§**2.**

```
int problem_count_when_creator_started;
int make_necessary_creations(parse_node *px, parse_node *py) {
    problem_count_when_creator_started = problem_count;
    if [[px == there]] {
        int i, np1, np2, rp1, rp2;
        if [[py == ... in ... : i --> np1, np2 ... rp1, rp2]] {
            if [[i, rp2 == which is/are ...]] rp1++;
            rp1--;
            px->word_ref1 = np1; px->word_ref2 = np2;
            py->word_ref1 = rp1; py->word_ref2 = rp2;
            make_assertion(px, py);
            return FALSE;
        }
        refine_parse_tree(py, ALLOW_CREATION); noun_creator(py, NULL, NULL);
        if (pn_get_node_type(py) == INSTANCE_NT)
            convert_instance_to_nounphrase(py, 0, FALSE);
        pn_set_node_type(px, NOUNPHRASE_NT);
        pn_set_refers_to(px, pn_get_refers_to(py));
    } else {
        refine_parse_tree(px, ALLOW_CREATION);
        refine_parse_tree(py, ALLOW_CREATION);
        if ((pn_get_node_type(px) == ACTION_NT) &&
```

```
            (pn_get_node_type(py) == CREATED_NT)) {
            pn_set_node_type(py, ACTION_NT);
            return TRUE;
        }
        if ((pn_get_node_type(px) == ACTION_NT) &&
            (pn_get_node_type(py) == KIND_NT)) {
            pn_set_node_type(px, CREATED_NT);
        }
        parse_node *govx = NULL, *govy = NULL;
        kind_of_value *kovx = NULL, *kovy = NULL;
        binary_predicate *bp = NULL;
        if (bp == NULL) bp = bp_of_subtree(py);
        if (bp == NULL) bp = bp_of_subtree(px);
        if (bp) {
            if (bp_term_kind(bp, 0)) kovx = kova(OBJECT_TY);
            if (bp_term_kind_of_value(bp, 0)) kovx = bp_term_kind_of_value(bp, 0);
            if (bp_term_kind(bp, 1)) kovy = kova(OBJECT_TY);
            if (bp_term_kind_of_value(bp, 1)) kovy = bp_term_kind_of_value(bp, 1);
            if (bp == a_contains_b_predicate) { kovx = kova(OBJECT_TY); kovy = kova(OBJECT_TY);
}
        }
        if ((kovx == NULL) || (kovy == NULL)) {
            kovx = kov_of_subtree(px, &govx);
            kovy = kov_of_subtree(py, &govy);
        }
        noun_creator(px, kovy, govy);
        noun_creator(py, kovx, govx);
    }
    if (problem_count > problem_count_when_creator_started) return FALSE;
    return TRUE;
}

binary_predicate *bp_of_subtree(parse_node *p) {
    if ((p) && (pn_get_node_type(p) == RELATIONSHIP_NT))
        return pn_get_relationship(p);
    return NULL;
}
```

The function make_necessary_creations is called from 8/mass.

§3.

```
void noun_creator(parse_node *p, kind_of_value *create_as, parse_node *governor) {
    switch (pn_get_node_type(p)) {
        case CALLED_NT: {
            parse_node *new_governor = NULL;
            kind_of_value *what = kov_of_subtree(p->down, &new_governor);
            if (what == NULL) { what = create_as; new_governor = governor; }
            noun_creator(p->down, NULL, NULL);
            noun_creator(p->down->next, what, new_governor);
            break;
        }
        case CREATED_NT: {
            int wording1 = p->word_ref1, wording2 = p->word_ref2;
            if (wording1 >= 0) ⟨Check that this is a name we will permit to be used 4⟩;
            p->word_ref1 = wording1; p->word_ref2 = wording2;
            if ((create_as == NULL) || (is_kova(create_as, OBJECT_TY))) {
                world_object *recent_creation = NULL;
                if (wording1 >= 0)
                    ⟨Create an object or kind of object rather than a value 5⟩;
                pn_set_node_type(p, NOUNPHRASE_NT);
                pn_set_refers_to(p, recent_creation);
            } else {
                specification *val = NULL;
                if (wording1 >= 0)
                    ⟨Create a value rather than an object 6⟩;
                pn_set_node_type(p, VALUE_NT);
                pn_set_evaluation(p, val);
            }
            break;
        }
        default: {
            parse_node *ch;
            for (ch = p->down; ch; ch = ch->next)
                noun_creator(ch, create_as, governor);
            break;
        }
    }
    switch (pn_get_node_type(p)) {
        case CALLED_NT:
            if (pn_get_node_type(p->down->next) == NOUNPHRASE_NT) {
                world_object *wo = pn_get_refers_to(p->down->next);
                if (wo) {
                    if (pn_int_annotation(p->down->next, nounphrase_article_ANNOT) == DEF_ART)
                        wo->article_is_the = TRUE;
                }
            }
            switch(pn_get_node_type(p->down)) {
                case VALUE_NT:
                    break;
                case INSTANCE_NT:
                    assert_kind_of_object(pn_get_refers_to(p->down->next), pn_get_refers_to(p->down));
                    pn_set_refers_to(p, pn_get_refers_to(p->down->next));
```

```
                    pn_set_node_type(p, NOUNPHRASE_NT);
                    p->word_ref1 = p->down->next->word_ref1;
                    p->word_ref2 = p->down->next->word_ref2;
                    if (spec_get_proposition(pn_get_evaluation(p->down)))
                        prop_true_in_world_model_about(
                            spec_get_proposition(pn_get_evaluation(p->down)),
                            pn_get_refers_to(p), NULL);
                    p->down = NULL;
                    break;
                case WITH_NT:
                    assert_kind_of_object(pn_get_refers_to(p->down->next), pn_get_refers_to(p->down->down));
                    pn_set_refers_to(p, pn_get_refers_to(p->down->next));
                    pn_set_node_type(p, NOUNPHRASE_NT);
                    p->word_ref1 = p->down->next->word_ref1;
                    p->word_ref2 = p->down->next->word_ref2;
                    assert_property_list(p, p->down->down->next);
                    p->down = NULL;
                    break;
                default:
                    LOG("$T\n", p);
                    sentence_problem(_P_(C8CalledWithoutKind),
                        "I can only make 'a something called whatever' when "
                        "the something is a kind I know",
                        "possibly qualified with adjectives. For instance, "
                        "'an open door called the Marble Door' is fine because "
                        "'door' is the name of a kind and 'open' is an adjective "
                        "which means something for doors. But 'a grand archway "
                        "called the Great Gates' would not normally mean anything "
                        "to me, because 'archway' is not one of the standard kinds "
                        "in Inform. (Try consulting the Kinds index.)");
                    pn_set_node_type(p, NOUNPHRASE_NT);
                    p->word_ref1 = p->down->next->word_ref1;
                    p->word_ref2 = p->down->next->word_ref2;
                    p->down = NULL;
                    break;
            }
            break;
    }
}
```

§4.

⟨Check that this is a name we will permit to be used 4⟩ ≡

```
int i;
if ((wording1 == wording2) &&
    (vocab_test_flags(wording1, ARTICLE_MC))) {
    sentence_problem(_P_(C8NameIsArticle),
        "this seems to give something a name which consists only of "
        "an article",
        "that is, 'a', 'an', 'the' or 'some'. This is not allowed since "
        "the potential for confusion is too high. (If you need, say, a "
        "room which the player sees as just 'A', you can get this "
        "effect with: 'A-Room is a room with printed name \"A\".')");
    wording1 = -1; wording2 = -1;
}
else if ((is_word_intermediate_inc(OPENBRACKET_V, wording1, wording2) >= 0) ||
    (is_word_intermediate_inc(CLOSEBRACKET_V, wording1, wording2) >= 0)) {
    int j;
    if [[current_sentence == to ... COMMA ... : j]]
    sentence_problem(_P_(C8NameWithBracketsTo),
        "reading this sentence strictly it seems to give something "
        "a name which contains brackets '(' or ')'",
        "which is forbidden, but in fact the 'To...' and the "
        "subsequent comma makes me wonder if you are trying to "
        "define a new phrase but have used a comma to divide the "
        "preamble from the definition: if so, it should be a colon.");
    else
    sentence_problem(_P_(C8NameWithBrackets),
        "this seems to give something a name which contains "
        "brackets '(' or ')'",
        "which is not allowed since the potential for confusion "
        "with other uses for brackets in Inform source text is "
        "too high. (If you need, say, a room which the player sees "
        "as 'Fillmore (West)', you can get this effect with: "
        "'Fillmore West is a room with printed name \"Fillmore "
        "(West)\".')");
    wording1 = -1; wording2 = -1;
}
else if [[wording1, wording2 == COMPILER_CRASH1]] {
    STREAM_WRITE(STDERR, "*** Exit(1) requested for testing purposes ***\n");
    STREAM_FLUSH(STDERR);
    sentence_problem(_P_(C8Crash1),
        "this uses the first secret hieroglyph of dreadful power",
        "which forces me to crash. (It's for testing the way I crash, in fact. "
        "If this is a genuine inconvenience to you, get in touch with my "
        "authors.)");
    exit(1);
}
else if [[wording1, wording2 == COMPILER_CRASH10]] {
    STREAM_WRITE(STDERR, "*** Exit(10) requested for testing purposes ***\n");
    STREAM_FLUSH(STDERR);
    sentence_problem(_P_(C8Crash10),
        "this uses the second secret hieroglyph of dreadful power",
        "which forces me to crash. (It's for testing the way I crash, in fact. "
```

```
        "If this is a genuine inconvenience to you, get in touch with my "
        "authors.)");
    exit(10);
}
else if [[wording1, wording2 == COMPILER_CRASH11]] {
    STREAM_WRITE(STDERR, "*** Exit(11) requested for testing purposes ***\n");
    STREAM_FLUSH(STDERR);
    sentence_problem(_P_(C8Crash11),
        "this uses the third secret hieroglyph of dreadful power",
        "which forces me to crash. (It's for testing the way I crash, in fact. "
        "If this is a genuine inconvenience to you, get in touch with my "
        "authors.)");
    exit(11);
}
else if [[wording1, wording2 == COMMA ...]] {
    sentence_problem(_P_(C8StartsWithComma),
        "this seems to refer to something whose name begins with a comma",
        "which is forbidden. Perhaps you used a comma in "
        "punctuating a sentence? Inform generally doesn't "
        "like this because it reserves commas for specific "
        "purposes such as dividing rules or 'if' phrases.");
}
else if [[wording1, wording2 == OPENBRACE ... CLOSEBRACE]] {
    sentence_problem(_P_(C8ObjectInBraces),
        "this seems to refer to something whose name is in braces { ... }",
        "which is forbidden. What has probably happened is that you "
        "intended it to be a list of values, since those are written in "
        "braces (for instance '{1, 2, 3}') - but Inform didn't recognise "
        "one or more of the values inside the braces. (This might be "
        "because they're mistyped, or it might be that they haven't "
        "yet been given a meaning. Note that spaces have to be used "
        "before commas, so '{2, 6}' is fine, but not '{2,6}'.)");
}
else if ([[wording1, wording2 == ... when ... : i]] ||
    [[wording1, wording2 == ... while ... : i]]) {
    quote_source(1, current_sentence);
    quote_words(2, wording1, wording2);
    handmade_problem(_P_(C8ObjectIncWhen));
    issue_problem_segment(
        "The sentence %1 seems to be talking about a previously unknown "
        "room or thing called %2. Ordinarily, I would create this, but "
        "because the name contains the word 'when' or 'while' I'm going "
        "say no. %PThat's because this far more often happens by mistake "
        "than deliberately. For instance, people sometimes type lines "
        "like 'Jumping when the actor is on the trampoline is "
        "high-jumping.' But in fact although 'jumping' is an action, "
        "'Jumping when...' is not - 'when' can't be used here (though "
        "it can be used in rule preambles). So the sentence is instead "
        "read as making an object 'jumping when the actor' and putting it "
        "on top of another one, 'trampoline is high-jumping'. This can "
        "lead to a lot of confusion. %PIf you genuinely do want an "
        "object whose name contains the word 'when', try something "
        "like: 'In the box is a thing called When worlds collide.'");
```

```
        issue_problem_end();
    }
    else {
        int j;
        for (j=wording1; j<=wording2; j++) {
            if (vocab_test_flags(j, (TEXT_MC+TEXTWITHSUBS_MC))) {
                assertion_problem(_P_(C8NameWithText),
                    "this seems to give something a name which contains "
                    "double-quoted text",
                    "which is not allowed. If you do need quotes in a name, "
                    "one option would be to write something like 'In the "
                    "Saloon is 'Black' Jacques Bernoulli.'; but this problem "
                    "message is often caused by an accident in punctuation, "
                    "in which case you never intended to create an object - "
                    "you thought that the text ended a sentence because "
                    "it finished with sentence-ending punctuation, when in "
                    "fact it didn't, so that I read the next words as "
                    "following on.");
                wording1 = -1; wording2 = -1;
            }
        }
    }
```

This code is used in §3.

§5.

⟨Create an object or kind of object rather than a value 5⟩ ≡

```
    recent_creation = parse_world_object(wording1, wording2, FALSE);
    if ((recent_creation) && (recent_creation->kind_flag))
        recent_creation = NULL;
    if ((recent_creation) && (recent_creation->word_ref1 >= 0) &&
        (compare_word_range(wording1, wording2, recent_creation->word_ref1, recent_creation->word_ref2)
== FALSE))
        recent_creation = NULL;
    if (recent_creation == NULL) {
        int is_a_kind = FALSE;
        if ((governor) && (pn_get_node_type(governor) == KIND_NT)) is_a_kind = TRUE;
        pcalc_prop *prop = prop_to_create_something(kova(OBJECT_TY), wording1, wording2);
        if (is_a_kind) prop = prop_concatenate(prop, prop_to_make_a_kind());
        prop_true_in_world_model(prop);
        recent_creation = get_latest_world_object();
        prop = NULL;
        if ((plural_named_aph) &&
            (pn_int_annotation(p, plural_reference_ANNOT)))
            prop = prop_concatenate(prop, atom_unary_PREDICATE_from_aph(plural_named_aph, FALSE));
        if ((proper_named_aph) &&
            (pn_int_annotation(p, nounphrase_article_ANNOT) == NO_ART) &&
            (is_a_kind == FALSE))
            prop = prop_concatenate(prop, atom_unary_PREDICATE_from_aph(proper_named_aph, FALSE));
        if (prop) {
            int cl = prevailing_mood; prevailing_mood = LIKELY_CE;
            prop_true_in_world_model_about(prop, recent_creation, NULL);
```

```
        prevailing_mood = cl;
    }
    if (pn_int_annotation(p, nounphrase_article_ANNOT) == NO_ART)
        recent_creation->propername = TRUE;
}
```

This code is used in §3.

§6.

⟨Create a value rather than an object 6⟩ ≡

```
specification *governing_spec = pn_get_evaluation(governor);
if (is_kova(create_as, LOCAL_VARIABLE_SPC)) {
    sentence_problem(_P_(C8VagueVariable),
        "'variable' is too vague a description",
        "because it doesn't say what kind of value should go into "
        "the variable. 'number variable' or 'a number that varies' - "
        "whatever kind of value you need - would be much clearer.");
} else
if ((spec_is_generic_CONSTANT(governing_spec)) &&
    (is_kova(spec_get_kind_of_value(governing_spec), ANY_VALUE_TY))) {
    pcalc_prop *prop = prop_to_create_something(NULL, wording1, wording2);
    prop = prop_concatenate(prop, prop_to_make_a_kov());
    prop_true_in_world_model(prop);
    val = new_generic_CONSTANT_type(latest_atomic_kov);
} else
if (spec_is_generic_NONLOCAL_VARIABLE(governing_spec)) {
    kind_of_value *domain = spec_get_kind_of_value(governing_spec);
    if (is_kova(domain, UNDERSTANDING_TY)) {
        assertion_problem(_P_(C8Case9a),
            "'topics that vary' are not allowed",
            "that is, a variable is not allowed to have 'topic' as its "
            "kind of value. (This would cause too much ambiguity with "
            "text variables, whose values look exactly the same.)");
    }
    pcalc_prop *prop = prop_to_create_something(domain, wording1, wording2);
    prop = prop_concatenate(prop, prop_to_make_a_var());
    prop_true_in_world_model(prop);
    val = new_actual_NONLOCAL_VARIABLE_type(latest_quantity);
} else
if (is_kova(create_as, RULEBOOK_TY)) {
    int f = PARAMETER_FOCUS;
    if (pn_int_annotation(governor, creation_modifier_ANNOT) == 0) f = ACTION_FOCUS;
    val = rulebook_to_RULEBOOK_spec(rb_new(wording1, wording2, f));
} else
if ((kov_represents_data(create_as)) && (kov_is_a_unit(create_as))) {
    sentence_problem(_P_(C8MixedConstants),
        "this is a kind of value which already has a semi-numerical "
        "specification",
        "so it can't have an entirely verbal form as well.");
} else
if ((kov_represents_data(create_as)) && (kov_has_named_constant_values(create_as))) {
    pcalc_prop *prop = prop_to_create_something(create_as, wording1, wording2);
```

```
        pcalc_prop *such_that = pn_get_creation_proposition(governor);
        if (such_that) prop = prop_concatenate(prop, such_that);
        prop_true_in_world_model(prop);
        val = spec_copy(qty_get_initial_value(latest_quantity));
    } else
    if (problem_count_when_creator_started == problem_count) {
        LOG("$W: $u\n$T\n", wording1, wording2, create_as, governor);
        quote_source(1, current_sentence);
        quote_words(2, wording1, wording2);
        quote_kov(3, create_as);
        handmade_problem(_P_(C8NoNewInstances));
        issue_problem_segment(
            "The sentence %1 reads to me as if '%2' refers to something "
            "I should create as brand new - %3. But that can't be right, "
            "because this is a kind of value where I can't simply invent "
            "new values. (Just as the numbers are ..., 1, 2, 3, ... and "
            "I can't invent a new one called 'Susan'.) %P"
            "Perhaps you wanted not to invent a constant but to make a "
            "variable - that is, to give a name for a value which will "
            "change during play. If so, try something like 'The bonus "
            "is a number which varies'. %P"
            "Or perhaps you wanted to create a name an an alias for a "
            "constant value. Some programming languages encourage this, "
            "but Inform doesn't: instead, make it a variable and then "
            "don't vary it - 'The lucky number is a number that varies. "
            "The lucky number is 8.'");
        issue_problem_end();
    }
    dref_position_of_symbol(&wording1, &wording2, FALSE);
    p->word_ref1 = wording1; p->word_ref2 = wording2;
```

This code is used in §3.

§7.

```
define NAME_IS_SUITABLE 1
define NAME_UNSUITABLE_BECAUSE_ARTICLE 2
define NAME_UNSUITABLE_BECAUSE_QUOTED 3
define NAME_UNSUITABLE_BECAUSE_PUNCTUATED 4

int suitable_name(int wording1, int wording2) {
    if ((wording1 == wording2) &&
        (vocab_test_flags(wording1, ARTICLE_MC))) return NAME_UNSUITABLE_BECAUSE_ARTICLE;
    if ((is_word_intermediate_inc(OPENBRACKET_V, wording1, wording2) >= 0) ||
        (is_word_intermediate_inc(CLOSEBRACKET_V, wording1, wording2) >= 0) ||
        (is_word_intermediate_inc(OPENBRACE_V, wording1, wording2) >= 0) ||
        (is_word_intermediate_inc(CLOSEBRACE_V, wording1, wording2) >= 0) ||
        (is_word_intermediate_inc(COMMA_V, wording1, wording2) >= 0))
        return NAME_UNSUITABLE_BECAUSE_PUNCTUATED;
    int j;
    for (j=wording1; j<=wording2; j++)
        if (vocab_test_flags(j, (TEXT_MC+TEXTWITHSUBS_MC)))
            return NAME_UNSUITABLE_BECAUSE_QUOTED;
    return NAME_IS_SUITABLE;
}
```

The function suitable_name is called from 9/prop.

§8.

```
kind_of_value *kov_of_subtree(parse_node *p, parse_node **governing) {
    if (p == NULL) return NULL;
    switch (pn_get_node_type(p)) {
        case AND_NT: {
            kind_of_value *left = kov_of_subtree(p->down, governing);
            kind_of_value *right = kov_of_subtree(p->down->next, governing);
            if (left) return left;
            return right;
        }
        case WITH_NT:
            return kov_of_subtree(p->down, governing);
        case KIND_NT: {
            if (p->down == NULL) { *governing = p; return kova(OBJECT_TY); }
            kind_of_value *kov = kov_of_subtree(p->down, governing);
            if (is_kova(kov, ANY_VALUE_TY)) return kov;
            if (kov) { *governing = p; return kov; }
            return NULL;
        }
        default: {
            specification *spec = pn_get_evaluation(p);
            if ((spec_is_generic_NONLOCAL_VARIABLE(spec)) ||
                (spec_is_generic_CONSTANT(spec))) {
                kind_of_value *found = spec_evaluates_to(spec);
                if (found) *governing = p;
                return found;
            }
            if (species_is(spec, DESCRIPTION_SPC)) {
                *governing = p;
                if (spec_get_described_kov(spec)) return spec_get_described_kov(spec);
                if (spec_get_described_kind(spec)) return kovko(spec_get_described_kind(spec));
                return kova(OBJECT_TY);
            }
        }
    }
    return NULL;
}
```

§**9. Instantiation.** If there is really such a word. The INSTANCE_NT node sometimes means to talk about things in general, sometimes things in particular: consider the two sentences

A container is usually open. A container is in the Box Room.

At initial parse tree time, and even now at resolution time, we cannot easily differentiate these meanings. We will only be able to do so later, when making assertions. When it turns out that the INSTANCE_NT is to be made into something nameless but tangible, as in the second sentence above, the following routine is used to transform it into a suitable NOUNPHRASE_NT referring to the newly created object.

```
void convert_instance_to_nounphrase(parse_node *p, int assertion_depth,
    int confect_name_flag) {
    int i, ic;
    parse_node *new_subtree = new_node(INSTANCE_NT);
    parse_node *pz, *attach_to = new_subtree, *original_next = p->next;
    if (pn_get_evaluation(p) == NULL) ic = 1;
    else ic = spec_get_quantification_parameter(pn_get_evaluation(p));
    if (ic<1) ic = 1;
    if (ic > 100) {
        assertion_problem(_P_(C8TooManyDuplicates),
            "at most 100 duplicates can be made at any one time",
            "so '157 chairs are in the UN General Assembly' will not be "
            "allowed. The system for handling duplicates during play "
            "becomes too slow and awkward when there are so many.");
        ic = 100;
    }
    for (i=1; i<=ic; i++) {
        pz = new_node(NOUNPHRASE_NT);
        pcalc_prop *prop = prop_to_create_something(kova(OBJECT_TY), -1, -1);
        prop_true_in_world_model(prop);
        world_object *recent_creation = get_latest_world_object();
        pn_set_refers_to(pz, recent_creation);
        if ((pn_has_annotation(current_sentence, implicit_in_creation_of_ANNOT)) &&
            (ic==1) && (confect_name_flag)) {
            world_object *owner = pn_get_implicit_in_creation_of(current_sentence);
            if ((owner->word_ref1 >= 0) && (pn_get_refers_to(p)) &&
                (pn_get_refers_to(p)->word_ref1 >= 0)) {
                char confected_name[1000];
                parse_node *confected_creator;
                int w1;
                if (owner == wo_yourself) {
                    sprintf(confected_name, "your ");
                } else {
                    print_raw_text_to_string(
                        owner->word_ref1, owner->word_ref2,
                        confected_name);
                    sprintf(confected_name+strlen(confected_name), "'s ");
                }
                print_raw_text_to_string(
                    pn_get_refers_to(p)->word_ref1,
                    pn_get_refers_to(p)->word_ref2,
                    confected_name+strlen(confected_name));
                sprintf(confected_name+strlen(confected_name), " ");
```

```
                w1 = lexer_wordcount;
                feed_into_lexer(confected_name, TRUE, FALSE);
                confected_creator = new_nounphrase_worldly(w1, lexer_wordcount-1, FALSE);
                pn_get_refers_to(pz)->named_after_w1 = pn_get_refers_to(p)->word_ref1;
                pn_get_refers_to(pz)->named_after_w2 = pn_get_refers_to(p)->word_ref2;
                pn_get_refers_to(pz)->word_ref1 = confected_creator->word_ref1;
                pn_get_refers_to(pz)->word_ref2 = confected_creator->word_ref2;
                register_wo_names(pn_get_refers_to(pz), confected_creator->word_ref1, confected_creator->word_
                if (owner == wo_yourself) {
                    pn_get_refers_to(pz)->propername = TRUE;
                } else {
                    pn_get_refers_to(pz)->propername = owner->propername;
                }
                pn_get_refers_to(pz)->named_after = owner;
                LOGIF(NP_RESOLUTION, "Confecting the name <%s>\n", confected_name);
            }
        }
        make_assertion_recursive(pz, p, assertion_depth+1);
        if (i < ic) {
            pn_set_node_type(attach_to, AND_NT);
            attach_to->down = new_node(NOUNPHRASE_NT);
            attach_to->down->next = new_node(NOUNPHRASE_NT);
            attach_to = attach_to->down;
        }
        pn_set_node_type(attach_to, NOUNPHRASE_NT);
        pn_set_refers_to(attach_to, pn_get_refers_to(pz));
        attach_to = attach_to->next;
    }
    pn_copy(p, new_subtree);
    p->next = original_next;
}

int convert_instances_in_subtree(parse_node *py, int d) {
    if (pn_get_node_type(py) == AND_NT) {
        int a = convert_instances_in_subtree(py->down, d);
        int b = convert_instances_in_subtree(py->down->next, d);
        return (a || b);
    }
    if (pn_get_node_type(py) == INSTANCE_NT) {
        convert_instance_to_nounphrase(py, d, FALSE);
        return TRUE;
    }
    return FALSE;
}
```

The function convert_instance_to_nounphrase is called from 8/mass.
The function convert_instances_in_subtree is called from 8/mass.

Make Assertions 8/mass

Purpose

To read assertion sentences, sort them according to a detailed grammatical classification and take the primary actions necessary either to generate problem messages, or to infer information.

8/mass.§2-3 The Assertion Matrix; §4 Splitting into cases; §5 Case 1; §6 Case 2; §7 Case 3; §8 Case 4; §9 Case 5; §10 Case 6; §11 Case 7; §12 Case 8; §13 Case 9; §14 Case 10; §15 Case 11; §16 Case 12; §17 Case 13; §18 Case 14; §19 Case 15; §20 Case 16; §21 Case 17; §22 Case 18; §23 Case 19; §24 Case 20; §25 Case 21; §26 Case 22; §27 Case 23; §28 Case 24; §29 Case 25; §30 Case 26; §31 Case 27; §32 Case 28; §33 Case 29; §34 Case 30; §35 Case 31; §36 Case 32; §37 Case 33; §38 Case 34; §39 Case 35; §40 Case 36; §41 Case 37; §42 Case 38; §43 Case 39; §44 Case 40; §45 Case 41; §46 Case 42; §47 Case 43; §48 Case 44; §49 Case 45; §50 Case 46; §51 Case 47; §52 Case 48; §53 Case 49; §54 Case 50; §55 Case 51; §56 Case 52; §57 Case 53; §58 Case 54; §59 Case 55; §60 Case 56; §61 Case 57; §62 Adjective list trees; §63 Equating values; §64-65 To Be and To Have

---

Definitions

¶1. In this section we are concerned with assertions, that is, declarations of the form "X is Y". For etymological reasons, the English verb "to be" is a mixture of several different verbs which have blurred together into one: consider "I am 5", "I am happy" and "I am Chloe". NI compounds the problem by allowing X and Y to be more complicated clauses than simple nouns, and also by distinguishing between the specific ("the lacquered box") and the non-specific ("a container"), and between the tangible ("box" again) and the intangible ("101").

The work of growing the parse tree and resolving its references now enables us to classify assertions into 196 grammatically different forms, which we group into 56 main cases. (The fact that relatively few of these split into sub-cases shows that the parse tree, when resolved at least, encodes meaning surprisingly well.) Of these cases, about half produce valid NI sentences in at least some situations, while the other half lead certainly to problem messages.

The verb "to be" occupies some 12 columns of the *Oxford English Dictionary* and they make useful reading in clarifying the problem. For instance, NI's distinction between spatial and property knowledge reflects the OED's distinction between meanings 5a and 9b respectively. NI's assertion cases construe the verb "to be" as follows:

(a) Cases 6, 7 and 8 are forms of "to have" in disguise as "to be allowed to have".
(b) Cases 10, 11, 13, 22 and 25 to 29 are "to be" with OED meaning 10, "to exist as the thing known by a certain name; to be identical with".
(c) Cases 15 to 21, 48 and 49 are "to be" with OED meaning 5a, "to have or occupy a place somewhere".
(d) Cases 43 to 47 are ungrammatical as "to be", and are in fact "to have": they are given here because common code is used to implement both verbs, which reflects the curious way that English "to be" has subsumed much of the function that "to have" serves in other languages.
(e) And all other cases are "to be" with OED meaning 9b, "to have a place among the things distinguished by a specified quality".

---

§**1.** Here is the main routine which asserts that subtree `px` "is" `py`. Note that references are "resolved" only on the first of the two passes through: making one assertion may change the way that the next one is resolved, so we cannot simply resolve the whole parse tree before making assertions.

```
void make_assertion(parse_node *px, parse_node *py) {
    if (traverse == 1) if (make_necessary_creations(px, py) == FALSE) return;
```

```
    if (trace_sentences) log_subtree(current_sentence, 1);
    if (![[px == there]]) make_assertion_recursive(px, py, 0);
    change_discussion_topic(pn_get_refers_to(px), pn_get_refers_to(py));
    if (pn_get_node_type(px) == AND_NT) subject_of_discussion_a_list();
}
```

The function make_assertion is called from 8/creat and 10/tab.

§**2. The Assertion Matrix.** The essential meaning of the assertion is reflected in the structure of `px` and `py`, which in turn is most easily distinguished by their node types. We want to be sure that we completely understand this process and that no possibilities escape notice. We therefore use a 14×14 matrix of possible cases, as follows. The cases are numbered from 1 to 57 inclusive and there is no significance to their ordering. Mathematically enough, `py` specifies the row and `px` the column.

define ASSERTION_MATRIX_DIM 14

```
typedef struct matrix_entry {
    int row_node_type;
    int cases[ASSERTION_MATRIX_DIM];
} matrix_entry;
matrix_entry assertion_matrix[ASSERTION_MATRIX_DIM] = {
                  W, A, A, A, A, K, I, L, N, P, P, V, X, E
{ WITH_NT,         {  3,  1,  1,  1,  1,  1,  1, 48,  1,  1,  1,  1,  1, 51 } },
{ AND_NT,          {  2,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4, 51 } },
{ ACTION_NT,       {  2,  5, 30, 33,  6, 33, 24, 21, 34, 33, 33, 57, 28, 51 } },
{ ADJECTIVE_NT,    {  2,  5, 32, 42,  6, 35, 36, 21, 36, 39, 39, 36, 54, 51 } },
{ ALLOWED_NT,      {  2,  5,  7,  7,  6,  7,  8,  7,  8,  7,  7,  7,  7,  8 } },
{ KIND_NT,         {  2,  5, 12, 12,  6, 12, 12, 12, 12, 12, 12, 12, 12, 51 } },
{ INSTANCE_NT,     {  2,  5, 24, 24,  6, 24, 16, 15, 14, 46, 24, 24, 24, 51 } },
{ RELATIONSHIP_NT, { 49,  5, 21, 21,  6, 21, 17, 18, 19, 21, 21, 56, 27, 53 } },
{ NOUNPHRASE_NT,   {  2,  5, 31, 38,  6, 35, 23, 20, 22, 44, 13, 10, 29, 51 } },
{ PROPERTYLIST_NT, {  2,  5, 33, 39,  6, 35, 45, 21, 43, 40, 40, 47, 28, 51 } },
{ PROPERTYNOUN_NT, {  2,  5, 32, 39,  6, 35, 24, 21, 37, 40, 40, 37, 28, 51 } },
{ VALUE_NT,        {  2,  5, 33, 41,  6, 35, 24, 55,  9, 47, 13, 11, 29, 51 } },
{ X_OF_Y_NT,       {  2,  5, 26, 26,  6, 26, 24, 21, 26, 26, 26, 26, 25, 51 } },
{ EVERY_NT,        { 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 52 } } };
```

The structure matrix_entry is private to this section.

§**3.** The following routine simply looks up which of the 56 cases the current pair of `px` and `py` falls into.

```
int which_assertion_case(parse_node *px, parse_node *py) {
    int i, x=-1, y=-1;
    if (px == NULL) internal_error("make assertion with px NULL");
    if (py == NULL) internal_error("make assertion with py NULL");
    if (pnt_allow_in_assertions(px) == FALSE) {
        log_subtree(px, 1);
        internal_error("make assertion with improper px node");
    }
    if (pnt_allow_in_assertions(py) == FALSE) {
        log_subtree(py, 1);
        internal_error("make assertion with improper py node");
    }
    for (i=0; i<ASSERTION_MATRIX_DIM; i++)
        if (assertion_matrix[i].row_node_type == pn_get_node_type(px)) x=i;
    for (i=0; i<ASSERTION_MATRIX_DIM; i++)
        if (assertion_matrix[i].row_node_type == pn_get_node_type(py)) y=i;
    if (x<0) {
        log_subtree(px, 1);
        internal_error("make assertion with px node not in matrix");
    }
    if (y<0) {
        log_subtree(py, 1);
        internal_error("make assertion with py node not in matrix");
    }
    return assertion_matrix[y].cases[x];
}
```

§**4. Splitting into cases.** The point of the matrix is to prove the following:

Proposition. *Every assertion either (a) adds to our knowledge of the model world (on at least one of the two passes) or (b) results in a problem message being issued (on at least one of the two passes).*

This may sound trivial to arrange, but the experience of the previous two drafts of the assertion maker showed that it was all too easy for certain assertions to fall through a net of comparisons and thus be ignored altogether. On each pass, each assertion is processed with exactly one of the cases and nothing else is done with it, except for one general rule: a requirement that we can only talk about specific things in definite terms. This is enforced by reference to the `prevailing_mood`, the level of certainty which the source text expressed about the assertion being made. (For instance, "A door is usually open" is a less certain assertion than "The black door is open".)

```
void make_assertion_recursive(parse_node *px, parse_node *py,
    int assertion_depth) {
    STREAM_INDENT(dl);
    make_assertion_recursive_inner(px, py, assertion_depth);
    STREAM_OUTDENT(dl);
}

void make_assertion_recursive_inner(parse_node *px, parse_node *py,
    int assertion_depth) {
    int ma_case;
    assertion_depth++;
```

```
    ma_case = which_assertion_case(px, py);
    LOGIF(ASSERTIONS, "[$W/$N] =%d [$W/$N]\n",
        px->word_ref1, px->word_ref2, pn_get_node_type(px), ma_case,
        py->word_ref1, py->word_ref2, pn_get_node_type(py));
    if (traverse == 2) {
        if ((prevailing_mood != UNKNOWN_CE) && (pn_get_refers_to(px) != NULL) &&
            (pn_get_refers_to(px)->kind_flag == FALSE)) {
            sentence_problem(_P_(C8VagueAboutSpecific),
                "you can only equivocate with 'usually', 'rarely', "
                "'always' and the like when talking about kinds of thing",
                "because when a specific thing is involved you should say "
                "definitely one way or another. 'A cave is usually dark' is "
                "fine, but not 'the Mystic Wood is usually dark'.");
            return;
        }
    }
    if (traverse == 2) {
        if (((pn_get_node_type(px) == INSTANCE_NT)
            && (pn_get_evaluation(px)) && (spec_get_quantification_parameter(pn_get_evaluation(px))
> 1)
            && (pn_get_node_type(py) != RELATIONSHIP_NT)) ||
            ((pn_get_node_type(py) == INSTANCE_NT)
            && (pn_get_evaluation(py)) && (spec_get_quantification_parameter(pn_get_evaluation(py))
> 1)
            && (pn_get_node_type(px) != RELATIONSHIP_NT))) {
            sentence_problem(_P_(C8MultiplyVague),
                "multiple objects can only be put into relationships",
                "by saying something like 'In the Drawing Room are two women.', "
                "and all other assertions with multiple objects are disallowed: "
                "so 'Three doors are open.' is rejected - I can't tell which three.");
            return;
        }
    }
    switch(ma_case) {
        case 1: ⟨MA1 - Y is WITH 5⟩; return;
        case 2: ⟨MA2 - X is WITH 6⟩; return;
        case 3: ⟨MA3 - X and Y are WITH - P 7⟩; return;
        case 4: ⟨MA4 - Y is AND 8⟩; return;
        case 5: ⟨MA5 - X is AND 9⟩; return;
        case 6: ⟨MA6 - X is ALLOWED - P 10⟩; return;
        case 7: ⟨MA7 - Y is ALLOWED - P 11⟩; return;
        case 8: ⟨MA8 - X is NOUNPHRASE or INSTANCE, Y is ALLOWED 12⟩; return;
        case 9: ⟨MA9 - X is NOUNPHRASE, Y is VALUE 13⟩; return;
        case 10: ⟨MA10 - X is VALUE, Y is NOUNPHRASE 14⟩; return;
        case 11: ⟨MA11 - X and Y are VALUEs 15⟩; return;
        case 12: ⟨MA12 - Y is KIND 16⟩; return;
        case 13: ⟨MA13 - X is PROPERTYNAME, Y is NOUNPHRASE or VALUE 17⟩; return;
        case 14: ⟨MA14 - X is NOUNPHRASE, Y is INSTANCE 18⟩; return;
        case 15: ⟨MA15 - X is RELATIONSHIP, Y is INSTANCE 19⟩; return;
        case 16: ⟨MA16 - X and Y are INSTANCE - P 20⟩; return;
        case 17: ⟨MA17 - X is INSTANCE, Y is RELATIONSHIP - P 21⟩; return;
        case 18: ⟨MA18 - X and Y are RELATIONSHIP 22⟩; return;
```

```
        case 19: ⟨MA19 - X is NOUNPHRASE, Y is RELATIONSHIP 23⟩; return;
        case 20: ⟨MA20 - X is RELATIONSHIP, Y is NOUNPHRASE 24⟩; return;
        case 21: ⟨MA21 - X or Y is RELATIONSHIP - P 25⟩; return;
        case 22: ⟨MA22 - X and Y are NOUNPHRASE 26⟩; return;
        case 23: ⟨MA23 - X is INSTANCE, Y is NOUNPHRASE - P 27⟩; return;
        case 24: ⟨MA24 - X is INSTANCE - P 28⟩; return;
        case 25: ⟨MA25 - X and Y are XOFY - P 29⟩; return;
        case 26: ⟨MA26 - Y is XOFY - P 30⟩; return;
        case 27: ⟨MA27 - X is XOFY, Y is RELATIONSHIP - P 31⟩; return;
        case 28: ⟨MA28 - X is XOFY - P 32⟩; return;
        case 29: ⟨MA29 - X is XOFY, Y is NOUNPHRASE or VALUE 33⟩; return;
        case 30: ⟨MA30 - X and Y are ACTION - P 34⟩; return;
        case 31: ⟨MA31 - X is ACTION, Y is NOUNPHRASE 35⟩; return;
        case 32: ⟨MA32 - X is ACTION, Y is ADJECTIVE or PROPERTYNOUN - P 36⟩; return;
        case 33: ⟨MA33 - X or Y is ACTION - P 37⟩; return;
        case 34: ⟨MA34 - X is NOUNPHRASE, Y is ACTION - P 38⟩; return;
        case 35: ⟨MA35 - X is KIND - P 39⟩; return;
        case 36: ⟨MA36 - X is INSTANCE or NOUNPHRASE or VALUE, Y is ADJECTIVE 40⟩; return;
        case 37: ⟨MA37 - X is NOUNPHRASE or VALUE, Y is PROPERTYNOUN - P 41⟩; return;
        case 38: ⟨MA38 - X is ADJECTIVE, Y is NOUNPHRASE - P 42⟩; return;
        case 39: ⟨MA39 - X, Y are ADJECTIVE, PROPERTYNOUN or PROPERTYLIST - P 43⟩; return;
        case 40: ⟨MA40 - X, Y are PROPERTYLIST or PROPERTYNOUN - P 44⟩; return;
        case 41: ⟨MA41 - X is ADJECTIVE, Y is VALUE - P 45⟩; return;
        case 42: ⟨MA42 - X, Y are ADJECTIVE 46⟩; return;
        case 43: ⟨MA43 - X is NOUNPHRASE, Y is PROPERTYLIST 47⟩; return;
        case 44: ⟨MA44 - X is PROPERTYLIST, Y is NOUNPHRASE 48⟩; return;
        case 45: ⟨MA45 - X is INSTANCE, Y is PROPERTYLIST 49⟩; return;
        case 46: ⟨MA46 - X is PROPERTYLIST, Y is INSTANCE 50⟩; return;
        case 47: ⟨MA47 - X, Y are PROPERTYLIST, VALUE 51⟩; return;
        case 48: ⟨MA48 - X is RELATIONSHIP, Y is WITH 52⟩; return;
        case 49: ⟨MA49 - X is WITH, Y is RELATIONSHIP 53⟩; return;
        case 50: ⟨MA50 - Y is EVERY 54⟩; return;
        case 51: ⟨MA51 - X is EVERY - P 55⟩; return;
        case 52: ⟨MA52 - X, Y are EVERY - P 56⟩; return;
        case 53: ⟨MA53 - X is EVERY, Y is RELATIONSHIP 57⟩; return;
        case 54: ⟨MA54 - X is XOFY, Y is ADJECTIVE 58⟩; return;
        case 55: ⟨MA55 - X is RELATIONSHIP, Y is VALUE 59⟩; return;
        case 56: ⟨MA56 - X is VALUE, Y is RELATIONSHIP 60⟩; return;
        case 57: ⟨MA57 - X is VALUE, Y is ACTION 61⟩; return;
        default: LOG("Unimplemented assertion case %d\n", ma_case);
            internal_error("No implementation for make assertion case");
    }
}
```

The function make_assertion_recursive is called from 8/creat.

§**5. Case 1.** "A is a B with I": process as "A is a B" followed by "A is I", which works nicely because we bend grammar to allow "is" in place of "has" when it comes to lists of property values. This construction is only meaningful for a few possible node types of A, but we don't provide any special problem messages for the rest, because recursion will do the job for us. But see also case 48.

The wickerwork box is a container with description "Pricy." The bat and ball are things with description "Cricket equipment." A trophy is a kind of container with score for finding 5.

⟨MA1 - Y is WITH 5⟩ ≡
```
make_assertion_recursive(px, py->down, assertion_depth);
if ((pn_get_node_type(px) == NOUNPHRASE_NT) &&
    (is_a_direction_object(pn_get_refers_to(px))) &&
    (pn_get_node_type(py->down) == NOUNPHRASE_NT))
make_assertion_recursive(py->down, py->down->next, assertion_depth);
else
make_assertion_recursive(px, py->down->next, assertion_depth);
```

This code is used in §4.

§**6. Case 2.** "An A with I is B": we treat this symmetrically with case 1, but only in order that the assertions we recurse into will produce a suitable problem message: so this does not really do anything constructive. It *is* possible for a left-hand WITH to do something meaningful, but only when a location is on the right, which is case 49, or when we have an implication.

A container with description "Solid." is the solid box.

⟨MA2 - X is WITH 6⟩ ≡
```
if ((pn_get_node_type(px->down) == INSTANCE_NT) &&
    (is_adjlist(px->down->next)) &&
    (is_adjlist(py))) {
    imp_new(px, py);
} else {
    int np = problem_count;
    make_assertion_recursive(px->down, py, assertion_depth);
    if (problem_count == np)
        make_assertion_recursive(px->down->next, py, assertion_depth);
}
```

This code is used in §4.

§**7. Case 3.** "A with B is C with D" must be incorrect.

A container with description "White" is a container with description "Black".

⟨MA3 - X and Y are WITH - P 7⟩ ≡
```
assertion_problem(_P_(C8Case3),
    "you can't say that one general description is another ",
    "for instance by writing 'A container with carrying capacity 10 "
    "is a dark room'.");
```

This code is used in §4.

§**8. Case 4.** "A is B and C": process as "A is B" then "A is C".

⟨MA4 - Y is AND 8⟩ ≡
```
{ parse_node *across = py->down;
    while (across) {
        make_assertion_recursive(px, across, assertion_depth);
        across = across->next;
    }
}
```

This code is used in §4.

§**9. Case 5.** "A and B are C": process as "A is C" then "B is C".

⟨MA5 - X is AND 9⟩ ≡
```
    parse_node *across = px->down;
    int np = problem_count;
    while (across) {
        if (np == problem_count)
            make_assertion_recursive(across, py, assertion_depth);
        across = across->next;
    }
```

This code is used in §4.

§**10. Case 6.** This can be proved never to happen, but just in case:

⟨MA6 - X is ALLOWED - P 10⟩ ≡
```
    internal_error("Forbidden case 6 in make assertion has occurred.");
```

This code is used in §4.

§**11. Case 7.** Permission granted for a property, where A is not an object.

A number has a number called square root.

⟨MA7 - Y is ALLOWED - P 11⟩ ≡
```
    if [[px == player]] {
        pn_make_player_noun_phrase(px);
        make_assertion_recursive(px, py, assertion_depth);
        return;
    }
    if (pn_get_node_type(px) == VALUE_NT) {
        specification *spec = pn_get_evaluation(px);
        kind_of_value *kov = spec_get_kind_of_value(spec);
        if ((spec_is_generic_CONSTANT(spec)) && (kov_has_properties(kov))) {
            if (traverse == 1) recursively_declare_properties(px, py->down);
            return;
        }
        if (spec_is_actual_CONSTANT(spec)) {
            if (is_kova(kov, ACTION_NAME_TY)) {
                action_name *an = ACTION_NAME_spec_to_action_name(spec);
                if (an == NULL) internal_error("failed to extract action-name structure");
                if (traverse == 2) an_add_variable(an, py->down);
                return;
            }
```

```
            if (is_kova(kov, ACTIVITY_TY)) {
                activity *av = ACTIVITY_spec_to_activity(spec);
                if (traverse == 2) {
                    if [[px == ... activity]]
                        av_add_variable(av, py->down);
                    else
                        assertion_problem(_P_(C8Case7c),
                    "an activity has to be formally referred to in a way making clear that "
                    "it is indeed a rulebook when we give it named values",
                    "to reduce the risk of ambiguity. So 'The printing the banner text "
                    "activity has a number called the accumulated vanity' is fine, but "
                    "'Printing the banner text has a number called...' is not. (I'm "
                    "insisting on the presence of the word 'activity' because the "
                    "syntax is so close to that for giving properties to objects, and "
                    "it's important to avoid mistakes here.)");
                    }
                    return;
                }
                if (is_kova(kov, RULEBOOK_TY)) {
                    rulebook *rb = RULEBOOK_spec_to_rulebook(spec);
                    if (traverse == 2) {
                        if [[px == ... rulebook]]
                            rb_add_variable(rb, py->down);
                        else
                            assertion_problem(_P_(C8Case7b),
                        "a rulebook has to be formally referred to in a way making clear that "
                        "it is indeed a rulebook when we give it named values",
                        "to reduce the risk of ambiguity. So 'The every turn rulebook has a "
                        "number called the accumulated bonus' is fine, but 'Every turn has a "
                        "number called...' is not. (I'm insisting on the presence of the word "
                        "'rulebook' because the syntax is so close to that for giving "
                        "properties to objects, and it's important to avoid mistakes here.)");
                        }
                        return;
                    }
                }
            }
            assertion_problem(_P_(C8Case7),
                "only an object, kind, rulebook, action or activity can be allowed to "
                "have properties or variables",
                "so for instance 'A door has a colour' is fine but "
                "'A number has a length' is not.");
```

This code is used in §4.

§12. Case 8. This time a valid permission.

A container has a number called security rating.

⟨MA8 - X is NOUNPHRASE or INSTANCE, Y is ALLOWED 12⟩ ≡

```
    if (traverse == 1) recursively_declare_properties(px, py->down);
```

This code is used in §4.

§**13. Case 9.** Two sorts of sentence can validly fall under this heading. The first is easy:

The innings total is a number that varies.

But the second takes some setting up. Only the last of these sentences falls into case 9:

Colour is a kind of value. Green and blue are colours. A thing has a colour. The barn door is green.

⟨MA9 - X is NOUNPHRASE, Y is VALUE 13⟩ ≡ {

```
    kind_of_value *kov;
    quantity *q = spec_get_constant_quantity_if_any(pn_get_evaluation(py));
    if (q) {
        property_name *pname = kov_get_coinciding_property(qty_kind_of_value(q));
        if (pname) {
            if (traverse == 2) assert_property_value_from_property_subtree(pname, pn_get_refers_to(px),
NULL, py);
            return;
        }
    }
    if (spec_is_generic_CONSTANT(pn_get_evaluation(py))) {
        if (traverse == 1) issue_value_equation_problem(px, py);
        return;
    }
    kov = spec_get_kind_of_value(pn_get_evaluation(py));
    if (kov_name_can_coincide_with_property(kov)) {
        property_name *pname = kov_get_coinciding_property(kov);
        if (pname) {
            if (traverse == 2) assert_property_value_from_property_subtree(pname, pn_get_refers_to(px),
NULL, py);
            return;
        }
    }
    assertion_problem(_P_(C8Case9),
        "this seems to say that a thing is a value",
        "like saying 'the chair is 10'.");
}
```

This code is used in §4.

§14. Case 10. Symmetrically, if questionably, we allow "A number variable is the departures counter". The only examples I can think of are poor style, so perhaps this case should in fact produce a problem.

The other test below, though, is important: it handles setting a variable to a value which happens to be a specifically named object, as in the second of these sentences:

The quest assignment is a thing that varies. The quest assignment is the Holy Grail.

⟨MA10 - X is VALUE, Y is NOUNPHRASE 14⟩ ≡

```
if ((pn_get_refers_to(py) != NULL) &&
    (spec_get_storage_form(pn_get_evaluation(px)) == NONLOCAL_VARIABLE_SPC)) {
    quantity *q = RETRIEVE_FROM_SPEC(pn_get_evaluation(px), quantity);
    if (traverse == 1) return;
    if (q == NULL) internal_error("Null quantity in value");
    if (qty_is_a_variable(q)) {
        kind_of_value *variable_kov = qty_kind_of_value(q);
        kind_of_value *constant_kov;
        specification *initialiser = world_object_to_OBJECT_spec(pn_get_refers_to(py));
        if ((pn_get_refers_to(py)) && (pn_get_refers_to(py)->kind))
            constant_kov = kovko(pn_get_refers_to(py)->kind);
        else
            constant_kov = kova(OBJECT_TY);
        if (is_kova(variable_kov, OBJECT_TY))
            variable_kov = kova(OBJECT_TY);                weaken for now: will check later
        if (can_we_cast_kovs(constant_kov, variable_kov) != ALWAYS_MATCH) {
            specification *dummy1 = new_generic_CONSTANT_type(variable_kov),
                *dummy2 = new_generic_CONSTANT_type(constant_kov);
            quote_source(1, current_sentence);
            quote_words(2, px->word_ref1, px->word_ref2);
            quote_words(3, py->word_ref1, py->word_ref2);
            quote_spec(4, dummy1);
            quote_spec(5, dummy2);
            handmade_problem(_P_(C8Case10a));
            issue_problem_segment(
                "The sentence %1 tells me that '%2', which is %4 that "
                "varies, should have value '%3', but this is %5 and not %4.");
            issue_problem_end();
            return;
        }
        initialise_global_variable(q, initialiser, py);
        return;
    }
}
if ((spec_is_generic_CONSTANT(pn_get_evaluation(px))) &&
    (kov_when_VALUE_is_evaluated(pn_get_evaluation(px)))) {
    if (traverse == 1) issue_value_equation_problem(py, px);
    return;
}
assertion_problem(_P_(C8Case10b),
    "this seems to say that a thing is a value",
    "like saying 'the chair is 10'.");
```

This code is used in §4.

§**15. Case 11.** Usually it makes no sense to equate two values: "5 is 10", for instance, and we produce a variety of more or less scornful errors.

```
10 is 15. 14 is a number. "Fish" is text. The turn count is a text. 19 is a rulebook. "Frog"
is a number. Before rules is a rule. The turn count is "Surprising."
```

But we do take polite notice when X is a variable name and Y is an initial value with a compatible type, as in the second sentence here:

```
The innings total is a number that varies. The innings total is 101.
```

⟨MA11 - X and Y are VALUEs 15⟩ ≡

```
int ng = 0;
if (spec_is_generic_CONSTANT(pn_get_evaluation(px))) ng++;
if (spec_is_generic_CONSTANT(pn_get_evaluation(py))) ng++;
if (ng == 1) {
    kind_of_value *g_kov = spec_get_kind_of_value(pn_get_evaluation(py));
    kind_of_value *a_kov = spec_get_kind_of_value(pn_get_evaluation(px));
    if ((a_kov) && (kov_compare(a_kov, g_kov))) {
        if ((is_kova(a_kov, NUMBER_TY)) || (is_kova(a_kov, TEXT_TY))) {
            quote_source(1, current_sentence);
            handmade_problem(_P_(C8Case11c));
            issue_problem_segment(
                "%1: Grateful as I generally am for your guidance, "
                "I think perhaps I could manage without this sentence.");
            issue_problem_end();
            return;
        }
        return;
    }
}
quantity *q = spec_get_constant_quantity_if_any(pn_get_evaluation(py));
if (q) {
    property_name *pname = kov_get_coinciding_property(qty_kind_of_value(q));
    if (pname) {
        if ((spec_get_constant_quantity_if_any(pn_get_evaluation(px))) ||
            (spec_is_generic_CONSTANT(pn_get_evaluation(px)))) {
            if (traverse == 2) assert_property_value_from_property_subtree(pname, NULL,
pn_get_evaluation(px), py);
            return;
        }
    }
}
if (traverse == 2) return;
if (spec_get_storage_form(pn_get_evaluation(px)) == NONLOCAL_VARIABLE_SPC) {
    if ((spec_is_generic_NONLOCAL_VARIABLE(pn_get_evaluation(px))) &&
        (spec_is_generic_NONLOCAL_VARIABLE(pn_get_evaluation(py)))) {
        quote_source(1, current_sentence);
        quote_words(2, px->word_ref1, px->word_ref2);
        quote_words(3, py->word_ref1, py->word_ref2);
        handmade_problem(_P_(C8Case11j));
        issue_problem_segment(
            "The sentence %1 seems to tell me that '%2', which describes "
            "a kind of variable, is the same as '%3', another description "
```

```
            "of a kind of variable - but that doesn't make sense to me. "
            "(Perhaps you intended one of these to be a specific variable, "
            "but chose a wording which looked accidentally like a "
            "general description?)");
        issue_problem_end();
        return;
    }
    quantity *q = RETRIEVE_FROM_SPEC(pn_get_evaluation(px), quantity);
    if (qty_is_a_variable(q)) {
        quantity *q2;
        kind_of_value *variable_kov = qty_kind_of_value(q);
        kind_of_value *constant_kov = spec_get_kind_of_value(pn_get_evaluation(py));
        if (spec_is_actual_NONLOCAL_VARIABLE(pn_get_evaluation(py))) {
            quantity *q2 = RETRIEVE_FROM_SPEC(pn_get_evaluation(py), quantity);
            constant_kov = qty_kind_of_value(q2);
        }
        q2 = spec_get_constant_quantity_if_any(pn_get_evaluation(py));
        if (q2) constant_kov = qty_kind_of_value(q2);
        if (spec_is_generic_NONLOCAL_VARIABLE(pn_get_evaluation(py))) {
            if (kov_compare(variable_kov, constant_kov)) return;
            LOG("$u, $u\n", variable_kov, constant_kov);
            quote_source(1, current_sentence);
            quote_words(2, px->word_ref1, px->word_ref2);
            quote_words(3, py->word_ref1, py->word_ref2);
            quote_kov(4, variable_kov);
            handmade_problem(_P_(C8Case11a));
            issue_problem_segment(
                "The sentence %1 seems to tell me that '%2', which has "
                "already been declared as %4 that varies, is instead "
                "'%3' - but that would be a contradiction.");
            issue_problem_end();
            return;
        }
        initialise_global_variable(q, pn_get_evaluation(py), py);
        return;
    }
}
if (kov_compare(spec_get_kind_of_value(pn_get_evaluation(px)),
        spec_get_kind_of_value(pn_get_evaluation(py)))) {
    quote_source(1, current_sentence);
    quote_words(2, px->word_ref1, px->word_ref2);
    quote_words(3, py->word_ref1, py->word_ref2);
    quote_type_of(4, pn_get_evaluation(px));
    quote_type_of(5, pn_get_evaluation(py));
    handmade_problem(_P_(C8Case11d));
    issue_problem_segment(
        "Before reading %1, I already knew that '%2' is %4 and "
        "'%3' likewise: so they are specific values, and saying "
        "that they are equal will not make it so.");
    issue_problem_end();
    return;
}
if ((spec_get_constant_quantity_if_any(pn_get_evaluation(px))) ||
```

```
        (spec_get_storage_form(pn_get_evaluation(px)) == NONLOCAL_VARIABLE_SPC)) {
        quote_source(1, current_sentence);
        quote_words(2, px->word_ref1, px->word_ref2);
        LOG("Type is: $S\n", pn_get_evaluation(px));
        quote_type_of(4, pn_get_evaluation(px));
        handmade_problem(_P_(C8Case11e));
        issue_problem_segment(
            "Before reading %1, I already knew that '%2' is %4, "
            "and it is too late to change now.");
        issue_problem_end();
        return;
    }
    if (is_kova(spec_get_kind_of_value(pn_get_evaluation(px)), NUMBER_TY)) {
        quote_source(1, current_sentence);
        handmade_problem(_P_(C8Case11f));
        issue_problem_segment(
            "%1: That, sir, is a damnable lie.");
        issue_problem_end();
        return;
    }
    if (is_kova(spec_get_kind_of_value(pn_get_evaluation(px)), TEXT_TY)) {
        quote_source(1, current_sentence);
        handmade_problem(_P_(C8Case11g));
        issue_problem_segment(
            "%1: And I am the King of Siam.");
        issue_problem_end();
        return;
    }
    quote_source(1, current_sentence);
    quote_words(2, px->word_ref1, px->word_ref2);
    quote_words(3, py->word_ref1, py->word_ref2);
    quote_type_of(4, pn_get_evaluation(px));
    quote_type_of(5, pn_get_evaluation(py));
    handmade_problem(_P_(C8Case11h));
    issue_problem_segment(
        "Before reading %1, I already knew that '%2' is %4 and "
        "'%3' is %5: so they are specific values, and saying "
        "that they are equal will not make it so.");
    issue_problem_end();
```

This code is used in §4.

§**16. Case 12.** "A is a kind of B". B, of course, is `py->down`, but there is also the possibility of "A is a kind", in which case `py->down` is null. We rely on `pn_get_refers_to(py)` being the kind in question, if B is the name of a kind, or on `py->down` being a value node for "a kind of value", etc. Only the last two of the following sentences do not issue problems:

> 100 is a kind. A container is a kind of door. A reptile is a kind of number. An objet d'art is a kind of object. An ethics is a kind of rulebook. A lecture hall is a kind of Cell. An amphibian is a kind of animal. A bird is a kind.

Note that the possible children of a KIND node have already been vetted once at resolution time.

⟨MA12 - Y is KIND 16⟩ ≡

```
if ((traverse == 1) && (pn_get_refers_to(px) == NULL) &&
    ((py->down == NULL) || (pn_get_node_type(py->down) != VALUE_NT))) {
    assertion_problem(_P_(BelievedImpossible),
        "that already has a meaning",
        "and can't be made into a new kind.");
    return;
}
if ((traverse == 1) && (pn_get_refers_to(px)) && (pn_get_refers_to(px)->kind_flag) &&
    (pn_get_refers_to(px)->creating_sentence != current_sentence)) {
    quote_object(1, pn_get_refers_to(px));
    quote_source(2, current_sentence);
    quote_source(3, pn_get_refers_to(px)->creating_sentence);
    handmade_problem(_P_(C8Case12c));
    issue_problem_segment(
        "You wrote %2, but that seems to reiterate that a kind already "
        "existing ('%1', created by %3) is to be a kind. To prevent a "
        "variety of possible misunderstandings, this is not allowed.");
    issue_problem_end();
    return;
}
if ((traverse == 1) && (pn_get_refers_to(px)) &&
    (pn_get_refers_to(px)->creating_sentence != current_sentence)) {
    quote_object(1, pn_get_refers_to(px));
    quote_source(2, current_sentence);
    quote_source(3, pn_get_refers_to(px)->creating_sentence);
    handmade_problem(_P_(C8Case12b));
    issue_problem_segment(
        "You wrote '%2', but that seems to say that some "
        "room or thing already created ('%1', created by '%3') is now to "
        "become a kind. To prevent a variety of possible misunderstandings, "
        "this is not allowed: when a kind is created, the name given has "
        "to be a name not so far used. (Sometimes this happens due to "
        "confusion between names. For instance, if a room called 'Marble "
        "archway' exists, then Inform reads 'An archway is a kind of thing', "
        "Inform will read 'archway' as a reference to the existing room, "
        "not as a new name. To solve this, put the sentences the other way "
        "round.)");
    issue_problem_end();
    return;
}
if (py->down != NULL) {
    if (pn_get_node_type(py->down) == VALUE_NT) {
        if (traverse == 2) return;
```

```
            if ((spec_is_generic_CONSTANT(pn_get_evaluation(py->down))) &&
                (is_kova(spec_get_kind_of_value(pn_get_evaluation(py->down)), ANY_VALUE_TY)))
{
                pn_set_refers_to(px, NULL);
                return;
            }
            if ((spec_is_generic_CONSTANT(pn_get_evaluation(py->down))) &&
                (is_kova(spec_get_kind_of_value(pn_get_evaluation(py->down)), OBJECT_TY))) {
                kind_of_object_problem(_P_(BelievedImpossible));
                return;
            }
            LOG("$X", pn_get_evaluation(py->down));
            assertion_problem(_P_(BelievedImpossible),
                "you aren't allowed to create kinds of non-object kinds "
                "except in the special case of 'value'",
                "so for example 'colour is a kind of value' is allowed but "
                "'prime is a kind of number' is not.");
            return;
        }
        if (pn_get_node_type(py->down) == WITH_NT) {
            assertion_problem(_P_(C8Case12e),
                "I don't recognise that as a kind",
                "such as 'room' or 'door': it would need to be straightforwardly "
                "the name of a kind, and not be qualified with adjectives like "
                "'open'.");
            return;
        }
    }
    if (traverse == 2) return;
    if (pn_get_refers_to(py) == NULL) {
        LOG("Subtree: $T\n", py);
        assertion_problem(_P_(BelievedImpossible),
            "I don't recognise that as a kind",
            "such as 'room' or 'door'");
        return;
    }
    if (pn_get_refers_to(py)->kind_flag == FALSE)
        internal_error("Kind of instance not caught at resolution time");
    pn_get_refers_to(px)->kind_flag = TRUE;
    assert_kind_of_object(pn_get_refers_to(px), pn_get_refers_to(py));
    if ((py->down) && (pn_get_node_type(py->down) == INSTANCE_NT)
        && (pn_get_evaluation(py->down)) && (spec_get_proposition(pn_get_evaluation(py->down))))
{
        prop_true_in_world_model_about(
            spec_get_proposition(pn_get_evaluation(py->down)), pn_get_refers_to(px), NULL);
    }
```

This code is used in §4.

§**17. Case 13.** We are allowed to equate a property noun with a value provided there is a current subject of conversation to be the owner of the property.

The strongbox is a container in the Pitch. The description is "Rough around the edges." The matching key is the inlaid ivory key.

⟨MA13 - X is PROPERTYNAME, Y is NOUNPHRASE or VALUE 17⟩ ≡

```
    if (get_current_subject() == NULL) {
        assertion_problem(_P_(C8Case13),
            "nothing is under discussion which might have this property",
            "so this is like starting with 'The description is \"Orange.\"': "
            "I can't tell what of.");
        return;
    }
    if (traverse == 2) {
        if [[py == not ...]] {
            negative_sentence_problem(_P_(BelievedImpossible));
            return;
        }
        assert_property_value_from_property_subtree(pn_get_property(px), get_current_subject(),
NULL, py);
    }
```

This code is used in §4.

§**18. Case 14.** We first worry about "East is a dead end", which must not change the kind of the "east" object. But we must be careful not to catch "East is a direction" in the same net because, of course, that does set its kind. We essentially rewrite as:

Z is a dead end. East is Z.

where Z is a newly created and nameless object, but we do not allow the first assertion to change the current subject of conversation, as then the second assertion would say that Z is east of Z, not east of the current room.

But most of the sentences falling into this case have the form "X is a Y." We can test both varieties with the latter two sentences in the following:

The Pitch is a room. East is a room. The lacquered box is a container.

⟨MA14 - X is NOUNPHRASE, Y is INSTANCE 18⟩ ≡

```
    if (pn_get_refers_to(px) == NULL) {
        assertion_problem(_P_(C8Case14),
            "'nowhere' cannot be made specific",
            "and so cannot have specific properties or be of any given kind.");
        return;
    }
    if ((is_a_direction_object(pn_get_refers_to(px))) &&
        (pn_get_refers_to(py) != kind_direction)) {
        convert_instance_to_nounphrase(py, assertion_depth, FALSE);
        make_assertion_recursive(px, py, assertion_depth);
        return;
    }
    if [[py == everywhere]] assert_object_everywhere(pn_get_refers_to(px));
    else assert_instance(px, py);
```

This code is used in §4.

§**19. Case 15.** "On the table is a container." Much the same as case 14a, but the code is duplicated to avoid having to `goto` between cases.

In the desk is a copy of Wisden.

⟨MA15 - X is RELATIONSHIP, Y is INSTANCE 19⟩ ≡

```
if ((px->down) && (pn_get_node_type(px->down) == EVERY_NT)) {
    make_generalisation(px, py);
    return;
}
if (pn_get_refers_to(py) == kind_region) {
    assertion_problem(_P_(C8Case15),
        "a region cannot be given a specific location",
        "since it contains what may be many rooms, which may not be "
        "contiguous and could be scattered about all over. (Sometimes "
        "this problem arises because an ambiguous sentence like 'East "
        "of Eden is a region.' has been used: Inform reads that as "
        "saying that a nameless region lies to the east of the room "
        "'Eden'. The desired effect can be got using 'called' to stop "
        "Inform taking 'East of' literally: for instance, 'The Land of "
        "Nod is in a region called East of Eden.'");
    return;
}
convert_instance_to_nounphrase(py, assertion_depth,
    (pn_get_relationship(px) == a_incorporates_b_predicate));
make_assertion_recursive(px, py, assertion_depth);
```

This code is used in §4.

§**20. Case 16.** This can involve a tautology, or a contradiction in terms:

A room is a room. A container is a door.

But it can also be valid, as in the final sentence here:

Colour is a kind of value. The colours are green and purple. A room has a colour. A room is usually purple.

⟨MA16 - X and Y are INSTANCE - P 20⟩ ≡

```
if (spec_get_proposition(pn_get_evaluation(px)) == NULL) {
    if [[py == everywhere]] assert_object_everywhere(pn_get_refers_to(px));
    assert_instance(px, py);
    return;
}
assertion_problem(_P_(C8Case16),
    "one description cannot equal another",
    "by saying something like 'a container is a room' (not true!) or "
    "even 'a dark room is a room' (true but a tautology, so not allowed "
    "because it suggests that something else was meant instead).");
```

This code is used in §4.

§**21. Case 17.** This is a finicky stylistic error, and it strikes me as quite likely that the beta-testers will object to it.

An animal is in the desk.

⟨MA17 - X is INSTANCE, Y is RELATIONSHIP - P 21⟩ ≡

```
if ((py->down) && (pn_get_node_type(py->down) == EVERY_NT)) {
    make_generalisation(py, px);
    return;
}
if (spec_get_quantification_parameter(pn_get_evaluation(px)) >= 1) {
    convert_instance_to_nounphrase(px, assertion_depth, FALSE);
    make_assertion_recursive(py, px, assertion_depth);
    return;
}
if ((pn_get_relationship(py)) &&
    (bp_sets_a_property(pn_get_relationship(py)))) {
    assert_subtree_in_relationship(px, py);
    return;
}
if ((pn_get_relationship(py)) &&
    (bp_relates_values_not_objects(pn_get_relationship(py)))) {
    assert_subtree_in_relationship(px, py);
    return;
}
assertion_problem(_P_(C8Case17),
    "something described only by its kind should not be given a "
    "specific place or role in the world",
    "to avoid ambiguity. For instance, suppose 'car' is a kind. Then "
    "we are not allowed to say 'a car is in the garage': there's too "
    "much risk of confusion between whether an individual (but "
    "nameless) car is referred to, or whether cars are generically to "
    "be found there. Sentences of this form are therefore prohibited, "
    "though more specific ones like 'a car called Genevieve is in the "
    "garage' are fine, as is the reverse, 'In the garage is a car.'");
```

This code is used in §4.

§**22. Case 18.** The first of these sample sentences is deservedly rejected, but the second makes a non-reciprocal map connection: "D of the R is E of the S" is construed as "D of the R is S" then "R is E of the S". We have to suppress NI's tendency to make tentative reciprocal map connections, because even though they will only be listed as "likely", we know they are in fact impossible in this case.

In the box is on the desk. East of the Pitch is north of the Pavilion.

⟨MA18 - X and Y are RELATIONSHIP 22⟩ ≡

```
if ((pn_int_annotation(px, relationship_node_type_ANNOT) == DIRECTION_RELN) &&
    (pn_int_annotation(py, relationship_node_type_ANNOT) == DIRECTION_RELN)) {
    enter_one_way_mapping_mode();
    make_assertion_recursive(px, py->down, assertion_depth);
    make_assertion_recursive(px->down, py, assertion_depth);
    exit_one_way_mapping_mode();
    return;
}
```

```
assertion_problem(_P_(C8Case18),
    "this says that two different relations are the same",
    "like saying that 'in the box is on the table'. (Sometimes this "
    "happens if I misinterpret names of places like 'In Prison' or "
    "'East of Fissure'.)");
```

This code is used in §4.

§**23. Case 19.** "A box is on the table." This makes "box" the subject of discussion. "The Gazebo is west of the Lawn" also falls into this case, since "west of the Lawn" parses to a `RELATIONSHIP_NT` subtree.

⟨MA19 - X is NOUNPHRASE, Y is RELATIONSHIP 23⟩ ≡

```
if ((py->down) && (convert_instances_in_subtree(py->down, assertion_depth))) {
    make_assertion_recursive(px, py, assertion_depth);
    return;
}
assert_subtree_in_relationship(px, py);
```

This code is used in §4.

§**24. Case 20.** "On the table is a box." This is not quite the same as its mirror case, case 19: it doesn't change the subject of discussion, which probably remains the table. But we don't see the difference here, because subject/object changes are handled higher up.

On the desk is a box. North of the Pitch is the Pavilion.

⟨MA20 - X is RELATIONSHIP, Y is NOUNPHRASE 24⟩ ≡

```
assert_subtree_in_relationship(py, px);
if ((px->down) && (pn_get_node_type(px->down) == NOUNPHRASE_NT) &&
    (pn_get_refers_to(px->down))) {
    set_current_subject(pn_get_refers_to(px->down));
}
```

This code is used in §4.

§**25. Case 21.** Hoovering up a variety of implausible things claimed to have a spatial location.

On the desk is 100. East of the Pitch is a rulebook.

⟨MA21 - X or Y is RELATIONSHIP - P 25⟩ ≡

```
if ((pn_get_node_type(px) == RELATIONSHIP_NT) &&
    (pn_get_node_type(py) == ADJECTIVE_NT) && (pn_get_relationship(px)) &&
    (bp_relates_values_not_objects(pn_get_relationship(px)))) {
    assert_subtree_in_relationship(py, px);
    return;
}
if ((pn_get_node_type(py) == RELATIONSHIP_NT) &&
    (pn_get_node_type(px) == ADJECTIVE_NT) && (pn_get_relationship(py)) &&
    (bp_relates_values_not_objects(pn_get_relationship(py)))) {
    assert_subtree_in_relationship(px, py);
    return;
}
if [[px == player]] {
    pn_make_player_noun_phrase(px);
    make_assertion_recursive(px, py, assertion_depth);
```

```
    return;
}
if [[py == player]] {
    pn_make_player_noun_phrase(py);
    make_assertion_recursive(px, py, assertion_depth);
    return;
}
assertion_problem(_P_(C8Case21),
    "this seems to give a worldly relationship to something intangible",
    "like saying that 'in the box is a text'. Perhaps it came "
    "to this because you gave something physical a name which was "
    "accidentally something meaningful to me in another context? "
    "If so, you may be able to get around it by rewording ('In the "
    "box is a medieval text') or in extremis by using 'called' "
    "('In the box is a thing called text').");
```

This code is used in §4.

§**26. Case 22.** We first take care of "East is the marble door", or similar, before being disdainful of the pitting of noun-phrase against noun-phrase.

East is the Pavilion. A rose is a rose.

⟨MA22 - X and Y are NOUNPHRASE 26⟩ ≡

```
if ((pn_get_refers_to(px)) && (is_a_direction_object(pn_get_refers_to(px)))) {
if (get_current_subject() == NULL) {
        assertion_problem(_P_(C8Case22a),
            "no location is under discussion to be the origin of this "
            "map connection",
            "so this is like starting with 'North is the Aviary': I "
            "can't tell where from.");
        return;
    }
    make_map_connection(get_current_subject(),
        pn_get_refers_to(py), pn_get_refers_to(px));
    return;
}
if (pn_get_refers_to(px) == pn_get_refers_to(py)) {
    assertion_problem(_P_(C8Case22b),
        "this seems to say that something is itself",
        "like saying 'the coin is the coin'. This is an odd thing "
        "to say, and makes me think that I've misunderstood you.");
    return;
}
if ((pn_get_refers_to(px)) && (pn_get_refers_to(py))) {
    LOG("pn_get_refers_to(px) is $O, pn_get_refers_to(py) is $O\n",
        pn_get_refers_to(px), pn_get_refers_to(py));
}
int name1 = -1, name2 = -1;
if (pn_get_refers_to(py)) {
    name1 = pn_get_refers_to(py)->word_ref1;
    name2 = pn_get_refers_to(py)->word_ref2;
}
if (name1 >= 0) {
```

```
    if [[name1, name2 == not ...]] {
        negative_sentence_problem(_P_(C8Case22f));
        return;
    }
    if [[name1, name2 == action]] {
        assertion_problem(_P_(C8Case22c),
            "it is not sufficient to say that something is an 'action'",
            "without giving the necessary details: for example, 'Unclamping "
            "is an action applying to one thing.'");
        return;
    }
}
quote_source(1, current_sentence);
quote_words(2, px->word_ref1, px->word_ref2);
quote_words(3, py->word_ref1, py->word_ref2);
if [[px == if/unless ...]] {
    handmade_problem(_P_(C8Case22d));
    issue_problem_segment(
        "I am reading the sentence %1 as a declaration of the initial "
        "state of the world, so I'm expecting that it will be definite. "
        "The only way I can construe it that way is by thinking that "
        "'%2' and '%3' are two different things, but that doesn't make "
        "sense, and the 'if' makes me think that perhaps you did not "
        "mean this as a definite statement after all. Although 'if...' "
        "is often used in rules and definitions of what to do in given "
        "circumstances, it shouldn't be used in a direct assertion.");
    issue_problem_end();
    return;
}
{ int variant;
    if (fix_rng_at_start_of_play) variant = 0;
    else variant = (time(0))&7;
    switch(variant) {
        case 1:
            quote_text(4, "the chalk"); quote_text(5, "the cheese");
            quote_text(6, "Dairy Products School"); break;
        case 2:
            quote_text(4, "St Peter"); quote_text(5, "St Paul");
            quote_text(6, "Pearly Gates"); break;
        case 3:
            quote_text(4, "Tom"); quote_text(5, "Jerry");
            quote_text(6, "Mouse-Hole"); break;
        case 4:
            quote_text(4, "Clark Kent"); quote_text(5, "Lex Luthor");
            quote_text(6, "Metropolis"); break;
        case 5:
            quote_text(4, "Ron"); quote_text(5, "Hermione");
            quote_text(6, "Hogsmeade"); break;
        case 6:
            quote_text(4, "Tarzan"); quote_text(5, "Jane");
            quote_text(6, "Treehouse"); break;
        case 7:
            quote_text(4, "Adam"); quote_text(5, "Eve");
```

```
            quote_text(6, "Land of Nod"); break;
        default:
            quote_text(4, "the hawk"); quote_text(5, "the handsaw");
            quote_text(6, "Elsinore"); break;
    }
}
handmade_problem(_P_(C8Case22e));
issue_problem_segment(
    "The sentence %1 appears to say two things are the "
    "same - I am reading '%2' and '%3' as two different things, and "
    "therefore it makes no sense to say that one is the other: "
    "it would be like saying that '%4 is %5'. It would be all "
    "right if the second thing were the name of a kind, perhaps "
    "with properties: for instance "
    "'%6 is a lighted room' says that something called %6 "
    "exists and that it is a 'room', which is a kind I know about, "
    "combined with a property called 'lighted' which I also know about.");
issue_problem_end();
```

This code is used in §4.

§27. Case 23.

⟨MA23 - X is INSTANCE, Y is NOUNPHRASE - P 27⟩ ≡

```
assertion_problem(_P_(C8Case23),
    "this seems to say that a general description is something specific",
    "like saying that 'a door is the oak tree'.");
```

This code is used in §4.

§28. Case 24. And in general, it is miscellaneously a bad idea to equate a description with anything much else.

⟨MA24 - X is INSTANCE - P 28⟩ ≡

```
if (pn_get_node_type(py) == VALUE_NT) {
    specification *spec = pn_get_evaluation(py);
    if (family_is(spec, STORAGE_FMY)) {
        assertion_problem(_P_(C8Case24b),
            "the name supplied for this new variable is a piece of text "
            "which is not available because it has a rival meaning already",
            "as a result of definitions made elsewhere. (Sometimes these "
            "are indirect: for instance, defining a column in a table "
            "called 'question' can make a name like 'container in question' "
            "suddenly ambiguous and thus unsuitable to be a variable "
            "name.) If you're getting this Problem message in the Standard "
            "Rules or some other extension you need to use, then your "
            "only option is to hunt through your own source text to see "
            "what you have defined which might cause this clash.");
        return;
    }
}
if (pn_get_node_type(py) == INSTANCE_NT) {
    specification *spec = pn_get_evaluation(py);
    if (spec_get_proposition(spec)) {
        specification *xts = pn_get_evaluation(px);
```

```
            if (xts) {
                prop_true_in_world_model_about(spec_get_proposition(spec), NULL, xts);
                return;
            } else if (pn_get_refers_to(px)) {
                prop_true_in_world_model_about(spec_get_proposition(spec), pn_get_refers_to(px),
NULL);
                return;
            }
        }
    }
    if ((pn_get_refers_to(px)) && (pn_get_refers_to(py)))
        equated_descriptions_problem(pn_get_refers_to(px), pn_get_refers_to(py));
    else
        assertion_problem(_P_(C8Case24),
        "this seems to say that a general description is something else",
        "like saying that 'a door is a number'.");
```

This code is used in §4.

§29. Case 25.

⟨MA25 - X and Y are XOFY - P 29⟩ ≡

```
    assertion_problem(_P_(C8Case25),
        "this seems to say two different properties are not simply equal "
        "but somehow the same thing",
        "like saying that 'the printed name of the millpond is the "
        "printed name of the village pond'. This puts me in a quandary: "
        "which should be changed to match the other, and what if I am "
        "unable to work out the value of either one?");
```

This code is used in §4.

§30. Case 26.

⟨MA26 - Y is XOFY - P 30⟩ ≡

```
    assertion_problem(_P_(C8Case26),
        "this is the wrong way around if you want to specify a property",
        "like saying that '10 is the score of the platinum pyramid', "
        "which is poor style. (Though sweet are the uses of adversity.)");
```

This code is used in §4.

§31. Case 27.

The position of the weathervane is east of the church.

⟨MA27 - X is XOFY, Y is RELATIONSHIP - P 31⟩ ≡

```
    assertion_problem(_P_(C8Case27),
        "this seems to say that a property of something is not simply equal "
        "to what happens at the moment to satisfy some relationship, but "
        "conceptually the same as that relationship",
        "like saying 'the position of the weathervane is east of the "
        "church'. It would be fine to say 'the position of the weathervane "
        "is east' or 'the position of the weathervane is the meadow', "
        "because 'east' and 'meadow' are definite things.");
```

This code is used in §4.

§**32. Case 28.** This is a catch-all sort of error. It might need narrowing into sub-cases later.

The description of the Pitch is open.

⟨MA28 - X is XOFY - P 32⟩ ≡

```
assertion_problem(_P_(C8Case28),
    "that is a very peculiar property value",
    "and ought to be something more definite and explicit.");
```

This code is used in §4.

§**33. Case 29.** At last, a correctly set property value. (See also case 54.)

The description of the Pitch is "Verdant." The desk is a container. The carrying capacity of the desk is 10.

⟨MA29 - X is XOFY, Y is NOUNPHRASE or VALUE 33⟩ ≡

```
if (traverse == 1) return;
if [[px->down == player]] pn_make_player_noun_phrase(px->down);
if [[py == not ...]] {
    negative_sentence_problem(_P_(C8Case29e));
    return;
}
if ((pn_get_node_type(px->down) == VALUE_NT) &&
    (px->down->next) &&
    [[px->down->next == specification]]) {
    specification *spec = pn_get_evaluation(px->down);
    if (spec == NULL) internal_error("unevaluated VALUE node");
    int nonliteral_spec = FALSE;
    if (spec_is_generic_CONSTANT(spec)) {
        if (is_kova(is_a_literal(py->word_ref1, py->word_ref2), TEXT_TY)) {
            int t = py->word_ref1;
            dequote_word(t);
            kov_set_specification_text(spec_get_kind_of_value(spec), lw_array[t].lw_text);
            return;
        } else {
            nonliteral_spec = TRUE;
        }
    } else if (spec_is_CONSTANT_of_kova(spec, ACTION_NAME_TY)) {
        if (is_kova(is_a_literal(py->word_ref1, py->word_ref2), TEXT_TY)) {
            action_name *an = ACTION_NAME_spec_to_action_name(spec);
            int t = py->word_ref1;
            dequote_word(t);
            an_set_specification_text(an, t);
            return;
        } else {
            nonliteral_spec = TRUE;
        }
    }
    if (nonliteral_spec) {
        assertion_problem(_P_(C8Case29d),
            "this tries to set a specification to something other "
            "than literal quoted text",
            "which will not work. 'Specification' is a special "
            "property used only to annotate the Index, and specifically "
```

```
            "the Kinds index, so it makes no sense to set this property "
            "to anything other than text.");
        return;
    }
    assertion_problem(_P_(C8Case29c),
        "this tries to set specification text for a particular value",
        "rather than a kind of value. 'Specification' is a special "
        "property used only to annotate the Index, and it makes no sense "
        "to set this property for anything other than a kind of value, "
        "a kind of object or an action to have specification text.");
    return;
}
if ((pn_get_node_type(px->down) == NOUNPHRASE_NT) ||
    (pn_get_node_type(px->down) == INSTANCE_NT)) {
    world_object *wo = pn_get_refers_to(px->down);
    if (wo == NULL) {
        assertion_problem(_P_(C8Case29b),
            "this tries to set a property for something more complicated "
            "than a single thing named without qualifications",
            "and that isn't allowed. For instance, 'The description of "
            "the Great Portal is \"It's open.\"' is fine, but 'The "
            "description of the Great Portal in the Palace is \"It's open.\"' "
            "is not allowed, because it tries to qualify 'Great Portal' "
            "with the extra clause 'in the Palace'. (If you need to make "
            "a description which changes based on where something is, or "
            "where it is seen from, try using text substitutions.)");
        return;
    }
    assert_property_value_from_property_subtree(value_property_name_ref(
        px->down->next->word_ref1, px->down->next->word_ref2),
        wo, NULL, py);
    return;
}
if (pn_get_node_type(px->down) == VALUE_NT) {
    assert_property_value_from_property_subtree(value_property_name_ref(
        px->down->next->word_ref1, px->down->next->word_ref2),
        NULL, pn_get_evaluation(px->down), py);
    return;
}
assertion_problem(_P_(C8Case29),
    "this tries to set a property for a complicated generality of "
    "items all at once",
    "which can lead to ambiguities. For instance, 'The description of "
    "an open door is \"It's open.\"' is not allowed: if we followed "
    "Inform's normal conventions strictly, that would be an instruction "
    "to create a new, nameless, open door and give it the description. "
    "But this is very unlikely to be what the writer intended, given "
    "the presence of the adjective to make it seem as if a particular "
    "door is meant. So in fact we reject such sentences unless they "
    "refer only to a kind, without adjectives: 'The description of a "
    "door is \"It's a door.\"' is fine. (If the idea is actually to "
    "make the description change in play, we could write a rule like "
    "'Instead of examining an open door, say \"It's open.\"'; or we "
```

```
        "could set the description of every door to "
        "\"[if open]It's open.[otherwise]It's closed.\".)");
```

This code is used in §4.

§34. Case 30.

⟨MA30 - X and Y are ACTION - P 34⟩ ≡

```
    if (traverse == 2) {
        action_pattern apx, apy;
        apx = parse_action_pattern(px->word_ref1, px->word_ref2, IS_TENSE);
        apy = parse_action_pattern(py->word_ref1, py->word_ref2, IS_TENSE);
        if ((ap_is_valid(&apy)) && (ap_is_named(&apy) == FALSE)) {
            log_action_pattern(&apx);
            log_action_pattern(&apy);
            assertion_problem(_P_(C8Case30),
                "two actions are rather oddly equated here",
                "which would only make sense if the second were a named "
                "pattern of actions like (say) 'unseemly behaviour'.");
            return;
        }
        categorise_action_as(apx, py->word_ref1, py->word_ref2);
    }
```

This code is used in §4.

§35. Case 31. When we need to create a new named action pattern, as in the sentence

Taking something is theft.

when nothing has previously been referred to as "theft", so that it is still being treated as a noun phrase (and presumed to be an object). What we do converts node Y to `ACTION_NT` type, so if it happens on pass 1, then it will fall into case 30 above on the second pass. But if we get here on pass 2, we had better take action directly, because there isn't going to be a pass 3.

⟨MA31 - X is ACTION, Y is NOUNPHRASE 35⟩ ≡

```
    assertion_problem(_P_(C8Case31),
        "an action can't be the same as a thing",
        "so my guess is that this is an attempt to categorise an action which "
        "went wrong because there was already something of that name in "
        "existence. For instance, 'Taking something is theft' would fail if "
        "'theft' was already a value. (But it can also happen with a sentence "
        "which tries to set several actions at once to a named kind of action, "
        "like 'Taking and dropping are manipulation.' - only one can be named "
        "at a time.)");
```

This code is used in §4.

§36. Case 32. Not as unlikely a mistake as it might seem:

Taking something is open.

⟨MA32 - X is ACTION, Y is ADJECTIVE or PROPERTYNOUN - P 36⟩ ≡

```
    assertion_problem(_P_(C8Case32),
        "that is already the name of a property",
        "so it will only confuse things if we use it for a kind of action.");
```

This code is used in §4.

§**37. Case 33.** Whereas this one is frankly remote.

Taking something is 100. The turn count is taking something.

⟨MA33 - X or Y is ACTION - P 37⟩ ≡
```
assertion_problem(_P_(C8Case33),
    "that means something else already",
    "so it will only confuse things if we use it for a kind of action.");
```

This code is used in §4.

§**38. Case 34.** A problem message issued purely on stylistic grounds, though it would be easy to comply with the likely intended sense of the sentence.

⟨MA34 - X is NOUNPHRASE, Y is ACTION - P 38⟩ ≡
```
assertion_problem(_P_(C8Case34),
    "that is putting the definition back to front",
    "since I need these categorisations of actions to take the form "
    "'Kissing a woman is love', not 'Love is kissing a woman'. (This "
    "is really because it is better style: love might be many other "
    "things too, and we don't want to imply that the present definition "
    "is all-inclusive.)");
```

This code is used in §4.

§**39. Case 35.**

⟨MA35 - X is KIND - P 39⟩ ≡
```
assertion_problem(_P_(C8Case35),
    "that seems to say that a new kind is the same as something else",
    "like saying 'A kind of container is a canister': which ought to "
    "be put the other way round, 'A canister is a kind of container'.");
```

This code is used in §4.

§**40. Case 36.** Equating something to a single adjective.

The desk is fixed in place. A container is usually fixed in place.

⟨MA36 - X is INSTANCE or NOUNPHRASE or VALUE, Y is ADJECTIVE 40⟩ ≡
```
if [[px == player]] {
    pn_make_player_noun_phrase(px);
    /* make_assertion_recursive(px, py, assertion_depth);
    return; */
}
if (traverse == 2) assert_property_list(px, py);
```

This code is used in §4.

§41. Case 37.

⟨MA37 - X is NOUNPHRASE or VALUE, Y is PROPERTYNOUN - P 41⟩ ≡

```
assertion_problem(_P_(C8Case37),
    "that seems to say that some object is a property",
    "like saying 'The brick building is the description': if you want "
    "to specify the description of the current object, try putting "
    "the sentence the other way around ('The description is...').");
```

This code is used in §4.

§42. Case 38. I am in two minds about the next nit-picking error message. But really this is a device used in English only for declamatory purposes or comedic intent. (As in Peter Schickele's spoof example of an 18th-century opera about a dog, *Collared Is Bowser*.)

⟨MA38 - X is ADJECTIVE, Y is NOUNPHRASE - P 42⟩ ≡

```
assertion_problem(_P_(C8Case38),
    "that seems to say that an adjective is a noun",
    "like saying 'Open are the doubled doors': which I'm picky about, "
    "preferring it written the other way about ('The doubled doors are "
    "open'). Less poetic, but clearer style.");
```

This code is used in §4.

§43. Case 39.

⟨MA39 - X, Y are ADJECTIVE, PROPERTYNOUN or PROPERTYLIST - P 43⟩ ≡

```
assertion_problem(_P_(C8Case39),
    "that seems to say that an either/or property is the same as "
    "a property which has a value",
    "like saying 'Open is the description': it makes no grammatical "
    "sense even if we know what object is being talked about.");
```

This code is used in §4.

§44. Case 40.

⟨MA40 - X, Y are PROPERTYLIST or PROPERTYNOUN - P 44⟩ ≡

```
assertion_problem(_P_(C8Case40),
    "that seems to say that two different properties are the same",
    "like saying 'The indefinite article is the printed name': that "
    "might be true for some things, some of the time, but it makes "
    "no sense in a general statement like this one.");
```

This code is used in §4.

§45. Case 41.

⟨MA41 - X is ADJECTIVE, Y is VALUE - P 45⟩ ≡

```
assertion_problem(_P_(C8Case41),
    "that suggests that an adjective has some sort of value",
    "like saying 'Open is a number' or 'Scenery is 5': but of course "
    "an adjective represents something which is either true or false.");
```

This code is used in §4.

§**46. Case 42.** Such implicatory sentences as

Scenery is usually fixed in place.

⟨MA42 - X, Y are ADJECTIVE 46⟩ ≡
```
if (traverse == 2) {
    imp_new(px, py);
}
```

This code is used in §4.

§**47. Case 43.** This is unlikely to be called as a top-level sentence with "is", but will instead occur through recursion from `WITH_NT` nodes, or as part of the handling of "to have" rather than "to be". Anyway, it behaves as if the verb were "has" not "is".

Property lists may contain references to things not yet created, if we assert them during pass 1: so we wait until pass 2.

⟨MA43 - X is NOUNPHRASE, Y is PROPERTYLIST 47⟩ ≡
```
if (traverse == 2) assert_property_list(px, py);
```

This code is used in §4.

§**48. Case 44.** And vice versa. This case can only arise through recursion.

⟨MA44 - X is PROPERTYLIST, Y is NOUNPHRASE 48⟩ ≡
```
if (traverse == 2) assert_property_list(py, px);
```

This code is used in §4.

§**49. Case 45.** Assigning a property list to something makes it tangible. This puts us back into case 43.

⟨MA45 - X is INSTANCE, Y is PROPERTYLIST 49⟩ ≡
```
if (current_sentence->down->next == px) {
    if (traverse == 2) assert_property_list(px, py);
} else {
    convert_instance_to_nounphrase(px, assertion_depth, FALSE);
    make_assertion_recursive(px, py, assertion_depth);
}
```

This code is used in §4.

§**50. Case 46.** And vice versa, but into case 44.

⟨MA46 - X is PROPERTYLIST, Y is INSTANCE 50⟩ ≡
```
if (current_sentence->down->next->next == py) {
    if (traverse == 2) assert_property_list(py, px);
} else {
    convert_instance_to_nounphrase(py, assertion_depth, FALSE);
    make_assertion_recursive(px, py, assertion_depth);
}
```

This code is used in §4.

§**51. Case 47.** This one used to be invalid, but became an active case in August 2008 when value properties for other values were put on a par with those for objects.

⟨MA47 - X, Y are PROPERTYLIST, VALUE 51⟩ ≡

```
if ((pn_get_node_type(px) == VALUE_NT) &&
    (spec_is_CONSTANT_of_kova(pn_get_evaluation(px), RULEBOOK_TY))) {
    if (traverse == 2)
        rb_parse_properties(
            RULEBOOK_spec_to_rulebook(pn_get_evaluation(px)),
            left_edge_of(py), right_edge_of(py));
    return;
}
if (traverse == 2) assert_property_list(px, py);
```

This code is used in §4.

§**52. Case 48.** "In A is a B with I": process as "In A is a B" followed by "the newly created B is I".

```
In the Pitch is a container with description "Made of wood."
```

⟨MA48 - X is RELATIONSHIP, Y is WITH 52⟩ ≡

```
make_assertion_recursive(px, py->down, assertion_depth);
make_assertion_recursive(py->down, py->down->next, assertion_depth);
```

This code is used in §4.

§**53. Case 49.** "An A with I is in B": similar.

⟨MA49 - X is WITH, Y is RELATIONSHIP 53⟩ ≡

```
make_assertion_recursive(px->down, py, assertion_depth);
make_assertion_recursive(px->down, px->down->next, assertion_depth);
```

This code is used in §4.

§**54. Case 50.** "X is every K" and other oddities.

⟨MA50 - Y is EVERY 54⟩ ≡

```
if [[py == everywhere]] {
    pn_set_node_type(py, INSTANCE_NT);
    pn_set_refers_to(py, kind_room);
    make_assertion_recursive(px, py, assertion_depth);
    return;
}
assertion_problem(_P_(C8Case50),
    "'every' can't be used in that way",
    "and should be reserved for sentences like 'A coin is in every room'.");
```

This code is used in §4.

§**55. Case 51.** "Every K is Y" and other oddities.

⟨MA51 - X is EVERY - P 55⟩ ≡
```
assertion_problem(_P_(C8Case51),
    "'every' can't be used in that way",
    "and should be reserved for sentences like 'A coin is in every room'.");
```

This code is used in §4.

§**56. Case 52.** "Every K is every L."

⟨MA52 - X, Y are EVERY - P 56⟩ ≡
```
assertion_problem(_P_(C8Case52),
    "I can't do that", "Dave.");
```

This code is used in §4.

§**57. Case 53.** "Every K is in L."

⟨MA53 - X is EVERY, Y is RELATIONSHIP 57⟩ ≡
```
make_generalisation(px, py);
```

This code is used in §4.

§**58. Case 54.** "The colour of the paint is white."

⟨MA54 - X is XOFY, Y is ADJECTIVE 58⟩ ≡
```
if (traverse == 1) return;
if ((pn_get_property(py)) && (pn_get_evaluation(py))) {
    if [[px->down == player]]
        pn_make_player_noun_phrase(px->down);
    if (pn_get_refers_to(px->down))
        assert_property_value_from_property_subtree(value_property_name_ref(
            px->down->next->word_ref1, px->down->next->word_ref2),
            pn_get_refers_to(px->down), NULL, py);
    else
        assert_property_value_from_property_subtree(value_property_name_ref(
            px->down->next->word_ref1, px->down->next->word_ref2),
            NULL, pn_get_evaluation(px->down), py);
} else {
    assertion_problem(_P_(C8Case54),
        "that property can't be used adjectivally as a value",
        "since it is an adjective applying to a thing but is "
        "not a name from a range of possibilities.");
}
```

This code is used in §4.

§**59. Case 55.** To do with value relations.

⟨MA55 - X is RELATIONSHIP, Y is VALUE 59⟩ ≡

```
if [[py == player]] {
    pn_make_player_noun_phrase(py);
    make_assertion_recursive(px, py, assertion_depth);
    return;
}
if ((pn_get_relationship(px)) &&
    (bp_relates_values_not_objects(pn_get_relationship(px))))
    assert_subtree_in_relationship(py, px);
else {
assertion_problem(_P_(C8Case55),
    "this seems to give a worldly relationship to something intangible",
    "possibly due to an accidental clash of names between a kind of "
    "value and something in the real world. "
    "I sometimes read sentences like 'There is a number on the "
    "door' or 'A text is in the prayer-box' literally - thinking "
    "you mean a whole number or a piece of double-quoted text, and "
    "not realising you intended to make a brass number-plate or "
    "an old book. If that's the trouble, you can use 'called': "
    "for instance, 'In the prayer-box is a thing called the text.'");
}
```

This code is used in §4.

§**60. Case 56.** To do with value relations.

⟨MA56 - X is VALUE, Y is RELATIONSHIP 60⟩ ≡

```
if [[px == player]] {
    pn_make_player_noun_phrase(px);
    make_assertion_recursive(px, py, assertion_depth);
    return;
}
if ((pn_get_relationship(py)) &&
    (bp_relates_values_not_objects(pn_get_relationship(py))))
    assert_subtree_in_relationship(px, py);
else {
assertion_problem(_P_(C8Case56),
    "this seems to give a worldly relationship to something intangible",
    "possibly due to an accidental clash of names between a kind of "
    "value and something in the real world. "
    "I sometimes read sentences like 'There is a number on the "
    "door' or 'A text is in the prayer-box' literally - thinking "
    "you mean a whole number or a piece of double-quoted text, and "
    "not realising you intended to make a brass number-plate or "
    "an old book. If that's the trouble, you can use 'called': "
    "for instance, 'In the prayer-box is a thing called the text.'");
}
```

This code is used in §4.

§**61. Case 57.** To catch confusions of action names with values.

⟨MA57 - X is VALUE, Y is ACTION 61⟩ ≡

```
quote_source(1, current_sentence);
quote_source(2, py);
handmade_problem(_P_(C8Case57));
issue_problem_segment(
    "You wrote %1: unfortunately %2 is already the name of an action, "
    "and it would only confuse things if we used it for a value as well.");
issue_problem_end();
```

This code is used in §4.

§**62. Adjective list trees.**

```
int is_adjlist(parse_node *p) {
    if (p == NULL) return FALSE;
    switch (pn_get_node_type(p)) {
        case ADJECTIVE_NT: return TRUE;
        case AND_NT: return ((is_adjlist(p->down)) && (is_adjlist(p->down->next)));
        default: return FALSE;
    }
}
```

§**63. Equating values.** This is never allowed, but there are a variety of possible problem messages, because it usually occurs as a symptom of a failed attempt to create something.

```
void issue_value_equation_problem(parse_node *px, parse_node *py) {
    if ((pn_get_node_type(px) != NOUNPHRASE_NT) || (pn_get_node_type(py) != VALUE_NT)) {
        log_subtree(px, 1); log_subtree(py, 1);
        internal_error("Assert PX of type PY on bad node types");
    }
    log_subtree(px,1); log_subtree(py,1);
    if ((spec_is_actual_CONSTANT(pn_get_evaluation(py)) == FALSE) &&
        (is_kova(spec_get_kind_of_value(pn_get_evaluation(py)), OBJECT_TY))) {
        sentence_problem(_P_(C8NoObjects),
            "that's not something which you can create new values of",
            "because you need to be more specific about what kind of "
            "object is intended. Thus 'The frog is an animal' or "
            "'The Hallway is a room' are fine, but not 'The cabinet "
            "is an object' or 'There is an object called the clock'. "
            "(If in doubt, call it a 'thing' instead of an 'object'.)");
        return;
    }
    if ((current_sentence) &&
        (is_word_intermediate_inc(something_V,
        current_sentence->word_ref1, current_sentence->word_ref2) >= 0)) {
        sentence_problem(_P_(C8EquatesSomethingToValue),
            "that seems to say that an object is the same as a value",
            "which must be wrong. This can happen if the word 'something' is "
            "used loosely - I read it as 'some thing', so I think it has to "
            "refer to a thing, which is a kind of object. A sentence like "
```

```
            "'Something called mauve is a colour' trips me up because mauve "
            "is a value, so it isn't an object, and doesn't match 'something'.");
        return;
    }
    if ((pn_get_refers_to(px)) &&
        (pn_get_refers_to(px)->creating_sentence != current_sentence)) {
        quote_words(1, px->word_ref1, px->word_ref2);
        quote_source(2, current_sentence);
        quote_source(3, pn_get_refers_to(px)->creating_sentence);
        handmade_problem(_P_(C8CantUncreate));
        issue_problem_segment(
            "In order to act on %2, I seem to need to give "
            "a new meaning to '%1', something which was created by the earlier "
            "sentence %3. That must be wrong somehow: I'm guessing that there "
            "is an accidental clash of names. This sometimes happens when "
            "adjectives are being made after objects whose names include them: "
            "for instance, defining 'big' as an adjective after having already "
            "made a 'big top'. The simplest way to avoid this is to define "
            "the adjectives in question first.");
        issue_problem_end();
        return;
    }
    sentence_problem(_P_(BelievedImpossible),
        "that seems to say that an object is the same as a value",
        "which must be wrong.");
}
```

§**64. To Be and To Have.** Well, that ought to be the name of an incomprehensible book by Sartre which dismisses Heidegger's seminal *To Have and To Be*, or something like that, but instead it is the name of a section which contains the two wrapper routines for the main English assertion verbs.

```
sentence_handler ASSERT_SH_handler = { SENTENCE_NT, ASSERT_VB, 0, to_be };
sentence_handler HAS_SH_handler = { SENTENCE_NT, HAS_VB, 0, to_have };
void to_be(parse_node *pv) {
    parse_node *px = pv->down->next;
    parse_node *py = pv->down->next->next;
    if ((px->word_ref1 >= 0) && (px->word_ref2 > px->word_ref1)
        && (vocab_test_flags(px->word_ref1, TEXT_MC))) {
        sentence_problem(_P_(C8TextNotClosing),
            "it looks as if perhaps you did not intend that to read as a "
            "single sentence",
            "and possibly the text in quotes was supposed to stand as "
            "as a sentence on its own? (The convention is that if text "
            "ends in a full stop, exclamation or question mark, perhaps "
            "with a close bracket or quotation mark involved as well, then "
            "that punctuation mark also closes the sentence to which the "
            "text belongs: but otherwise the words following the quoted "
            "text are considered part of the same sentence.)");
        return;
    }
    make_assertion(px, py);
}
```

§**65.** Note that `to_have` converts its sentence subtree to a case of "to be", so that it is only called on traverse 1: by the time of traverse 2, all "have" sentences have been converted. Generally speaking we construe the Y in "X has Y" as a property list, but with certain exceptions:

X has an A called B → X is allowed to have an A called B

X has A → X is A

X has Q → X is allowed to have Q

The last of these handles sentences like "X has a colour", where "colour" is a kind of value which will be allowed to coincide with the name of a property. ("Is allowed to have..." is pseudo-code here: this is no longer syntax which I7 recognises directly, although it used to be in early drafts of NI. What really happens is that we manufacture a suitable `ALLOWED_NT` structure.)

```
void to_have(parse_node *pv) {
    int make_pl = TRUE;
    parse_node *px = pv->down->next;
    parse_node *py = pv->down->next->next;
    if (pn_get_node_type(py) == X_OF_Y_NT) {
        LOG("Sentence subtree:\n$T\n", current_sentence);
        sentence_problem(_P_(C8SuperfluousOf),
            "the 'of' here appears superfluous",
            "assuming the sentence aims to give a property value of something. "
            "(For instance, if we want to declare the carrying capacity of "
            "something, the normal Inform practice is to say 'The box has "
            "carrying capacity 10' rather than 'The box has a carrying capacity "
            "of 10'.)");
        return;
    }
    if (pn_get_node_type(py) == WITH_NT) {
        LOG("Sentence subtree:\n$T\n", current_sentence);
        sentence_problem(_P_(C8SuperfluousWith),
            "the 'has ... with' here appears to be a mixture of two ways to "
            "give something properties",
            "that is, 'The box is a container with capacity 10.' and 'The box "
            "has capacity 10.'");
        return;
    }
    if (pn_get_node_type(py) == CALLED_NT) {
        pn_set_node_type(py, PROPERTYCALLED_NT);
        px->next = new_node(ALLOWED_NT);
        px->next->down = py;
        py = px->next;
        make_pl = FALSE;
    } else {
        if ((py->word_ref1 >= 0) && (py->word_ref2 >= py->word_ref1)) {
            if (qty_parse(py->word_ref1, py->word_ref2)) make_pl = FALSE;
            else {
                if (SP_excerpt(DESIGNED_TYPE_MC, py->word_ref1, py->word_ref2)) {
                    px->next = new_node(ALLOWED_NT);
                    px->next->down = py;
                    py = px->next;
                    make_pl = FALSE;
```

```
                }
            }
        }
    }
    if (make_pl) pn_set_node_type(py, PROPERTYLIST_NT);
    pn_annotate_int(pv->down, verb_id_ANNOT, ASSERT_VB);
    to_be(pv);
}
```

Assemblies 8/assem

Purpose

To build the complex multi-object assemblies which result from allowing the source text to say things like "in every room is a vehicle".

Definitions

¶1. Generalisations are essentially fragments of parse tree stored for later use. They handle sentences like

> In every container is a coin.

which are done by recognising the prototype part ("in every container") in the parse tree and grafting on a duplicate of the assembly part ("a coin") in place of the EVERY subtree ("every container"). Sometimes the EVERY subtree is the whole prototype subtree ("Every coin is on a table"), in which case `px` and `substitute_at` in the following structure coincide.

Each kind (in this example "container") keeps a linked list of the generalisations which apply to it.

```
typedef struct generalisation {
    struct parse_node *px;                          prototype situation to look for
    struct parse_node *py;                          subtree for what to assemble
    struct parse_node *substitute_at;               position under px of the EVERY node
    struct parse_node copied;                       a deep copy of py
    struct generalisation *next;                    next in list of generalisations about kind
    MEMORY_MANAGEMENT
} generalisation;
```

The structure generalisation is private to this section.

---

§**1.** Assemblies are made when an object of a given kind is created, and when generalisations about that kind mean that further creations are also needed. For instance: if a generalisation has said that every container contains a shoe, then each time a container is created, we also need to create a shoe, and assert a spatial relationship between them. In practice we do this by a simple process which involves cutting and pasting of subtrees of the parse tree, for which the following utility is helpful:

```
void deep_copy_tree(parse_node *from, parse_node *to, int level) {
    if ((from == NULL) || (to == NULL)) internal_error("Null deep copy");
    pn_copy(to, from);
    if (from->down) {
        to->down = new_node(NOUNPHRASE_NT);
        deep_copy_tree(from->down, to->down, level+1);
    }
    if ((level>0) && (from->next)) {
        to->next = new_node(NOUNPHRASE_NT);
        deep_copy_tree(from->next, to->next, level);
    }
}
```

§**2.** Now the actual assembly code.

```
int contains_instance_of_k(parse_node *chunk, world_object *k, int level) {
    if ((pn_get_node_type(chunk) == INSTANCE_NT) && (pn_get_refers_to(chunk) == k))
        return TRUE;
    if ((chunk->down) && (contains_instance_of_k(chunk->down, k, level+1)))
        return TRUE;
    if ((level>0) && (chunk->next) && (contains_instance_of_k(chunk->next, k, level)))
        return TRUE;
    return FALSE;
}

int implicit_recursion_exception = FALSE;
void satisfies_generalisation(world_object *wo, generalisation *g, world_object *k) {
    parse_node *csn = current_sentence->next;
    if (wo->kind_flag) return;
    if (pn_int_annotation(current_sentence, implicitness_count_ANNOT) >= 500) {
        if (implicit_recursion_exception) return;
        implicit_recursion_exception = TRUE;
        object_problem(_P_(C8AssemblyLoop),
            k, "seems to be associated with an assembly which can never end",
            "or which at any rate led to some 500 further constructions "
            "before I panicked. This problem tends to occur if instructions "
            "are given which cause kinds to create each other forever: "
            "for instance, 'Every device is on a supporter. Every supporter "
            "is in a container. Every container is part of a device.'");
        return;
    }
    current_sentence->next = new_node(SENTENCE_NT);
    current_sentence->next->next = csn;
    csn = current_sentence->next;
    pn_set_implicit_in_creation_of(csn, wo);
    pn_annotate_int(csn, implicitness_count_ANNOT,
        pn_int_annotation(current_sentence, implicitness_count_ANNOT) + 1);
    csn->word_ref1 = current_sentence->word_ref1;
    csn->word_ref2 = current_sentence->word_ref2;
    csn->down = new_node(VERB_NT);
    pn_annotate_int(csn->down, verb_id_ANNOT, ASSERT_VB);
    pn_set_node_type(g->substitute_at, NOUNPHRASE_NT);
    pn_set_refers_to(g->substitute_at, wo);
    g->substitute_at->word_ref1 = -1;
    g->substitute_at->word_ref2 = -1;
    LOGIF(ASSEMBLIES, "Object $O satisfies generalisation (from $O):\n$P$P",
        wo, k, g->px, g->py);
    csn->down->next = new_node(NOUNPHRASE_NT);
    deep_copy_tree(g->px, csn->down->next, 0);
    csn->down->next->next = new_node(NOUNPHRASE_NT);
    deep_copy_tree(g->py, csn->down->next->next, 0);
    csn->down->next->next->next = NULL;
    pn_set_node_type(g->substitute_at, EVERY_NT);
    pn_set_refers_to(g->substitute_at, k);
    LOGIF(ASSEMBLIES, "Result of the generalisation after deep copy:\n$P$P",
        g->px, g->py);
}
```

```
void make_generalisation(parse_node *px, parse_node *py) {
    world_object *k = NULL, *wo;
    parse_node *pz = NULL;
    specification *val;
    generalisation *g;
    if (traverse == 2) return;
    if (pn_get_node_type(px) == EVERY_NT) { k = pn_get_refers_to(px); pz = px; }
    else {
        if ((px->down) && (pn_get_node_type(px->down) == EVERY_NT)) {
            k = pn_get_refers_to(px->down); pz = px->down;
        }
    }
    if (pz == NULL) {
        LOG("Generalisation:\n");
        log_subtree(px, 1);
        log_subtree(py, 1);
        internal_error("Generalisation without EVERY node");
    }
    if ((contains_instance_of_k(px,k,0)) || (contains_instance_of_k(py,k,0))) {
        LOG("Generalisation:\n");
        log_subtree(px, 1);
        log_subtree(py, 1);
        sentence_problem(_P_(C8AssemblyRegress),
            "this generalisation would be too dangerous",
            "because it would lead to infinite regress in the assembly "
            "process. Sometimes this happens if you have set up matters with "
            "text like 'A container is in every container.'.");
        return;
    }
    g = CREATE(generalisation);
    g->px = px;
    val = pn_get_evaluation(py);
    if ((val) && (spec_get_described_kind(val)) && (number_of_adjectives_applied_to(val) > 0))
        grow_as_NP_with_adjectives(py, spec_copy(val));
    g->py = py;
    g->substitute_at = pz;
    g->next = k->generalisation_list;
    g->copied = *py;
    k->generalisation_list = g;
    if (dl_this(ASSEMBLIES_DA)) {
        LOG("New generalisation made concerning $O:\n", k);
        log_subtree(g->px, 1);
        log_subtree(g->py, 1);
    }
    LOOP_OVER(wo, world_object)
        if (wo_of_kind(wo, k))
            satisfies_generalisation(wo, g, k);
}
void satisfies_generalisations_about(world_object *wo, world_object *k) {
    generalisation *g = k->generalisation_list;
    if (wo->kind_flag) return;
    k->currently_making_instance = TRUE;
```

```
    for (; g; g=g->next) satisfies_generalisation(wo, g, k);
    k->currently_making_instance = FALSE;
}
```

The function make_generalisation is called from 8/mass.
The function satisfies_generalisations_about is called from 9/kind.

Character Knowledge 8/knowc

Purpose

Character in the sense of the nature of things – both the kind of an object, but also more generally that it satisfies some proposition.

§1. Asserting kinds.

```
void assert_kind_of_object(world_object *wo, world_object *k) {
    prop_true_in_world_model_about(prop_to_set_kind(wo, k), wo, NULL);
}
```

The function assert_kind_of_object is called from 8/creat, 8/mass, 9/spabp and 9/model.

§2. Inferring from INSTANCE nodes.

```
void assert_instance(parse_node *px, parse_node *py) {
    if ((pn_get_node_type(px) != NOUNPHRASE_NT) && (pn_get_node_type(px) != INSTANCE_NT))
        internal_error_on_node_type(px);
    if ((pn_get_node_type(py) != ADJECTIVE_NT) && (pn_get_node_type(py) != INSTANCE_NT))
        internal_error_on_node_type(py);
    world_object *wo = pn_get_refers_to(px);
    if (pn_get_refers_to(py) != NULL) {
        world_object *k = pn_get_refers_to(py);
        if (wo == k) { equated_descriptions_problem(k, k); return; }
        if ((current_sentence != wo->creating_sentence) &&
            (wo_of_kind(wo, kind_region) == FALSE) && (k == kind_region)) {
            object_problem_at_sentence(_P_(C8ExistingRegion),
                wo,
                "(which I notice in another sentence) seems now to "
                "be declared as a new region",
                "which is suspect. Perhaps I have misinterpreted what was "
                "meant to be a new name for an old one?");
            return;
        }
        if ((wo) && (wo->kind_flag) && (k) && (k->kind_flag)) {
            if (((wo == kind_backdrop) || (wo_of_kind(wo, kind_backdrop))) &&
                (is_word_intermediate_inc(everywhere_V,
                    current_sentence->word_ref1, current_sentence->word_ref2))) {
                object_problem_at_sentence(_P_(C8KindOfBackdropEverywhere),
                    wo,
                    "seems to be said to be 'everywhere' in some way",
                    "which doesn't make sense. An individual backdrop can be 'everywhere',
"
                    "but here we're talking about a whole kind, and it's not allowed "
                    "to talk about general locations of a whole kind of things at once.");
                return;
            } else equated_descriptions_problem(wo, k); return;
        }
```

```
    }
    pcalc_prop *prop = prop_from_spec(pn_get_evaluation(py), FALSE);
    if ((vars_number_free(prop) == 0) && (wo)) {
        object_problem_at_sentence(_P_(C8SubjectNotFree),
            wo,
            "seems to be set equal to something in a complicated relationship "
            "with something else again",
            "which is too much for me. Perhaps you're trying to do two things "
            "at once, and it would be clearer to write it as two sentences?");
        return;
    }
    prop_true_in_world_model_about(prop, wo, NULL);
}
```

The function assert_instance is called from 8/mass.

§3.

define NAME_DESCRIPTION_CLASH_NOTE

```
    "Sometimes this happens because I've read too much into a "
    "name - for instance, 'A dark room is a room' makes me read "
    "'dark room' as 'dark' (an adjective I know) plus 'room', "
    "but maybe the writer actually meant a photographer's workshop. "
    "If you need to call something by a name which confuses me, one "
    "way is to use 'called': for instance, 'West is a room called "
    "the dark room.' Another way is to call it something else here, "
    "and set the 'printed name' property to what you want the player "
    "to see - for instance, 'The photo lab is a room. The printed name "
    "of the photo lab is \"dark room\".'"

void equated_descriptions_problem(world_object *X, world_object *Y) {
    quote_source(1, current_sentence);
    quote_object(2, X);
    quote_object(3, Y);
    if (X == Y) {
        handmade_problem(_P_(C8SameKindEquated));
        issue_problem_segment(
            "The sentence %1 seems to be telling me that two descriptions, "
            "both forms of %2, are the same. That's a little puzzling - "
            "like saying that 'An open container is a container.' %P"
            NAME_DESCRIPTION_CLASH_NOTE);
        issue_problem_end();
    } else {
        handmade_problem(_P_(C8DescriptionsEquated));
        issue_problem_segment(
            "The sentence %1 seems to be telling me that two descriptions, "
            "one a form of %2 and the other of %3, are the same. That's a "
            "little puzzling - like saying that 'An open door is a container.' %P"
            NAME_DESCRIPTION_CLASH_NOTE);
        issue_problem_end();
    }
}
```

The function equated_descriptions_problem is called from 8/mass.

Property Knowledge 8/knowp

Purpose

This section draws inferences from assertions which seem to be about the properties of things, independent of their location.

8/knowp.§1 One way to assert global variable values; §2-4 Three ways to assert properties

---

§**1. One way to assert global variable values.** A global like "score" is in effect a property belonging to no single object, and which therefore exists only once.

```
void initialise_global_variable(quantity *q, specification *val,
    parse_node *where_from) {
    kind_of_value *variable_kov = qty_kind_of_value(q);
    kind_of_value *constant_kov = spec_evaluates_to(val);
    if (can_we_cast_kovs(constant_kov, variable_kov) != ALWAYS_MATCH) {
        specification *dummy1 = new_generic_CONSTANT_type(variable_kov),
            *dummy2 = new_generic_CONSTANT_type(constant_kov);
        quote_source(1, current_sentence);
        quote_words(2, q->word_ref1, q->word_ref2);
        if (where_from) quote_words(3, where_from->word_ref1, where_from->word_ref2);
        else quote_spec(3, val);
        quote_spec(4, dummy1);
        quote_spec(5, dummy2);
        if ((kovko_get_kind(variable_kov)) && (spec_is_nothing_object_constant(val))) {
            handmade_problem(_P_(C8QuantityKindNothing));
            issue_problem_segment(
                "The sentence %1 tells me that '%2', which should be %4 "
                "that varies, is to have the initial value 'nothing'. "
                "This is allowed as an 'object which varies', but the "
                "rules are stricter for %4.");
            issue_problem_end();
        } else {
            handmade_problem(_P_(C8GlobalKindWrong));
            issue_problem_segment(
                "The sentence %1 tells me that '%2', which is %4 that varies, "
                "should start out with the value '%3', but this is %5 and not %4.");
            issue_problem_end();
        }
        return;
    }
    specification *var = new_actual_NONLOCAL_VARIABLE_type(q);
    int spm = prevailing_mood;
    if (prevailing_mood == UNKNOWN_CE) prevailing_mood = CERTAIN_CE;
    prevailing_mood = spm;
    prop_true_in_world_model(prop_to_set_relation(a_is_b_predicate, NULL, var, NULL, val));
}
```

The function initialise_global_variable is called from 8/mass, 10/tab and 10/bib.

§**2. Three ways to assert properties.** In these three alternative routines, we can assert that a given owner – specified either as an object, a value or a subtree – should have

(a) a given single property equal to a value given as a subtree,
(b) a given single property equal to an explicit value, or
(c) a whole list of properties and their values.

```
void assert_property_value_from_property_subtree(property_name *prn,
    world_object *owner_wo, specification *owner_spec, parse_node *val_subtree) {
    if (property_owner_acceptable(owner_wo, owner_spec, NULL))
        prop_true_in_world_model_about(prop_from_property_subtree(prn, val_subtree),
            owner_wo, owner_spec);
}

void assert_property_value_explicitly(property_name *prn,
    world_object *owner_wo, specification *owner_spec, specification *val) {
    if (property_owner_acceptable(owner_wo, owner_spec, NULL))
        prop_true_in_world_model_about(prop_to_set_property(prn, val),
            owner_wo, owner_spec);
}

void assert_property_list(parse_node *owner_subtree, parse_node *list_subtree) {
    world_object *owner_wo = pn_get_refers_to(owner_subtree);
    specification *owner_spec = pn_get_evaluation(owner_subtree);
    kind_of_value *kov = property_owner_acceptable(owner_wo, owner_spec, owner_subtree);
    if (owner_wo) owner_spec = NULL;
    if (kov)
        prop_true_in_world_model_about(prop_from_property_list(list_subtree, kov),
            owner_wo, owner_spec);
}
```

The function assert_property_value_from_property_subtree is called from 8/mass, 9/pp and 10/tab.
The function assert_property_value_explicitly is called from 9/model.
The function assert_property_list is called from 6/treec, 8/creat and 8/mass.

§**3.**

```
kind_of_value *property_owner_acceptable(world_object *owner_wo, specification *owner_spec,
    parse_node *owner_subtree) {
    kind_of_value *kov = NULL;
    if (species_is(owner_spec, DESCRIPTION_SPC)) {
        if ((number_of_adjectives_applied_to(owner_spec) > 0) ||
            (prop_contains_adjective(spec_get_proposition(owner_spec)))) {
            quote_source(1, current_sentence);
            quote_source(2, owner_subtree);
            handmade_problem(_P_(C8AdjectiveAPL));
            issue_problem_segment(
                "The sentence %1 looked to me as if it might be trying "
                "to assign certain properties to something described "
                "in a way (%2) which involved an adjective which can't be "
                "properly understood until the game is running, so "
                "that at this point it's impossible to act upon.");
            issue_problem_end();
            return NULL;
        }
        if (prop_contains_binary_predicate(spec_get_proposition(owner_spec))) {
```

```
            quote_source(1, current_sentence);
            quote_source(2, owner_subtree);
            handmade_problem(_P_(C8RelationAPL));
            issue_problem_segment(
                "The sentence %1 looked to me as if it might be trying "
                "to assign certain properties to something described "
                "in a way (%2) which involved a clause which can't be "
                "properly determined until the game is running, so "
                "that at this point it's impossible to act upon.");
            issue_problem_end();
            return NULL;
        }
    }
    if (owner_subtree) {
        if ((spec_is_actual_CONSTANT(owner_spec) || spec_is_generic_CONSTANT(owner_spec))) {
            owner_wo = NULL;
        } else if (owner_wo) {
            kov = kova(OBJECT_TY); owner_spec = NULL;
        } else { owner_spec = NULL; owner_wo = NULL; }
    }
    if (owner_spec) kov = spec_get_kind_of_value(owner_spec);
    else if (owner_wo) kov = kova(OBJECT_TY);
    else issue_bad_owner_problem();
    return kov;
}
void issue_bad_owner_problem(void) {
    quote_source(1, current_sentence);
    handmade_problem(_P_(C8BadPropertyOwner));
    issue_problem_segment(
        "The sentence %1 looked to me as if it might be trying "
        "to assign certain properties to something "
        "which is not allowed to have properties at all.");
    issue_problem_end();
}
```

The function issue_bad_owner_problem is called from 9/inf.

§4. The following assumes that the subtree `py` describes a value which the `prn` property of something will have; it issues problem messages if this would be impossible, returning `NULL`, or else silently returns the value. It's used both above and by the tree-conversion code in the predicate calculus engine.

```
specification *property_value_from_property_subtree(property_name *prn, parse_node *py) {
    specification *val = NULL;
    int type_mismatch = FALSE;
    if (prn_is_either_or(prn)) {
        quote_source(1, current_sentence);
        quote_words(2, py->word_ref1, py->word_ref2);
        quote_property(3, prn);
        handmade_problem(_P_(BelievedImpossible));
        issue_problem_segment(
            "In %1 you give a value of the %3 property as '%2', but %3 is an "
            "either/or property - something is either %3 or not, so there is "
```

```
            "no value involved. For instance, something can be 'opaque' or "
            "not, but it can't be 'printed name' or not; whereas it can be "
            "'with printed name \"marble\"' but it can't be 'with opaque'. "
            "'Opaque' is an either/or property in Inform, and 'printed name' "
            "is a value property.");
        issue_problem_end();
        return NULL;
    }
    switch(pn_get_node_type(py)) {
        case ADJECTIVE_NT:
        case VALUE_NT:
            if ((prn_is_value_property(prn)) &&
                (can_we_cast_kovs(spec_evaluates_to(pn_get_evaluation(py)),
                prn_get_kind_of_value(prn)) != ALWAYS_MATCH)) type_mismatch = TRUE;
            else {
                val = pn_get_evaluation(py);
                if ((val) && (spec_is_generic(val))) {
                    specification *spec_expected =
                        new_actual_CONSTANT_spec(prn_get_kind_of_value(prn));
                    quote_source(1, current_sentence);
                    quote_words(2, py->word_ref1, py->word_ref2);
                    quote_type_of(3, spec_expected);
                    quote_property(4, prn);
                    quote_type_of(5, val);
                    handmade_problem(_P_(C8PropertyObj2));
                    issue_problem_segment(
                        "In %1 you give a value of the %4 property as '%2', but it "
                        "seems to be a general description of values (%5) rather than "
                        "nailing down exactly what the value is.");
                    issue_problem_end();
                    return NULL;
                }
            }
            break;
        case NOUNPHRASE_NT:
            if (pn_get_refers_to(py)) val = world_object_to_OBJECT_spec(pn_get_refers_to(py));
            else val = pn_get_evaluation(py);
            if (can_we_cast_kovs(prn_get_kind_of_value(prn), kova(OBJECT_TY)) != ALWAYS_MATCH)
{
                specification *spec_expected =
                    new_actual_CONSTANT_spec(prn_get_kind_of_value(prn));
                quote_source(1, current_sentence);
                quote_words(2, py->word_ref1, py->word_ref2);
                quote_type_of(3, spec_expected);
                quote_property(4, prn);
                handmade_problem(_P_(C8PropertyObj));
                issue_problem_segment(
                    "In %1 you give a value of the %4 property as '%2', but I can "
                    "only suppose that must be the name of a room or thing, since "
                    "it does not seem to be %3 - which this property calls for.");
                issue_problem_end();
                return NULL;
            }
```

```
        break;
    case X_OF_Y_NT:
        sentence_problem(_P_(BelievedImpossible),
            "something grammatically odd has happened here",
            "possibly to do with the unexpected 'of' in what seems to "
            "be a list of property values?");
        return NULL;
    case INSTANCE_NT:
        LOG("Bad property value is at: $T\nValuation: $S\n", py,
            pn_get_evaluation(py));
        sentence_problem(_P_(C8PropertyInstance),
            "this property value makes no sense to me",
            "since it looks as if it contains a relative clause. Sometimes "
            "this happens if a clause follows directly on, and I have "
            "misunderstood to what it belongs. For instance: 'The widget "
            "is a door with matching key a thing in the Ballroom' is "
            "read with 'a thing in the Ballroom' as the value of the "
            "'matching key' property, not with 'in the Ballroom' applying "
            "to the door.");
        return NULL;
    case RELATIONSHIP_NT:
        sentence_problem(_P_(BelievedImpossible),
            "this property value makes no sense to me",
            "since it looks as if it contains a relative clause. Sometimes "
            "this happens if a clause follows directly on, and I have "
            "misunderstood to what it belongs. For instance, 'Sleeping "
            "Beauty is a woman who is asleep in the Tower' is too complex "
            "an assertion for me, even if I can understand 'asleep' and "
            "'in the Tower' where they occur in simpler sentences.");
        return NULL;
    default:
        log_subtree(current_sentence, 1);
        internal_error_on_node_type(py);
}
if (type_mismatch) {
    specification *spec_expected =
        new_actual_CONSTANT_spec(prn_get_kind_of_value(prn));
    quote_source(1, current_sentence);
    quote_words(2, py->word_ref1, py->word_ref2);
    quote_type_of(3, spec_expected);
    quote_property(4, prn);
    quote_type_of(5, pn_get_evaluation(py));
    handmade_problem(_P_(C8PropertyType));
    issue_problem_segment(
        "In %1 you give a value of the %4 property as '%2', "
        "but this ought to be %3, not %5.");
    issue_problem_end();
    return NULL;
}
if ((val) && (spec_is_evaluating(val)) && (species_is(val, CONSTANT_SPC) == FALSE)) {
    specification *spec_expected =
        new_actual_CONSTANT_spec(prn_get_kind_of_value(prn));
    quote_source(1, current_sentence);
```

```
        quote_words(2, py->word_ref1, py->word_ref2);
        quote_type_of(3, spec_expected);
        quote_property(4, prn);
        handmade_problem(_P_(C8PropertyNonConstant));
        issue_problem_segment(
            "In %1 you give a value of the %4 property as '%2', "
            "and while this does make sense as %3, it is a value which "
            "exists and changes during play, and which doesn't make sense to "
            "use when setting up the initial state of things.");
        issue_problem_end();
        return NULL;
    }
    return val;
}
```

The function property_value_from_property_subtree is called from 6/treec.

Relation Knowledge 8/relv

Purpose

This section draws inferences about the relationships between objects or values.

§**1. Relationship nodes.** Here we have a relationship between subtrees T_X and T_Y, where T_X must be a list of values or objects (joined into an `AND_NT` tree), and T_Y must be a `RELATIONSHIP_NT` subtree – which is usually a node annotated with the predicate meant, and beneath that another list of objects or values, but there are two exceptional cases to take care of.

```
void assert_subtree_in_relationship(parse_node *px, parse_node *relationship_subtree) {
    if ((px == NULL) || (relationship_subtree == NULL))
        internal_error("assert relation between null subtrees");
    if (pn_get_node_type(relationship_subtree) != RELATIONSHIP_NT)
        internal_error("asserted malformed relationship subtree");
    if (pn_get_node_type(px) == AND_NT) {
        assert_subtree_in_relationship(px->down, relationship_subtree);
        assert_subtree_in_relationship(px->down->next, relationship_subtree);
        return;
    }
    if (traverse == 1) return;
    switch(pn_int_annotation(relationship_subtree, relationship_node_type_ANNOT)) {
        case PARENTAGE_HERE_RELN: ⟨Exceptional relationship nodes for placing objects "here" 3⟩;
        case DIRECTION_RELN: ⟨Exceptional relationship nodes for map connections 4⟩;
        case STANDARD_RELN: ⟨Standard relationship nodes (the vast majority) 2⟩;
        default: internal_error("unknown RELATIONSHIP node type");
    }
}
```

The function assert_subtree_in_relationship is called from 8/mass.

§**2.**

⟨Standard relationship nodes (the vast majority) 2⟩ ≡

```
    binary_predicate *bp = bp_get_reversal(pn_get_relationship(relationship_subtree));
    if (bp == NULL) internal_error("asserted bp-less relationship subtree");
    fix_property_bp(bp);
    assert_relation_between_subtrees(px, bp, relationship_subtree->down);
    break;
```

This code is used in §1.

§3.

⟨Exceptional relationship nodes for placing objects "here" 3⟩ ≡

```
    if (relationship_subtree->down) internal_error("malformed PARENTAGE");
    if (traverse == 1) return;
    world_object *wo = pn_get_refers_to(px);
    specification *spec = pn_get_evaluation(px);
    if (wo) spec = NULL;
    prop_true_in_world_model_about(prop_to_put_here(), wo, spec);
    break;
```

This code is used in §1.

§4.

⟨Exceptional relationship nodes for map connections 4⟩ ≡

```
    if (relationship_subtree->down == NULL) internal_error("malformed DIRECTION");
    if (pn_get_node_type(relationship_subtree->down) != NOUNPHRASE_NT) {
        sentence_problem(_P_(BelievedImpossible),
            "this is not straightforward enough in saying which "
            "room (or door) leads away from",
            "and should just name the source.");
        break;
    }
    if (relationship_subtree->down->next == NULL) internal_error("malformed DIRECTION");
    if (relationship_subtree->down->next->next != NULL) internal_error("malformed DIRECTION");
    if (pn_get_node_type(relationship_subtree->down->next) != NOUNPHRASE_NT) {
        sentence_problem(_P_(BelievedImpossible),
            "this is not straightforward enough in saying which "
            "direction the room (or door) lies in",
            "and should just name the direction.");
        break;
    }
    if (traverse == 1) return;
    world_object *oy = pn_get_refers_to(relationship_subtree->down);
    world_object *od = pn_get_refers_to(relationship_subtree->down->next);
    if ((oy == NULL) || (od == NULL) || (is_a_direction_object(od) == FALSE))
        internal_error("malformed directional subtree");
    make_map_connection(oy, pn_get_refers_to(px), od);
    break;
```

This code is used in §1.

§**5. Relating subtrees.** And here we have a relationship between subtrees T_X and T_Y, where T_X is a single value or object and T_Y is a list of values or objects (joined into an `AND_NT` tree).

```
void assert_relation_between_subtrees(parse_node *px,
    binary_predicate *bp, parse_node *py) {
    if (pn_get_node_type(py) == AND_NT) {
        assert_relation_between_subtrees(px, bp, py->down);
        assert_relation_between_subtrees(px, bp, py->down->next);
        return;
    }
    if (pn_get_node_type(py) == EVERY_NT) {
        sentence_problem(_P_(C8EveryWrongSide),
            "'every' can only be used on the other side of the verb",
            "because of limitations in Inform (but also to avoid certain "
            "possible ambiguities). In general, 'every' should be applied "
            "to the subject of an assertion sentence and not the object. "
            "Thus 'Sir Francis prefers every blonde' is not allowed, but "
            "'Every blonde is preferred by Sir Francis' is.");
            return;
    }
    reverse the relation (and swap the terms) to ensure it's the right way round
    if (bp_is_the_wrong_way_round(bp)) {
        parse_node *pz = px; px = py; py = pz;
        bp = bp_get_reversal(bp);
    }
    treat adjectives like "green" as nouns if possible
    if (pn_get_node_type(px) == ADJECTIVE_NT) pn_set_node_type(px, VALUE_NT);
    if (pn_get_node_type(py) == ADJECTIVE_NT) pn_set_node_type(py, VALUE_NT);
    treat "the player" as referring to the default player, not the variable
    turn_player_to_yourself(px);
    turn_player_to_yourself(py);
    if (((pn_get_node_type(px) != NOUNPHRASE_NT) &&
        (pn_get_node_type(px) != INSTANCE_NT) &&
        (pn_get_node_type(px) != VALUE_NT)) ||
        ((pn_get_node_type(py) != NOUNPHRASE_NT) &&
        (pn_get_node_type(py) != INSTANCE_NT) &&
        (pn_get_node_type(py) != VALUE_NT))) {
        sentence_problem(_P_(C8BadRelation),
            "this description of a relationship makes no sense to me",
            "and should be something like 'X is in Y' (or 'on' or 'part of Y'); "
            "or else 'X is here' or 'X is east of Y'.");
    }
    if ((bp_relates_values_not_objects(bp)) &&
        (((pn_get_refers_to(px)) && (pn_get_refers_to(px)->kind_flag)) ||
        ((pn_get_refers_to(py)) && (pn_get_refers_to(py)->kind_flag)))) {
        sentence_problem(_P_(C8KindRelatedToValue),
            "relations between objects and values have to be made one "
            "object at a time",
            "not using kinds of object to make multiple relationships in "
            "a single sentence. (Sorry for this restriction.)");
        return;
    }
```

```
    if (traverse == 1) return;
    world_object *wo0 = NULL, *wo1 = NULL;
    specification *spec0 = NULL, *spec1 = NULL;
    if (pn_get_refers_to(px)) wo0 = pn_get_refers_to(px);
    else spec0 = pn_get_evaluation(px);
    if (pn_get_refers_to(py)) wo1 = pn_get_refers_to(py);
    else spec1 = pn_get_evaluation(py);
    prop_true_in_world_model(prop_to_set_relation(bp, wo0, spec0, wo1, spec1));
}
```

§**6. The map-connector.** Now we come to the code which creates map connections. This needs special treatment only so that asserting $M(X, Y)$ also asserts $M'(Y, X)$, where M' is the predicate for the opposite direction to M; but since this is only a guess, it drops from `CERTAIN_CE` to merely `LIKELY_CE`.

However, the map-connector can also run in one-way mode, where it doesn't make this guess; so we begin with switching in and out.

```
int oneway_map_connections_only = FALSE;
void enter_one_way_mapping_mode(void) {
    oneway_map_connections_only = TRUE;
}
void exit_one_way_mapping_mode(void) {
    oneway_map_connections_only = FALSE;
}
```

The function enter_one_way_mapping_mode is called from 8/mass.
The function exit_one_way_mapping_mode is called from 8/mass.

§**7.** Note that, in order to make the conjectural reverse map direction, we need to look up the "opposite" property of the forward one. This relies on all directions having their opposites defined before any map is built, and is the reason for Inform's insistence that directions are always created in matched pairs.

```
void make_map_connection(world_object *go_from, world_object *go_to,
    world_object *forwards_dir) {
    world_object *reverse_dir = get_value_of_opposite_property(forwards_dir);
    if (traverse == 1) return;
    oneway_map_connection(go_from, go_to, forwards_dir, CERTAIN_CE);
    if (oneway_map_connections_only) return;
    if ((reverse_dir) && (go_to))
        oneway_map_connection(go_to, go_from, reverse_dir, LIKELY_CE);
}
void oneway_map_connection(world_object *go_from, world_object *go_to,
    world_object *forwards_dir, int certainty_level) {
    binary_predicate *bp = bp_for_mapping_direction(forwards_dir);
    if (bp == NULL) internal_error("map connection in non-direction");
    int pm = prevailing_mood;
    prevailing_mood = certainty_level;
    prop_true_in_world_model_about(prop_to_set_mapping_relation(bp, go_to),
        go_from, NULL);
    prevailing_mood = pm;
}
```

The function make_map_connection is called from 8/mass.

§**8. Everywhere.** Lastly, a simply routine to say that something is everywhere.

```
void assert_object_everywhere(world_object *wo) {
    if (wo_of_kind(wo, kind_backdrop) == FALSE) {
        object_problem(_P_(C8OnlyBackdropsEverywhere),
            wo, "must first be established to be a backdrop",
            "before it can be said to be 'everywhere'. Only backdrops "
            "can be in more than one place at once, and we are only allowed "
            "to talk about one at a time - so we can't say 'A backdrop is "
            "always everywhere', for instance.");
        return;
    }
    prop_true_in_world_model_about(prop_to_put_everywhere(), wo, NULL);
}
```

The function assert_object_everywhere is called from 8/mass.

Implications 8/imp

Purpose

To keep track of a dangerous form of super-assertion called an implication, which is allowed to generalise about properties.

8/imp.§1-4 Implication checking

Definitions

¶1. Implications are structures are used to store the information in sentences like "Something worn is usually wearable and initially carried." The "something worn" part must turn out to be a description of a category of objects; the "usually" part translates into a level of certainty. We regard these as implications in the sense of IF condition A, THEN condition B, but note that A is quite restricted in what it can be: it must be a simple-to-test description only, whereas B could be any subtree of an assertion.

```
typedef struct implication {
    struct specification *if_spec;                    which objects are affected
    struct parse_node *then_pn;                       what assertion is implied about them
    int implied_likelihood;                           with what certainty level
    struct implication *next_implication;             in list of implications
    MEMORY_MANAGEMENT
} implication;
```

The structure implication is private to this section.

¶2. We also need a little piece of storage attached to each property name:

```
typedef struct possession_marker {
    int possessed;                    temporary use when checking implications about objects
    int possession_certainty;                                                        ditto
} possession_marker;
```

The structure possession_marker is private to this section.

```
void imp_new(parse_node *px, parse_node *py) {
    implication *imp;
    world_object *wo;
    property_name *prn;
    if (prevailing_mood == CERTAIN_CE) {
        sentence_problem(_P_(C8ImplicationCertain),
            "that's an implication which is too certain for me",
            "since a sentence like this talks about a generality of things "
            "in terms of one either/or property implying another, and "
            "I can only handle those as likelihoods. You should probably "
            "add 'usually' somewhere: e.g., 'An open door is usually "
            "openable'. (But implications can have unpredictable "
            "consequences: best to avoid them altogether where possible.)");
        return;
    }
    if (pn_get_node_type(py) == AND_NT) {
```

```
        imp_new(px, py->down);
        imp_new(px, py->down->next);
        return;
    }
    imp = CREATE(implication);
    if (pn_get_node_type(px) == WITH_NT) {
        wo = pn_get_refers_to(px->down);
        imp->if_spec = pn_get_evaluation(px->down);
        if (wo == NULL) internal_error("Implication for null instance");
    } else {
        wo = kind_thing;
        imp->if_spec = pn_get_full_phrase_evaluation(px);
    }
    imp->then_pn = py;
    prn = pn_get_property(imp->then_pn);
    if ((prn) && (prn_is_either_or(prn) == FALSE)) {
        sentence_problem(_P_(C8ImplicationValueProperty),
            "that's an implication where the outcome is a value property",
            "rather than a simple either/or property, which is the only form "
            "of implication I can handle.");
        return;
    }
    imp->implied_likelihood = prevailing_mood;
    imp->next_implication = wo->implies;
    wo->implies = imp;
    LOGIF(IMPLICATIONS, "Forming implication for $O: type $S implies\n  $T",
        wo, imp->if_spec, imp->then_pn);
}
```

The function imp_new is called from 8/mass.

§**1. Implication checking.** For instance, if there is an inference asserting that object X is worn, and there is an implication that what is worn is usually also wearable, then we must generate an inference that X is wearable: in effect, this is a deduction from a syllogism. We should however not generate such an inference if we already have definite knowledge that X is not wearable. We do this for each object X individually.

The following sets flags attached to each boolean property to say whether X has that property, and if so, with what level of certainty.

```
void set_possessed_flags(world_object *wo) {
    inference *inf;
    if ((wo->kind) && (wo->kind != kind_kind)) set_possessed_flags(wo->kind);
    KNOWLEDGE_LOOP(inf, wo, PROPERTY_INF) {
        property_name *prn = inf_get_property_name(inf);
        if (prn_is_either_or(prn)) {
            int truth = TRUE, abc = inf_get_certainty(inf);
            if (abc < 0) {
                abc = -abc;
                truth = FALSE;
            }
            if (abc != 0) {
                possession_marker *pom = prn_get_possession_marker(prn);
                if (pom->possession_certainty < abc) {
                    pom->possessed = truth;
```

```
                    pom->possession_certainty = abc;
                }
                if (prn_either_or_get_negation(prn)) {
                    possession_marker *pom =
                        prn_get_possession_marker(prn_either_or_get_negation(prn));
                    if (pom->possession_certainty < abc) {
                        pom->possessed = 1-truth;
                        pom->possession_certainty = abc;
                    }
                }
            }
        }
    }
}
```

§**2.** Implications are associated with given objects, but usually in fact with kinds: an implication for all containers will apply to any instance of "container", and so forth. In the following, `wo` is the object whose implications are being considered, and `for_wo` is the instance which may be affected.

```
void check_implications_of(world_object *wo, world_object *for_wo) {
    implication *imp;
    if ((wo->kind) && (wo->kind != kind_kind)) check_implications_of(wo->kind, for_wo);
    if (wo->implies) {
        LOGIF(IMPLICATIONS, "Considering implication of $O for $O:\n",
            wo, for_wo);
    }
    for (imp = wo->implies; imp; imp = imp->next_implication) {
        int f = TRUE, changed_state = TRUE;
        property_name *then_prop = pn_get_property(imp->then_pn);
        possession_marker *pom;
        if (then_prop == NULL) continue;
        if (imp->if_spec == NULL) internal_error("implication without premiss");
        if (pn_int_annotation(imp->then_pn, negated_boolean_ANNOT))
            changed_state = FALSE;
        if (imp->implied_likelihood < 0) changed_state = 1 - changed_state;
        LOGIF(IMPLICATIONS, "$S => $T (certainty %d; changed state %d)\n",
            imp->if_spec, imp->then_pn, imp->implied_likelihood, changed_state);
        if (does_object_provide(for_wo, then_prop) == FALSE) {
            LOGIF(IMPLICATIONS, "IMPOSSIBLE: property not provided\n");
            continue;
        }
        pom = prn_get_possession_marker(then_prop);
        LOGIF(IMPLICATIONS, "Possession marker has (certainty %d; possessed state %d)\n",
            pom->possession_certainty, pom->possessed);
        if (pom->possessed == changed_state) {
            LOGIF(IMPLICATIONS, "REDUNDANT: property already correct\n");
            continue;
        }
        if (pom->possession_certainty == CERTAIN_CE) {
            LOGIF(IMPLICATIONS, "IRRELEVANT: property already settled\n");
            continue;
        }
```

```
        adjective_list_entry *ale;
        LOOP_THROUGH_ADJECTIVE_LIST(ale, imp->if_spec) {
            adjectival_phrase *aph = get_adjective_from_list_entry(ale);
            property_name *prn = aph_has_EORP_meaning(aph);
            if (prn == NULL) internal_error("non-EORP adjective in implication");
            pom = prn_get_possession_marker(prn);
            if (pom->possessed == FALSE) f = FALSE;
        }
        if (f) {
            LOGIF(IMPLICATIONS, "PASS: changing property $Y of $O\n", then_prop, for_wo);
            adjectival_phrase *aph = prn_either_or_get_aph(then_prop);
            pcalc_prop *set_prop = atom_unary_PREDICATE_from_aph(aph, FALSE);
            if (changed_state == FALSE) {
                set_prop = prop_concatenate(atom_new(NEGATION_OPEN_ATOM), set_prop);
                set_prop = prop_concatenate(set_prop, atom_new(NEGATION_CLOSE_ATOM));
            }
            int pm = prevailing_mood;
            prevailing_mood = CERTAIN_CE;
            prop_true_in_world_model_about(set_prop, for_wo, NULL);
            prevailing_mood = pm;
            pom = prn_get_possession_marker(then_prop);
            pom->possessed = changed_state;
            pom->possession_certainty = CERTAIN_CE;
            if (prn_either_or_get_negation(then_prop)) {
                pom = prn_get_possession_marker(prn_either_or_get_negation(then_prop));
                pom->possessed = 1-changed_state;
                pom->possession_certainty = CERTAIN_CE;
            }
        } else {
            LOGIF(IMPLICATIONS, "FAIL: take no action\n");
        }
    }
}
```

§**3.** We begin by checking implications associated with X and applying to X, but in fact because the routine above recurses depth-first through the kinds, a typical object X – a container, say – will first have implications associated with "thing" applied to it, then with those associated with "container", and only then its own implications.

```
void consider_implications(world_object *wo) {
    property_name *prn;
    if (wo->kind_flag) return;
    LOOP_OVER(prn, property_name) {
        possession_marker *pom = prn_get_possession_marker(prn);
        pom->possessed = FALSE;
        pom->possession_certainty = UNKNOWN_CE;
    }
    set_possessed_flags(wo);
    LOGIF(IMPLICATIONS, "Started implications about $O\n", wo);
    check_implications_of(wo, wo);
    LOGIF(IMPLICATIONS, "Finished implications about $O\n", wo);
}
```

The function consider_implications is called from 9/model.

9 Inference and Model

`9/rsdt`: *Runtime Support for Data Types.w* To compile I6 material needed at runtime to enable data types to function as they should.

`9/qty`: *Quantities.w* To manage initial and current values of named values, which may be either constants or variables, but which have global scope.

`9/scene`: *Scenes.w* Scenes are periods of time during play: at any given moment, several may be going on, or none. They are started and stopped when certain conditions are met, or by virtue of having been anchored together.

`9/wo`: *World Objects.w* Who is World, I hear you cry, and to what does he object? Just my little joke. This section contains code to create, uncreate and parse the names of world objects, which are objects or kinds of objects used in the model of the physical world.

`9/kind`: *Kinds.w* To manage kinds of world object, which are analogous to classes in the sense of Inform 6.

`9/spabp`: *Spatial Relations.w* To define the binary predicates corresponding to basic spatial relationships such as containment, support and being part of.

`9/mapbp`: *Map Connection Relations.w* To define one binary predicate for each map direction, such as "mapped north of".

`9/exrel`: *Explicit Relations.w* To draw inferences from the relations created explicitly by the source text.

`9/prop`: *Properties.w* World objects and kinds have "properties" associated with them. Some are either/or, while others hold values: a value-property always has a kind of value it must hold, and type-checking is used to ensure that it always does hold that kind of value. (Inform does not have a boolean type: so an either/or property is not a value property whose value is required to be of boolean type.) In this section of code, we consider "property names" – the properties, such as "open" or "carrying capacity", rather than knowledge about their values in the case of individual objects.

`9/pp`: *Property Permissions.w* To enforce the domain of properties: for instance, that a door can be open or closed but that an animal cannot, or that a person can have a carrying capacity but that a door cannot.

`9/provr`: *The Provision Relation.w* To define the provision relation, which determines which properties can be held by which objects.

`9/madj`: *Measurement Adjectives.w* To define adjectives such as large, wide or roomy, which make implicit comparisons of the size of some numerical property, and which (unlike other adjectives) lead to comparative and superlative forms.

`9/cmpbp`: *Comparative Relations.w* When a measurement adjective like "tall" is defined, so is a comparative relation like "taller".

`9/vpbp`: *Value-Property Relations.w* Each value property has two associated relations, one to set the value for a single holder and the other to compare its value between two holders.

`9/inf`: *Inferences.w* To manage the individual pieces of information gathered, with varying degrees of certainty, from assertion sentences. This is mostly information about which objects have what properties.

`9/model`: *Make Model World.w* Once the assertions have all been read and reduced to inferences, and all the world objects are created, we then have to work out their likely kinds. Mostly this is

easy: if we were told that X is on Y, for instance, we can immediately see that Y needs to be a supporter. But we go through a fairly thorough process in order to make sure that every piece of information is considered, even in easy cases, so that any contradiction is caught and reported back. "Creating a program often means that you have to create a small universe" (Donald Knuth).

`9/cot`: *Compile Object Tree.w* To write out the I6 object or class schemas as needed, looking through the inferences as needed to find the property values required, and to declare the objects in foundational order, as I6 (unlike I7) requires.

`9/inpw`: *Index Physical World.w* To index the kind hierarchy and the object tree.

`9/map`: *Spatial Map.w* To fit the map of the rooms in the game into a cubical grid, preserving distances and angles where possible, and then render it as HTML using a grid of icons for the World Index and also as an EPS file.

`9/tmap`: *Temporal Map.w* Parallel to the World index of space is the Scenes index of time, and in this section we render it as HTML.

Quantities 9/qty

Purpose

To manage initial and current values of named values, which may be either constants or variables, but which have global scope.

9/qty.§8 Parsing quantity names; §9 List of quantities in the Contents index; §10-11 Quantity compilation; §12 Scenes as quantities

---

Template interpreter commands

```
9   {-callv:index_quantities}
11  {-callv:declare_quantities}
11  {-array:Global_Vars}
```

Definitions

¶1. Quantities are named values: some are constants, some are variables, but all have more than temporary scope. (So, for instance, local variables are not stored as quantities.) In the text

> The colours are red, blue and green. The score to beat is a number that varies.

four quantities are created: "red", "blue" and "green", which are constants of type "colour" made to differ from all other known "colour" values (including each other), and "score to beat", which is a variable of type "number".

"Library defined" quantities are those which give NI names to constants or global variables created within the I6 library, like `score` or `location`.

```
typedef struct quantity {
    int word_ref1, word_ref2;                          text of the name
    struct parse_node *quantity_created_at;            sentence creating the quantity
    struct parse_node *initialised_at;                 where an initial value was given
    int q_documentation_symbol_wn;                     reference to manual, if any
    int is_a_variable;                                 rather than a constant
    struct specification *initial_value;               initial value (vars) or value (consts)
    int adjectivally_used;                             constant value used as an adjective, like "red"?
    struct adjectival_phrase *usage_as_aph;            if so, what adjective
    struct stacked_variable *is_a_stv;                 its identity as a STV (if any)
    struct inference *knowledge_about_constant;        if a constant
    struct kind_of_value *quantity_kov;                i.e., the data type for the variable
    int externally_defined;                            declared by I6 code already, e.g. by I6 library
    int index_in_i6_table;                             or -1 if not stored in a table
    char qty_I6_identifier[32];                        an I6 identifier
    MEMORY_MANAGEMENT
} quantity;
```

The structure quantity is private to this section.

¶2. Five more miscellaneous quantities (besides the ones treated in Bibliographic Data, that is): the initial time of day, the maximum score, "the player" – which needs special handling because Inform users believe it to be a constant, when in fact it is a variable; and a quantity to hold the I6 variable `subst__v`, used in the substitution of an object-loop variable into unbound object specifications. `i6_nothing_quantity` is by contrast a constant: it compiles to `nothing`, which in I6 is zero, but for type-checking reasons is distinguished from the number 0.

```
quantity *time_of_day_quantity = NULL;
quantity *max_score_quantity = NULL;
quantity *player_quantity = NULL;
quantity *i6_nothing_quantity = NULL;
quantity *subst_variable_quantity = NULL;                    I6 subst__v, substituted into objspecs
quantity *real_location_quantity = NULL;
quantity *actor_location_quantity = NULL;
quantity *parameter_object_quantity = NULL;
quantity *i6_glob_quantity = NULL;
quantity *I6_noun_quantity = NULL;
quantity *I6_second_quantity = NULL;
quantity *I6_actor_quantity = NULL;
```

¶3. We record the one most recently made:

```
quantity *latest_quantity = NULL;
```

---

§**1.** Some quantities can be used adjectivally, but not others. This happens when the type name coincides with a name for a property held by a world object, as might for instance happen with "colour". In other words, because it is reasonable that a ball might have a colour, we can declare that "the ball is green", or speak of "something blue": whereas we are not allowed to use "score to beat" adjectivally since (a) it is a variable, and (b) "number" is not a coinciding property: we would not ordinarily write "the ball is 4". (A quirk in English would allow this for children, say, by construing number as an age property belonging to the child in question. But we don't go there in NI.)

Therefore some quantities are entered twice into the dictionary of excerpt meanings, once in the context of nouns, once as adjectives. The following routine does the necessary supplementary work for any quantity established as possibly adjectival.

```
adjective_meaning *ENUMERATIVE_KADJ_parse(parse_node *q,
    int sense,
    int adj_name_w1, int adj_name_w2,
    int dom_name_w1, int dom_name_w2,
    int cond_w1, int cond_w2,
    int called_w1, int called_w2) {
    return NULL;
}

void ENUMERATIVE_KADJ_compiling_soon(adjective_meaning *am, quantity *q, int T) {
}

int ENUMERATIVE_KADJ_compile(quantity *q, int T, OUTPUT_STREAM, ph_stack_frame *phsf) {
    return FALSE;
}

int ENUMERATIVE_KADJ_assert(quantity *q,
```

```
    world_object *wo_to_assert_on, specification *val_to_assert_on, int parity) {
    inference_subject infs;
    if (parity == FALSE) return FALSE;
    if (wo_to_assert_on) infs = inference_subject_wo(wo_to_assert_on);
    else infs = inference_subject_spec(val_to_assert_on);
    property_name *prn = kov_get_coinciding_property(q->quantity_kov);
    if (prn == NULL) internal_error("enumerative adjective on non-property");
    inference *i = create_property_inference(infs, prn, q->initial_value);
    join_inference(i, infs);
    return TRUE;
}

int ENUMERATIVE_KADJ_index(quantity *q) {
    property_name *prn = kov_get_coinciding_property(q->quantity_kov);
    if (prn_condition_of_which_object(prn) == FALSE) {
        property_permission *pp = prn_permission_list(prn);
        if (pp) { INDEX("(of "); pp_index_applicabilities(pp); INDEX(") "); }
        INDEX("having this ");
        print_raw_text_to_file(prn->word_ref1, prn->word_ref2, ifl);
    } else {
        quantity *q2;
        int nq = 0, i = 0;
        INDEX("a condition which is otherwise ");
        LOOP_OVER(q2, quantity)
            if ((q2 != q) &&
                (q2->is_a_variable == FALSE) &&
                (kov_compare(q2->quantity_kov, q->quantity_kov))) nq++;
        LOOP_OVER(q2, quantity)
            if ((q2 != q) &&
                (q2->is_a_variable == FALSE) &&
                (kov_compare(q2->quantity_kov, q->quantity_kov))) {
                i++;
                INDEX("</i>");
                print_raw_text_to_file(q2->word_ref1, q2->word_ref2, ifl);
                INDEX("<i>");
                if (i == nq-1) INDEX(" or ");
                else if (i < nq) INDEX(", ");
            }
    }
    return TRUE;
}
```

The function ENUMERATIVE_KADJ_parse is called from 5/aph.
The function ENUMERATIVE_KADJ_compiling_soon is called from 5/aph.
The function ENUMERATIVE_KADJ_compile is called from 5/aph.
The function ENUMERATIVE_KADJ_assert is called from 5/aph.
The function ENUMERATIVE_KADJ_index is called from 5/aph.

§2.

```
void register_as_adjectival_quantity(quantity *q, property_name *prn) {
    LOGIF(QUANTITY_COMPILATION, "RACQ on $W property $Y (%d)\n",
        q->word_ref1, q->word_ref2, prn, spec_get_data(q->initial_value, QUANTITY_ENUMERATION_SPDATA));
    pp_register_as_adjectival(q, prn);
}
void register_as_adjectival_quantity_domain(quantity *q, property_name *prn,
    kind_of_value *domain_kov, world_object *domain_wo) {
    if (qty_is_a_variable(q) == FALSE) {
        if (domain_kov == NULL) {
            if (q->adjectivally_used) return;
            q->adjectivally_used = TRUE;
        }
        LOGIF(QUANTITY_COMPILATION, "RACQ on $W property $Y (%d), WO=$O, KOV=$u\n",
            q->word_ref1, q->word_ref2, prn, spec_get_data(q->initial_value, QUANTITY_ENUMERATION_SPDATA),
domain_wo,
            domain_kov);
        if (domain_kov)
        LOGIF(QUANTITY_COMPILATION, "RACQ sees range %d, offset %d in ID %d\n",
            pp_get_prn_offset_in_kov(NULL, domain_kov),
            pp_get_prn_offset_in_kov(prn, domain_kov),
            kov_I6_ID(domain_kov));
        adjective_meaning *am = am_new(ENUMERATIVE_KADJ, STORE_POINTER_quantity(q),
            q->word_ref1, q->word_ref2);
        q->usage_as_aph = am_declare(am, q->word_ref1, q->word_ref2);
        if (domain_wo)
            am_set_domain_from_wo(am, domain_wo);
        else if (domain_kov)
            am_set_domain_from_kov(am, domain_kov);
        else
            am_set_domain_from_wo(am, NULL);
        i6_schema *sch = am_set_i6_schema(am, TEST_ADJECTIVE_TASK, FALSE);
        if (domain_kov)
        sch_write_to_existing_3d(sch,
            "PropertyKOV(KOVP_%d,%d,*1) == %d",
                kov_I6_ID(domain_kov),
                pp_get_prn_offset_in_kov(prn, domain_kov),
                spec_get_data(q->initial_value, QUANTITY_ENUMERATION_SPDATA));
        else
        sch_write_to_existing_1s1d(sch,
            "ValueProperty(*1,%s) == %d", prn_get_i6_identifier(prn),
                spec_get_data(q->initial_value, QUANTITY_ENUMERATION_SPDATA));
        sch = am_set_i6_schema(am, NOW_ADJECTIVE_TRUE_TASK, FALSE);
        if (domain_kov)
        sch_write_to_existing_3d(sch,
            "WritePropertyKOV(KOVP_%d,%d,*1,%d)",
                kov_I6_ID(domain_kov),
                pp_get_prn_offset_in_kov(prn, domain_kov),
                spec_get_data(q->initial_value, QUANTITY_ENUMERATION_SPDATA));
        else
        sch_write_to_existing_1s1d(sch,
            "WriteValueProperty(*1,%s,%d)", prn_get_i6_identifier(prn),
```

```
                spec_get_data(q->initial_value, QUANTITY_ENUMERATION_SPDATA));
    }
}
```

The function register_as_adjectival_quantity_domain is called from 9/pp.

§**3.** And this is the routine with the power to make a property and a type name coincide:

```
void make_type_coincident(kind_of_value *kov, property_name *pn) {
    kov_set_coinciding_property(kov, pn);
    update_adjectival_forms(pn);
}

void update_adjectival_forms(property_name *pn) {
    if (prn_is_either_or(pn) == TRUE) return;
    kind_of_value *kov = prn_get_kind_of_value(pn);
    if (pn == kov_get_coinciding_property(kov)) {
        quantity *q;
        LOOP_OVER(q, quantity)
            if (kov_compare(q->quantity_kov, kov))
                register_as_adjectival_quantity(q, pn);
    }
}
```

The function make_type_coincident is called from 7/data and 9/prop.
The function update_adjectival_forms is called from 9/pp.

§**4.** With those defined, here is the routine which creates a new quantity and decides what name it should have in the I6 source code.

NI is a typed language but I6 is not. The quantities "number understood", "time understood", etc., really all represent the same thing – the value which was parsed in a command at run-time – but they are all distinct NI quantities because they need to have different types. In I6 they revert to being all the same variable `parsed_number`.

The other complications in this routine are to set a dozen or so global
(NI) variables to the quantities created with particular names – they are created, mostly, in the Standard Rules; to give some quantities specific I6 names defined in the I6 library, while letting others have mechanically generated new I6 names; and to notify the scene machinery when a new scene constant is created. (To create a constant scene is to create an actual scene.)

```
int no_i6_table_entries_used = 0;

quantity *new_model_quantity(int w1, int w2, kind_of_value *kov, int varies) {
    PROTECTED_MODEL_PROCEDURE;
    return new_quantity(w1, w2, kov, varies, NULL);
}

quantity *new_stacked_quantity(int w1, int w2, kind_of_value *kov, stacked_variable *stv) {
    if (stv == NULL) internal_error("not a stacked quantity at all");
    return new_quantity(w1, w2, kov, TRUE, stv);
}

quantity *new_quantity(int w1, int w2, kind_of_value *kov, int varies, stacked_variable *stv)
{
    quantity *q = CREATE(quantity);
    latest_quantity = q;
    q->q_documentation_symbol_wn = dref_position_of_symbol(&w1, &w2, FALSE);
    q->word_ref1 = w1; q->word_ref2 = w2;
```

```
q->is_a_variable = varies;
q->initial_value = new_UNKNOWN_spec();
q->initialised_at = NULL;
q->quantity_created_at = current_sentence;
if (kov == NULL) internal_error("created quantity without kind of value");
q->quantity_kov = kov;
q->adjectivally_used = FALSE;
q->usage_as_aph = NULL;
q->externally_defined = FALSE;
q->index_in_i6_table = -1;
q->is_a_stv = NULL;
q->knowledge_about_constant = NULL;
if (stv) {
    q->is_a_stv = stv;
    q->index_in_i6_table = -2;
    if (varies == FALSE) internal_error("stacked variable quantities must be var");
} else {
    if ((prn_condition_of_which_object(kov_get_coinciding_property(kov))) == NULL)
        register_excerpt_meaning(QUANTITY_MC, 0, w1, w2, STORE_POINTER_quantity(q));
}
if [[w1, w2 == ... understood]] {
    int d1, d2;
    kov_get_name(kov, &d1, &d2, FALSE);
    if ((d1 >= 0) && (compare_word_range(w1, w2-1, d1, d2))) {
        strcpy(q->qty_I6_identifier, "parsed_number");
        q->externally_defined = TRUE;
        return q;
    }
    if ((d1 < 0) && (w2 == w1+1) &&
        (is_kova(kov, SNIPPET_TY) == FALSE) &&
        (compare_word(w1, kov_get_parsing_name(kov)))) {
        strcpy(q->qty_I6_identifier, "parsed_number");
        q->externally_defined = TRUE;
        return q;
    }
}
if (varies == FALSE) {
    if (kov_has_named_constant_values(kov)) {
        q->initial_value = new_QUANTITY_spec(q);
        spec_set_data(q->initial_value, QUANTITY_ENUMERATION_SPDATA,
            kov_new_enumerated_value(q->quantity_kov));
    }
    if (is_kova(kov, SCENE_TY)) new_scene_made(q, 0);
    if (kov_get_coinciding_property(kov))
        register_as_adjectival_quantity(q, kov_get_coinciding_property(kov));
}
if [[w1, w2 == story author]]        story_author = q;
if [[w1, w2 == story headline]]      story_headline = q;
if [[w1, w2 == story genre]]         story_genre = q;
if [[w1, w2 == story description]]   story_description = q;
if [[w1, w2 == story creation year]] story_creation_year = q;
if [[w1, w2 == release number]]      story_release_number = q;
if [[w1, w2 == noun]]                I6_noun_quantity = q;
```

```
    if [[w1, w2 == player]]              player_quantity = q;
    if [[w1, w2 == location]]            real_location_quantity = q;
    if [[w1, w2 == actor_location]]      actor_location_quantity = q;
    if [[w1, w2 == i6HYPHENnothingHYPHENconstant]] i6_nothing_quantity = q;
    if [[w1, w2 == substitutionHYPHENvariable]]  subst_variable_quantity = q;
    if [[w1, w2 == second noun]]         I6_second_quantity = q;
    if [[w1, w2 == person asked]]        I6_actor_quantity = q;
    if [[w1, w2 == maximum score]]       max_score_quantity = q;
    if [[w1, w2 == i6HYPHENvaryingHYPHENglobal]] i6_glob_quantity = q;
    if [[w1, w2 == story title]]         story_title = q;
    if [[w1, w2 == parameter_object]]    parameter_object_quantity = q;
    if [[w1, w2 == time of day]]         time_of_day_quantity = q;
    if (qty_is_a_variable(q)) q->index_in_i6_table = no_i6_table_entries_used++;
    isn_compose_identifier(q->qty_I6_identifier, 'Q', q->allocation_id, w1, w2);
    LOGIF(QUANTITY_CREATIONS, "Created non-library quantity: $Q\n", q);
    return q;
}
```

The function new_model_quantity is called from 6/asp.
The function new_stacked_quantity is called from 12/stv.

§**5.** Most quantities are given automatically generated Inform 6 names in the compiled code: `Q4_green`, for instance. A few must however correspond to names of significance in the I6 library.

```
void qty_translates(parse_node *pn) {
    parse_node *p1 = pn->down->next;
    parse_node *p2 = pn->down->next->next;
    quantity *q = qty_parse(p1->word_ref1, p1->word_ref2);
    if ((q == NULL) || ((qty_is_a_variable(q) == FALSE))) {
        sentence_problem(_P_(C9NonQuantityTranslated),
            "this is not the name of a variable",
            "or at any rate not one global in scope.");
        return;
    }
    if (q->externally_defined) {
        sentence_problem(_P_(C9QuantityTranslatedAlready),
            "this property has already been translated",
            "so there must be some duplication somewhere.");
        return;
    }
    strcpy(q->qty_I6_identifier, lw_array[p2->word_ref1].lw_text);
    q->externally_defined = TRUE;
    q->index_in_i6_table = -1;
    LOGIF(QUANTITY_CREATIONS, "Translated quantity: $Q as %s\n", q, lw_array[p2->word_ref1].lw_text);
}
```

The function qty_translates is called from 2/isn.

§**6.** And some miscellaneous services:

```
quantity *qty_temporary_quantity(char *I6_form) {
    strcpy(i6_glob_quantity->qty_I6_identifier, I6_form);
    return i6_glob_quantity;
}

void log_quantity(quantity *q) {
    if (q== NULL) { LOG("<null quantity>"); return; }
    LOG("%s%s[$u]", q->qty_I6_identifier, q->is_a_variable?"(var)":"", q->quantity_kov);
}

char *qty_get_I6_representation(quantity *q) {
    return q->qty_I6_identifier;
}

specification *qty_get_initial_value(quantity *q) {
    return q->initial_value;
}

adjectival_phrase *qty_get_adjectival_phrase(quantity *q) {
    return q->usage_as_aph;
}

inference **qty_get_knowledge(quantity *q) {
    return &(q->knowledge_about_constant);
}

void qty_now_has_initial_value_set(quantity *q) {
    if (q == NULL) internal_error("non-quantity in initial value");
    q->initialised_at = current_sentence;
}

int qty_has_initial_value_set(quantity *q) {
    if (q->initialised_at) return TRUE;
    return FALSE;
}

void deduce_initial_value(quantity *q) {
    inference *inf;
    inference_subject infs = inference_subject_q(q);
    GENERAL_POSITIVE_KNOWLEDGE_LOOP(inf, infs, GLOBAL_INF) {
        specification *val = inf_get_literal_value(inf);
        q->initial_value = spec_copy(val);
    }
}

int qty_treat_as_plain_text_word(quantity *q) {
    deduce_initial_value(q);
    spec_set_kind_of_value(q->initial_value, kova(TEXT_TY));
    return q->initial_value->word_ref1;
}

int qty_is_a_variable(quantity *q) {
    return q->is_a_variable;
}

int qty_is_a_global_variable(quantity *q) {
    if (q->is_a_stv) return FALSE;
    return q->is_a_variable;
}
```

```
int qty_is_an_I6_library_variable(specification *spec) {
    quantity *q;
    if (spec == NULL) return FALSE;
    if (spec_get_storage_form(spec) != NONLOCAL_VARIABLE_SPC) return FALSE;
    q = RETRIEVE_FROM_SPEC(spec, quantity);
    if (q == I6_noun_quantity) return TRUE;
    if (q == I6_second_quantity) return TRUE;
    if (q == I6_actor_quantity) return TRUE;
    return FALSE;
}
kind_of_value *qty_kind_of_value(quantity *q) {
    return q->quantity_kov;
}
int qty_get_value_index(quantity *q) {
    return spec_get_data(q->initial_value, QUANTITY_ENUMERATION_SPDATA) - 1;
}
```

The function qty_temporary_quantity is called from 11/ap and 11/los.
The function log_quantity is called from 2/dl.
The function qty_get_I6_representation is called from 7/kov, 7/vasp, 7/stsp, 9/prop and 11/ap.
The function qty_get_initial_value is called from 6/asp, 7/specv, 8/creat, 10/tab and 10/bib.
The function qty_get_adjectival_phrase is called from 5/varc, 6/term and 6/pform.
The function qty_get_knowledge is called from 9/inf.
The function qty_now_has_initial_value_set is called from 6/equal.
The function qty_has_initial_value_set is called from 10/tab and 10/bib.
The function qty_treat_as_plain_text_word is called from 10/bib.
The function qty_is_a_variable is called from 2/lexi, 4/edoc, 5/candd, 7/kov, 9/rsdt, 7/kix, 7/specv, 8/refpt, 8/mass, 9/prop, 10/tab, 11/ap and 13/gprv.
The function qty_is_a_global_variable is called from 4/edoc, 9/rsdt and 7/stsp.
The function qty_is_an_I6_library_variable is called from 6/atoms.
The function qty_kind_of_value is called from 2/lexi, 4/edoc, 5/mlc, 6/pform, 7/kov, 9/rsdt, 7/kix, 7/specv, 7/stsp, 7/tc, 8/refpt, 8/mass, 8/knowp, 9/prop, 12/stv, 12/cinv and 13/gprv.
The function qty_get_value_index is called from 9/cot.

§**7.** Actually compiling a valid I6 expression for a given quantity is now fairly easy.

```
void compile_quantity_name(OUTPUT_STREAM, quantity *q) {
    if (q->index_in_i6_table >= 0)
        WRITE("(Global_Vars-->%d)", q->index_in_i6_table);
    else
        WRITE("%s", q->qty_I6_identifier);
}
void compile_quantity_name_to_string(OUTPUT_STREAM, quantity *q) {
    if (q->is_a_stv)
        stv_compile_lvalue(OUT, q->is_a_stv);
    else {
        if (q->index_in_i6_table >= 0)
            WRITE("(Global_Vars-->%d)", q->index_in_i6_table);
        else
            WRITE("%s", q->qty_I6_identifier);
    }
}
```

The function compile_quantity_name is called from 9/rsdt and 13/gprv.
The function compile_quantity_name_to_string is called from 7/vasp and 7/stsp.

§8. Parsing quantity names.

```
quantity *qty_parse(int w1, int w2) {
    meaning_list *ml;
    [[w1, w2 == the ... --> w1, w2]];
    ml = SP_excerpt(QUANTITY_MC, w1, w2);
    if (ml) return RETRIEVE_POINTER_quantity(em_data(ml_meaning(ml)));
    return NULL;
}
```

The function qty_parse is called from 8/mass.

§9. List of quantities in the Contents index. The following is part of the Contents index, and acts as an aide-memoire of the names of the variable quantities created in the source text.

```
void index_quantities(void) {
    quantity *q;
    heading *definition_area, *current_area = NULL;
    INDEX("<p><hr><p>");
    index_anchor("NAMES");
    INDEX("<b>List of named values</b> (<i>About these</i>");
    index_doc_link("VARIABLES"); INDEX(")<p>\n");
    LOOP_OVER(q, quantity)
        if ((qty_is_a_variable(q)) && (q->word_ref1 >= 0) && (q->is_a_stv == NULL)) {
            definition_area =
                heading_of(lw_array[q->word_ref1].lw_source);
            if (hd_indexed(definition_area) == FALSE) continue;
            if (definition_area != current_area) {
                int x1, x2, j;
                hd_get_text_of_heading(definition_area, &x1, &x2);
                INDEX("<p>");
                if ((x1 >= 0) && ((j = is_word_intermediate(HYPHEN_V, x1, x2))>=0)) {
                    x1 = j+1;
                    INDEX("<b>");
                    print_raw_text_to_file(x1, x2, ifl);
                    INDEX("</b><br>");
                }
            }
            current_area = definition_area;
            qty_index_single(q);
        }
}

void qty_index_single(quantity *q) {
    print_raw_text_to_file(q->word_ref1, q->word_ref2, ifl);
    index_link(lw_array[q->word_ref1].lw_source);
    if (q->q_documentation_symbol_wn >= 0) {
        index_doc_link(lw_array[q->q_documentation_symbol_wn].lw_rawtext);
    }
    INDEX(" ... <i>");
    index_lowercase_kov(q->quantity_kov);
```

```
    INDEX("</i><br>\n");
}

world_object *qty_initial_value(quantity *q) {
    if (q == NULL) return NULL;
    return wo_of_CONSTANT_OBJECT_if_any(q->initial_value);
}
```

The function index_quantities is invoked by a command in a `.i6t` template file.
The function qty_index_single is called from 12/stv.
The function qty_initial_value is called from 9/cot.

§**10. Quantity compilation.** The following routine compiles the correct initial value for the given quantity. If it has no known initial value, it is given the initial value for its type where possible: note that this may not be possible if the source text says something like

> Thickness is a kind of value. The carpet nap is a thickness that varies.

without specifying any thicknesses: the set of legal thickness values is empty, so the carpet nap variable cannot be created in a way which makes its type safe. Hence the error messages.

```
void compile_quantity_initial(OUTPUT_STREAM, quantity *q) {
    TEMPORARY_STREAM;
    LOGIF(QUANTITY_COMPILATION,
        "Compiling initial value of quantity: $Q\n"
        "From initial value $S\n", q, q->initial_value);
    compile_quantity_initial_inner(TEMP, q);
    STREAM_COPY(OUT, TEMP);
    CLOSE_TEMPORARY_STREAM;
}

void compile_quantity_initial_inner(OUTPUT_STREAM, quantity *q) {
    if (q->is_a_variable) deduce_initial_value(q);
    if (q->quantity_created_at) current_sentence = q->quantity_created_at;
    if (spec_is_UNKNOWN(q->initial_value)) {
        LOGIF(QUANTITY_COMPILATION, "Initialising missing initial type: $u\n",
            q->quantity_kov);
        if (is_kova(q->quantity_kov, TABLE_TY)) {
            quote_words(1, q->word_ref1, q->word_ref2);
            handmade_problem(_P_(C9UnspecifiedTable));
            issue_problem_segment(
                "The variable '%1' must always contain a table, but has never "
                "been given an initial value, and it seems a bad idea to allow "
                "me to guess one (as I would for a mere number variable, for "
                "instance): the potential for error is too great. So please "
                "add a sentence specifying the initial value: e.g., 'The "
                "current lexicon is Table 12.'");
            issue_problem_end();
            return;
        }
        switch(compile_default_value(OUT, q->quantity_kov,
            q->word_ref1, q->word_ref2, "variable")) {
            case NOT_APPLICABLE: return;
            case FALSE: {
                int w1, w2;
                kov_get_name(q->quantity_kov, &w1, &w2, FALSE);
```

```
                quote_words(1, q->word_ref1, q->word_ref2);
                quote_words(2, w1, w2);
                handmade_problem(_P_(C9EmptyDataType));
                issue_problem_segment(
                    "I am unable to put any value into the variable '%1', because "
                    "%2 is a kind of value with no actual values.");
                issue_problem_end(); return;
            }
        }
    } else {
        if (q->initialised_at) current_sentence = q->initialised_at;
        if (spec_get_storage_form(q->initial_value) == NONLOCAL_VARIABLE_SPC) {
            quantity *q2 = RETRIEVE_FROM_SPEC(q->initial_value, quantity);
            if (q2 == NULL) internal_error(
                "Tried to compile initial value of quantity as null quantity");
            if (q2 == q) {
                quote_source(1, q->initialised_at);
                quote_words(2, q->word_ref1, q->word_ref2);
                quote_kov(3, q->quantity_kov);
                handmade_problem(_P_(C9InitialiseQ2));
                issue_problem_segment(
                    "The sentence %1 tells me that '%2', which should be %3 "
                    "that varies, is to have an initial value equal to itself - "
                    "this is such an odd thing to say that I think I must have "
                    "misunderstood.");
                issue_problem_end();
                return;
            }
            quote_source(1, q->initialised_at);
            quote_words(2, q->word_ref1, q->word_ref2);
            quote_kov(3, q->quantity_kov);
            quote_words(4, q2->word_ref1, q2->word_ref2);
            quote_kov(5, q2->quantity_kov);
            handmade_problem(_P_(C9InitialiseQ1));
            issue_problem_segment(
                "The sentence %1 tells me that '%2', which should be %3 "
                "that varies, is to have an initial value equal to '%4', "
                "which in turn is %5 that varies. At the start of play, "
                "variable values have to be set equal to definite constants, "
                "so this is not allowed.");
            issue_problem_end();
            return;
        }
        spec_compile(OUT, q->initial_value);
    }
}
```

The function compile_quantity_initial is called from 9/cot, 10/tab, 10/bib and 12/stv.

§**11.** Global variables and constants in I6 code need to be explicitly declared, and this we do as follows. Note that certain quantities ("score", for instance) correspond to I6 variables already defined by the I6 library code without our intervention, so we skip those.

```
void declare_quantities(OUTPUT_STREAM) {
    quantity *q;
    LOGIF(QUANTITY_COMPILATION, "Compiling quantity declarations\n");
    LOOP_OVER(q, quantity)
        if ((q->externally_defined == FALSE) && (q->index_in_i6_table == -1)) {
            LOGIF(QUANTITY_COMPILATION, "Declaring $Q\n", q);
            if (qty_is_a_variable(q)) WRITE("Global ");
            else WRITE("Constant ");
            compile_quantity_name(OUT, q);
            WRITE(" = ");
            if (qty_is_a_variable(q)) compile_quantity_initial(OUT, q);
            else WRITE("%d", spec_get_data(q->initial_value, QUANTITY_ENUMERATION_SPDATA));
            WRITE(";\n");
        } else {
            LOGIF(QUANTITY_COMPILATION, "Not declaring $Q\n", q);
        }
}
void compile_Global_Vars_array(OUTPUT_STREAM) {
    quantity *q;
    if (no_i6_table_entries_used > 0) {
        int ind = 0;
        WRITE("Array Global_Vars -->\n");
        LOOP_OVER(q, quantity)
            if (q->index_in_i6_table >= ind) {
                while (q->index_in_i6_table > ind) {
                    WRITE("  (0) ! Skipped\n");
                    ind++;
                }
                ind++;
                WRITE("  (");
                BEGIN_COMPILATION_MODE;
                COMPILATION_MODE_EXIT(DEREFERENCE_POINTERS_CMODE);
                compile_quantity_initial(OUT, q);
                END_COMPILATION_MODE;
                WRITE(") ! %d: %s\n", ind, q->qty_I6_identifier);
            }
        WRITE("  NULL\n");
        WRITE("  ; ! extent %d words\n", no_i6_table_entries_used+1);
    }
}
```

The function declare_quantities is invoked by a command in a `.i6t` template file.

The function compile_Global_Vars_array is invoked by a command in a `.i6t` template file.

§**12. Scenes as quantities.** When a new scene is created, with text such as "Train Stop is a scene.", what actually happens is that a constant quantity ("Train Stop") is created, whose kind of value is `SCENE`. The following routines place a scene pointer into, or extract one from, a quantity.

```
void write_scene_to_quantity(quantity *q, scene *sc) {
    int ix = spec_get_data(q->initial_value, QUANTITY_ENUMERATION_SPDATA);
    q->quantity_kov = kova(SCENE_TY);
    q->initial_value = scene_to_SCENE_spec(sc);
    spec_set_data(q->initial_value, QUANTITY_ENUMERATION_SPDATA, ix);
}

scene *read_scene_from_quantity(quantity *q) {
    if (is_kova(q->quantity_kov, SCENE_TY) == FALSE)
        internal_error("tried to find scene from non-scenic quantity");
    return SCENE_spec_to_scene(q->initial_value);
}

int qty_is_a_scene_name(quantity *q) {
    if (is_kova(q->quantity_kov, SCENE_TY)) return TRUE;
    return FALSE;
}
```

The function write_scene_to_quantity is called from 9/scene.

The function read_scene_from_quantity is called from 9/scene and 12/phrcd.

The function qty_is_a_scene_name is called from 9/scene, 12/phud and 12/phrcd.

Scenes 9/scene

Purpose

Scenes are periods of time during play: at any given moment, several may be going on, or none. They are started and stopped when certain conditions are met, or by virtue of having been anchored together.

9/scene.§3 Anchors; §4 Run-time storage for scenes; §5 Scene-changing machinery at run-time

Template interpreter commands

```
5   {-routine:DetectSceneChange}
5   {-routine:PrintSceneName}
5   {-routine:ShowSceneStatus}
5   {-array:scene_recurs}
```

Definitions

¶1. Scenes are gated intervals of time, but there are more than two gates: for while there is only one past, there are many possible futures. These gates are called "ends" in the code below, and are numbered end 0 (the beginning), end 1 (the usual end), and then any named ends ("ends badly" or "ends triumphantly", for instance, might be ends 2 and 3). Each end has a condition which can cause it, or can be "anchored" to any number of ends of other scenes – to express which, the `scene_connector` structure is used.

define `MAX_SCENE_ENDS 32`

```
typedef struct scene_connector {
    struct scene *connect_to;                                   scene connected to
    int end;                                                    end number
    struct scene_connector *next;                               next in list of connectors for a scene end
    struct parse_node *where_said;                              where this linkage was specified in source
} scene_connector;

typedef struct scene {
    struct quantity *q;                                         the constant for the name of the scene
    int once_only;                                              cannot repeat during play
    int start_of_play;                                          if begins when play begins
    int marker;
    int no_ends;                                                how many ends the scene has
    int end_names_w1[MAX_SCENE_ENDS];                           for ends 2, 3, ...: e.g. "badly"
    int end_names_w2[MAX_SCENE_ENDS];
    struct rulebook *end_rulebook[MAX_SCENE_ENDS];              rules to apply then
    struct specification *anchor_condition[MAX_SCENE_ENDS];
    struct scene_connector *anchor_scene[MAX_SCENE_ENDS];       linked list
    int indexed;                                  temporary storage during Scenes index creation
    struct parse_node *scene_declared_at;                       where defined
    struct parse_node *anchor_condition_set[MAX_SCENE_ENDS];    where set
    MEMORY_MANAGEMENT
} scene;
```

The structure scene_connector is shared with 9/tmap.
The structure scene is shared with 9/tmap.

---

§**1.** Each scene ending has its own rulebook, and this is where they are created.

```
void new_scene_rulebook(scene *sc, int end) {
    int wn = lexer_wordcount;
    feed_into_lexer("when", TRUE, FALSE);
    splice_words(sc->q->word_ref1, sc->q->word_ref2);
    feed_into_lexer((end==0)?"begins":"ends", TRUE, FALSE);
    if (end >= 2) splice_words(sc->end_names_w1[end], sc->end_names_w2[end]);
    sc->end_rulebook[end] =
        rb_new_automatic(wn, lexer_wordcount-1, ACTION_FOCUS,
            NO_OUTCOME, FALSE, FALSE, FALSE);
    wn = lexer_wordcount;
    feed_into_lexer("when the ", TRUE, FALSE);
    splice_words(sc->q->word_ref1, sc->q->word_ref2);
    feed_into_lexer((end==0)?"begins":"ends", TRUE, FALSE);
    if (end >= 2) splice_words(sc->end_names_w1[end], sc->end_names_w2[end]);
    register_excerpt_meaning(RULEBOOK_MC, 0, wn, lexer_wordcount-1,
        STORE_POINTER_rulebook(sc->end_rulebook[end]));
    rb_set_alt_name(sc->end_rulebook[end], wn, lexer_wordcount-1);
    if (end >= 2) {
        char i6_code[128];
        wn = lexer_wordcount;
        feed_into_lexer("To decide if ", TRUE, FALSE);
        splice_words(sc->q->word_ref1, sc->q->word_ref2);
        feed_into_lexer(" ended ", TRUE, FALSE);
        splice_words(sc->end_names_w1[end], sc->end_names_w2[end]);
        make_sentence_node(wn, lexer_wordcount-1, ':');
        wn = lexer_wordcount;
        sprintf(i6_code, " (- (scene_latest_ending-->%d == %d) -) ",
            sc->allocation_id, end);
        feed_into_lexer(i6_code, TRUE, FALSE);
        make_sentence_node(wn, lexer_wordcount-1, '.');
        wn = lexer_wordcount;
        feed_into_lexer("To decide if ", TRUE, FALSE);
        splice_words(sc->q->word_ref1, sc->q->word_ref2);
        feed_into_lexer(" did not end ", TRUE, FALSE);
        splice_words(sc->end_names_w1[end], sc->end_names_w2[end]);
        make_sentence_node(wn, lexer_wordcount-1, ':');
        wn = lexer_wordcount;
        sprintf(i6_code, " (- (scene_latest_ending-->%d ~= 0 or %d) -) ",
            sc->allocation_id, end);
        feed_into_lexer(i6_code, TRUE, FALSE);
        make_sentence_node(wn, lexer_wordcount-1, '.');
        register_recently_lexed_phrases();
    }
}
```

§**2.** The following routine is so named because it is called by the quantity-maker. In effect, the quantities code notifies us of the creation of a new `SCENE` constant: we take it from there.

```
void new_scene_made(quantity *q, int modifiers) {
    int end;
    scene *sc = CREATE(scene);
    sc->q = q;
    sc->once_only = TRUE; if (modifiers & 1) sc->once_only = FALSE;
    sc->indexed = FALSE;
    sc->no_ends = 2;
    sc->start_of_play = FALSE;
    sc->scene_declared_at = current_sentence;
    for (end=0; end<2; end++) {
        sc->anchor_condition[end] = NULL;
        sc->anchor_scene[end] = NULL;
    }
    write_scene_to_quantity(q, sc);
    new_scene_rulebook(sc, 0);
    new_scene_rulebook(sc, 1);
}

quantity *scene_quantity(scene *sc) {
    if (sc == NULL) return NULL;
    return sc->q;
}
```

The function new_scene_made is called from 9/qty.
The function scene_quantity is called from 7/specv.

§**3. Anchors.** These are joins between the endings of different scenes. Any end of any scene can be anchored to any end of any scene (including itself).

```
sentence_handler BEGINS_WHEN_SH_handler =
    { SENTENCE_NT, BEGINS_WHEN_VB, 2, begins_or_ends_when };
sentence_handler ENDS_WHEN_SH_handler =
    { SENTENCE_NT, ENDS_WHEN_VB, 2, begins_or_ends_when };

void begins_or_ends_when(parse_node *p) {
    model_world_constructed = TRUE;
    new_scene_anchor(p);
    model_world_constructed = FALSE;
}

void new_scene_anchor(parse_node *p) {
    int cw1 = p->down->next->next->word_ref1;
    int cw2 = p->down->next->next->word_ref2;
    int sw1 = p->down->next->word_ref1;
    int sw2 = p->down->next->word_ref2;
    int en1 = -1, en2 = -1;
    int oen1 = -1, oen2 = -1;
    int end = 0, otherend = -1, i;
    specification *spec1, *spec2;
    quantity *q = NULL, *q2 = NULL;
    scene *sc, *sc2;
    if (pn_int_annotation(p->down, verb_id_ANNOT) == ENDS_WHEN_VB) {
        end = 1;
```

```
    [[sw1, sw2 == ... ends ... : i --> sw1, sw2 ... en1, en2]];
}
spec1 = parse_expression(sw1, sw2, VALUE_EXPCON);
if [[cw1, cw2 == play ends]] {
    sentence_problem(_P_(C9ScenesNotPlay),
        "'play' is not really a scene",
        "so although you can write '... when play begins' "
        "you cannot write '... when play ends'. But there's no need "
        "to do so, anyway. When play ends, all scenes end.");
    return;
}
if [[cw1, cw2 == ... begins]] otherend = 0;
if [[cw1, cw2 == ... ends]] otherend = 1;
if (otherend == -1) {
    if [[cw1, cw2 == ... ends ... : i --> cw1, cw2 ... oen1, oen2]] {
        cw2 = i; otherend = 1;
    }
}
if [[cw1, cw2 == play begins]] otherend = -2;
if (word_range_includes_a_calling(cw1, cw2)) {
    LOG("Scene word range = $W\n", cw1, cw2);
    sentence_problem(_P_(C9ScenesDisallowCalled),
        "'(called ...)' is not allowed within conditions for a "
        "scene to begin or end",
        "since calling gives only a temporary name to something, "
        "for the purpose of further instructions which immediately "
        "follow in. Here there is no room for such further instructions, "
        "so a calling would have no effect. Anyway - not allowed!");
    return;
}
spec2 = new_UNKNOWN_spec();
if ((otherend >= 0) && (cw2>cw1))
    spec2 = parse_expression(cw1, cw2-1, VALUE_EXPCON);
if ((spec_is_UNKNOWN(spec2)) && (cw1 >= 0))
    spec2 = parse_expression(cw1, cw2, CONDITION_EXPCON);
if ((q = spec_get_constant_quantity_if_any(spec1)) &&
    (qty_is_a_scene_name(q))) {
    sc = read_scene_from_quantity(q);
    if (en1 >= 0) {
        for (i=2; i<sc->no_ends; i++)
            if (compare_word_range(en1, en2,
                sc->end_names_w1[i], sc->end_names_w2[i]))
                end = i;
        if (end == 1) {
            end = sc->no_ends++;
            sc->end_names_w1[end] = en1;
            sc->end_names_w2[end] = en2;
            new_scene_rulebook(sc, end);
        }
    }
    if (sc == NULL) internal_error("Scene has no underlying structure");
} else {
```

```
        LOG("Not a scene: $X", spec1);
        sentence_problem(_P_(C9ScenesOnly),
            "'begins when' and 'ends when' can only be applied to scenes",
            "which have already been defined with a sentence like "
            "'The final confrontation is a scene.'");
        return;
    }
    if ((otherend == -2) ||
        ((q2 = spec_get_constant_quantity_if_any(spec2)) &&
        (qty_is_a_scene_name(q2)))) {
        scene_connector *scon = CREATE(scene_connector);
        if (otherend == -2) {
            sc->start_of_play = TRUE;
            return;
        }
        sc2 = read_scene_from_quantity(q2);
        if (sc2 == NULL)
            internal_error("Connecting scene has no underlying structure");
        if (oen1 >= 0) {
            for (i=2; i<sc2->no_ends; i++)
                if (compare_word_range(oen1, oen2,
                    sc2->end_names_w1[i], sc2->end_names_w2[i]))
                    otherend = i;
            if (otherend == 1) {
                sentence_problem(_P_(C9ScenesUnknownEnd),
                    "that's not one of the known ends for that scene",
                    "which must be declared with something like 'Confrontation "
                    "ends happily when...' or 'Confrontation ends tragically "
                    "when...'.");
                return;
            }
        }
        scon->connect_to = sc2;
        scon->end = otherend;
        scon->next = sc->anchor_scene[end];
        scon->where_said = current_sentence;
        sc->anchor_scene[end] = scon;
        return;
    }
    if (sc->anchor_condition[end]) {
        sentence_problem(_P_(C9ScenesOversetEnd),
            "you have already told me a condition for when that happens",
            "and although a scene can be linked to the beginning or ending "
            "of any number of other scenes, it can only have a single "
            "condition such as 'when the player is in the Dining Car' "
            "to trigger it from outside the scene machinery.");
        return;
    }
    sc->anchor_condition[end] = spec2;
    sc->anchor_condition_set[end] = current_sentence;
}
```

§**4. Run-time storage for scenes.** At run-time, we need to store information about the current state of each scene: whether it is currently playing or not, when the last change occurred, and so on. This data is stored in I6 arrays as follows.

First, each scene has a unique ID number, used as an index `X` to these arrays. This ID number is what is stored as an I6 value for the kind of value `scene`, and it agrees with the allocation ID for the I7 scene structure.

`scene_status-->X` is 0 if the scene is not playing, but may do so in future; 1 if the scene is playing; or 2 if the scene is not playing and will never play again.

`scene_started-->X` is the value of `the_time` when the scene last started, or 0 if it has never started.

`scene_ended-->X` is the value of `the_time` when the scene last ended, or 0 if it has never ended. (The "starting" end does not count as ending for this purpose.)

`scene_endings-->X` is a bitmap recording which ends have been used, including bit 1 which records whether the scene has started.

`scene_latest_ending-->X` holds the end number of the most recent ending (or 0 if the scene has never ended).

§**5. Scene-changing machinery at run-time.** We now compile an I6 routine, `DetectSceneChange`, which detects scene changes and runs the appropriate rulebooks when they occur. ■

```
void compile_scene_end_dash(OUTPUT_STREAM, scene *sc, int end) {
    int skip_rules = FALSE;
    if (end == -1) { end = 1; skip_rules = TRUE; }
    if (sc->marker) return;
    if (end >= 2) compile_scene_end_dash(OUT, sc, -1);
    WRITE("if (debug_scenes) print \"[Scene '");
    if (sc->q)
    print_raw_text_to_file(sc->q->word_ref1, sc->q->word_ref2, OUT);
    WRITE("' ");
    if (end == 0) WRITE("begins");
    if (end == 1) WRITE("ends");
    if (end >= 2) { WRITE("ends ");
        print_raw_text_to_file(sc->end_names_w1[end], sc->end_names_w2[end], OUT);
    }
    WRITE("]^\";\n");
    if (end == 0)
        WRITE("scene_status-->%d = 1; ", sc->allocation_id);
    else {
        WRITE("if (PropertyKOV(KOVP_%d, 0, %d)) scene_status-->%d = 0; ",
            SCENE_TY, sc->allocation_id+1, sc->allocation_id);
        WRITE("else scene_status-->%d = 2; ",
            sc->allocation_id);
    }
    if (skip_rules == FALSE)
        WRITE("ProcessRulebook(%d);\n",
            sc->end_rulebook[end]->allocation_id);
    else
        WRITE("\n");
    if (end == 0)
        WRITE("scene_started-->%d = the_time;\n",
            sc->allocation_id);
    else
```

```
        WRITE("scene_ended-->%d = the_time;\n",
            sc->allocation_id);
    WRITE("scene_endings-->%d = (scene_endings-->%d)|%d;\n",
        sc->allocation_id, sc->allocation_id, 1 << end);
    WRITE("scene_latest_ending-->%d = %d;\n",
        sc->allocation_id, end);
    sc->marker = TRUE;
    if (skip_rules == FALSE)
        compile_consequent_ends(OUT, sc, end);
    if (end >= 2) {
        WRITE("ProcessRulebook(%d);\n",
            sc->end_rulebook[1]->allocation_id);
        compile_consequent_ends(OUT, sc, 1);
    }
}

void compile_consequent_ends(OUTPUT_STREAM, scene *sc, int end) {
    int i; scene *sc2; scene_connector *scon;
    LOOP_OVER(sc2, scene)
        for (i=0; i<sc2->no_ends; i++)
            for (scon = sc2->anchor_scene[i]; scon; scon = scon->next)
                if ((scon->connect_to == sc) && (scon->end == end)) {
                    if (i >= 1)
                    WRITE("if (scene_status-->%d == 1) {\n",
                        sc2->allocation_id);
                    else
                    WRITE("if (scene_status-->%d == 0) {\n",
                        sc2->allocation_id);
                    INDENT;
                    compile_scene_end_dash(OUT, sc2, i);
                    OUTDENT; WRITE("}\n");
                }
}

void compile_scene_end(OUTPUT_STREAM, scene *sc, int end) {
    scene *sc2;
    LOOP_OVER(sc2, scene) sc2->marker = FALSE;
    compile_scene_end_dash(OUT, sc, end);
}

void test_scene_end(OUTPUT_STREAM, scene *sc, int end) {
    specification *spec = sc->anchor_condition[end];
    specification *cond_type = new_generic_CONDITION_type();
    if (spec) {
        current_sentence = sc->anchor_condition_set[end];
        if (spec_is_UNKNOWN(spec)) {
            spec = parse_expression(spec->word_ref1, spec->word_ref2, CONDITION_EXPCON);
            sc->anchor_condition[end] = spec;
        }
        if (typecheck(spec, cond_type) == NEVER_MATCH) {
            LOG("Condition: $S\n", spec);
            sentence_problem(_P_(C9ScenesBadCondition),
                "'begins when' and 'ends when' must be followed by a condition",
                "which this does not seem to be.");
        } else {
            WRITE("if (");
```

```
            current_sentence = sc->anchor_condition_set[end];
            spec_compile(OUT, spec);
            WRITE(") {\n"); INDENT;
            WRITE("ch++;\n");
            compile_scene_end(OUT, sc, end);
            WRITE("jump CScene;\n");
            OUTDENT; WRITE("}\n");
        }
    }
}
void compile_DetectSceneChange_routine(OUTPUT_STREAM) {
    scene *sc;
    int ix, i;
    phsf_create_nonphrase_stack_frame();
    we_need_scene_locals();
    OUT = begin_compiling_phrase(OUT);
    begin_code_blocks();
    INDENT;
    LOOP_OVER(sc, scene) {
        ix = sc->allocation_id;
        WRITE("if (scene_status-->%d == 1) {\n", ix); INDENT;
        for (i=sc->no_ends-1; i>=1; i--) test_scene_end(OUT, sc, i);
        OUTDENT; WRITE("}\n");
        WRITE("if (scene_status-->%d == 0) {\n", ix); INDENT;
        if (sc->start_of_play) {
            WRITE("if (((scene_endings-->%d) & 1) == 0) {\n", ix); INDENT;
            compile_scene_end(OUT, sc, 0);
            OUTDENT; WRITE("}\n");
        }
        test_scene_end(OUT, sc, 0);
        OUTDENT; WRITE("}\n");
    }
    WRITE(".CScene;\n");
    WRITE("if (chs>10) \">--> The scene change machinery is stuck.\";\n\n");
    WRITE("if (ch>0) DetectSceneChange(++chs);\n\n");
    OUTDENT; WRITE("];\n");
    OUT = write_routine_header();
    WRITE("[ DetectSceneChange ");
    copy_compiled_phrase();
    end_code_blocks();
    phsf_remove_nonphrase_stack_frame();
}
void compile_PrintSceneName_routine(OUTPUT_STREAM) {
    scene *sc;
    int ix;
    WRITE("[ PrintSceneName sc;\n"); INDENT;
    WRITE("switch (sc) {\n"); INDENT;
    LOOP_OVER(sc, scene) {
        ix = sc->allocation_id;
        WRITE("%d: print \"", ix+1);
        print_raw_text_to_file(sc->q->word_ref1, sc->q->word_ref2, OUT);
        WRITE("\";\n");
    }
```

```
    WRITE("default: print \"<no-such-scene>\";\n");
    OUTDENT; WRITE("}\n");
    OUTDENT; WRITE("];\n\n");
}

void compile_ShowSceneStatus_routine(OUTPUT_STREAM) {
    scene *sc;
    int ix, end;
    WRITE("[ ShowSceneStatus chs sc ch;\n"); INDENT;
    LOOP_OVER(sc, scene) {
        ix = sc->allocation_id;
        WRITE("if (scene_status-->%d == 1) {\n", ix); INDENT;
        WRITE("print \"Scene '");
        print_raw_text_to_file(sc->q->word_ref1, sc->q->word_ref2, OUT);
        WRITE("' playing (for \", the_time-(scene_started-->%d), \" mins now)^\";\n",
            ix);
        OUTDENT; WRITE("} else {\n"); INDENT;
        WRITE("if (scene_latest_ending-->%d > 0) {\n", ix); INDENT;
        WRITE("print \"Scene '");
        print_raw_text_to_file(sc->q->word_ref1, sc->q->word_ref2, OUT);
        WRITE("' ended \";\n");
        if (sc->no_ends > 2) {
            WRITE("switch(scene_latest_ending-->%d) {\n", ix); INDENT;
            for (end=2; end<sc->no_ends; end++) {
                WRITE("%d: print \"", end);
                print_raw_text_to_file(sc->end_names_w1[end], sc->end_names_w2[end], OUT);
                WRITE("\";\n");
            }
            OUTDENT; WRITE("}\n");
        }
        WRITE("print \"^\";\n");
        OUTDENT; WRITE("}\n");
        OUTDENT; WRITE("}\n");
    }
    OUTDENT; WRITE("];\n\n");
}

void compile_scene_recurs_array(OUTPUT_STREAM) {
    scene *sc;
    WRITE("Array scene_recurs -> ");
    LOOP_OVER(sc, scene) {
        WRITE("%d ", (sc->once_only)?0:1);
    }
    WRITE("0 0;\n");
    note_VM_usage("scene", -1, -1, NULL, 5, 1, TRUE);
}
```

The function compile_DetectSceneChange_routine is invoked by a command in a `.i6t` template file.
The function compile_PrintSceneName_routine is invoked by a command in a `.i6t` template file.
The function compile_ShowSceneStatus_routine is invoked by a command in a `.i6t` template file.
The function compile_scene_recurs_array is invoked by a command in a `.i6t` template file.

World Objects 9/wo

Purpose

Who is World, I hear you cry, and to what does he object? Just my little joke. This section contains code to create, uncreate and parse the names of world objects, which are objects or kinds of objects used in the model of the physical world.

9/wo.¶3-0 Special world objects; §2 Access to details; §3 Namelessness; §4-6 Parsing names of world objects; §7 The debugging of knowledge

---

Definitions

¶1. At this point, we have essentially disposed of the original source text altogether. As we read it in, we converted it to a set of distinct objects, together with properties which they might or might not have. For each object, a list of inferences was drawn up about its spatial relationship to other objects, the values of its properties, and so on. Although we rejected obvious contradictions and discarded redundancies as we went, we never compared the knowledge about one object with the knowledge about another, and nor did we have to take "what is the most likely thing to have been meant?" decisions. In this chapter, we have to get from that collection of knowledge to a specific and explicit model of the world, in the form of a set of I6 object, class, property and attribute declarations which we can be certain will compile under I6 without error.

Other, more abstract, things were also created from time to time in the assertions-maker, in particular variables and tables: those are deferred to the next chapter.

¶2. A world object is a physical thing, room, direction or region, or a kind of any of those. We think of it as part of the model world, in a way that a kind of value is not.

World objects are originally created during assertion parsing, when sentences like "A quick brown fox is in the undergrowth" are read. At this stage, the data structure below is only very sketchily filled in, and indeed might never be finished. Because the assertion-maker sometimes jumps the gun by creating objects too soon, we also need the ability to destroy them again if it turned out that the initially plausible source text did not refer to something physical after all.

Later, when all assertions have been turned into inferences, the known facts are weighed to work out the spatial relationships between world objects, and also to finish off deciding what kinds they have. The world objects are then assembled in an object tree in the usual interactive fiction manner, and compiled into I6 object or class directives.

At this point, NI is usually only about half done in compiling a source text (all the phrases and rules must still be made), and the data structure goes on being useful. It contains very many fields used temporarily in different contexts, and consumes a good deal of storage per object, but there are too few objects to make any drastic economies meaningful.

```
typedef struct world_object {
    int word_ref1, word_ref2;                          my name, if I have one
    struct parse_node *creating_sentence;              under which assertion sentence
    struct heading *under_what_heading;                in turn under what heading
    struct world_object *next_under_heading;           next in the list under that
    struct parse_node *creator_plural;                 where plural name specified, if it was
    struct excerpt_meaning *principal_meaning;         main singular excerpt meaning this
    struct generalisation *generalisation_list;        assembly instructions, if any
    int currently_making_instance;
    int propername;                                    has a proper name, such as "Jane"
```

```
int pluralname;                                    has a plural name, such as "marbles"
int article_is_the;                                uses "the" as its indefinite article
struct world_object *named_after;                  name derived from another: e.g. "Jane's nose"
int named_after_w1, named_after_w2;                text of the derived part, e.g. "nose"
int nameless;                                      source text name not to be used for parsing at run-time

struct grammar_verb *understand_as_this_object;    grammar for parsing the name at run-time

int search_score;                                  used when searching world objects to parse names in NI
struct world_object *next_to_search;               similarly

struct world_object *kind;                         what kind of object am I?
struct parse_node *kind_set_at;                    and where in the parse tree does it say?
int room_flag;                                     am I a room?
int door_flag;                                     am I a door?
int kind_flag;                                     am I myself a kind?
int instance_count;                                if so, how many instances do I have? (used in indexing)
int instance_list_length;                          and how long is my optimised loop list? (if any)
char *wo_documentation_symbol;                     where I am documented in the manual (if I am)

struct inference *knowledge;                       list of inferences: what is contingently true
struct implication *implies;                       list of implications: what is necessarily true
struct specification *quoted_appearance;           quoted text describing me
struct parse_node *quoted_appearance_sentence;     where it was
int initially_carried;                             am I initially carried by the player character?
int defines_printed_name;                          does the source specify my "printed name"?
int defines_plural_name;                           does the source specify my "printed plural name"?
int defines_description;                           does the source specify my "description"?

struct world_object *object_tree_parent;           in/on/worn by/carried by tree structure
struct world_object *object_tree_child;
struct world_object *object_tree_sibling;
struct world_object *part_of;                      part-of tree structure
struct world_object *first_part;
struct world_object *next_part;
struct world_object *in_region;                    smallest region containing me (rooms only)
int here_flag;                                     declared simply as being "here"

struct world_object *next_in_I6_order;             linked list in order of declaration in I6
int I6_definition_depth;                           i.e., how many arrows -> appear in its I6 header

struct world_object *map_connection_a;             temporary: used to place doors
struct world_object *map_connection_b;
struct world_object *map_connection_c;
struct world_object *map_direction_a;
struct world_object *map_direction_b;
struct world_object *map_direction_c;

int index_appearances;                             count of mentions in the World index
char *world_index_description;                     notations such as "carried" or "part"
int rough_array_memory_used;                       in words, not bytes; for kinds only

int direction_index;                               directions only: counts 0, 1, 2, ..., in order of creation
struct world_object *exits[MAX_DIRECTIONS];        temporary: for World index (rooms only)
int lock_exit;
struct world_object *lock_exit_to;
int grid_x, grid_y, grid_z;
struct map_component *component;
char *world_index_colour;                          an HTML colour for the room square (rooms only)

struct map_parameter_scope local_map_parameters;   temporary: used in EPS mapping
```

```
    int eps_x, eps_y;
    char wo_I6_identifier[32];                    Name to be used in Inform 6 output
    char direction_property[32];
    int rightc, leftc;                            temporary: used in constructing relation arrays
    int relation_indices[2];
    struct inference *right_infs[2];
    struct kind_of_value *kovko_of_this;          cached result of kovko(W)
    MEMORY_MANAGEMENT
} world_object;
```

The structure world_object is shared with 2/index, 2/lexi, 2/prob2, 2/prob3, 4/edoc, 4/head, 5/rel, 5/conj, 5/aph, 5/parse, 5/mlc, 6/asp, 7/kov, 7/kix, 7/vasp, 7/stsp, 7/cosp, 7/tc, 8/sob, 8/refpt, 8/creat, 8/mass, 8/assem, 8/knowc, 8/relv, 8/imp, 9/kind, 9/spabp, 9/pp, 9/inf, 9/model, 9/cot, 9/inpw, 9/map, 10/list, 10/tab, 11/ap, 13/gv and 13/gpr.

¶3. Special world objects. One room is special in being where play begins, one object as being where the player starts (not necessarily a room: it might be, say, a barber's chair inside a room) and a third object as being the player character, the one with printed name `"yourself"`:

```
world_object *start_room = NULL;
world_object *start_object = NULL;
world_object *wo_yourself = NULL;
world_object *wo_up_direction = NULL;
world_object *wo_down_direction = NULL;
```

---

§**1.** World object creation is complicated by two factors: first, that all world objects are stored in a linked "search list" giving the priority of name-searching when parsing text – a complication because each object creation means having to update this list; and second, world objects are frequently brought into being only for a short time before being destroyed again.

Note that we require the first 10 kinds created to be the right things. That all happens in the Standard Rules: if the Standard Rules ever reorder these kind creations, disaster will ensue. Once these basic kinds are created, we can revise the binary predicate definitions. Some of these BPs contain references saying that their terms must be objects of these basic kinds – but up to now, it has been impossible to use a pointer to the relevant kind in a BP structure, because that kind did not exist. The revision process makes that good.

```
int no_wos_now_existing = 0, no_wo_creations = 0;        different because of deletions
world_object *latest_world_object = NULL;
world_object *get_latest_world_object(void) {
    return latest_world_object;
}
int wo_get_no_wos_now_existing(void) {
    return no_wos_now_existing;
}
world_object *create_world_object(int w1, int w2, int a_kind) {
    PROTECTED_MODEL_PROCEDURE;
    world_object *new_object = CREATE(world_object);
    int i;
    hd_disturb_headings();
    new_object->kind = NULL;
```

```
switch(no_wo_creations++) {
    case 0: kind_kind = new_object; new_object->kind = kind_kind; break;
    case 1: kind_room = new_object; break;
    case 2: kind_thing = new_object; break;
    case 3: kind_direction = new_object; break;
    case 4: kind_door = new_object; break;
    case 5: kind_container = new_object; break;
    case 6: kind_supporter = new_object; break;
    case 7: kind_backdrop = new_object; break;
    case 8: kind_person = new_object; break;
    case 9: kind_region = new_object;
        write_kind_references_into_BPs();                    now that all basic kinds exist
        break;
}
no_wos_now_existing++;

new_object->object_tree_parent = NULL;
new_object->object_tree_child = NULL;
new_object->object_tree_sibling = NULL;
new_object->first_part = NULL;
new_object->next_part = NULL;
new_object->part_of = NULL;
new_object->world_index_description = NULL;

new_object->I6_definition_depth = 0;
new_object->next_in_I6_order = NULL;

new_object->kind_set_at = NULL;
new_object->kind_flag = a_kind;
new_object->instance_count = 0;
new_object->instance_list_length = 0;

new_object->principal_meaning = NULL;
new_object->word_ref1 = w1;
new_object->word_ref2 = w2;
new_object->creator_plural = NULL;
new_object->pluralname = FALSE;
new_object->propername = FALSE;
new_object->defines_printed_name = FALSE;
new_object->defines_plural_name = FALSE;
new_object->named_after = NULL;
new_object->named_after_w1 = -1; new_object->named_after_w2 = -1;
new_object->nameless = NOT_APPLICABLE;

if (w1 >= 0) register_wo_names(new_object, w1, w2);

new_object->room_flag = FALSE;
new_object->door_flag = FALSE;
new_object->map_connection_a = NULL;
new_object->map_connection_b = NULL;
new_object->map_connection_c = NULL;
new_object->map_direction_a = NULL;
new_object->map_direction_b = NULL;
new_object->map_direction_c = NULL;
new_object->in_region = NULL;
new_object->world_index_colour = NULL;
new_object->quoted_appearance = NULL;
new_object->quoted_appearance_sentence = NULL;
```

```
    new_object->defines_description = FALSE;
    new_object->generalisation_list = NULL;
    new_object->currently_making_instance = FALSE;
    new_object->under_what_heading = NULL;
    new_object->next_under_heading = NULL;
    new_object->next_to_search = NULL;
    new_object->article_is_the = FALSE;
    new_object->creating_sentence = current_sentence;
    new_object->understand_as_this_object = NULL;
    new_object->direction_index = -1;
    for (i=0; i<MAX_DIRECTIONS; i++) new_object->exits[i] = NULL;
    new_object->lock_exit = -1;
    new_object->lock_exit_to = NULL;
    new_object->grid_x = 0;
    new_object->grid_y = 0;
    new_object->grid_z = 0;
    new_object->component = 0;
    new_object->index_appearances = 0;
    new_object->wo_documentation_symbol = NULL;
    new_object->here_flag = FALSE;
    new_object->kovko_of_this = NULL;
    prepare_map_parameter_scope(&(new_object->local_map_parameters));
    latest_world_object = new_object;
    if (new_object != kind_kind) attach_wo_to_heading(new_object);
    verify_heading_divisions();
    LOGIF(OBJECT_CREATIONS, "Created object: $O\n", new_object);
    return new_object;
}
```

The function get_latest_world_object is called from 8/creat.
The function wo_get_no_wos_now_existing is called from 4/head.
The function create_world_object is called from 6/asp.

§2. Access to details.

```
parse_node *wo_get_creating_sentence(world_object *wo) {
    return wo->creating_sentence;
}
char *wo_get_I6_representation(world_object *wo) {
    if (wo == NULL) return "nothing";
    return wo->wo_I6_identifier;
}
void wo_set_I6_representation(world_object *wo, char *new) {
    int i, trim = FALSE;
    char *p = wo->wo_I6_identifier;
    if (*new == '"') { new++; trim = TRUE; }
    for (i=0; (i<31) && (new[i]); i++) p[i] = new[i]; p[i] = 0;
    if ((trim) && (p[0])) p[strlen(p)-1] = 0;
}
excerpt_meaning *wo_get_principal_meaning(world_object *wo) {
    return wo->principal_meaning;
}
```

The function wo_get_I6_representation is called from 5/bp, 5/rel, 5/aph, 6/atoms, 7/kov, 7/vasp, 9/pp, 9/inf, 9/cot, 9/inpw, 11/ap, 12/cinv, 13/gtok, 13/test and 14/i6t.
The function wo_set_I6_representation is called from 9/model.
The function wo_get_principal_meaning is called from 5/ml.

§3. **Namelessness.** This is for objects whose names in the source text won't be compiled into I6 `name` properties, so that they won't ordinarily be referred to by the player at run-time.

```
int wo_is_nameless(world_object *wo) {
    while (wo) {
        if (wo == kind_kind) break;
        if (wo->nameless != NOT_APPLICABLE) return wo->nameless;
        wo = wo->kind;
    }
    return FALSE;
}
```

The function wo_is_nameless is called from 9/cot.

§4. **Parsing names of world objects.** The following routine registers the name, and plural name, of a new object. Note that everything gets a plural name: even something palpably unique, like "the Koh-i-Noor diamond". The plural dictionary supports multiple plurals, so there may be any number of names registered: for instance, the kind "person" is registered with plurals "persons" and "people".

```
void register_wo_names(world_object *new_object, int w1, int w2) {
    plural_dictionary_entry *pde = NULL;
    if [[w1, w2 == yourself]] wo_yourself = new_object;
    if (w1 >= 0) {
        int k = 0;
        new_object->principal_meaning = register_excerpt_meaning(
            WORLD_OBJECT_MC, 0, w1, w2, STORE_POINTER_world_object(new_object));
        new_object->creator_plural = new_node(NOUNPHRASE_NT);
        do {
            k++;
            pde = make_plural_of(w1, w2, pde);
            if (plw1 >= 0) {
                LOGIF(PLURALS, "(%d) Reading plural of <$W> as <$W>\n", k,
                    w1, w2, plw1, plw2);
                if (new_object->creator_plural->word_ref1 == -1) {
                    new_object->creator_plural->word_ref1 = plw1;
                    new_object->creator_plural->word_ref2 = plw2;
                    LOGIF(PLURALS, "(%d) Setting plural of <$W> as <$W>\n", k,
                        w1, w2, plw1, plw2);
                }
                register_excerpt_meaning(
                    WORLD_OBJECT_MC, 0, plw1, plw2,
                    STORE_POINTER_world_object(new_object));
            }
        } while (pde);
    }
}

int wo_has_ING_name(world_object *wo) {
```

```
    int i;
    if ((wo == NULL) || (wo->word_ref1 < 0) || (wo->word_ref1 < 0))
        return FALSE;
    for (i = wo->word_ref1; i <= wo->word_ref2; i++)
        if (vocab_test_flags(i, ING_MC)) return TRUE;
    return FALSE;
}
```

The function register_wo_names is called from 8/creat.
The function wo_has_ING_name is called from 11/ap.

§**5.** Note that some WOs are nameless (those with `creator` null), and not all necessarily have plurals, so the set of world object names by no means corresponds to the set of world objects. The actual parsing is done by a method which will be explained later, but the gist of it is to produce a list of all possible matches.

```
world_object *parse_world_object(int w1, int w2, int kinds_only) {
    meaning_list *ml;
    if (w2<0) w2=w1; if (w1<0) return NULL;
    if (is_a_literal(w1, w2)) return NULL;
    ml = SP_excerpt(WORLD_OBJECT_MC, w1, w2);
    if (ml == NULL) return NULL;
    return disambiguate_world_object(ml, kinds_only);
}
```

The function parse_world_object is called from 5/rel, 5/candd, 8/creat, 9/model, 9/map, 10/tab, 13/tfg, 13/test and 14/i6t.

§**6.** The tricky part is to choose from the aforementioned list of possible world objects intended. If the list is empty or contains only one choice, there is no problem. Otherwise we will probably have to reorder the object search list, and then run through it. The code below looks as if it picks out the match with highest score, so that the ordering is unimportant, but in fact the score assigned to a match is based purely on the number of words missed out (see later): that means that ambiguities often arise between two lexically similar objects, e.g., a "blue chair" or a "red chair" when the text simply specifies "chair". Since the code below accepts the *first* object with the highest score, the outcome is thus determined by which of the blue and red chairs ranks highest in the search list: and that is why the search list is so important.

```
world_object *disambiguate_world_object(meaning_list *ml, int kinds_only) {
    meaning_list *ml2;
    world_object *wo, *best_wo = NULL;
    int best_score = 0;
    if (ml == NULL) return NULL;
    if (ml_sideways(ml) == NULL) {
        wo = RETRIEVE_POINTER_world_object(em_data(ml_meaning(ml)));
        if ((kinds_only == FALSE) || (wo->kind_flag)) return wo;
        return NULL;
    }
    construct_wo_search_list();
    LOOP_OVER_WO_SEARCH_LIST(wo) wo->search_score = 0;
    for (ml2 = ml; ml2; ml2 = ml_sideways(ml2)) {
        wo = RETRIEVE_POINTER_world_object(em_data(ml_meaning(ml2)));
        if ((kinds_only == FALSE) || (wo->kind_flag))
```

```
            wo->search_score = ml_match_score(ml2);
    }
    LOOP_OVER_WO_SEARCH_LIST(wo)
        if ((wo->search_score > 0) && (wo->search_score > best_score)) {
            best_wo = wo;
            best_score = wo->search_score;
        }
    return best_wo;
}
```

The function disambiguate_world_object is called from 5/mlc.

§7. The debugging of knowledge.

```
void log_world_object(world_object *wo) {
    if (wo == NULL) { LOG("<null>"); return; }
    if (logging_to_I6_text == FALSE) {
        if (wo->kind_flag) LOG("K"); else LOG("O");
        LOG("%d", wo->allocation_id);
    }
    if (wo->word_ref1 >= 0) {
        LOG("'");
        if (wo->word_ref1 < 0) LOG("<nameless creator>");
        else {
            int i;
            for (i=wo->word_ref1; i<=wo->word_ref2; i++) {
                LOG("%s", lw_array[i].lw_text);
                if (i < wo->word_ref2) LOG(" ");
            }
        }
        LOG("'");
    }
}
void log_knowledge_about_object(world_object *wo) {
    LOG("$O (parent $O)", wo, wo->object_tree_parent);
    if (wo->creator_plural)
        LOG(" (plural $W)", wo->creator_plural->word_ref1, wo->creator_plural->word_ref2);
    LOG("\n");
    if (wo->kind) LOG("  Kind: $O\n", wo->kind);
    if (wo->knowledge) {
        inference *in;
        LOG("  Inferences:\n");
        for (in = wo->knowledge; in; in = in->next) LOG("    $I\n", in);
    }
}
void log_object_tree(void) {
    world_object *wo;
    LOOP_OVER(wo, world_object)
        if (wo->object_tree_parent == NULL)
            log_object_tree_recursively(wo, 0);
}
void log_object_tree_recursively(world_object *wo, int depth) {
```

```
    int i = depth;
    while (i>0) { LOG("  "); i--; }
    LOG("$O\n", wo);
    if (wo->object_tree_child)
        log_object_tree_recursively(wo->object_tree_child, depth+1);
    if (wo->object_tree_sibling)
        log_object_tree_recursively(wo->object_tree_sibling, depth);
}
```

The function log_world_object is called from 2/dl and 9/model.
The function log_knowledge_about_object is called from 2/prob3.
The function log_object_tree is called from 9/model.

Kinds 9/kind

Purpose

To manage kinds of world object, which are analogous to classes in the sense of Inform 6.

9/kind.§5 KOVA cache

Template interpreter commands

```
{-callv:compile_direction_object_constants}
```

Definitions

¶1. There is not very much to say about kinds, which are simply world objects with the `kind_flag` set.

The following kinds are special ones which NI needs to deal with directly: they are defined by the Standard Rules, and these pointers are set as NI notices them arriving. Unlike the special world objects, they cannot be left unset due to neglect, so need not be initialised to `NULL`.

```
world_object *kind_kind = NULL;
world_object *kind_room;
world_object *kind_thing;
world_object *kind_direction;
world_object *kind_door;
world_object *kind_container;
world_object *kind_supporter;
world_object *kind_backdrop;
world_object *kind_person;
world_object *kind_region;
world_object *latest_world_object_made_a_room = NULL;
```

---

§**1.** Every world object has one and only one kind: this routine sets it – which can happen more than once when asserting, since kinds are revised in the light of knowledge. Such revisions are allowed to specialise the kind (e.g., by changing a "person" to a "man") but not to contradict it (e.g., by changing a "supporter" to a "container").

```
int registered_directions = 0;
void set_kind(world_object *wo, world_object *k) {
    PROTECTED_MODEL_PROCEDURE;
    world_object *k2, *old_kind;
    if (wo == NULL) {
        LOG("Tried to set kind to $O\n", k);
        internal_error("Tried to set the kind of a null object");
    }
    if ((!(wo_of_kind(wo, kind_room))) &&
        ((k == kind_room) || (wo_of_kind(k, kind_room)))) {
        latest_world_object_made_a_room = wo;
    }
    if ((wo->kind_set_at != NULL) && (wo->kind != k) && (wo->kind != NULL)) {
```

```
        if (wo_of_kind(wo->kind, k)) return;
        if (wo_of_kind(k, wo->kind) == FALSE) {
            LOG("Tried to set kind of $O to $O\n", wo, k);
            quote_source(1, current_sentence);
            quote_source(2, wo->kind_set_at);
            quote_object(3, k);
            quote_object(4, wo->kind);
            handmade_problem(_P_(C9KindsIncompatible));
            issue_problem_segment(
                "You wrote %1, but that seems to contradict %2, as %3 and %4 "
                "are incompatible. (If %3 were a kind of %4 or vice versa "
                "there'd be no problem, but they aren't.)");
            issue_problem_end();
            return;
        }
    }
    if ((wo->kind != k) && (k == kind_direction)) {
        char i6_identifier_constant[32];
        int w1 = wo->word_ref1, w2 = wo->word_ref2;
        sprintf(i6_identifier_constant, "DirectionObject_%d", registered_directions);
        if (w1 < 0) {
            sentence_problem(_P_(C9NamelessDirection),
                "nameless directions are not allowed",
                "so writing something like 'There is a direction.' is forbidden.");
            return;
        }
        if (w2 > w1+2) {
            sentence_problem(_P_(C9DirectionTooLong),
                "directions may only have names of three or fewer words",
                "so writing something like 'North by north west is a direction.' is "
                "not allowed.");
            return;
        }
        if [[w1, w2 == up]] wo_up_direction = wo;
        if [[w1, w2 == down]] wo_down_direction = wo;
        wo->direction_index = registered_directions++;
        make_mapped_predicate(wo, i6_identifier_constant);
    }
    if (wo == k) {
        LOG("Tried to set kind of $O to itself\n", k);
        internal_error("Tried to set the kind of something to itself");
    }
    old_kind = wo->kind;
    wo->kind = k;
    for (k2 = k; k2 && (k2 != kind_kind) && (k2 != old_kind); k2=k2->kind)
        satisfies_generalisations_about(wo, k2);
    wo->kind_set_at = current_sentence;
    if (k == kind_room) {
        inference *i = create_inference(ISAROOM_INF, CERTAIN_CE);
        join_inference(i, inference_subject_wo(wo));
    }
    LOGIF(KIND_CHANGES, "Setting kind of $O to $O\n", wo, k);
}
```

```
void compile_direction_object_constants(OUTPUT_STREAM) {
    world_object *wo;
    int c = 0;
    WRITE("! Table of direction object alias constants:\n");
    LOOP_OVER(wo, world_object)
        if (wo_of_kind(wo, kind_direction))
            WRITE("Constant DirectionObject_%d = %s;\n", c++, wo->wo_I6_identifier);
}
```

The function set_kind is called from 6/asp.
The function compile_direction_object_constants is invoked by a command in a `.i6t` template file.

§**2.** Detecting whether a given world object is of a given kind involves recursion: if O is of kind "container", then it should pass the test for "thing" as well, since "container" is a kind of "thing". Note that only "kind" is allowed to be its own kind.

```
int wo_of_kind(world_object *wo, world_object *kd) {
    if ((wo == NULL) || (kd == NULL)) return FALSE;
    if (wo->kind == kd) return TRUE;
    if (wo == kind_kind) return FALSE;
    if (wo == wo->kind) {
        LOG("Offending self-kind object is $O\n", wo);
        internal_error("A world object has itself as its kind");
    }
    return wo_of_kind(wo->kind, kd);
}
```

The function wo_of_kind is called from 5/rel, 5/aph, 6/simp, 6/asp, 7/kov, 7/kix, 7/spec, 7/cosp, 8/assem, 8/knowc, 8/relv, 9/spabp, 9/pp, 9/model, 9/cot, 9/inpw, 9/map, 10/list and 11/ap.

§**3.** The number of objects of this kind.

```
int instance_count_for_kind(world_object *wo) {
    if (wo->kind_flag) {
        world_object *wo2;
        int c = 0;
        LOOP_OVER(wo2, world_object) {
            world_object *wo3 = wo2;
            if (wo2->kind_flag) continue;
            while (wo3 != kind_kind) {
                if (wo3->kind == wo) { c++; break; }
                wo3 = wo3->kind;
            }
        }
        return c;
    }
    return 0;
}
```

The function instance_count_for_kind is called from 10/tab and 14/i6t.

§**4.** Directions play a special role because sentences like "east of the treehouse is the garden" are parsed differently from sentences like "the nearby place property of the treehouse is the garden".

```
int is_a_direction_object(world_object *wo) {
    return wo_of_kind(wo, kind_direction);
}
```

The function is_a_direction_object is called from 8/sob, 8/mass and 8/relv.

§**5. KOVA cache.** This is managed by the "Kinds of Value" section, and explained there.

```
kind_of_value **wo_get_kova_cache_location(world_object *k) {
    if (k) return &(k->kovko_of_this);
    return NULL;
}
```

The function wo_get_kova_cache_location is called from 7/kov.

Spatial Relations 9/spabp

Purpose

To define the binary predicates corresponding to basic spatial relationships such as containment, support and being part of.

Definitions

¶1. The spatial relationships can be divided into three:

fundamental spatial relationships

```
binary_predicate *a_contains_b_predicate = NULL;
binary_predicate *a_supports_b_predicate = NULL;
binary_predicate *a_incorporates_b_predicate = NULL;
binary_predicate *a_carries_b_predicate = NULL;
binary_predicate *a_holds_b_predicate = NULL;
binary_predicate *a_wears_b_predicate = NULL;
binary_predicate *a_has_b_predicate = NULL;
```

indirect spatial relationships

```
binary_predicate *a_can_see_b_predicate = NULL;
binary_predicate *a_can_touch_b_predicate = NULL;
binary_predicate *a_conceals_b_predicate = NULL;
binary_predicate *a_encloses_b_predicate = NULL;
```

geographical spatial relationships

```
binary_predicate *a_adjacent_to_b_predicate = NULL;
binary_predicate *a_region_contains_b_predicate = NULL;
```

---

§**1. Initial stock.** These relations are all hard-wired in.

```
void SPATIAL_KBP_create_initial_stock(void) {
    ⟨Make built-in spatial relationships 2⟩;
    ⟨Make built-in indirect spatial relationships 3⟩;
    ⟨Make built-in geographical relationships 4⟩;
}
```

The function SPATIAL_KBP_create_initial_stock is called from 5/bp.

§**2.** Containment, support, incorporation, carrying, holding, wearing and possession. The "loop parent optimisation" is explained elsewhere, but the basic idea is that given a fixed y you can search for all x such that $B(x, y)$ by looking at the object-tree children of y at run-time. On a large work of IF, this cuts the number of cases to check by a factor of 100 or more. (But it can't be used for component parts, since those are not stored in the I6 object tree; nor for the holding relation, since that's a union of the others, and therefore includes incorporation.)

⟨Make built-in spatial relationships 2⟩ ≡

```
a_contains_b_predicate =
    make_pair_of_BPs(SPATIAL_KBP,
        bptd_early_new(-1, sch_new("ContainerOf(*1)")),
        bptd_blank(),
        "contains", "is-in", CONTAINSTHINGS_INF,
        NULL, sch_new("MoveObject(*2,*1)"), NULL,
        containment_V);
a_contains_b_predicate->loop_parent_optimisation_proviso = "ContainerOf";
a_contains_b_predicate->loop_parent_optimisation_ranger = "TestContainmentRange";
a_supports_b_predicate =
    make_pair_of_BPs(SPATIAL_KBP,
        bptd_early_new(kind_supporter_WR, sch_new("SupporterOf(*1)")),
        bptd_early_new(kind_thing_WR, NULL),
        "supports", "is-on", -1,
        NULL, sch_new("MoveObject(*2,*1)"), NULL,
        support_V);
a_supports_b_predicate->loop_parent_optimisation_proviso = "SupporterOf";
a_incorporates_b_predicate =
    make_pair_of_BPs(SPATIAL_KBP,
        bptd_early_new(kind_thing_WR, sch_new("(*1.component_parent)")),
        bptd_early_new(kind_thing_WR, NULL),
        "incorporates", "is-part-of", PARTOF_INF,
        NULL, sch_new("MakePart(*2,*1)"), NULL,
        incorporation_V);
a_carries_b_predicate =
    make_pair_of_BPs(SPATIAL_KBP,
        bptd_early_new(kind_person_WR, sch_new("CarrierOf(*1)")),
        bptd_early_new(kind_thing_WR, NULL),
        "carries", "is-carried-by", -1,
        NULL, sch_new("MoveObject(*2,*1)"), NULL,
        carrying_V);
a_carries_b_predicate->loop_parent_optimisation_proviso = "CarrierOf";
a_holds_b_predicate =
    make_pair_of_BPs(SPATIAL_KBP,
        bptd_early_new(kind_person_WR, sch_new("HolderOf(*1)")),
        bptd_early_new(kind_thing_WR, NULL),
        "holds", "is-held-by", -1,
        NULL, sch_new("MoveObject(*2,*1)"), NULL,
        holding_V);
can't be optimised, because parts are also held
a_wears_b_predicate =
    make_pair_of_BPs(SPATIAL_KBP,
        bptd_early_new(kind_person_WR, sch_new("WearerOf(*1)")),
        bptd_early_new(kind_thing_WR, NULL),
        "wears", "is-worn-by", -1,
        NULL, sch_new("WearObject(*2,*1)"), NULL,
```

```
        wearing_V);
    a_wears_b_predicate->loop_parent_optimisation_proviso = "WearerOf";
    a_has_b_predicate =
        make_pair_of_BPs(SPATIAL_KBP,
            bptd_early_new(-1, sch_new("OwnerOf(*1)")),
            bptd_blank(),
            "has", "is-had-by", -1,
            NULL, sch_new("MoveObject(*2,*1)"), NULL,
            possession_V);
    a_has_b_predicate->loop_parent_optimisation_proviso = "OwnerOf";
```

This code is used in §1.

§**3.** Visibility, touchability, concealment and enclosure: all relations which can be tested at run-time, but which can't be asserted or made true or false.

⟨Make built-in indirect spatial relationships 3⟩ ≡

```
    a_can_see_b_predicate =
        make_pair_of_BPs(SPATIAL_KBP,
            bptd_early_new(kind_thing_WR, NULL),
            bptd_early_new(kind_thing_WR, NULL),
            "can-see", "can-be-seen-by", -1,
            NULL, NULL, sch_new("TestVisibility(*1,*2)"),
            visibility_V);
    a_can_touch_b_predicate =
        make_pair_of_BPs(SPATIAL_KBP,
            bptd_early_new(kind_thing_WR, NULL),
            bptd_early_new(kind_thing_WR, NULL),
            "can-touch", "can-be-touched-by", -1,
            NULL, NULL, sch_new("TestTouchability(*1,*2)"),
            touchability_V);
    a_conceals_b_predicate =
        make_pair_of_BPs(SPATIAL_KBP,
            bptd_early_new(kind_thing_WR, NULL),
            bptd_early_new(kind_thing_WR, NULL),
            "conceals", "is-concealed-by", -1,
            NULL, NULL, sch_new("TestConcealment(*1,*2)"),
            concealment_V);
    a_encloses_b_predicate =
        make_pair_of_BPs(SPATIAL_KBP,
            bptd_early_new(kind_thing_WR, NULL),
            bptd_early_new(kind_thing_WR, NULL),
            "encloses", "is-enclosed-by", -1,
            NULL, NULL, sch_new("IndirectlyContains(*1,*2)"),
            enclosure_V);
```

This code is used in §1.

§4. Adjacency is a simple relation checked by looking at the current map. Regional containment is more subtle, because the meaning of "in" in the S-parser can be either containment or regional containment, depending on context.

⟨Make built-in geographical relationships 4⟩ ≡

```
    a_adjacent_to_b_predicate =
        make_pair_of_BPs(SPATIAL_KBP,
            bptd_early_new(kind_room_WR, NULL),
            bptd_early_new(kind_room_WR, NULL),
            "adjacent-to", "adjacent-from", -1,
            NULL, NULL, sch_new("TestAdjacency(*1,*2)"),
            adjacency_V);
    a_region_contains_b_predicate =
        make_pair_of_BPs(SPATIAL_KBP,
            bptd_early_new(kind_room_WR, NULL),
            bptd_early_new(kind_room_WR, NULL),
            "region-contains", "in-region", -1,
            NULL, NULL, sch_new("TestRegionalContainment(*2,*1)"),
            regional_containment_V);
```

This code is used in §1.

§5. **Second stock.** There is none – this is a family of relations which is all built in.

```
void SPATIAL_KBP_create_second_stock(void) {
}
```

The function SPATIAL_KBP_create_second_stock is called from 5/bp.

§6. Typechecking.

```
int SPATIAL_KBP_typecheck(binary_predicate *bp,
        kind_of_value **kovs_of_terms, kind_of_value **kovs_required, tc_problem_kit *tck) {
    return DECLINE_TO_MATCH;
}
```

The function SPATIAL_KBP_typecheck is called from 5/bp.

§7. Assertion.

```
int SPATIAL_KBP_assert(binary_predicate *bp,
        world_object *wo0, specification *spec0,
        world_object *wo1, specification *spec1) {
    if ((wo0) && (wo1)) {
        place_ox_in_relation_to_oy(bp, wo0, wo1);
        return TRUE;
    }
    return FALSE;
}
```

The function SPATIAL_KBP_assert is called from 5/bp.

§**8.** The next routine is used when we are told that something is in a relationship with something else. From this we infer not merely the relationship, but also some contextual information implied by the fact that it is even possible for such a relationship to exist.

"In" requires delicate handling, because of the way that English uses it sometimes transitively and sometimes not. "The passport is in the desk", "The passport is in the Dining Room" and "The passport is in Venezuela" place the same object in a container, a room or a region respectively. Moreover, while we normally assume that nothing can be in more than one place at once, there is also the exception of "backdrop" objects: intended to represent widely spread, probably background, things, such as the sky. These generate `FOUNDIN_INF` rather than `PARENTAGE_INF` inferences to avoid piling up bogus inconsistencies.

```
void place_ox_in_relation_to_oy(binary_predicate *relation,
    world_object *wo0, world_object *wo1) {
    inference *i;
    int wo1_is_backdrop = FALSE, wo1_is_region = FALSE, wo0_is_region = FALSE;
    if ((wo1) && (wo_of_kind(wo1, kind_backdrop))) wo1_is_backdrop = TRUE;
    if ((wo1) && (wo_of_kind(wo1, kind_region))) wo1_is_region = TRUE;
    if ((wo0) && (wo_of_kind(wo0, kind_region))) wo0_is_region = TRUE;
    ⟨PIRS - Reject non-assertable relations 9⟩;
    ⟨PIR - Relations to regions 10⟩;
    ⟨PIR - Reject parts of rooms 11⟩;
    ⟨PIR - Deal with spatial relations 12⟩;
}
```

§**9.** This is the point at which non-assertable relations are thrown out.

⟨PIRS - Reject non-assertable relations 9⟩ ≡

```
    if ((relation != a_region_contains_b_predicate) &&
        (bp_can_be_made_true_at_runtime(relation) == FALSE)) {
        sentence_problem(_P_(C9Unassertable),
            "the relationship you describe is not exact enough",
            "so that I do not know how to make this assertion come true. "
            "For instance, saying 'The Study is adjacent to the Hallway.' "
            "is not good enough because I need to know in what direction: "
            "is it east of the Hallway, perhaps, or west?");
        return;
    }
```

This code is used in §8.

§**10.** Regions are a special enough case to be worth diverting now.

⟨PIR - Relations to regions 10⟩ ≡

```
    if (wo0_is_region) {
        if ((relation == a_incorporates_b_predicate) ||
            (relation == a_contains_b_predicate) ||
            (relation == a_region_contains_b_predicate)) {
            if (wo1_is_backdrop) {
                world_object *ref1;
                if (wo0) ref1 = wo0; else ref1 = latest_world_object;
                i = create_inference(FOUNDIN_INF, prevailing_mood);
                inf_set_object_references(i, ref1, wo0);
                join_inference(i, inference_subject_wo(wo1));
                return;
            }
            if (wo1_is_region == FALSE) {
                specification *spec = world_object_to_OBJECT_spec(wo0);
                assert_kind_of_object(wo1, kind_room);
                i = create_property_inference(inference_subject_wo(wo1), region_PRN, spec);
                inf_set_object_reference(i, wo0);
                join_inference(i, inference_subject_wo(wo1));
                wo1->in_region = wo0;
                return;
            }
            if ((wo1->in_region) && (wo1->in_region != wo0)) {
                LOG("Here OX is $O and is to be in both $O and $O\n", wo1,
                    wo1->in_region, wo0);
                sentence_problem(_P_(C9RegionInTwoRegions),
                    "each region can only be declared to be inside a single "
                    "other region",
                    "since although regions can be placed inside each other, "
                    "they are not permitted to overlap.");
                return;
            }
            wo1->in_region = wo0;
        } else {
            if (relation != a_region_contains_b_predicate) {
                sentence_problem(_P_(C9RegionRelation),
                    "regions can only contain rooms",
                    "and have no other relationships.");
                return;
            }
        }
    }
```

This code is used in §8.

§**11.** People sometimes try, a little hopefully, to subdivide rooms. Alas for them.

⟨PIR - Reject parts of rooms 11⟩ ≡
```
    if ((relation == a_incorporates_b_predicate) && (wo_of_kind(wo0, kind_room))) {
        sentence_problem(_P_(C9PartOfRoom),
            "this asks to make something a part of a room",
            "when only things are allowed to have parts.");
        return;
    }
```
This code is used in §8.

§**12.** Spatial relations are all essentially ways of saying that X is spatially within Y: perhaps it is inside a cavity of Y, or part of Y, carried by Y, worn by Y, and so on on. These relationships contain contextual information which go beyond what we can deduce for other relations. For instance,

On the desk is a liquorice twist.

generates the following three inferences:

O80⟨chocolatier's desk⟩ - SUPPORTSTHINGS_INF - Certain - 1:O86⟨liquorice twist⟩
O86⟨liquorice twist⟩ - PARENTAGE_INF - Certain - 1:O80⟨chocolatier's desk⟩
O86⟨liquorice twist⟩ - ISAROOM_INF - Impossible

⟨PIR - Deal with spatial relations 12⟩ ≡
```
{ world_object *ref1 = NULL, *ref2 = NULL;
    if (wo1 == wo0) {
        sentence_problem(_P_(C9MiseEnAbyme),
            "this asks to put something inside itself",
            "like saying 'the bottle is in the bottle'. ");
        return;
    }
    if ((relation == a_contains_b_predicate) ||
        (relation == a_supports_b_predicate)){
        if (wo1 != NULL) {
            i = create_inference(PARTOF_INF, IMPOSSIBLE_CE);
            inf_set_object_reference(i, wo0);
            join_inference(i, inference_subject_wo(wo1));
        }
    }
    if (relation == a_incorporates_b_predicate) {
        We now know that X is a component part
        if (wo1 != NULL) {
            i = create_inference(PARTOF_INF, CERTAIN_CE);
            inf_set_object_reference(i, wo0);
            join_inference(i, inference_subject_wo(wo1));
        }
    } else {
        if ((wo1_is_backdrop == FALSE) && (wo0_is_region == FALSE) && (wo0 != NULL)) {
            We now know that Y contains or supports things
            int kind_of_inference = bp_get_kind_of_inference(relation);
            if (kind_of_inference != -1) {
                i = create_inference(kind_of_inference, CERTAIN_CE);
                inf_set_object_reference(i, wo1);
                join_inference(i, inference_subject_wo(wo0));
            }
```

```
        }
    }
    We also know the parentage of X
    if (wo1_is_backdrop) {
        i = create_inference(FOUNDIN_INF, CERTAIN_CE);
        ref2 = wo0;
    } else {
        i = create_inference(PARENTAGE_INF, CERTAIN_CE);
    }
    if (wo0) ref1 = wo0; else ref1 = latest_world_object;
    inf_set_object_references(i, ref1, ref2);
    join_inference(i, inference_subject_wo(wo1));
    And that it cannot be a room
    i = create_inference(ISAROOM_INF, IMPOSSIBLE_CE);
    join_inference(i, inference_subject_wo(wo1));
    if (relation == a_wears_b_predicate) {
        i = create_inference(ISWORN_INF, CERTAIN_CE);
        join_inference(i, inference_subject_wo(wo1));
        if ((wearable_PRN = find_property_translating("wearable")) != NULL) {
            i = create_property_inference(inference_subject_wo(wo1), wearable_PRN, NULL);
            inf_set_certainty(i, LIKELY_CE);
            join_inference(i, inference_subject_wo(wo1));
        }
    }
}
```

This code is used in §8.

§13. **Compilation.** We need do nothing special: these relations can be compiled from their schemas.

```
int SPATIAL_KBP_compile(int task, binary_predicate *bp, annotated_i6_schema *asch) {
    return FALSE;
}
```

The function SPATIAL_KBP_compile is called from 5/bp.

§14. Problem message text.

```
int SPATIAL_KBP_describe_for_problems(binary_predicate *bp, char *problem_buffer) {
    return FALSE;
}
```

The function SPATIAL_KBP_describe_for_problems is called from 5/bp.

Map Connection Relations 9/mapbp

Purpose

To define one binary predicate for each map direction, such as "mapped north of".

Definitions

¶1. This section creates a family of implicit relations (implemented as binary predicates) corresponding to the different directions.

For every direction created, a predicate is created for the possibility of a map connection. For instance, "if Versailles is mapped north of the Metro" tests the "mapped-north" BP.

---

§**1. Initial stock.** There is none, since at the start of Inform's run no direction objects exist yet.

```
void MAP_CONNECTING_KBP_create_initial_stock(void) {
}
```

The function MAP_CONNECTING_KBP_create_initial_stock is called from 5/bp.

§**2. Subsequent creations.** Every direction created has a relation associated with it: for instance, "north" has the relation "X is mapped north of Y". Now a direction is a kind of object, but objects aren't created until after relations used to parse sentences are needed. In fact, however, directions are "noticed" at an earlier stage in Inform's run, so another two-step is needed:

```
binary_predicate *bp_create_sketchy_mapping_direction(int w1, int w2) {
    binary_predicate *bp;
    ⟨Create the mapping BP for the new direction 3⟩;
    return bp;
}
```

The function bp_create_sketchy_mapping_direction is called from 4/ofs.

§**3.** The names of these relations might look odd, for two reasons: (i) why is "mapped" optional in the case of above and below, but not the others? and (ii) why define "to be mapped east of" rather than "to be east of"? After all, that seems to be what is used in assertions like:

The Bakery is east of Pudding Lane.

In fact, the A-parser reads sentences like that by looking out specially for direction names plus "of" – so this is parsed without using the mapping predicate for "east". But it cannot read:

The Flour Cellar is below the Bakery.

as a direction name plus "of", since "below" is not the name of the direction "down", and anyway there is no "of". This is why `mapped_up_predicate` and `mapped_down_predicate` were special cases.

The answer to question (ii) is that the A-parser is better at avoiding ambiguities in assertions like

North of Patio Area is East of Gardens.

We don't want to define the relation "to be east of" to avoid clashing with the A-parser's method. We especially don't want to define "to be inside", because "inside" already has a meaning as containment ("the pearl in inside the jewellery box"). So we include the word "mapped" in the names of these relations, and while that is a little fussy, it avoids all such problems.

⟨Create the mapping BP for the new direction 3⟩ ≡

```
bp_term_details room_term = bptd_blank();
char dln[3*MAX_WORD_LENGTH+10];                 name of BP in debugging log, e.g., "north-map"
vocabulary_entry *rel_name;                     source text name of relation, e.g., "mapping-north"
vocabulary_entry *ve[5];                                          for name of prepositional usage
int i;
⟨Confect the various mapping relation names 4⟩;
bp = make_pair_of_BPs(MAP_CONNECTING_KBP,
    room_term, room_term, dln, NULL, -1, NULL, NULL, NULL, rel_name);
ve[0] = mapped_V;
for (i = 1; i <= w2-w1+1; i++) ve[i] = lw_array[w1+i-1].lw_identity;
if (![[w1, w2 == inside/outside]]) ve[i++] = of_V;
for (; i<5; i++) ve[i] = NULL;
register_pu(ve[0], ve[1], ve[2], ve[3], ve[4], FALSE, bp);
```

This code is used in §2.

§**4.** Some string hackery:

⟨Confect the various mapping relation names 4⟩ ≡

```
int wn = lexer_wordcount, i;
char relname[3*MAX_WORD_LENGTH+10];
print_text_to_string(w1, w2, dln); sprintf(dln + strlen(dln), "-map");
for (i=0; dln[i]; i++) if (dln[i] == ' ') dln[i] = '-';
sprintf(relname, "mapping-"); print_text_to_string(w1, w2, relname + strlen(relname));
for (i=0; relname[i]; i++) if (relname[i] == ' ') relname[i] = '-';
feed_into_lexer(relname, TRUE, FALSE);
rel_name = lw_array[wn].lw_identity;
```

This code is used in §3.

§**5.** And then, later, when the new direction comes into existence (i.e., not when the underlying object `wo` is created, but when its kind is first realised to be "direction"), the following routine is called to fill out the details of the BP. `i6_identifier` can be any string of text which evaluates in I6 to the object number of the direction object. It seems redundant here because surely if we know `wo`, we know its runtime representation; but that's not true – we need to call this routine at a time when the final identifier names for I6 objects have not yet been settled.

```
int mmp_call_counter = 0;
void make_mapped_predicate(world_object *wo, char *i6_identifier) {
    int w1 = wo->word_ref1, w2 = wo->word_ref2;
    if ((w1<0) || (w2 > w1+2)) internal_error("bad direction name");
    complete_mapped_predicate(wo, i6_identifier, relation_noticed(mmp_call_counter++));
}
void complete_mapped_predicate(world_object *wo, char *i6_identifier, binary_predicate *bp)
{
    if (bp == NULL) {
        sentence_problem(_P_(C9ImproperlyMadeDirection),
            "directions must be created by only the simplest possible sentences",
            "in the form 'North-north-west is a direction' only. Using adjectives, "
            "'called', 'which', and so on is not allowed. (In practice this is not "
            "too much of a restriction. I won't allow 'Clockwise is a privately-named "
            "direction.', but I will allow 'Clockwise is a direction. Clockwise "
            "is privately-named.')");
        return;
    }
    bp->term_details[0] = bptd_blank();                    bptd_new(kind_room, NULL);
    bp->term_details[1] = bptd_blank();                    bptd_new(kind_room, NULL);
    bp->test_function = sch_new_1("(MapConnection(*2,%s) == *1)", i6_identifier);
    bp->make_true_function = sch_new_1("AssertMapConnection(*2,%s,*1)", i6_identifier);
    bp->make_false_function = sch_new_1("AssertMapUnconnection(*2,%s,*1)", i6_identifier);
    bp->direction_object = wo;
}
```

The function make_mapped_predicate is called from 9/kind.

§**6. Second stock.** By this time, they all exist; there is nothing to add.

```
void MAP_CONNECTING_KBP_create_second_stock(void) {
}
```

The function MAP_CONNECTING_KBP_create_second_stock is called from 5/bp.

§7. Typechecking.

```
int MAP_CONNECTING_KBP_typecheck(binary_predicate *bp,
        kind_of_value **kovs_of_terms, kind_of_value **kovs_required, tc_problem_kit *tck) {
    int t;
    for (t=0; t<2; t++)
        if ((can_we_cast_kovs(kovs_of_terms[t], kovko(kind_room)) == NEVER_MATCH) &&
            (can_we_cast_kovs(kovs_of_terms[t], kovko(kind_door)) == NEVER_MATCH)) {
        LOG("Term %d is $u but should be a room or door\n", t, kovs_of_terms[t]);
        issue_bp_typecheck_error(bp, kovs_of_terms[0], kovs_of_terms[1], tck);
        return NEVER_MATCH;
    }
    return ALWAYS_MATCH;
}
```

The function MAP_CONNECTING_KBP_typecheck is called from 5/bp.

§8. Assertion.

```
int MAP_CONNECTING_KBP_assert(binary_predicate *bp,
        world_object *wo0, specification *spec0,
        world_object *wo1, specification *spec1) {
    world_object *o_dir = bp_get_mapping_direction(bp);
    world_object *o_from = wo0;
    world_object *o_to = wo1;
    To have exits one must be a room
    inference *i = create_inference(ISAROOM_INF, prevailing_mood);
    join_inference(i, inference_subject_wo(o_from));
    i = create_inference(DIRECTION_INF, prevailing_mood);
    inf_set_object_references(i, o_to, o_dir);
    join_inference(i, inference_subject_wo(o_from));
    return TRUE;
}
```

The function MAP_CONNECTING_KBP_assert is called from 5/bp.

§9. Compilation. We need do nothing special: these relations can be compiled from their schemas.

```
int MAP_CONNECTING_KBP_compile(int task, binary_predicate *bp, annotated_i6_schema *asch) {
    return FALSE;
}
```

The function MAP_CONNECTING_KBP_compile is called from 5/bp.

§10. The correspondence with directions. (Speed really does not matter here.)

```
binary_predicate *bp_for_mapping_direction(world_object *dir) {
    binary_predicate *bp;
    LOOP_OVER(bp, binary_predicate)
        if (bp->direction_object == dir)
            return bp;
    return NULL;
}

world_object *bp_get_mapping_direction(binary_predicate *bp) {
    return bp->direction_object;
}
```

The function bp_for_mapping_direction is called from 8/relv.
The function bp_get_mapping_direction is called from 8/refpt.

§11. Problem message text.

```
int MAP_CONNECTING_KBP_describe_for_problems(binary_predicate *bp, char *problem_buffer) {
    return FALSE;
}
```

The function MAP_CONNECTING_KBP_describe_for_problems is called from 5/bp.

Explicit Relations 9/exrel

Purpose

To draw inferences from the relations created explicitly by the source text.

9/exrel.§1-11 Managing the BPs generated

---

§**1. Managing the BPs generated.** The relations created in this section belong to the EXPLICIT_KBP family, named so because their definitions are explicit in the source text. Initially, there are none.

```
void EXPLICIT_KBP_create_initial_stock(void) {
}
void EXPLICIT_KBP_create_second_stock(void) {
}
```

The function EXPLICIT_KBP_create_initial_stock is called from 5/bp.
The function EXPLICIT_KBP_create_second_stock is called from 5/bp.

§**2.** They typecheck by the default rule only:

```
int EXPLICIT_KBP_typecheck(binary_predicate *bp,
        kind_of_value **kovs_of_terms, kind_of_value **kovs_required, tc_problem_kit *tck) {
    return DECLINE_TO_MATCH;
}
```

The function EXPLICIT_KBP_typecheck is called from 5/bp.

§**3.** They are asserted thus:

```
int EXPLICIT_KBP_assert(binary_predicate *bp,
        world_object *wo0, specification *spec0,
        world_object *wo1, specification *spec1) {
    if ((bp_term_kind_of_value(bp, 0)) || (bp_term_kind_of_value(bp, 1))) {
        inference *i = create_inference(VALUEARBITRARY_INF, prevailing_mood);
        inf_set_all_references(i, spec0, wo0, spec1, wo1);
        inf_set_bp(i, bp);
        join_inference(i, inference_subject_bp(bp));
        return TRUE;
    } else {
        if ((wo0) && (wo1)) {
            nons_place_ox_in_relation_to_oy(bp, wo1, wo0);
            return TRUE;
        }
    }
    return FALSE;
}
```

The function EXPLICIT_KBP_assert is called from 5/bp.

§**4.**

```
void nons_place_ox_in_relation_to_oy(binary_predicate *relation,
    world_object *ox, world_object *oy) {
    inference *i;
    ⟨PIR - Reject non-assertable relations 7⟩;
    ⟨PIR - Deal with equivalence relations 5⟩;
    ⟨PIR - Deal with other non-exclusive relations 6⟩;
    ⟨PIR - Deal with non-spatial relations 8⟩;
}
```

§**5.** Equivalence relations are handled entirely differently from other relations, because of their representation at run-time being so very different.

⟨PIR - Deal with equivalence relations 5⟩ ≡

```
    if ((bp_get_form_of_relation(relation) == Relation_Equiv) && (ox) && (oy)) {
        i = create_inference(MATCHING_INF, prevailing_mood);
        inf_set_property_name(i, bp_get_i6_storage_property(relation));
        inf_set_object_references(i, ox, ox);
        inf_set_bp(i, relation);
        join_inference(i, inference_subject_wo(oy));
        return;
    }
```

This code is used in §4.

§**6.** Now we get rid of other relationships where $B(x, y)$ and $B(x, z)$ are not necessarily contradictory where $y \neq z$. Note that if we have a symmetric various-to-various relation then we need to behave as if $B(y, x)$ had also been asserted whenever $B(x, y)$ has.

⟨PIR - Deal with other non-exclusive relations 6⟩ ≡

```
    if ((bp_allow_arbitrary_assertions(relation)) && (ox) && (oy)) {
        i = create_inference(ARBITRARY_INF, prevailing_mood);
        inf_set_object_references(i, ox, ox);
        inf_set_bp(i, relation);
        join_inference(i, inference_subject_wo(oy));
        if (bp_get_form_of_relation(relation) == Relation_Sym_VtoV) {
            i = create_inference(ARBITRARY_INF, prevailing_mood);
            inf_set_object_references(i, oy, oy);
            inf_set_bp(i, relation);
            join_inference(i, inference_subject_wo(ox));
        }
        return;
    }
```

This code is used in §4.

§**7.** This is the point at which non-assertable relations are thrown out.

⟨PIR - Reject non-assertable relations 7⟩ ≡
```
    if (bp_can_be_made_true_at_runtime(relation) == FALSE) {
        sentence_problem(_P_(C9Unassertable2),
            "the relationship you describe is not exact enough",
            "so that I do not know how to make this assertion come true. "
            "For instance, saying 'The Study is adjacent to the Hallway.' "
            "is not good enough because I need to know in what direction: "
            "is it east of the Hallway, perhaps, or west?");
        return;
    }
```

This code is used in §4.

§**8.** Those relations which involve arbitrary pairs of objects have already been dealt with, so we must now be left with one-to-one, one-to-various, various-to-one, or else spatial relations. We get rid of all but the tricky spatial relations now. Again, note the enforcement of symmetric behaviour: if $B(x, y)$ is asserted then we behave as if $B(y, x)$ had also been asserted.

⟨PIR - Deal with non-spatial relations 8⟩ ≡
```
    if (bp_is_explicit_with_runtime_storage(relation)) {
        infer_property_based_relation(relation, ox, oy);
        if ((bp_get_form_of_relation(relation) == Relation_Sym_OtoO) && (ox != oy))
            infer_property_based_relation(relation, oy, ox);
        return;
    }
```

This code is used in §4.

§**9.** We need do nothing special: these relations can be compiled from their schemas.

```
int EXPLICIT_KBP_compile(int task, binary_predicate *bp, annotated_i6_schema *asch) {
    return FALSE;
}
```

The function EXPLICIT_KBP_compile is called from 5/bp.

§**10.** Problem message text:

```
int EXPLICIT_KBP_describe_for_problems(binary_predicate *bp, char *problem_buffer) {
    return FALSE;
}
```

The function EXPLICIT_KBP_describe_for_problems is called from 5/bp.

§**11.** This routine converts the knowledge that $R(ox, oy)$ into a single inference. It can only be used for a simple subclass of the relations: those which store `oy`, the only thing related to `ox`, in a given property of `ox`. The beauty of this is that the "only thing related to" business is then enforced by the inference mechanism, since an attempt to assert both $R(x, y)$ and $R(x, z)$ will result in contradictory property value inferences for y and z.

```
void infer_property_based_relation(binary_predicate *relation,
    world_object *ox, world_object *oy) {
    if (bp_get_form_of_relation(relation) == Relation_VtoO) {
        world_object *swap=ox; ox=oy; oy=swap;
    }
    specification *spec = world_object_to_OBJECT_spec(oy);
    property_name *prn = bp_get_i6_storage_property(relation);
    inference *i = create_property_inference(inference_subject_wo(ox), prn, spec);
    join_inference(i, inference_subject_wo(ox));
}
```

Properties 9/prop

Purpose

World objects and kinds have "properties" associated with them. Some are either/or, while others hold values: a value-property always has a kind of value it must hold, and type-checking is used to ensure that it always does hold that kind of value. (Inform does not have a boolean type: so an either/or property is not a value property whose value is required to be of boolean type.) In this section of code, we consider "property names" – the properties, such as "open" or "carrying capacity", rather than knowledge about their values in the case of individual objects.

Template interpreter commands

```
17  {-callv:compile_attributes}
18  {-callv:compile_stub_properties}
18  {-routine:CreatePropertyOffsets}
```

Definitions

¶1. A property name is just that: a name given to a property, such as "carrying capacity" or "fixed in place". Property names do not correspond exactly to I6 properties in the compiled code, because some either/or properties have two names: "open" and "closed" refer to the same property, but with different implicit values for it.

Most properties belong to world objects (directions, rooms, things and so on), and are "physical", but some are "abstract" and belong to values. These are in effect the other columns in a table whose first column defines the possibilities for a new kind of value. (An implicit restriction here is that the same name cannot be both a physical and an abstract property.)

Sometimes the name of a property is the same as that of a kind of value. For instance, we might define a kind of value called "weight", and then say that a thing has a weight: that makes a property also called "weight", which is a value property whose value is always a weight.

Finally, some properties are "conditions", in the sense of "What condition is this in?": for instance the sentence

> The cask can be wedged open, bolted closed or stoved in.

sets up a new property as the condition of the cask, whose value must be one of these three named possibilities. (A new kind of value is set up whose three possible values these are.)

```
typedef struct property_name {
    int word_ref1, word_ref2;                          name of property
    struct property_permission *applicable_to;         lists who has permission to have this
    int used_by_i6_library_only;                       is this for private use of the I6 library?
    int shown_in_index;                                is this property shown in the indexes?
    int indexed_flag;                                  and has it been, thus far in index construction?
    int either_or;                                     is this an either/or property?
    int implemented_as_attribute;                      if so: is it an I6 attribute at run-time?
    int ephemeral;                                     and does it flicker on and off in a temporary way?
    struct property_name *negation;                    and which property name (if any) negates it?
```

```
    struct adjective_meaning *adjectival_meaning_registered;          and has it been made an
adjective yet?
    struct adjectival_phrase *adjectival_phrase_registered;                              similarly
    struct grammar_verb *eo_parsing_grammar;                          exotic forms used in parsing
    struct kind_of_value *property_value_kov;      if not either/or, what kind of value does it hold?
    struct binary_predicate *setting_bp;                                  and which relation sets it?
    int also_a_type;                            and is its name the same as that of the kind of value?
    int abstract_flag;                            and is it an abstract rather than physical property?
    int stores_1to1_relation;                          or is it used to implement a 1 to 1 relation?
    struct world_object *condition_of;                                 or is it a condition of an object?
    int condition_anonymously_named;                              if so, is it named just "... condition"?
    int translated;                           translated into an I6 library-defined property/attribute?
    char i6_name[32];                          the I6 name (whether defined by the I6 library or not)
    int table_offset;                    position in the property_metadata word array at run-time
    struct world_object *last_initialised_for;                    temporary use when compiling objects
    struct possession_marker pom;                      for temporary use when checking implications
    MEMORY_MANAGEMENT
} property_name;
```

The structure property_name is private to this section.

¶2. A few properties are treated specially by NI, and need to have their references stored in global variables. They are set as they are created, during the parsing of assertions in the Standard Rules.

```
property_name *wearable_PRN = NULL;                  needed as it interacts with the wearing relation
property_name *Inform_property_PRN = NULL;                  I6 dir_prop, for doors and directions
property_name *specification_PRN = NULL;                      a meta-property for indexing kinds
property_name *proper_named_PRN = NULL;                         an either/or property for names
property_name *plural_named_PRN = NULL;                         an either/or property for names
property_name *region_PRN = NULL;                          an I6 property giving the region of a room
property_name *other_side_PRN = NULL;                      an I6 property for the other side of a door
```

---

§**1.** Two special cases are "I6 direction property" and "I6 direction object", which are used by the Standard Rules as part of the link between I7 and the lower-level I6 library. We keep track of them mainly in order to exclude them from the index, as we don't want the user to see them or think about them.

A different sort of special case is the "specification" of a kind, which is text used in indexing, and not something inherited by its instances in the usual way.

```
property_name *create_property_name(int w1, int w2) {
    property_name *pn;
    specification *spec;
    kind_of_value *data_type;
    if [[w1, w2 == value]] {
        sentence_problem(_P_(BelievedImpossible),
            "the single word 'value' cannot be used as the name of a property",
            "because it has a much broader meaning already. Inform uses the "
            "word 'value' to mean any number, time of day, name of something, "
            "etcetera: and because of that very broadness, Inform cannot decide "
            "what kind of value a simple 'value' might be. So 'A door has "
```

```
        "a value' is not allowed; but 'A door has a number called the "
        "room number' would be fine.");
}
if (suitable_name(w1, w2) != NAME_IS_SUITABLE) {
    quote_source(1, current_sentence);
    quote_words(2, w1, w2);
    handmade_problem(_P_(C9PropertyNameUnsuitable));
    issue_problem_segment(
        "The sentence %1 seems to create a new property called '%2', but "
        "this is not a good name, and I think I must have misread what "
        "you wanted. Maybe the punctuation is wrong?");
    issue_problem_end();
}
pn = CREATE(property_name);
pn->word_ref1 = w1; pn->word_ref2 = w2;
pn->applicable_to = NULL;
pn->adjectival_meaning_registered = NULL;
pn->adjectival_phrase_registered = NULL;
pn->either_or = FALSE;
pn->setting_bp = NULL;
pn->ephemeral = FALSE;
pn->implemented_as_attribute = TRUE;
pn->abstract_flag = FALSE;
pn->condition_of = NULL;
pn->condition_anonymously_named = FALSE;
pn->negation = NULL;
pn->translated = FALSE;
pn->used_by_i6_library_only = FALSE;
pn->indexed_flag = FALSE;
pn->property_value_kov = NULL;
pn->last_initialised_for = NULL;
pn->stores_1to1_relation = FALSE;
pn->shown_in_index = TRUE;
pn->eo_parsing_grammar = NULL;
pn->also_a_type = FALSE;
[[w1, w2 == a/an ... --> w1, w2]];
if [[w1, w2 == matching key]] strcpy(pn->i6_name, "with_key");
else isn_compose_identifier(pn->i6_name, 'p', pn->allocation_id, w1, w2);
int okay = FALSE;
spec = parse_expression(w1, w2, TYPE_EXPCON);
if (spec_is_generic_CONSTANT(spec)) {
        okay = TRUE;
    data_type = spec_get_kind_of_value(spec);
    if ((data_type)
        && (is_kova(data_type, OBJECT_DESCRIPTION_TY) == FALSE)) {
        pn->property_value_kov = data_type;
        pn->also_a_type = TRUE;
        if (kov_name_can_coincide_with_property(pn->property_value_kov))
            make_type_coincident(pn->property_value_kov, pn);
    }
}
if (okay == FALSE) {
```

```
        spec = parse_expression(w1, w2, TYPE_OR_VALUE_EXPCON);
        if (spec_is_actual_CONSTANT_of_kova(spec, TABLE_COLUMN_TY)) okay = TRUE;
        if (spec_is_actual_CONSTANT_of_kova(spec, PROPERTY_TY)) okay = TRUE;
        if (species_is(spec, DESCRIPTION_SPC)) okay = TRUE;
        if (species_is(spec, NONLOCAL_VARIABLE_SPC)) okay = TRUE;
        if ((okay == FALSE) && (spec_is_UNKNOWN(spec) == FALSE)) {
            LOG("Offending SP: $X", spec);
            quote_source(1, current_sentence);
            quote_words(2, w1, w2);
            quote_type_of(3, spec);
            handmade_problem(_P_(C9PropertyNameClash));
            issue_problem_segment(
                "You wrote %1, but '%2' is not free to be the name of a fresh "
                "property: it already has a meaning (as %3).");
            issue_problem_end();
        }
    }
    if [[w1, w2 == specification]] {
        specification_PRN = pn;
        pn->property_value_kov = kova(TEXT_TY);
    }
    if [[w1, w2 == plural_named]] plural_named_PRN = pn;
    if [[w1, w2 == proper_named]] proper_named_PRN = pn;
    if [[w1, w2 == map region]] region_PRN = pn;
    if [[w1, w2 == other side]] other_side_PRN = pn;
    if [[w1, w2 == i6 library direction property]] {
        pn->used_by_i6_library_only = TRUE;
        Inform_property_PRN = pn;
    }
    if [[w1, w2 == initially carried]] pn->used_by_i6_library_only = TRUE;
    if [[w1, w2 == privately_named]] pn->used_by_i6_library_only = TRUE;
    if [[w1, w2 == publically_named]] pn->used_by_i6_library_only = TRUE;
    register_excerpt_meaning(PROPERTY_MC, 0, w1, w2, STORE_POINTER_property_name(pn));
    register_reworded_meaning(PROPERTY_MC, 0, property_V, 0, w1, w2, 0,
        STORE_POINTER_property_name(pn));
    LOGIF(PROPERTY_CREATIONS, "Created property: %s (ubyi6 = %d)\n", pn->i6_name,
        pn->used_by_i6_library_only);
    return pn;
}
void log_property_name(property_name *pn) {
    if (pn == NULL) { LOG("NULL-PROPERTY"); return; }
    LOG("'");
    log_word_range(pn->word_ref1, pn->word_ref2);
    LOG("'");
    if (logging_to_I6_text) return;
    LOG(" - ");
    if (pn->either_or) {
        LOG("CONDITION_FMY");
        if (pn->negation != NULL) {
            LOG("/~'");
            log_word_range(pn->negation->word_ref1,
                pn->negation->word_ref2);
            LOG("'");
```

```
        }
    } else {
        log_kind_of_value(pn->property_value_kov);
    }
    LOG(" -> %s", pn->i6_name);
}

int prn_is_used_by_i6_library_only(property_name *prn) {
    return prn->used_by_i6_library_only;
}
int prn_is_shown_in_index(property_name *prn) {
    return prn->shown_in_index;
}
void prn_exclude_from_index(property_name *prn) {
    prn->shown_in_index = FALSE;
}

void prn_set_indexed_flag(property_name *prn, int spec) {
    prn->indexed_flag = spec;
}
int prn_get_indexed_flag(property_name *prn) {
    return prn->indexed_flag;
}
```

The function log_property_name is called from 2/dl and 9/model.
The function prn_is_used_by_i6_library_only is called from 4/edoc and 9/inpw.
The function prn_is_shown_in_index is called from 4/edoc.
The function prn_exclude_from_index is called from 5/rel.
The function prn_set_indexed_flag is called from 9/inpw.
The function prn_get_indexed_flag is called from 9/inpw.

§**2. Parsing property names.** In almost all cases (but see below) properties are parsed using the standard excerpt parser.

```
property_name *parse_property_name(int w1, int w2) {
    meaning_list *ml;
    ml = SP_excerpt(PROPERTY_MC, w1, w2);
    if (ml == NULL) return NULL;
    return RETRIEVE_POINTER_property_name(em_data(ml_meaning(ml)));
}
```

The function parse_property_name is called from 4/ofs, 6/treec, 7/data, 8/refpt, 9/pp, 9/madj, 9/vpbp, 12/cinv, 13/tfg and 14/i6t.

§**3.** The following asserts that `(w1,w2)` is certainly a property, which might or might not already exist, so must be created if it does not.

```
property_name *value_property_name_ref(int w1, int w2) {
    meaning_list *ml;
    ml = SP_excerpt(PROPERTY_MC, w1, w2);
    property_name *pn;
    if (ml == NULL) {
        pn = create_property_name(w1, w2);
        binary_predicate *bp = find_set_property_BP(w1, w2);
        if (bp == NULL) bp = make_set_property_BP(w1, w2);
        fix_property_bp(bp);
        fix_property_bp(bp_get_reversal(bp));
        pn->setting_bp = bp;
    } else pn = RETRIEVE_POINTER_property_name(em_data(ml_meaning(ml)));
    return pn;
}

property_name *property_name_ref(int w1, int w2) {
    meaning_list *ml;
    ml = SP_excerpt(PROPERTY_MC, w1, w2);
    property_name *pn;
    if (ml == NULL) {
        pn = create_property_name(w1, w2);
    } else pn = RETRIEVE_POINTER_property_name(em_data(ml_meaning(ml)));
    return pn;
}
```

The function value_property_name_ref is called from 4/ofs, 8/mass, 9/pp and 10/tab.

§**4.** Now the same: except that we require the property to have a given data type, if it already exists. We can widen the type if need be, but in the event that such a property already exists and has an incompatible type, we now have the possibility of failure – a problem message and a return value of a null pointer. So this routine does not guarantee success, like the last.

```
property_name *typed_property_name_ref(int w1, int w2, kind_of_value *kov) {
    property_name *prn = parse_property_name(w1, w2);
    if (kov == NULL) kov = kova(OBJECT_TY);
    if (prn) {
        switch(can_we_cast_kovs(kov, prn->property_value_kov)) {
            case ALWAYS_MATCH: break;
            case SOMETIMES_MATCH:
                if (can_we_cast_kovs(prn->property_value_kov, kov)
                    == ALWAYS_MATCH) {
                    prn->property_value_kov = kov;
                    break;
                }
            case NEVER_MATCH:
                sentence_problem(_P_(C9BadKOVForRelationProperty),
                    "that property already exists and contains a kind of value "
                    "incompatible with what we need here",
                    "so you will need to give it a different name.");
                return NULL;
        }
```

```
    } else {
        prn = create_property_name(w1, w2);
        prn->property_value_kov = kov;
    }
    return prn;
}
```

The function typed_property_name_ref is called from 5/rel.

§**5.** The following slow routine, not used very often, finds the length of the longest property name at the start of the excerpt `(w1,w2)`. This information is then used to cut text like "discovery score 13" into a name part ("discovery score") and a value part ("13").

```
int match_longest_pname(int w1, int w2) {
    int j, x1, x2, maxlen = -1;
    property_name *pn;
    LOOP_OVER(pn, property_name) {
        x1 = pn->word_ref1;
        x2 = pn->word_ref2;
        if (x2-x1 <= w2-w1) {
            for (j=0; j<=x2-x1; j++)
            for (j=0; j<=x2-x1; j++)
                if (compare_words(w1+j, x1+j) == FALSE) goto NotThisOne;
            if (maxlen < x2-x1) maxlen = x2-x1+1;
        }
        NotThisOne: ;
    }
    return maxlen;
}
```

The function match_longest_pname is called from 6/treec.

§**6. Properties of objects or values.** Some properties belong to objects, others to values. The default is objects, so we only need routines to change from this.

```
void prn_make_property_of_value(property_name *prn) {
    prn->abstract_flag = TRUE;
}
int prn_belongs_to_value(property_name *prn) {
    return prn->abstract_flag;
}
```

The function prn_make_property_of_value is called from 9/pp.
The function prn_belongs_to_value is called from 7/tc and 9/pp.

§**7. Permissions.** Each property has a list of permissions for its usage attached. These are important enough to have their own section: here, all we do is...

```
property_permission *prn_permission_list(property_name *prn) {
    return prn->applicable_to;
}
void prn_set_permission_list(property_name *prn, property_permission *pp) {
    prn->applicable_to = pp;
}
```

The function prn_permission_list is called from 9/qty, 9/pp and 13/gpr.
The function prn_set_permission_list is called from 9/pp.

§**8. Either/or properties.** A property name structure with the either/or flag set represents one which is true or false, but there is an additional oddity. "Open" and "closed", for instance, refer to two different property name structures, even though they are arguably better regarded as the names of the two possible mutually exclusive values of a single property. The `negation` field of such a property name refers to its antonym, so for "open" it points to "closed" and vice versa. (All the same, note that some either/or properties do not have explicit antonyms, so it should not be assumed that either/or property name structures always occur in pairs.)

The following routine, at any rate, makes an existing property name structure usable as either/or, registering its name as one which can be used adjectivally ("an open door"): it will of course already be registered as one which can be used as a noun in the context of properties.

```
property_name *prn_either_or_new(int w1, int w2, kind_of_value *kov) {
    property_name *prn;
    prn = property_name_ref(w1, w2);
    prn->either_or = TRUE;
    if ((kov == NULL) && (prn->adjectival_meaning_registered == NULL)) {
        adjective_meaning *am = am_new(EORP_KADJ, STORE_POINTER_property_name(prn),
            w1, w2);
        am_declare(am, w1, w2);
        am_set_domain_from_kov(am, kova(OBJECT_TY));
        prn->adjectival_phrase_registered = am_get_aph(am);
        prn->adjectival_meaning_registered = am;
    }
    if (kov) {
        adjective_meaning *am = am_new(EORP_KADJ, STORE_POINTER_property_name(prn),
            w1, w2);
        am_declare(am, w1, w2);
        am_set_domain_from_kov(am, kov);
        prn->adjectival_phrase_registered = am_get_aph(am);
    }
    return prn;
}
adjective_meaning *EORP_KADJ_parse(parse_node *q,
    int sense,
    int adj_name_w1, int adj_name_w2,
    int dom_name_w1, int dom_name_w2,
    int cond_w1, int cond_w2,
    int called_w1, int called_w2) {
    return NULL;
}
```

The function prn_either_or_new is called from 9/pp.
The function EORP_KADJ_parse is called from 5/aph.

§**9.** There are actually only two run-time implementations for either/or properties – as I6 attributes, and (because there are not many I6 attributes to play with) as I6 properties whose contents are always true or false – but it looks like four implementations because of the wrinkle that either/or properties tend to come in antonymical pairs, only one of which is actually stored at run-time. (Thus, "closed" is the absence of being "open".) Anyway, here are the four combinations.

```
void EORP_KADJ_compiling_soon(adjective_meaning *am, property_name *prn, int T) {
    if (am == NULL) internal_error("Unregistered adjectival either/or property in either/or
atom");
    if (am_get_ready_flag(am)) return;
    am_set_ready_flag(am);
    kind_of_value *kov = am_get_domain(am);
    if (is_kova(kov, OBJECT_TY) == FALSE) {
        if (prn_either_or_stored_in_negation(prn)) {
            property_name *prnbar = prn_either_or_get_negation(prn);
            int offset = pp_get_prn_offset_in_kov(prnbar, kov);
            int id = kov_I6_ID(kov);
            i6_schema *sch = am_set_i6_schema(am, TEST_ADJECTIVE_TASK, FALSE);
            sch_write_to_existing_2d(sch, "PropertyKOV(KOVP_%d, %d, *1) == false", id, offset);
            sch = am_set_i6_schema(am, NOW_ADJECTIVE_TRUE_TASK, FALSE);
            sch_write_to_existing_2d(sch, "WritePropertyKOV(KOVP_%d, %d, *1)", id, offset);
            sch = am_set_i6_schema(am, NOW_ADJECTIVE_FALSE_TASK, FALSE);
            sch_write_to_existing_2d(sch, "WritePropertyKOV(KOVP_%d, %d, *1, true)", id, offset);
        } else {
            int offset = pp_get_prn_offset_in_kov(prn, kov);
            int id = kov_I6_ID(kov);
            i6_schema *sch = am_set_i6_schema(am, TEST_ADJECTIVE_TASK, FALSE);
            sch_write_to_existing_2d(sch, "PropertyKOV(KOVP_%d, %d, *1)", id, offset);
            sch = am_set_i6_schema(am, NOW_ADJECTIVE_TRUE_TASK, FALSE);
            sch_write_to_existing_2d(sch, "WritePropertyKOV(KOVP_%d, %d, *1, true)", id, offset);
            sch = am_set_i6_schema(am, NOW_ADJECTIVE_FALSE_TASK, FALSE);
            sch_write_to_existing_2d(sch, "WritePropertyKOV(KOVP_%d, %d, *1)", id, offset);
        }
        return;
    }
    i6_schema *sch = am_set_i6_schema(am, TEST_ADJECTIVE_TASK, FALSE);
    if (prn_either_or_implemented_as_attribute(prn)) {
        if (prn_either_or_stored_in_negation(prn)) {
            property_name *prnbar = prn_either_or_get_negation(prn);
            char *identifier = prn_get_i6_identifier(prnbar);
            sch_write_to_existing_1(sch, "GetEitherOrProperty(*1,%s)==false", identifier);
        } else {
            char *identifier = prn_get_i6_identifier(prn);
            sch_write_to_existing_1(sch, "GetEitherOrProperty(*1,%s)", identifier);
        }
    } else {
        if (prn_either_or_stored_in_negation(prn)) {
            property_name *prnbar = prn_either_or_get_negation(prn);
            char *identifier = prn_get_i6_identifier(prnbar);
```

```
            sch_write_to_existing_1(sch, "GetEitherOrProperty(*1,%s)==false", identifier);
        } else {
            char *identifier = prn_get_i6_identifier(prn);
            sch_write_to_existing_1(sch, "GetEitherOrProperty(*1,%s)", identifier);
        }
    }
    sch = am_set_i6_schema(am, NOW_ADJECTIVE_TRUE_TASK, FALSE);
    if (prn_either_or_implemented_as_attribute(prn)) {
        if (prn_either_or_stored_in_negation(prn)) {
            property_name *prnbar = prn_either_or_get_negation(prn);
            char *identifier = prn_get_i6_identifier(prnbar);
            sch_write_to_existing_1(sch, "SetEitherOrProperty(*1,%s,true)", identifier);
        } else {
            char *identifier = prn_get_i6_identifier(prn);
            sch_write_to_existing_1(sch, "SetEitherOrProperty(*1,%s)", identifier);
        }
    } else {
        if (prn_either_or_stored_in_negation(prn)) {
            property_name *prnbar = prn_either_or_get_negation(prn);
            char *identifier = prn_get_i6_identifier(prnbar);
            sch_write_to_existing_1(sch, "SetEitherOrProperty(*1,%s,true)", identifier);
        } else {
            char *identifier = prn_get_i6_identifier(prn);
            sch_write_to_existing_1(sch, "SetEitherOrProperty(*1,%s)", identifier);
        }
    }
    sch = am_set_i6_schema(am, NOW_ADJECTIVE_FALSE_TASK, FALSE);
    if (prn_either_or_implemented_as_attribute(prn)) {
        if (prn_either_or_stored_in_negation(prn)) {
            property_name *prnbar = prn_either_or_get_negation(prn);
            char *identifier = prn_get_i6_identifier(prnbar);
            sch_write_to_existing_1(sch, "SetEitherOrProperty(*1,%s)", identifier);
        } else {
            char *identifier = prn_get_i6_identifier(prn);
            sch_write_to_existing_1(sch, "SetEitherOrProperty(*1,%s,true)", identifier);
        }
    } else {
        if (prn_either_or_stored_in_negation(prn)) {
            property_name *prnbar = prn_either_or_get_negation(prn);
            char *identifier = prn_get_i6_identifier(prnbar);
            sch_write_to_existing_1(sch, "SetEitherOrProperty(*1,%s)", identifier);
        } else {
            char *identifier = prn_get_i6_identifier(prn);
            sch_write_to_existing_1(sch, "SetEitherOrProperty(*1,%s,true)", identifier);
        }
    }
}
int EORP_KADJ_compile(property_name *prn, int T, OUTPUT_STREAM, ph_stack_frame *phsf) {
    return FALSE;
}
int EORP_KADJ_assert(property_name *prn,
    world_object *wo_to_assert_on, specification *val_to_assert_on, int parity) {
```

```
    inference_subject infs;
    if (wo_to_assert_on) infs = inference_subject_wo(wo_to_assert_on);
    else infs = inference_subject_spec(val_to_assert_on);
    inference *i = create_property_inference(infs, prn, NULL);
    if (parity == FALSE) inf_reverse_certainty(i);
    join_inference(i, infs);
    return TRUE;
}
int EORP_KADJ_index(property_name *prn) {
    property_name *prnbar = prn_either_or_get_negation(prn);
    property_permission *pp = prn_permission_list(prn);
    if ((pp == NULL) && (prnbar)) pp = prn_permission_list(prnbar);
    INDEX("either/or property");
    if (pp) { INDEX(" of "); pp_index_applicabilities(pp); }
    if (prnbar) {
        INDEX(", opposite of </i>");
        print_raw_text_to_file(prnbar->word_ref1, prnbar->word_ref2, ifl);
        INDEX("<i>");
    }
    return TRUE;
}
void compile_KOV_properties(OUTPUT_STREAM, kind_of_value *kov) {
    int s = pp_get_prn_offset_in_kov(NULL, kov);
    int i;
    quantity *q;
    property_name *prn_offsets[1024];
    if (s == 0) return;
    if (s > 1024) internal_error("too many KOV properties");
    for (i=0; i<s; i++) prn_offsets[i] = NULL;
    property_name *prn;
    LOOP_OVER(prn, property_name) {
        int j = pp_get_prn_offset_in_kov(prn, kov);
        if (j<s) prn_offsets[j] = prn;
    }
    WRITE("Array KOVP_%d --> %d ", kov_I6_ID(kov), s);
    LOOP_OVER(q, quantity)
        if ((kov_compare(qty_kind_of_value(q), kov)) &&
            (qty_is_a_variable(q) == FALSE)) {
            WRITE("\n! Properties for %s\n    ",
                qty_get_I6_representation(q));
            for (i=0; i<s; i++) {
                property_name *prn = prn_offsets[i];
                if ((compile_inferred_value(OUT, inference_subject_q(q), prn) == FALSE)
                    && (compile_inferred_value(OUT, inference_subject_kov(kov), prn) == FALSE))
{
                    if (prn->either_or) WRITE("0");
                    else compile_pval_default(OUT, prn);
                }
                WRITE(" ");
            }
        }
    WRITE(";\n");
}
```

```
void allocate_heap_for_KOV_properties(OUTPUT_STREAM, kind_of_value *kov) {
    int s = pp_get_prn_offset_in_kov(NULL, kov);
    int i, j;
    quantity *q;
    property_name *prn_offsets[1024];
    if (s == 0) return;
    if (s > 1024) internal_error("too many KOV properties");
    for (i=0; i<s; i++) prn_offsets[i] = NULL;
    property_name *prn;
    LOOP_OVER(prn, property_name) {
        int j = pp_get_prn_offset_in_kov(prn, kov);
        if (j<s) prn_offsets[j] = prn;
    }
    j = 1;
    LOOP_OVER(q, quantity)
        if ((kov_compare(qty_kind_of_value(q), kov)) &&
            (qty_is_a_variable(q) == FALSE)) {
            for (i=0; i<s; i++) {
                property_name *prn = prn_offsets[i];
                if (prn_is_either_or(prn) == FALSE) {
                    kind_of_value *pkov = prn_get_kind_of_value(prn);
                    if (kov_uses_pointer_values(pkov)) {
                        WRITE("pv = KOVP_%d-->%d; ", kov_I6_ID(kov), j);
                        if (kov_has_pointer_value_constants(pkov)) WRITE("if (pv==0) ");
                        WRITE("KOVP_%d-->%d = ", kov_I6_ID(kov), j);
                        compile_heap_allocation(OUT, pkov, 1, TRUE);
                        WRITE(";\n");
                    }
                }
                j++;
            }
        }
}

int prn_is_either_or(property_name *prn) {
    return prn->either_or;
}

int prn_either_or_implemented_as_attribute(property_name *prn) {
    if (prn->either_or == FALSE)
        internal_error("asked if a value property implemented as attribute");
    return prn->implemented_as_attribute;
}

void prn_either_or_make_negations(property_name *prn, property_name *prnbar) {
    if ((prn->either_or == FALSE) || (prnbar->either_or == FALSE))
        internal_error("tried to negate a value property");
    prn->negation = prnbar;
    prnbar->negation = prn;
}

adjectival_phrase *prn_either_or_get_aph(property_name *prn) {
    if (prn->either_or == FALSE)
        internal_error("asked AM for a value property");
    return prn->adjectival_phrase_registered;
}
```

```
int prn_either_or_stored_in_negation(property_name *prn) {
    if (prn->either_or == FALSE)
        internal_error("asked if a value property stored in negation");
    if ((prn->negation) && (prn->negation->translated)) return TRUE;
    return FALSE;
}

property_name *prn_either_or_get_negation(property_name *prn) {
    if (prn->either_or == FALSE)
        internal_error("asked for the negation of a value property");
    return prn->negation;
}

int prn_is_ephemeral(property_name *prn) {
    if (prn->either_or == FALSE) return FALSE;
    return prn->ephemeral;
}

grammar_verb *prn_get_parsing_grammar(property_name *prn) {
    if (prn->either_or == FALSE) return NULL;
    return prn->eo_parsing_grammar;
}

void prn_set_parsing_grammar(property_name *prn, grammar_verb *gv) {
    if (prn->either_or == FALSE)
        internal_error("tried to set parsing grammar for a value property");
    prn->eo_parsing_grammar = gv;
}
```

The function EORP_KADJ_compiling_soon is called from 5/aph.
The function EORP_KADJ_compile is called from 5/aph.
The function EORP_KADJ_assert is called from 5/aph.
The function EORP_KADJ_index is called from 5/aph.
The function compile_KOV_properties is called from 9/rsdt.
The function allocate_heap_for_KOV_properties is called from 9/rsdt.
The function prn_is_either_or is called from 6/pform, 6/treec, 6/sconv, 9/rsdt, 7/vasp, 7/stsp, 7/tc, 8/knowp, 8/imp, 9/qty, 9/pp, 9/vpbp, 9/inf, 9/model, 9/cot, 9/inpw, 13/tfg and 13/gpr.
The function prn_either_or_implemented_as_attribute is called from 7/vasp and 9/pp.
The function prn_either_or_make_negations is called from 9/pp.
The function prn_either_or_get_aph is called from 6/pform, 6/sconv and 8/imp.
The function prn_either_or_stored_in_negation is called from 7/vasp and 9/pp.
The function prn_either_or_get_negation is called from 7/vasp, 8/imp, 9/pp, 9/inf, 9/model, 9/cot, 9/inpw and 13/gpr.
The function prn_is_ephemeral is called from 9/inpw.
The function prn_get_parsing_grammar is called from 13/gv and 13/gpr.
The function prn_set_parsing_grammar is called from 13/gv.

§**10. Value properties.** All other properties store values, as the following routine expresses:

```
int prn_is_value_property(property_name *prn) {
    if (prn->either_or == FALSE) return TRUE;
    return FALSE;
}
binary_predicate *prn_get_setting_bp(property_name *prn) {
    if (prn->either_or) internal_error("either/or property has no setting BP");
    return prn->setting_bp;
}
int prn_is_value_property_made_by_ni(property_name *prn) {
    if ((prn_is_either_or(prn) == FALSE) &&
        (prn->translated == FALSE)) return TRUE;
    return FALSE;
}
kind_of_value *prn_get_kind_of_value(property_name *prn) {
    if (prn->either_or)
        internal_error("asked for kind of value of either/or property");
    return prn->property_value_kov;
}
void prn_set_kind_of_value(property_name *prn, kind_of_value *kov) {
    if (prn->either_or)
        internal_error("asked to set kind of value of either/or property");
    prn->property_value_kov = kov;
}
int prn_coincides_with_kind_of_value(property_name *prn) {
    if (prn->either_or)
        internal_error("asked if either/or property coincides with kov");
    return prn->also_a_type;
}
```

The function prn_is_value_property is called from 7/tc, 8/knowp, 9/pp, 9/provr, 9/vpbp, 9/model and 9/inpw.
The function prn_get_setting_bp is called from 6/treec.
The function prn_is_value_property_made_by_ni is called from 9/pp.
The function prn_get_kind_of_value is called from 6/treec, 6/sconv, 9/rsdt, 7/stsp, 7/tc, 8/knowp, 9/qty, 9/pp, 9/madj, 9/vpbp, 9/model, 9/inpw, 13/tfg and 13/gpr.
The function prn_set_kind_of_value is called from 7/data, 9/pp and 10/tab.
The function prn_coincides_with_kind_of_value is called from 6/sconv and 8/refpt.

§**11. Relation-storing properties.** Some value properties are used for the run-time storage of 1-1 relations, as the following notes.

```
void prn_now_stores_1to1_relation(property_name *prn) {
    prn->stores_1to1_relation = TRUE;
}
int prn_stores_1to1_relation(property_name *prn) {
    return prn->stores_1to1_relation;
}
```

The function prn_now_stores_1to1_relation is called from 5/rel.
The function prn_stores_1to1_relation is called from 9/model.

§**12. Condition properties.** Nothing to do with conditions in the usual computing sense: these are value properties describing the current state, or "condition", of a world object which is simply modelled as having one of an enumerated range of possibilities. The name is always the object's name plus "condition": for instance, "lounge table condition".

```
property_name *prn_condition_new(world_object *wo, int n1, int n2) {
    property_name *prn;
    int w1, w2, anon;
    if (n1 >= 0) {
        w1 = n1; w2 = n2; anon = FALSE;
    } else {
        int ct = 0;
        LOOP_OVER(prn, property_name)
            if ((prn->condition_of == wo) && (prn->condition_anonymously_named))
                ct++;
        w1 = lexer_wordcount;
        splice_words(wo->word_ref1, wo->word_ref2);
        feed_into_lexer(" condition ", FALSE, FALSE);
        if (ct > 0) {
            char numb[32];
            sprintf(numb, " %d ", ct+1);
            feed_into_lexer(numb, FALSE, FALSE);
        }
        w2 = lexer_wordcount-1; anon = TRUE;
    }
    pcalc_prop *prop = prop_to_create_something(NULL, w1, w2);
    prop = prop_concatenate(prop, prop_to_make_a_kov());
    prop_true_in_world_model(prop);
    prn = value_property_name_ref(w1, w2);
    prn->either_or = FALSE;
    prn->condition_of = wo;
    prn->condition_anonymously_named = anon;
    return prn;
}
world_object *prn_condition_of_which_object(property_name *prn) {
    if (prn == NULL) return NULL;
    return prn->condition_of;                    which will be null if not a condition property
}
```

The function prn_condition_new is called from 9/pp.

The function prn_condition_of_which_object is called from 2/lexi, 7/stsp, 9/qty, 9/pp, 9/inf and 12/cinv.

§**13. I6 names of properties.** Some properties have I6 names mechanically generated by NI (indeed all properties initially have, as we saw above), but others must have names corresponding to those used in the I6 library: these are, we say, "translated". The following routine accomplishes that. It is brought to bear in response to explicit sentences in the standard rules file.

```
char *prn_get_i6_identifier(property_name *prn) {
    if (prn == NULL) internal_error("I6 identifier requested for null property");
    return prn->i6_name;
}
void set_translation(property_name *pn, char *t) {
    int i; char *to = pn->i6_name;
    for (i=0; ((t[i]) && (i<31)); i++) {
        if ((isalpha(t[i])) || (isdigit(t[i])) || (t[i] == '_')) to[i] = t[i];
        else to[i] = '_';
    }
    to[i] = 0;
    pn->translated = TRUE;
}
void compile_has_property(OUTPUT_STREAM, property_name *pn) {
    if (pn->implemented_as_attribute) {
        if (prn_either_or_stored_in_negation(pn))
            WRITE(" hasnt %s", pn->negation->i6_name);
        else
            WRITE(" has %s", pn->i6_name);
    } else {
        if (prn_either_or_stored_in_negation(pn))
            WRITE(".%s == false", pn->negation->i6_name);
        else
            WRITE(".%s == true", pn->i6_name);
    }
}
```

The function prn_get_i6_identifier is called from 5/rel, 6/equal, 6/defer, 6/cdefp, 9/rsdt, 7/vasp, 9/qty, 9/pp, 9/madj, 9/cmpbp, 9/vpbp, 9/model, 9/cot, 9/inpw, 12/cinv, 13/gtok, 13/gpr and 14/i6t.
The function compile_has_property is called from 9/inpw and 13/gpr.

§**14.** Conversely, the following function finds which NI property name represents, say, "plural-name", or some other I6 library attribute or property. (In fact at present it is used only to find "initial", "scenery", "description" and "wearable", and runs only about 200 times on a large game, so its inefficiency is unimportant.)

```
property_name *find_property_translating(char *t) {
    property_name *pn;
    LOOP_OVER(pn, property_name)
        if ((pn->translated) && (strcmp(pn->i6_name, t) == 0))
            return pn;
    return NULL;
}
```

The function find_property_translating is called from 9/spabp and 9/model.

§**15.** And this is the routine which is called by the assertion maker to set property name translations.

```
void property_translates(parse_node *pn) {
    parse_node *p1 = pn->down->next;
    parse_node *p2 = pn->down->next->next;
    char *p = lw_array[p2->word_ref1].lw_text;
    property_name *pname = parse_property_name(p1->word_ref1, p1->word_ref2);
    if (pname == NULL)  {
        sentence_problem(_P_(C9NonPropertyTranslated),
            "this property does not exist",
            "so cannot be translated.");
        return;
    }
    if (pname -> translated) {
        sentence_problem(_P_(C9TranslatedTwice),
            "this property has already been translated",
            "so there must be some duplication somewhere.");
        return;
    }
    if (pname->either_or) {
        pname->implemented_as_attribute = TRUE;
        if (pname->negation)
            pname->negation->implemented_as_attribute = TRUE;
    }
    if ((strcmp(p, "workflag") == 0) ||
        (strcmp(p, "concealed") == 0) ||
        (strcmp(p, "mentioned") == 0)) {
        pname->ephemeral = TRUE;
        if (pname->negation)
            pname->negation->ephemeral = TRUE;
    }
    LOGIF(PROPERTY_TRANSLATIONS, "Property <$Y", pname);
    set_translation(pname, p);
    LOGIF(PROPERTY_TRANSLATIONS, "> translates as <$Y>\n", pname);
}
```

The function property_translates is called from 2/isn.

§**16. Compilation.** This subsection begins with code made necessary by oddities in the syntax of Inform 6. I6 distinguishes either/or properties as "attributes", which must be declared before use, but are then present for every object; and others as "properties", which need not be declared before use, but are only present as specified.

The following routine is, perhaps surprisingly, used only when declaring objects: not when compiling regular code.

```
void write_attribute_name(OUTPUT_STREAM, property_name *pn, int truth_state) {
    if (pn->implemented_as_attribute) {
        WRITE("  has ");
        if (pn->translated) {
            if (truth_state == FALSE) WRITE("~");
            WRITE("%s", pn->i6_name);
            return;
        }
        if (prn_either_or_stored_in_negation(pn)) {
            if (truth_state == TRUE) WRITE("~");
            WRITE("%s", pn->negation->i6_name);
            return;
        }
        if (truth_state == FALSE) WRITE("~");
        WRITE("%s", pn->i6_name);
    } else {
        WRITE("  with ");
        if (pn->translated) {
            WRITE("%s", pn->i6_name);
            if (truth_state == FALSE) WRITE(" false");
            else WRITE(" true");
            return;
        }
        if (prn_either_or_stored_in_negation(pn)) {
            WRITE("%s", pn->negation->i6_name);
            if (truth_state == FALSE) WRITE(" true");
            else WRITE(" false");
            return;
        }
        WRITE("%s", pn->i6_name);
        if (truth_state == FALSE) WRITE(" false");
        else WRITE(" true");
    }
}
```

The function write_attribute_name is called from 9/cot.

§**17.** We make one exception to the rule about not declaring properties explicitly: those explicitly declared are handled slightly more efficiently at run-time, and can also be made "additive", meaning that they inherit values from classes in a way which concatenates lists. This is what we want for the special property which records the kind of an object (which we need in order to check, at run-time, that various property manipulations are safe and legal).

```
int no_attributes_made = 0, i7_fbna_made = FALSE;
void compile_attributes(OUTPUT_STREAM) {
    property_name *prn;
    LOOP_OVER(prn, property_name) {
        if ((prn->either_or) && (prn->implemented_as_attribute)) {
            if (prn_either_or_stored_in_negation(prn)) continue;
            if (prn->translated == FALSE) {
                if (++no_attributes_made >= 13) {
                    prn->implemented_as_attribute = FALSE;
                    if (prn->negation)
                        prn->negation->implemented_as_attribute = FALSE;
                } else {
                    WRITE("Attribute %s;\n", prn->i6_name);
                }
                if (no_attributes_made == 13) {
                    i7_fbna_made = TRUE;
                    WRITE("Constant FBNA_PROP_NUMBER = %s;\n", prn->i6_name);
                }
                prn->translated = TRUE;
            }
        }
    }
}
```

The function compile_attributes is invoked by a command in a `.i6t` template file.

§**18.** Various things mis-compile in I6 if it should happen that a property, by not being used anywhere, never gets created as a symbol. So we compile stub definitions to give meanings (it doesn't matter what) to property names which would otherwise go unrecognised.

The FBNA is the First Boolean Not to be an Attribute, and relates to the dodge we use to store further attributes in I6 when the Z-machine's stock has run out.

```
void compile_stub_properties(OUTPUT_STREAM) {
    property_name *prn;
    binary_predicate *bp;
    LOOP_OVER(prn, property_name) {
        if ((prn->either_or == TRUE)
            && (prn->negation != NULL)
            && (prn->negation->translated)) continue;
        WRITE("#ifndef %s; Constant %s = 0; #endif;\n",
            prn->i6_name, prn->i6_name);
    }
    LOOP_OVER(bp, binary_predicate)
        if (bp_allow_arbitrary_assertions(bp)) {
            WRITE("#ifndef relation_index_L%d; Constant relation_index_L%d = 0; #endif;",
                bp->allocation_id, bp->allocation_id);
            WRITE("#ifndef relation_index_R%d; Constant relation_index_R%d = 0; #endif;",
```

```
                bp->allocation_id, bp->allocation_id);
        }
    if (i7_fbna_made == FALSE) WRITE("Constant FBNA_PROP_NUMBER = 10000;\n");
}

void prn_offset_in_runtime_metadata_table_is(property_name *prn, int pos) {
    prn->table_offset = pos;
}

void compile_CreatePropertyOffsets_routine(OUTPUT_STREAM) {
    property_name *prn;
    WRITE("[ CreatePropertyOffsets i;\n"); INDENT;
    WRITE("for (i=0: i<%d: i++) attribute_offsets-->i = -1;\n",
        50 + NUMBER_CREATED(property_name));
    WRITE("for (i=0: i<%d: i++) property_offsets-->i = -1;\n",
        100 + NUMBER_CREATED(property_name));
    LOOP_OVER(prn, property_name) {
        if (prn->either_or == FALSE) continue;
        if (prn->negation != NULL) {
            if (prn->negation->translated) continue;
        }
        if (prn->implemented_as_attribute)
            WRITE("attribute_offsets-->%s = %d;\n", prn->i6_name,
                prn->table_offset);
        else
            WRITE("attribute_offsets-->(50 + %s - FBNA_PROP_NUMBER) = %d;\n",
                prn->i6_name, prn->table_offset);
    }
    LOOP_OVER(prn, property_name) {
        if (prn->either_or == TRUE) continue;
        WRITE("property_offsets-->%s = %d;\n", prn->i6_name,
            prn->table_offset);
    }
    OUTDENT; WRITE("];\n");
}

possession_marker *prn_get_possession_marker(property_name *prn) {
    return &(prn->pom);
}

void prn_set_last_initialised_for(property_name *prn, world_object *wo) {
    prn->last_initialised_for = wo;
}

world_object *prn_get_last_initialised_for(property_name *prn) {
    return prn->last_initialised_for;
}
```

The function compile_stub_properties is invoked by a command in a `.i6t` template file.
The function prn_offset_in_runtime_metadata_table_is is called from 9/pp.
The function compile_CreatePropertyOffsets_routine is invoked by a command in a `.i6t` template file.
The function prn_get_possession_marker is called from 8/imp.
The function prn_set_last_initialised_for is called from 9/cot.
The function prn_get_last_initialised_for is called from 9/pp.

Property Permissions 9/pp

Purpose

To enforce the domain of properties: for instance, that a door can be open or closed but that an animal cannot, or that a person can have a carrying capacity but that a door cannot.

9/pp.§2-3 Visible properties; §4 Compiling unspecified but permitted properties; §5-6 Considering visible properties; §7-8 Actual provision; §9 Declaring properties; §10-12 "Can be" assertions declaring properties; §13 Numberspace forcer; §14 Property metadata

---

Template interpreter commands

```
13    {-callv:compile_property_numberspace_forcer}
14    {-array:property_metadata}
```

Definitions

¶1. Each property name has a linked list of objects and kinds which are allowed to provide it. There is a tacit form of multiple-inheritance going on here, even though NI is officially a single-inheritance language from an objects-and-classes point of view: the property "open", for instance, exists for both doors and containers, even though neither is a kind of the other.

```
typedef struct property_permission {
    struct property_name *property_granted;                    which property is permitted
    struct parse_node *where_granted;                          sentence granting the permission
    struct world_object *wo_with_permission;                   either a world object has permission...
    int visible_to_parser;                                     if so, does the run-time I6 parser recognise it?
    int visibility_condition_w1, visibility_condition_w2;      (at least if...?)
    struct kind_of_value *kov_with_permission;                 ...or a kind of value has permission
    int storage_table_id, storage_column_id;                   the table and column reference
    struct property_permission *next;                          in list of permissions
    MEMORY_MANAGEMENT
} property_permission;
```

The structure property_permission is private to this section.

¶2.

```
property_permission *latest_property_permission = NULL;
```

---

§**1.** The following routine declares that permission is granted: as can be seen, each world object keeps a linked list of its own permissions.

```
property_permission *new_property_permission(void) {
    property_permission *pp = CREATE(property_permission);
    pp->wo_with_permission = NULL;
    pp->kov_with_permission = NULL;
    pp->property_granted = NULL;
    pp->next = NULL;
    pp->visible_to_parser = FALSE;
    pp->visibility_condition_w1 = -1;
    pp->visibility_condition_w2 = -1;
    pp->where_granted = current_sentence;
    pp->storage_table_id = -1;
    pp->storage_column_id = -1;
    latest_property_permission = pp;
    return pp;
}
void kind_provides_property(world_object *k, property_name *pn) {
    PROTECTED_MODEL_PROCEDURE;
    property_permission *pp, *pp2;
    LOGIF(PROVISION, "Allowing $O to provide $Y\n", k, pn);
    if (k == NULL) internal_error("Tried to give permission to null object");
    if (k == kind_kind)
        sentence_problem(_P_(C9PropertyOfKind),
            "this seems to give a property to all kinds",
            "rather than to objects or values, which are the only things "
            "capable of having properties. For instance, 'A vehicle has a "
            "number called maximum speed' is fine, but not 'A kind has a "
            "number called coolness rating'.");
    pp = new_property_permission();
    pp->wo_with_permission = k;
    pp->property_granted = pn;
    pp2 = prn_permission_list(pn);
    if (pp2 == NULL) {
        prn_set_permission_list(pn, pp);
    } else {
        while (pp2->next != NULL) {
            if (pp2->wo_with_permission == k) { DESTROY(pp, property_permission); return; }
            pp2 = pp2->next;
        }
        if (pp2->wo_with_permission == k) { DESTROY(pp, property_permission); return; }
        pp2->next = pp;
    }
    update_adjectival_forms(pn);
}
```

The function kind_provides_property is called from 9/provr.

§**2. Visible properties.** A visible property is one which can be used to describe an object: for instance, if colour is a visible property of a car, then it can be called "green car" if and only if the current value of the colour of the car is "green".

Properly speaking it is not the property which is visible, but the combination of property and object (or kind): thus the following test depends on a property permission and not a mere property.

```
int pp_visible_to_wo(property_permission *pp, world_object *wo,
    int allow_inheritance) {
    if ((pp->wo_with_permission) && (pp->visible_to_parser > 0)) {
        if (pp->wo_with_permission == wo) return TRUE;
        if ((allow_inheritance) &&
            (pp->wo_with_permission->kind_flag) && (wo_of_kind(wo, pp->wo_with_permission)))
            return TRUE;
    }
    return FALSE;
}
```

The function pp_visible_to_wo is called from 13/gpr.

§**3.** Rolling this up to see if anything is visible:

```
int pr_any_visible_to_wo(world_object *wo, int allow_inheritance) {
    property_name *pr;
    LOOP_OVER(pr, property_name) {
        property_permission *pp;
        for (pp = prn_permission_list(pr); pp; pp = pp->next)
            if (pp_visible_to_wo(pp, wo, allow_inheritance))
                return TRUE;
    }
    return FALSE;
}
```

The function pr_any_visible_to_wo is called from 13/gpr.

§**4. Compiling unspecified but permitted properties.** If the source specified that, say, "A container has a number called sufficiency", then we need to compile the property even when no explicit value was given for it (so that it does not occur in the inferences for the object `wo`). Its initial value will be the default initial value for its type.

This stricture does not apply to boolean properties, as they are stored as I6 attributes and always exist: and they already have the default value of being false, which is what we want. So we need do nothing.

```
void compile_permitted_but_unspecified_properties(OUTPUT_STREAM, world_object *wo) {
    property_permission *pp;
    LOOP_OVER(pp, property_permission) {
        property_name *prn = pp->property_granted;
        if ((pp->wo_with_permission == wo) &&
            (prn_is_value_property_made_by_ni(prn)) &&
            (prn_get_last_initialised_for(prn) != wo)) {
            WRITE(" with %s ", prn_get_i6_identifier(prn));
            compile_pval_default(OUT, prn);
            WRITE(",\n");
        }
        if ((pp->wo_with_permission == wo) && (prn_is_either_or(prn)) &&
```

```
            (prn_either_or_implemented_as_attribute(prn) == FALSE) &&
            (prn_get_last_initialised_for(prn) != wo)) {
            WRITE(" with %s false,\n", prn_get_i6_identifier(prn));
        }
    }
}
void compile_pval_default(OUTPUT_STREAM, property_name *prn) {
    kind_of_value *kov = prn_get_kind_of_value(prn);
    LOGIF(OBJECT_COMPILATION, "Initialising: $Y\n", prn);
    if (compile_default_value(OUT,
        kov, prn->word_ref1, prn->word_ref2, "property") == FALSE) {
        quote_words(1, prn->word_ref1, prn->word_ref2);
        handmade_problem(_P_(C9PropertyUninitialisable));
        if (prn_get_kind_of_value(prn) == NULL)
            issue_problem_segment(                                    believed not to happen
            "I am unable to put any value into the property '%1', because "
            "it was created so vaguely that I cannot tell what kind of "
            "value it should hold. (A number, a time, some text perhaps?)");
        else issue_problem_segment(
            "I am unable to put any value into the property '%1', because "
            "it seems to have a kind of value which has no actual values.");
        issue_problem_end();
    }
}
```

The function compile_permitted_but_unspecified_properties is called from 9/cot.
The function compile_pval_default is called from 9/prop.

§5. Considering visible properties.

```
int pp_visibility_level(property_permission *pp) {
    return pp->visible_to_parser;
}
specification *pp_get_visibility_condition(property_permission *pp) {
    specification *spec;
    if (pp->visibility_condition_w1 == -1) return NULL;
    spec = parse_expression(pp->visibility_condition_w1, pp->visibility_condition_w2, CONDITION_EXPCON);
    if (validate_when(spec) == FALSE) {
        LOG("$X", spec);
        current_sentence = pp->where_granted;
        sentence_problem(_P_(C9BadVisibilityWhen),
            "the condition after 'when' makes no sense to me",
            "although otherwise this worked - it is only the part after 'when' "
            "which I can't follow.");
        pp->visibility_condition_w1 = -1;
        pp->visibility_condition_w2 = -1;
        return NULL;
    }
    return spec;
}
```

The function pp_visibility_level is called from 13/gpr.
The function pp_get_visibility_condition is called from 13/gpr.

§**6.** Kinds of value can also provide properties, when they have been defined by table.

```
void kov_provides_property(kind_of_value *kov, property_name *pn) {
    PROTECTED_MODEL_PROCEDURE;
    property_permission *pp, *pp2;
    LOGIF(PROVISION, "Allowing $u to provide $Y\n", kov, pn);
    if (kov == NULL) internal_error("Tried to give permission to null type");
    pp = new_property_permission();
    pp->kov_with_permission = kov;
    prn_make_property_of_value(pn);
    pp->property_granted = pn;
    pp2 = prn_permission_list(pn);
    if (pp2 == NULL) {
        prn_set_permission_list(pn, pp);
    } else {
        while (pp2->next != NULL) {
            if (kov_compare(pp2->kov_with_permission, kov)) {
                DESTROY(pp, property_permission);
                return;
            }
            pp2 = pp2->next;
        }
        if (kov_compare(pp2->kov_with_permission, kov)) {
            DESTROY(pp, property_permission);
            return;
        }
        pp2->next = pp;
    }
    update_adjectival_forms(pn);
}

void pp_set_table_storage(property_permission *pp, int t, int i) {
    pp->storage_table_id = t;
    pp->storage_column_id = i;
}

void pp_register_as_adjectival(quantity *q, property_name *prn) {
    property_permission *pp;
    int make_object_form = FALSE;
    for (pp = prn_permission_list(prn); pp; pp = pp->next) {
        if (pp->kov_with_permission)
            register_as_adjectival_quantity_domain(q, prn,
                pp->kov_with_permission, NULL);
        else make_object_form = TRUE;
    }
    if (make_object_form)
        register_as_adjectival_quantity_domain(q, prn,
            NULL, prn_condition_of_which_object(prn));
}

int pp_get_prn_offset_in_kov(property_name *pn, kind_of_value *kov) {
    int ct = 0;
    property_name *try_prn;
    LOOP_OVER(try_prn, property_name) {
        if ((prn_is_either_or(try_prn)) && (prn_either_or_get_negation(try_prn))) {
            property_name *prnbar = prn_either_or_get_negation(try_prn);
```

```
            if (prnbar->allocation_id < try_prn->allocation_id) continue;
        }
        if (try_prn == pn) break;
        property_permission *pp = prn_permission_list(try_prn);
        while (pp) {
            if ((pp->storage_table_id == -1) &&
                (pp->kov_with_permission) &&
                (kov_compare(pp->kov_with_permission, kov))) {
                    ct++; break;
            }
            pp = pp->next;
        }
    }
    return ct;
}

int does_kov_provide(kind_of_value *kov, property_name *prn) {
    int ct = pp_get_prn_offset_in_kov(NULL, kov);
    int j = pp_get_prn_offset_in_kov(prn, kov);
    if (j<ct) return TRUE;
    return FALSE;
}

int compile_value_properties(OUTPUT_STREAM) {
    property_name *prn;
    int there_are_abstracts = FALSE;
    LOOP_OVER(prn, property_name)
        if (prn_belongs_to_value(prn)) {
            property_permission *pp; int counter = 0;
            there_are_abstracts = TRUE;
            pp = prn_permission_list(prn); while (pp) { counter++; pp = pp->next; }
            if (counter > 1) {
                current_sentence = prn_permission_list(prn)->where_granted;
                property_problem(_P_(C9PropOfValueAmbiguous),
                    prn, "has more than one interpretation",
                    "that is, it has meaning for more than one kind of thing "
                    "or value. This is fine for properties of things, but "
                    "not allowed (at present) for properties of values.");
            }
        }
    WRITE("Object ValuePropertyHolder\n");
    LOOP_OVER(prn, property_name)
        if ((prn_belongs_to_value(prn)) &&
            (prn_permission_list(prn)->storage_table_id >= 0)) {
            WRITE("  with %s tab_%d_%d\n",
                prn_get_i6_identifier(prn),
                prn_permission_list(prn)->storage_table_id,
                prn_permission_list(prn)->storage_column_id);
        }
    WRITE(";\n");
    return there_are_abstracts;
}

int seek_permission(property_name *pr, world_object *wo,
    int level, int when1, int when2) {
    property_permission *pp, *pp2;
```

```
    world_object *k;
    int tries = 1;
    SeekProperty: k = wo;
    while (k) {
        if (k == kind_kind) break;
        pp = prn_permission_list(pr);
        while (pp) {
            if (pp->wo_with_permission == k) {
                if (wo != k) {
                    pp = new_property_permission();
                    pp->wo_with_permission = wo;
                    pp->property_granted = pr;
                    if (prn_permission_list(pr)) {
                        pp2 = prn_permission_list(pr);
                        while (pp2->next) pp2 = pp2->next;
                        pp2->next = pp;
                    } else prn_set_permission_list(pr, pp);
                }
                pp->visible_to_parser = level;
                pp->visibility_condition_w1 = when1;
                pp->visibility_condition_w2 = when2;
                return TRUE;
            }
            pp = pp->next;
        }
        k = k->kind;
    }
    if ((tries == 1) && (prn_either_or_get_negation(pr))) {
        tries = 2; pr = prn_either_or_get_negation(pr); goto SeekProperty;
    }
    return FALSE;
}
void pp_index_applicabilities(property_permission *first_pp) {
    property_permission *pp;
    int ac = 0;
    for (pp = first_pp; pp; pp = pp->next) ac++;
    for (pp = first_pp; pp; pp = pp->next) {
        INDEX("</i>");
        if (pp->wo_with_permission)
            print_raw_text_to_file(pp->wo_with_permission->word_ref1,
                pp->wo_with_permission->word_ref2, ifl);
        else {
            int w1, w2;
            kov_get_name(pp->kov_with_permission, &w1, &w2, FALSE);
            print_raw_text_to_file(w1, w2, ifl);
        }
        INDEX("<i>");
        ac--;
        if (ac == 1) INDEX(" or ");
        if (ac > 1) INDEX(", ");
    }
}
```

The function kov_provides_property is called from 9/provr.
The function pp_set_table_storage is called from 10/tab.
The function pp_register_as_adjectival is called from 9/qty.
The function pp_get_prn_offset_in_kov is called from 7/stsp, 9/qty and 9/prop.
The function does_kov_provide is called from 9/provr.
The function compile_value_properties is called from 9/cot.
The function seek_permission is called from 13/tfg.
The function pp_index_applicabilities is called from 9/qty and 9/prop.

§**7. Actual provision.** It is one thing to have permission to have a given property: quite another actually to have it. So the following code looks to see, and is somewhat analogous to I6's "provides" operator. We can only find out by looking through the inferences stored for the object. We need to go to a little trouble with boolean properties: if a "closed" inference can be found, then "open" is provided as well, even though there is no explicit reference to it.

```
inference *provision_inference; int provision_sign = 1;
int does_kind_provide_dash(world_object *k, property_name *prn) {
    inference *inf;
    property_permission *pp;
    while ((k != NULL) && (k != kind_kind)) {
        KNOWLEDGE_LOOP(inf, k, PROPERTY_INF) {
            if ((inf_get_property_name(inf) == prn)
                && (provision_inference == NULL)) {
                provision_sign = 1;
                provision_inference = inf;
            }
            if ((prn_is_either_or(prn))
                && (prn_either_or_get_negation(prn))
                && (inf_get_property_name(inf) == prn_either_or_get_negation(prn))
                && (provision_inference == NULL)) {
                provision_sign = -1;
                provision_inference = inf;
            }
        }
        pp = prn_permission_list(prn);
        while (pp != NULL) {
            if (pp->wo_with_permission == k) return TRUE;
            pp = pp->next;
        }
        k = k->kind;
    }
    return FALSE;
}
int does_kind_provide(world_object *k, property_name *prn) {
    int f;
    provision_sign = 1;
    provision_inference = NULL;
    f = does_kind_provide_dash(k, prn);
    if ((f) || (prn_is_value_property(prn)) ||
        (prn_either_or_get_negation(prn) == NULL)) return f;
    return does_kind_provide_dash(k, prn_either_or_get_negation(prn));
}
```

The function does_kind_provide is called from 9/model and 9/inpw.

§**8.** Provision is handled identically for objects and kinds, but it was not always so, and might one day not be so again. We therefore keep two different routines, even though they do the same thing right now.

```
int does_object_provide(world_object *k, property_name *prn) {
    return does_kind_provide(k, prn);
}
```

The function does_object_provide is called from 7/tc, 8/imp and 9/model.

§**9. Declaring properties.** The following handles sentences like

A container has a number called rating.

in which `kind_ref` would be "a container" and `pname` would be "a rating".

```
property_name *recursively_declare_properties(parse_node *kind_ref,
    parse_node *pname) {
    property_name *prn;
    specification *spec;
    kind_of_value *kov;
    world_object *owner_wo = NULL;
    kind_of_value *owner_kov = NULL;
    if (pn_get_node_type(kind_ref) == VALUE_NT) {
        specification *vts = pn_get_evaluation(kind_ref);
        owner_kov = spec_get_kind_of_value(vts);
    } else {
        owner_wo = pn_get_refers_to(kind_ref);
    }
    if ((owner_wo == NULL) && (owner_kov == NULL)) {
        sentence_problem(_P_(BelievedImpossible),
            "only an object, kind, rulebook, action or activity can be given "
            "properties or variables",
            "so 'a room has a number called capacity' is allowed, but '20 "
            "has a number called capacity' would not be.");
        return NULL;
    }
    switch(pn_get_node_type(pname)) {
        case PROPERTYCALLED_NT:
            prn = recursively_declare_properties(kind_ref, pname->down->next);
            spec = parse_expression(pname->down->word_ref1, pname->down->word_ref2,
                TYPE_EXPCON);
            if (species_is(spec, DESCRIPTION_SPC)) {
                if ((spec_get_described_kind(spec)) && (number_of_adjectives_applied_to(spec)
== 0)) {
                    spec = new_generic_CONSTANT_type(kovko(spec_get_described_kind(spec)));
                } else {
                    quote_source(1, current_sentence);
                    quote_words(2, pname->down->word_ref1, pname->down->word_ref2);
                    handmade_problem(_P_(C9PropertyTooSpecific));
                    issue_problem_segment(
                        "You wrote %1, which I am reading as a request to make "
```

```
            "a new named property - a value associated with an "
            "object and which has a name. The request seems to say "
            "that the value in question is '%2', but this is too "
            "specific a description. (Instead, a kind of value "
            "(such as 'number') or a kind of object (such as 'room' "
            "or 'thing') should be given. To get a property whose "
            "contents can be any kind of object, use 'object'.)");
        issue_problem_end();
        return NULL;
    }
}
if ((spec_is_generic_CONSTANT(spec) == FALSE) &&
    (spec_is_generic_NONLOCAL_VARIABLE(spec) == FALSE)) {
    LOG("Offending SP: $X", spec);
    quote_source(1, current_sentence);
    quote_words(2, pname->down->word_ref1, pname->down->word_ref2);
    handmade_problem(_P_(C9PropertyKindUnknown));
    issue_problem_segment(
        "You wrote %1, but '%2' is not the name of a kind of "
        "value which I know (such as 'number' or 'text').");
    issue_problem_end();
    return NULL;
}
kov = spec_get_kind_of_value(spec);
if (is_kova(kov, ANY_VALUE_TY)) {
    quote_source(1, current_sentence);
    quote_words(2, pname->down->word_ref1, pname->down->word_ref2);
    handmade_problem(_P_(C9PropertyKindVague));
    issue_problem_segment(
        "You wrote %1, but saying that a property is a 'value' "
        "does not give me a clear enough idea what it will hold. "
        "You need to say what kind of value: for instance, 'A door "
        "has a number called street address.' is allowed because "
        "'number' is specific about the kind of value.");
    issue_problem_end();
    return NULL;
}
if (prn != NULL) {
    if (prn_get_kind_of_value(prn) == NULL)
        prn_set_kind_of_value(prn, kov);
    else {
        if (kov_compare(prn_get_kind_of_value(prn), kov) == FALSE) {
            quote_source(1, current_sentence);
            quote_words(2, pname->down->word_ref1, pname->down->word_ref2);
            quote_property(3, prn);
            spec = new_generic_CONSTANT_type(prn_get_kind_of_value(prn));
            quote_spec(4, spec);
            handmade_problem(_P_(C9PropertyKindClashes));
            issue_problem_segment(
                "You wrote %1, but '%2' contradicts what I previously "
                "thought about the property %3, which was that it was %4.");
            issue_problem_end();
            return NULL;
```

```
                    }
                }
            }
            return prn;
        case AND_NT:
            recursively_declare_properties(kind_ref, pname->down);
            recursively_declare_properties(kind_ref, pname->down->next);
            return NULL;
        case NOUNPHRASE_NT:
            prn = value_property_name_ref(pname->word_ref1, pname->word_ref2);
            prop_true_in_world_model_about(prop_to_provide_property(prn),
                owner_wo, kov_as_spec(owner_kov));
            return prn;
        default:
            internal_error("recursively_declare_properties on a node of unknown type");
    }
    return NULL;
}
```

The function recursively_declare_properties is called from 8/mass.

§**10. "Can be" assertions declaring properties.** The following handles sentences like

A container can be voluminous, middling or poky.

The next routine breaks up the list of choices, returning the number of possibilities noticed (so in this example, 3).

```
int or_list(parse_node *p) {
    int w1 = p->word_ref1;
    int w2 = p->word_ref2;
    int i = w1, count = 0;
    if (pn_get_node_type(p) == AND_NT) {
        return or_list(p->down) + or_list(p->down->next);
    }
    while (i<w2) {
        i++;
        if [[word i == COMMA/or]] {
            graft(new_nounphrase_raw(w1, i-1), p);
            pn_set_node_type(p, AND_NT);
            w1 = i+1; count++;
        }
    }
    if (w1 <= w2) { graft(new_nounphrase_raw(w1, w2), p); count++; }
    return count;
}
```

§**11.** Here we declare a subtree of such options as constants of some given type. We return the first-mentioned option as a quantity.

```
quantity *declare_kov_values(parse_node *p, kind_of_value *kov, int top,
    quantity *earliest) {
    if (p == NULL) return earliest;
    if [[p == either ...]] {
        sentence_problem(_P_(C9EitherOnThree),
            "that looks like an attempt to use 'either' on a list of "
            "three or more possibilities",
            "which is not allowed. (Technically it ought to be legal "
            "to have a property whose name actually starts with 'either' "
            "but the confusion would be just too awful to contemplate.)");
        if (p->word_ref1 < p->word_ref2) p->word_ref1++;
    }
    int w1, w2, cw1, cw2, i;
    if ([[p == ... OPENBRACKET ... CLOSEBRACKET : i --> w1, w2 ... cw1, cw2]] &&
        [[cw1, cw2 == this is ... --> cw1, cw2]]) {
        p->word_ref1 = w1; p->word_ref2 = w2;
    }
    if (pn_get_node_type(p) != AND_NT) {
        int pw1, pw2;
        quantity *q;
        specification *spec;
        [[pw1, pw2 <-- p]];
        spec = parse_expression(pw1, pw2, TYPE_OR_VALUE_EXPCON);
        if ((spec_is_UNKNOWN(spec) == FALSE) &&
            (species_is(spec, DESCRIPTION_SPC) == FALSE) &&
            (spec_is_actual_CONSTANT_of_kova(spec, PROPERTY_TY) == FALSE)) {
            LOG("Already means: $S\n", spec);
            quote_source(1, current_sentence);
            quote_words(2, pw1, pw2);
            quote_spec(3, spec);
            handmade_problem(_P_(C9PropertyAlreadyKnown));
            issue_problem_segment(
                "In %1, one of the values you supply as a possibility is '%2', "
                "but this already has a meaning (as %3).");
            issue_problem_end();
        }
        pcalc_prop *prop = prop_to_create_something(kov, pw1, pw2);
        prop_true_in_world_model(prop);
        q = latest_quantity;
        if (earliest == NULL) earliest = q;
    }
    if ((top) || (pn_get_node_type(p) == AND_NT))
        earliest =
            declare_kov_values(p->down, kov, FALSE, earliest);
    if (top == FALSE)
        earliest =
            declare_kov_values(p->next, kov, FALSE, earliest);
    return earliest;
}
```

§**12.** Lastly, the actual routine to handle the assertion. If there are only one or two alternatives, then we implement the property as a boolean, and the default is for it not to hold; but if there are three or more, then we create a new type associated with the given object (or kind), called "N condition", where "N" is the name of the object in question. In this case, the default value is the *first* mentioned. The beta-testers felt that this was an anomaly in the language, and I suspect they're right, but it isn't obvious to me what a better system would be.

```
sentence_handler CANBE_SH_handler =
    { SENTENCE_NT, CANBE_VB, 1, declare_property_can_be };

void declare_property_can_be(parse_node *p) {
    property_name *pname, *pnamebar;
    world_object *wo = NULL;
    kind_of_value *kov = NULL;
    int i, w1, w2, count, cw1, cw2;
    cw1 = p->word_ref1; cw2 = p->word_ref2;
    if ([[cw1, cw2 == ... OPENBRACKET ... CLOSEBRACKET : i --> w1, w2 ... cw1, cw2]] &&
        [[cw1, cw2 == this is ... --> cw1, cw2]]) {
        [[cw1, cw2 == the/its/his/her/their ... --> cw1, cw2]];
        [[cw1, cw2 == ... property --> cw1, cw2]];
    } else { cw1 = -1; cw2 = -1; }

    p = p->down->next;
    resolve_references(p);
    if (pn_get_node_type(p) == WITH_NT) {
        sentence_problem(_P_(C9QualifiedCanBe),
            "only a room, a thing or a kind can have such adjectives applied "
            "to it",
            "and qualifications cannot be used. It makes no sense to say "
            "'An open door can be rickety or sturdy' because the door still "
            "has to have the property even at times when it is not open: "
            "we must instead just say 'A door can be rickety or sturdy'.");
        return;
    }

    count = or_list(p->next);
    w1 = p->next->down->word_ref1; w2 = p->next->down->word_ref2;
    if (count == 2) [[w1, w2 == either ... --> w1, w2]];

    if (cw1 >= 0) {
        if (count < 3) {
            sentence_problem(_P_(C9ThisIsEitherOr),
                "a name can only be supplied using '... (this is...)' when "
                "a new property is being made with three or more named "
                "alternatives",
                "whereas here a simpler either/or property is being made "
                "with just one or two possibilities - which means these named "
                "outcomes are the property names themselves. For instance, "
                "'A book can be mint or foxed' makes two either/or properties, "
                "one called 'mint', the other called 'foxed'. So 'A book can "
                "be mint or foxed (this is the cover state)' is not allowed.");
            return;
        }
    }

    specification *spec = pn_get_evaluation(p);

    if (spec_is_generic_CONSTANT(spec)) {
```

```
    kov = spec_get_kind_of_value(spec);
    if (count > 2) {
        sentence_problem(_P_(C9BadConditionOwner),
            "only a room, a thing or a kind can have such adjectives applied "
            "to it",
            "where three or more alternatives are given. (Simpler adjectives, "
            "with just one or two choices, can be used with a wider range "
            "of possibilities - scenes, for instance.)");
        return;
    }
    if (kov_has_properties(kov) == FALSE) {
        quote_source(1, current_sentence);
        quote_kov(2, kov);
        handmade_problem(_P_(C9ValueCantHaveProperties));
        issue_problem_segment(
            "The sentence %1 looked to me as if it might be trying "
            "to create an either/or property which would be held by all "
            "of the values of a rather large kind (%2). But this is a kind "
            "which is not allowed to have properties, because the "
            "storage requirements would be too difficult. For instance, "
            "scenes can have properties like this, but numbers can't: "
            "that's because there are only a few, named scenes, but there "
            "are an almost unlimited range of numbers. (That doesn't mean "
            "you can't create adjectives using 'Definition: ...' - it's "
            "only when storage would be needed that this limitation kicks in.)");
        issue_problem_end();
        return;
    }
} else {
    wo = pn_get_refers_to(p);
    if (wo == NULL) {
        sentence_problem(_P_(C9NonObjectCanBe),
            "only a room, a thing or a kind can have such adjectives applied "
            "to it",
            "so that 'a dead end can be secret' is fine but 'taking can be "
            "secret' would not be, since 'taking' is an action and not a "
            "room, thing or kind.");
        return;
    }
}
if (count <= 2) {
    char *error_text = NULL;
    int ew1 = w1, ew2 = w2;
    property_name *already = parse_property_name(w1, w2);
    if (already) {
        if (prn_is_either_or(already) == FALSE)
            error_text = "this already has a meaning as a value property";
    } else {
        specification *spec = parse_expression(w1, w2, TYPE_OR_VALUE_EXPCON);
        LOGIF(PROVISION, "Meaning as type: $S\n", spec);
        if ((spec_is_UNKNOWN(spec) == FALSE)
            && (species_is(spec, DESCRIPTION_SPC) == FALSE)) {
            error_text = "this already has a meaning";
```

```
        }
    }
    if ((count == 2) && (error_text == NULL)) {
        property_name *alreadybar = NULL;
        ew1 = p->next->down->next->word_ref1;
        ew2 = p->next->down->next->word_ref2;
        if [[ew1, ew2 == not ...]] {
            property_name *not_what = parse_property_name(ew1+1, ew2);
            if ((not_what) && (not_what != already))
                error_text =
                "this is 'not' compounded with an existing either/or "
                "property, which would cause horrible ambiguities";
        }
        alreadybar = parse_property_name(ew1, ew2);
        if (alreadybar) {
            if (prn_is_either_or(alreadybar) == FALSE)
                error_text = "this already has a meaning as a value property";
            else if ((already) &&
                (prn_either_or_get_negation(already) != alreadybar))
                error_text = "this is not the same negation as the last "
                        "time this either/or property was used";
            else if ((already == NULL) ||
                (prn_either_or_get_negation(already) != alreadybar))
                error_text =
                    "this already has a meaning as an either/or property";
        } else {
            specification *specb = parse_expression(ew1, ew2, TYPE_OR_VALUE_EXPCON);
            LOGIF(PROVISION, "Meaning as type: $S\n", specb);
            if ((spec_is_UNKNOWN(specb) == FALSE)
                && (species_is(specb, DESCRIPTION_SPC) == FALSE)) {
                error_text = "this already has a meaning";
            }
        }
    }
    if (error_text) {
        quote_source(1, current_sentence);
        quote_words(2, ew1, ew2);
        quote_text(3, error_text);
        handmade_problem(_P_(C9MiscellaneousEOProblem));
        issue_problem_segment(
            "In %1, you proposed the new either/or property '%2': but %3.");
        issue_problem_end();
    }
}
switch(count) {
    case 1:
    case 2:
        pname = prn_either_or_new(w1, w2, kov);
        break;
    default:
        pname = prn_condition_new(wo, cw1, cw2);
        break;
}
```

```
    prop_true_in_world_model_about(prop_to_provide_property(pname),
        wo, kov_as_spec(kov));
    if (count == 2) {
        w1 = p->next->down->next->word_ref1;
        w2 = p->next->down->next->word_ref2;
        pnamebar = prn_either_or_new(w1, w2, kov);
        prop_true_in_world_model_about(prop_to_provide_property(pnamebar),
            wo, kov_as_spec(kov));
        prn_either_or_make_negations(pname, pnamebar);
    }
    if (count >= 3) {
        quantity *initial_val;
        parse_node *piv;
        int i = prevailing_mood;
        initial_val = declare_kov_values(p->next,
            prn_get_kind_of_value(pname), TRUE, NULL);
        if (initial_val) {
            prevailing_mood = LIKELY_CE;
            piv = new_nounphrase_raw(initial_val->word_ref1, initial_val->word_ref2);
            pn_set_node_type(piv, VALUE_NT);
            pn_set_evaluation(piv, new_QUANTITY_spec(initial_val));
            assert_property_value_from_property_subtree(pname, wo, NULL, piv);
            prevailing_mood = i;
        }
    }
}
```

§**13. Numberspace forcer.** This is to do with I6 creating properties in a different order from I7, so that the I6 property numbering sequence does not match the I7 creation sequence: as a result no run-time test by I6 could safely determine whether or not I7 had chosen to implement an either/or property as an I6 attribute or an I6 property. We might for example define lots of new either/or properties first for thing, thus kicking I7 to the point where it is forced to use I6 attributes; then sneak back and define an either/or property for a room. As NI defines the room class before the thing class, the new room property gets a lower I6 property number than the thing properties, which means numerical comparisons with the magic threshold property number `FBNA_PROP_NUMBER` would fail.

We get around this by creating the following spurious object before the class hierarchy and object tree are created: its properties are therefore all new creations, and since we declare them in I7 creation order, they are now allocated I6 property numbers in a sequence matching this.

```
void compile_property_numberspace_forcer(OUTPUT_STREAM) {
    property_name *prn;
    WRITE("Object property_numberspace_forcer\n");
    LOOP_OVER(prn, property_name)
        if ((prn_is_either_or(prn)) &&
            (prn_either_or_implemented_as_attribute(prn) == FALSE) &&
            (prn_either_or_stored_in_negation(prn) == FALSE))
                WRITE("  with %s false\n", prn_get_i6_identifier(prn));
    WRITE(";\n");
}
```

The function compile_property_numberspace_forcer is invoked by a command in a `.i6t` template file.

§14. Property metadata. The run-time system needs to know quite a lot of what NI knows about properties: in particular, what is allowed to belong to what.

The I6 routine `CreatePropertyOffsets` looks a little odd at first sight. The table contains no data which is in any sense variable, so why not simply initialise it explicitly here in NI? Why have to run a routine at run-time to do the job? The answer is that in NI we know the names we are giving the properties in the I6 source – "open", for instance – but not what property numbers I6 will compile these to. (And we cannot predict in what order I6 will create them: it would be hazardous to guess, anyway, because I6 or its library may well change.)

```
void compile_property_metadata_array(OUTPUT_STREAM) {
    int pos; property_name *prn;
    WRITE("Array property_metadata --> \n    ");
    pos = 0;
    LOOP_OVER(prn, property_name) {
        property_permission *pp;
        if ((prn_is_either_or(prn))
            && (prn_either_or_stored_in_negation(prn))) continue;
        prn_offset_in_runtime_metadata_table_is(prn, pos);
        pp = prn_permission_list(prn);
        WRITE("\"");
        print_text_to_file(prn->word_ref1, prn->word_ref2, OUT);
        WRITE("\" ");
        pos++;
        while (pp != NULL) {
            if (pp->wo_with_permission == NULL) WRITE("0 ");
            else
                if (wo_get_I6_representation(pp->wo_with_permission))
                    WRITE("%s ", wo_get_I6_representation(pp->wo_with_permission));
            pos++;
            pp = pp->next;
        }
        if (strcmp(prn_get_i6_identifier(prn), "light") == 0) {
            WRITE("%s %s ",
                wo_get_I6_representation(kind_room),
                wo_get_I6_representation(kind_thing));
            pos+=2;
        }
        WRITE("NULL\n    "); pos++;
    }
    WRITE(";\n");
}
```

The function compile_property_metadata_array is invoked by a command in a `.i6t` template file.

The Provision Relation 9/provr

Purpose

To define the provision relation, which determines which properties can be held by which objects.

Definitions

¶1.

```
binary_predicate *a_provides_b_predicate = NULL;
```

---

§**1. Initial stock.** This relation is hard-wired in.

```
void PROVISION_KBP_create_initial_stock(void) {
    a_provides_b_predicate =
        make_pair_of_BPs(PROVISION_KBP,
            bptd_blank(), bptd_blank(),
            "provides", NULL, -1, NULL, NULL, NULL,
            provision_V);
}
```

The function PROVISION_KBP_create_initial_stock is called from 5/bp.

§**2. Second stock.** There is none, of course.

```
void PROVISION_KBP_create_second_stock(void) {
}
```

The function PROVISION_KBP_create_second_stock is called from 5/bp.

§**3. Typechecking.**

```
int PROVISION_KBP_typecheck(binary_predicate *bp,
        kind_of_value **kovs_of_terms, kind_of_value **kovs_required, tc_problem_kit *tck) {
    if (is_kova(kovs_of_terms[1], PROPERTY_TY)) return ALWAYS_MATCH;
    if (is_kova(kovs_of_terms[1], OBJECT_TY)) return ALWAYS_MATCH;
    quote_kov(4, kovs_of_terms[1]);
    tcp_problem(_P_(C9BadProvides), tck,
        "that asks whether something provides something, and in Inform "
        "'to provide' means that an object (or value) has a property attached - "
        "for instance, containers provide the property 'carrying capacity'. "
        "Here, though, we have %4 rather than the name of a property.");
    return NEVER_MATCH;
}
```

The function PROVISION_KBP_typecheck is called from 5/bp.

§4. Assertion.

```
int PROVISION_KBP_assert(binary_predicate *bp,
        world_object *wo0, specification *spec0,
        world_object *wo1, specification *spec1) {
    kind_of_value *kov = NULL;
    property_name *prn = spec_get_property_name_if_any(spec1);
    if (prn) {
        if (spec_is_generic_CONSTANT(spec0)) kov = spec_get_kind_of_value(spec0);
        if (wo0) kind_provides_property(wo0, prn);
        else if (kov) kov_provides_property(kov, prn);
        else return FALSE;
        return TRUE;
    }
    return FALSE;
}
```

The function PROVISION_KBP_assert is called from 5/bp.

§5. Compilation. Run-time is too late to change which objects provide what, so this relation can't be changed at compile time.

```
int PROVISION_KBP_compile(int task, binary_predicate *bp, annotated_i6_schema *asch) {
    if (task == TEST_ATOM_TASK) {
        kind_of_value *kov = cind_kind_of_value_of_term(asch->pt0);
        kind_of_value *kovp = cind_kind_of_value_of_term(asch->pt1);
        if (is_kova(kovp, OBJECT_TY)) return FALSE;
        property_name *prn = spec_get_property_name_if_any(asch->pt1.constant);
        if (prn) {
            if (is_kova(kov, OBJECT_TY)) {
                if (prn_is_value_property(prn))
                    sch_write_to_existing(asch->schema, "WhetherProvides(*1, false, *2)");
                else
                    sch_write_to_existing(asch->schema, "WhetherProvides(*1, true, *2)");
            } else if ((kov) && (does_kov_provide(kov, prn)))
                sch_write_to_existing(asch->schema, "true");
            else if ((kov) && (does_kov_provide(kov, prn) == FALSE))
                sch_write_to_existing(asch->schema, "false");
            else return FALSE;
            return TRUE;
        }
    }
    return FALSE;
}
```

The function PROVISION_KBP_compile is called from 5/bp.

§6. Problem message text.

```
int PROVISION_KBP_describe_for_problems(binary_predicate *bp, char *problem_buffer) {
    return FALSE;
}
```

The function PROVISION_KBP_describe_for_problems is called from 5/bp.

Measurement Adjectives 9/madj

Purpose

To define adjectives such as large, wide or roomy, which make implicit comparisons of the size of some numerical property, and which (unlike other adjectives) lead to comparative and superlative forms.

Template interpreter commands

```
1   {-callv:validate_definitions}
```

Definitions

¶1. A typical example would be:

> Definition: A container is roomy if its carrying capacity is 10 or more.

Here the domain of the definition is "container", and we must assign an adjective meaning for "roomy" which involves the comparison of a property (here "carrying capacity") against a threshold value ("10") with a given sign (+1, indicating that we need to be at this threshold or above, not at it or below). We then need to create the comparative form "roomier" as a relation, and the superlative "roomiest" as a phrasal form – not in general an adjective, since its domain is too ambiguous in text such as:

> if the canvas bag is roomiest, ...

which begs the question: roomiest out of what? All containers, or implicitly some subcollection of them? So we avoid the problem by allowing superlatives only when explicitly followed by a domain:

> roomiest container in Heathrow Terminal 5

The word "roomy" is the headword, so called in the lexicography sense – other forms are derived from it but they all appear under "roomy" in the Phrasebook.

The implementation of measurement adjectives is difficult for reasons of timing during NI's run: the names of kinds, properties and values become available at different times; whereas we need the name of the adjective itself to become available very early on. This is why the structure below appears to record a lot of extraneous clutter apparently needed only temporarily during parsing – because parsing does not happen all at once, and partial results have to be parked in the structure after one stage to be picked up at the next.

¶2. Details are held in the following structure:

```
typedef struct measurement_definition {
    struct parse_node *measurement_node;                    where the actual definition is
    struct property_name *prop;                             the property being compared, if any
    int headword;                    and the number of the word being defined (must be single word)
    int property_sign;                                      +1 if it must be >= the threshold, -1 if <=
    int property_threshold;                                 numerical value of threshold
    int property_threshold_evaluated;                       have we evaluated this one yet?
    int property_threshold_w1, property_threshold_w2;       text of threshold value
    int pname_w1, pname_w2;                          text of the name of the property to compare
    int property_schema_written;                            I6 schema for testing written yet?
    int superlative_wn;                                     the word number of the superlative form
    struct adjective_meaning *am_of_mdef;                   which adjective meaning
    MEMORY_MANAGEMENT
} measurement_definition;
```

The structure measurement_definition is private to this section.

---

§**1.** Property measurement adjectives.

```
measurement_definition *mdef_new(parse_node *q) {
    measurement_definition *mdef = CREATE(measurement_definition);
    mdef->measurement_node = q;
    mdef->prop = NULL;
    mdef->headword = -1;
    mdef->property_sign = 1;
    mdef->property_threshold = 0;
    mdef->property_threshold_evaluated = FALSE;
    mdef->property_threshold_w1 = -1; mdef->property_threshold_w2 = -1;
    mdef->pname_w1 = -1; mdef->pname_w2 = -1;
    mdef->property_schema_written = FALSE;
    mdef->superlative_wn = -1;
    mdef->am_of_mdef = NULL;
    return mdef;
}
adjective_meaning *MEASUREMENT_KADJ_parse(parse_node *q,
    int sense,
    int adj_name_w1, int adj_name_w2,
    int dom_name_w1, int dom_name_w2,
    int cond_w1, int cond_w2,
    int called_w1, int called_w2) {
    if (sense == 0) return NULL;
    if (![[cond_w1, cond_w2 == its/his/her/their ... --> cond_w1, cond_w2]]) return NULL;
    property_name *prop = NULL;
    int pw1, pw2, thr1, thr2, i;
    int psign = 0;
    if [[cond_w1, cond_w2 == ... or more --> cond_w1, cond_w2]] psign = 1;
    else if [[cond_w1, cond_w2 == ... or less --> cond_w1, cond_w2]] psign = -1;
    if (adj_name_w1 != adj_name_w2) {
        if (psign != 0)
            definition_problem(_P_(C9MultiwordGrading),
```

```
            q, "a grading adjective must be a single word",
            "as in 'Definition: a container is large if its "
            "carrying capacity is 10 or more.': 'fairly large' "
            "would not be allowed because it would make no sense "
            "to talk about 'fairly larger' or 'fairly largest'.");
    return NULL;
}
if (called_w1 >= 0) {
    if (psign != 0)
        definition_problem(_P_(C9GradingCalled),
            q, "callings are not allowed when defining grading adjectives",
            "so 'Definition: a container is large if its "
            "carrying capacity is 10 or more.' is fine, but "
            "so 'Definition: a container (called the bag) is large if its "
            "carrying capacity is 10 or more.' is not - then again, there's "
            "very little call for it.");
    return NULL;
}
if (sense != 1) {
    if (psign != 0)
        definition_problem(_P_(C9GradingUnless),
            q, "'unless' is not allowed when defining grading adjectives",
            "so 'Definition: a container is large if its "
            "carrying capacity is 10 or more.' is fine, but "
            "so 'Definition: a container is modest unless its "
            "carrying capacity is 10 or more.' is not - of course a similar "
            "effect could be achieved by "
            "'Definition: a container is modest if its "
            "carrying capacity is 9 or less.'");
    return NULL;
}
if ((([[cond_w1, cond_w2 == ... is ... : i --> pw1, pw2 ... thr1, thr2]] ||
    [[cond_w1, cond_w2 == ... are ... : i --> pw1, pw2 ... thr1, thr2]]) &&
    (![[thr1, thr2 == not ...]])) {
    prop = parse_property_name(pw1, pw2);
} else {
    if (psign != 0) {
        LOG("Tried <$W> psign %d\n", cond_w1, cond_w2, psign);
        definition_problem(_P_(C9GradingMisphrased),
            q, "that definition is wrongly phrased",
            "assuming it was meant to be a grading adjective "
            "like 'Definition: a container is large if its "
            "carrying capacity is 10 or more.'");
    }
    return NULL;
}
if ((psign == 0) && (is_a_literal(thr1, thr2) == NULL)) return NULL;
measurement_definition *mdef = mdef_new(q);
mdef->property_sign = psign;
mdef->pname_w1 = pw1; mdef->pname_w2 = pw2;
mdef->property_threshold_w1 = thr1; mdef->property_threshold_w2 = thr2;
mdef->prop = prop;
```

```
    mdef->headword = adj_name_w1;
    if (psign != 0) {
        mdef->superlative_wn = make_superlative(mdef->headword);
        char superlative_phrase[1024]; int x1, x2;
        sprintf(superlative_phrase,
            " To decide which object is %s ( S - description ) ",
            lw_array[mdef->superlative_wn].lw_text);
        x1 = lexer_wordcount;
        feed_into_lexer(superlative_phrase, FALSE, FALSE);
        x2 = lexer_wordcount-1;
        make_sentence_node(x1, x2, ':');
        LOG("Defining superlative:\n%s:\n", superlative_phrase);
        sprintf(superlative_phrase,
            " (- {-extremal%s",
            (mdef->property_sign>0)?">":"<");
        print_text_to_string(mdef->pname_w1, mdef->pname_w2,
            superlative_phrase+strlen(superlative_phrase));
        sprintf(superlative_phrase+strlen(superlative_phrase),
            ":S}  -) ");
        x1 = lexer_wordcount;
        feed_into_lexer(superlative_phrase, FALSE, FALSE);
        x2 = lexer_wordcount-1;
        make_sentence_node(x1, x2, '.');
        register_recently_lexed_phrases();
        LOG("    %s\n", superlative_phrase);
    }
    adjective_meaning *am = am_new(MEASUREMENT_KADJ,
        STORE_POINTER_measurement_definition(mdef), q->word_ref1, q->word_ref2);
    mdef->am_of_mdef = am;
    am_declare(am, adj_name_w1, adj_name_w2);
    am_pass_task_to_support_routine(am, TEST_ADJECTIVE_TASK);
    am_set_domain_from_word_range(am, dom_name_w1, dom_name_w2);
    return am;
}
void mdef_read_property_details(measurement_definition *mdef, property_name **prn, int *sign)
{
    if (prn) *prn = mdef->prop;
    if (sign) *sign = mdef->property_sign;
}
int mdef_get_headword(measurement_definition *mdef) {
    return mdef->headword;
}
void validate_definitions(void) {
    measurement_definition *mdef;
    LOOP_OVER(mdef, measurement_definition) validate_mdef(mdef);
}
void validate_mdef(measurement_definition *mdef) {
    if ((mdef->prop == NULL) && (mdef->pname_w1 >= 0)) {
        mdef->prop = parse_property_name(mdef->pname_w1, mdef->pname_w2);
        if (mdef->prop == NULL) {
            LOG("Validating mdef with headword $W... <$W>\n",
                mdef->headword, mdef->headword,
```

```
                mdef->pname_w1, mdef->pname_w2);
            definition_problem(_P_(C9GradingUnknownProperty),
                mdef->measurement_node,
                "that definition involves an unknown property",
                "assuming it was meant to be a definition in the "
                "form 'Definition: a container is large if its "
                "carrying capacity is 10 or more.'");
            return;
        }
    }
    if (mdef->property_threshold_evaluated == FALSE) {
        kind_of_value *kov =
            is_a_literal(mdef->property_threshold_w1, mdef->property_threshold_w2);
        if (kov) {
            mdef->property_threshold = last_literal_evaluated;
            if (can_we_cast_kovs(kov,
                prn_get_kind_of_value(mdef->prop)) != ALWAYS_MATCH) {
            definition_problem(_P_(C9GradingWrongKOV),
                mdef->measurement_node,
                "the property value given here is the wrong kind",
                "and does not match the property being looked at.");
            }
        } else {
            LOG("Can't get literal from %d, %d, <$W>\n",
                mdef->property_threshold_w1, mdef->property_threshold_w2,
                mdef->property_threshold_w1, mdef->property_threshold_w2);
            definition_problem(_P_(C9GradingNonLiteral),
                mdef->measurement_node,
                "that definition is wrongly phrased",
                "assuming it was meant to be a grading adjective "
                "like 'Definition: a container is large if its "
                "carrying capacity is 10 or more.'");
        }
        mdef->property_threshold_evaluated = TRUE;
    }
}

void MEASUREMENT_KADJ_compiling_soon(adjective_meaning *am,
    measurement_definition *mdef, int T) {
    if ((mdef->prop) && (mdef->property_threshold_evaluated) &&
        (mdef->property_schema_written == FALSE)) {
        char *operator = "";
        i6_schema *sch = am_set_i6_schema(mdef->am_of_mdef, TEST_ADJECTIVE_TASK, FALSE);
        switch (mdef->property_sign) {
            case 1: operator = ">="; break;
            case 0: operator = "=="; break;
            case -1: operator = "<="; break;
            default: internal_error("unknown property sign");
        }
        sch_write_to_existing_2sd(sch, "t_0.%s %s %d",
            prn_get_i6_identifier(mdef->prop), operator, mdef->property_threshold);
        mdef->property_schema_written = TRUE;
    }
}
```

```
int MEASUREMENT_KADJ_compile(measurement_definition *mdef, int T, OUTPUT_STREAM, ph_stack_frame
*phsf) {
    return FALSE;
}

int MEASUREMENT_KADJ_assert(measurement_definition *mdef,
    world_object *wo_to_assert_on, specification *val_to_assert_on, int parity) {
    validate_mdef(mdef);
    if ((mdef->prop) && (parity == TRUE)) {
        specification *val = new_actual_CONSTANT_spec(prn_get_kind_of_value(mdef->prop));
        val->word_ref1 = mdef->property_threshold_w1;
        val->word_ref2 = mdef->property_threshold_w2;
        join_inference(
            create_property_inference(inference_subject_wo(wo_to_assert_on), mdef->prop, val),
                inference_subject_wo(wo_to_assert_on));
        return TRUE;
    }
    return FALSE;
}

int MEASUREMENT_KADJ_index(measurement_definition *mdef) {
    return FALSE;
}
```

The function MEASUREMENT_KADJ_parse is called from 5/aph.
The function mdef_read_property_details is called from 6/defer, 6/cdefp and 9/cmpbp.
The function mdef_get_headword is called from 9/cmpbp.
The function validate_definitions is invoked by a command in a `.i6t` template file.
The function validate_mdef is called from 6/defer and 9/cmpbp.
The function MEASUREMENT_KADJ_compiling_soon is called from 5/aph.
The function MEASUREMENT_KADJ_compile is called from 5/aph.
The function MEASUREMENT_KADJ_assert is called from 5/aph.
The function MEASUREMENT_KADJ_index is called from 5/aph.

Comparative Relations 9/cmpbp

Purpose

When a measurement adjective like "tall" is defined, so is a comparative relation like "taller".

Definitions

¶1. This section handles the `PROPERTY_COMPARISON_KBP` relations. Unlike the other relations to do with property values, these do not correspond exactly with the properties. Some properties, like "carrying capacity", might never be compared with measurement adjectives; others, like "height", might be compared with more than one ("short", "tall").

§**1. Initial stock.** There is no initial stock of these, since there are no value properties yet when Inform starts up.

```
void PROPERTY_COMPARISON_KBP_create_initial_stock(void) {
}
```

The function PROPERTY_COMPARISON_KBP_create_initial_stock is called from 5/bp.

§**2. Second stock.** When an adjective is defined so that it performs an inequality comparison of a property value, like so:

> Definition: A woman is tall if her height is 68 or more.

...Inform automatically generates comparative ("taller") and superlative ("tallest") forms of the adjective being defined. The "or more" tells Inform that having greater height makes someone more tall, rather than less, so that X is "taller than" Y if the height of X is greater than the height of Y. This is handled as a "comparative" relation.

```
void PROPERTY_COMPARISON_KBP_create_second_stock(void) {
    measurement_definition *mdef;
    LOOP_OVER(mdef, measurement_definition) {
        int sign;
        property_name *prn;
        mdef_read_property_details(mdef, &prn, &sign);
        if (prn) validate_mdef(mdef);          make sure definition is otherwise syntactically sound
        if (prn) {
            int headword = mdef_get_headword(mdef);                word number of, e.g., "tall"
            int comparative_form = make_comparative(headword);                          "taller"
            vocabulary_entry *quiddity = lw_array[make_quiddity(headword)].lw_identity;    "tall-
ness"
            i6_schema *schema_to_compare_property_values;
            ⟨Work out property comparison schema 3⟩;
            ⟨Construct a BP named for the quiddity and tested using the comparative schema 4⟩;
        }
    }
}
```

The function PROPERTY_COMPARISON_KBP_create_second_stock is called from 5/bp.

§**3.** The I6 code needed numerically to compare the property values:

⟨Work out property comparison schema 3⟩ ≡

```
char *i6_pname = prn_get_i6_identifier(prn);
char *i6_op;
switch (sign) {
    case 1: i6_op = ">"; break;
    case -1: i6_op = "<"; break;
    default: i6_op = "=="; break;
}
schema_to_compare_property_values =
    sch_new_3("(*1.%s %s *2.%s)", i6_pname, i6_op, i6_pname);
```

This code is used in §2.

§**4.** The BP arising from "tall" would be called the "tallness relation", for instance. Note that we don't say anything about the domain of x and y in $T(x, y)$. That's because we a value property like "height" may exist for more than one kind of object, and there is not necessarily any unifying kind of value K such all objects satisfying K possess a height and all others do not. (That's intentional: it is Inform's one concession towards multiple-inheritance, and it allows containers and doors to share lockability behaviour despite being of mutually incompatible kinds.)

⟨Construct a BP named for the quiddity and tested using the comparative schema 4⟩ ≡

```
binary_predicate *bp;
bp = make_pair_of_BPs(PROPERTY_COMPARISON_KBP,
    bptd_blank(), bptd_blank(),
    vocab_get_exemplar(quiddity, FALSE), NULL, -1, NULL, NULL,
    schema_to_compare_property_values, quiddity);
bp->comparison_sign = sign; bp->comparative_property = prn;
register_comparative_preposition(comparative_form, bp);
```

This code is used in §2.

§5. Typechecking.

```
int PROPERTY_COMPARISON_KBP_typecheck(binary_predicate *bp,
    kind_of_value **kovs_of_terms, kind_of_value **kovs_required, tc_problem_kit *tck) {
    property_name *prn = kov_get_coinciding_property(kovs_of_terms[1]);
    if ((prn) && (prn != bp->comparative_property)) {
        if (tck->log_to_I6_text)
            LOG("Comparative misapplied to $Y not $Y\n", prn, bp->comparative_property);
        quote_property(4, bp->comparative_property);
        quote_property(5, prn);
        tcp_problem(_P_(C9ComparativeMisapplied), tck,
            "that ought to make a comparison of %4 not %5.");
        return NEVER_MATCH;
    }
    if ((kovs_required[0]) &&
        (can_we_cast_kovs(kovs_of_terms[0], kovs_required[0]) == NEVER_MATCH)) {
        LOG("Term 0 is $u not $u\n", kovs_of_terms[0], kovs_required[0]);
        issue_bp_typecheck_error(bp, kovs_of_terms[0], kovs_of_terms[1], tck);
        return NEVER_MATCH;
    }
    return ALWAYS_MATCH;
}
```

The function PROPERTY_COMPARISON_KBP_typecheck is called from 5/bp.

§6. Assertion.

```
int PROPERTY_COMPARISON_KBP_assert(binary_predicate *bp,
        world_object *wo0, specification *spec0,
        world_object *wo1, specification *spec1) {
    return FALSE;
}
```

The function PROPERTY_COMPARISON_KBP_assert is called from 5/bp.

§7. **Compilation.** We need do nothing special: these relations can be compiled from their schemas.

```
int PROPERTY_COMPARISON_KBP_compile(int task, binary_predicate *bp, annotated_i6_schema *asch)
{
    if (task == TEST_ATOM_TASK) {
        kind_of_value *st[2];
        st[0] = cind_kind_of_value_of_term(asch->pt0);
        st[1] = cind_kind_of_value_of_term(asch->pt1);
        if ((can_we_cast_kovs(st[0], kova(OBJECT_TY)) != NEVER_MATCH) &&
            (kov_name_can_coincide_with_property(st[1]))) {
            char *i6_op = NULL;
            property_name *prn = kov_get_coinciding_property(st[1]);
            if (prn) {
                if (bp->comparison_sign == 1) i6_op = ">";
                if (bp->comparison_sign == 2) i6_op = "==";
                if (bp->comparison_sign == -1) i6_op = "<";
                if (i6_op) {
                    sch_write_to_existing_2(asch->schema,
                        "*1.%s %s *2", prn_get_i6_identifier(prn), i6_op);
                    return TRUE;
                }
            }
        }
    }
    return FALSE;
}
```

The function PROPERTY_COMPARISON_KBP_compile is called from 5/bp.

§8. Problem message text.

```
int PROPERTY_COMPARISON_KBP_describe_for_problems(binary_predicate *bp, char *problem_buffer)
{
    return FALSE;
}
```

The function PROPERTY_COMPARISON_KBP_describe_for_problems is called from 5/bp.

Value-Property Relations 9/vpbp

Purpose

Each value property has two associated relations, one to set the value for a single holder and the other to compare its value between two holders.

Definitions

¶1. This section handles two different kinds of BP: `PROPERTY_SETTING_KBP` and `PROPERTY_SAME_KBP`.

---

§**1. Initial stock.** There is no initial stock of these, since there are no value properties yet when Inform starts up.

```
void PROPERTY_SETTING_KBP_create_initial_stock(void) {
}
void PROPERTY_SAME_KBP_create_initial_stock(void) {
}
```

The function PROPERTY_SETTING_KBP_create_initial_stock is called from 5/bp.
The function PROPERTY_SAME_KBP_create_initial_stock is called from 5/bp.

§**2. Subsequent creations.** BPs like this lead to a timing problem, because we have to create the relation early enough that we can make sense of the sentences in the source text; but at that early time, the properties haven't been created yet. We therefore store the text of the property name (say, "weight") in `property_pending_w1`, `property_pending_w2` and come back to it later on.

```
binary_predicate *make_set_property_BP(int w1, int w2) {
    binary_predicate *bp = make_pair_of_BPs(PROPERTY_SETTING_KBP,
        bptd_thing(), bptd_blank(),
        "set-property", NULL, -1, NULL, NULL, NULL, NULL);
    bp->property_pending_w1 = w1;
    bp->property_pending_w2 = w2;
    bp->reversal->property_pending_w1 = w1;
    bp->reversal->property_pending_w2 = w2;
    return bp;
}
```

The function make_set_property_BP is called from 5/conj and 9/prop.

§**3.** Meanwhile, we can't look up the BP with reference to the property, since the property may not exist yet; we have to use the text of the name of the property as a key, clumsy as that may seem.

```
binary_predicate *find_set_property_BP(int w1, int w2) {
    binary_predicate *bp;
    LOOP_OVER(bp, binary_predicate)
        if (compare_word_range(w1, w2, bp->property_pending_w1, bp->property_pending_w2))
            if (bp->right_way_round)
                return bp;
    return NULL;
}
```

The function find_set_property_BP is called from 9/prop.

§**4.** ...And now it's "later on". We fix these BPs in original-reversal pairs, so that the two can never fall out of step.

```
void fix_property_bp(binary_predicate *bp) {
    property_name *prn;
    binary_predicate *bpr = bp->reversal;
    int w1 = bp->property_pending_w1, w2 = bp->property_pending_w2;
    if (w1 < 0) return;
    prn = parse_property_name(w1, w2);
    bp->property_pending_w1 = -1; bp->property_pending_w2 = -1;
    bpr->property_pending_w1 = -1; bpr->property_pending_w2 = -1;
    if (prn == NULL) {
        current_sentence = bp->bp_created_at;
        sentence_problem(_P_(C9RelationWithBadProperty),
            "that doesn't seem to be a property",
            "perhaps because you haven't defined it yet?");
        return;
    }
    if (prn_is_either_or(prn)) {
        current_sentence = bp->bp_created_at;
        sentence_problem(_P_(C9RelationWithEitherOrProperty),
            "verbs can only set properties with values",
            "not either/or properties like this one.");
        return;
    }
    bp->set_property = prn; bpr->set_property = prn;
    if (bp->right_way_round) set_property_BP_schemas(bp, prn);
    else set_property_BP_schemas(bpr, prn);
}
```

The function fix_property_bp is called from 8/relv and 9/prop.

§**5. Second stock.** The following is called after all properties have been created, making it the perfect opportunity to go over all of the property-setting BPs:

```
void PROPERTY_SETTING_KBP_create_second_stock(void) {
    binary_predicate *bp;
    LOOP_OVER(bp, binary_predicate)
        if (bp->property_pending_w1 >= 0)
            fix_property_bp(bp);
}
```

The function PROPERTY_SETTING_KBP_create_second_stock is called from 5/bp.

§**6.** Note that we read and write to the property directly, without asking the template layer to check if the given object has permission to possess that property. We can afford to do this because type-checking at compile time guarantees that it does have permission, and as a result we gain some speed and simplicity.

```
void set_property_BP_schemas(binary_predicate *bp, property_name *prn) {
    bp->test_function = sch_new_1("*1.%s == *2", prn_get_i6_identifier(prn));
    bp->make_true_function = sch_new_1("*1.%s = *2", prn_get_i6_identifier(prn));
    bp->term_details[1].implies_kov = prn_get_kind_of_value(prn);
}
```

§**7.** Next, an equality comparison between property values also exists automatically for every value property. If, as in the above example, there is a value property called "height" then a BP is constructed in order to serve as the meaning of "the same height as" in text like this:

if Ms Cregg is the same height as Big Bird, ...

We again have two schemas, because it makes sense not only to perform the comparison but also to force it true thus:

now Ms Cregg is the same height as Big Bird;

(That couldn't be arranged for strict inequality comparisons like "taller than" because it is unclear just how much taller than Big Bird we would have to make C. J.)

```
void PROPERTY_SAME_KBP_create_second_stock(void) {
    property_name *prn;
    LOOP_OVER(prn, property_name) {
        if (prn_is_value_property(prn)) {
            vocabulary_entry *rel_name;
            char *i6_pname = prn_get_i6_identifier(prn);
            ⟨Work out the name for the same-property-value-as relation 8⟩;
            binary_predicate *bp = make_pair_of_BPs(PROPERTY_SAME_KBP,
                bptd_blank(), bptd_blank(),
                vocab_get_exemplar(rel_name, FALSE), NULL, -1, NULL,
                sch_new_2("*1.%s = *2.%s", i6_pname, i6_pname),
                sch_new_2("*1.%s == *2.%s", i6_pname, i6_pname),
                rel_name);
            bp->same_property = prn;
            register_same_property_as_preposition(bp);
        }
    }
}
```

The function PROPERTY_SAME_KBP_create_second_stock is called from 5/bp.

§**8.** In I7 source text, this relation is called "same-height-as", but we don't mention this in the documentation because (for timing reasons) it doesn't exist when the new-verb sentences are being parsed: so writing

> The verb to be level with implies the same-height-as relation.

cannot work. Nothing is really lost by this, since it's easy enough to define an identically-behaving relation by hand:

> Levelling relates a person (called Mr X) to a person (called Mr Y) when the height of Mr X is the height of Mr Y.
>
> The verb to be level with implies the levelling relation.

Relations need to have single-word names, but properties don't, so we shrink spaces to hyphens: thus, for instance, "same-carrying-capacity-as". We also truncate to a reasonable length, ensuring that the result doesn't exceed `MAX_WORD_LENGTH` or overflow our storage for a debugging-log name.

⟨Work out the name for the same-property-value-as relation 8⟩ ≡

```
int w1 = prn->word_ref1, w2 = prn->word_ref2;
int i7_name_wn = lexer_wordcount;
char i7_name[STRING_TOLERANCE_LIMIT + 10];
sprintf(i7_name, "same-");
print_modest_sized_text_to_string(w1, w2, i7_name + strlen(i7_name));
sprintf(i7_name + strlen(i7_name), "-as");
int i; for (i=0; i7_name[i]; i++) if (i7_name[i] == ' ') i7_name[i] = '-';
feed_into_lexer(i7_name, TRUE, FALSE);
rel_name = lw_array[i7_name_wn].lw_identity;
```

This code is used in §7.

§9. Typechecking.

```
int PROPERTY_SETTING_KBP_typecheck(binary_predicate *bp,
        kind_of_value **kovs_of_terms, kind_of_value **kovs_required, tc_problem_kit *tck) {
    property_name *prn = bp->set_property;
    kind_of_value *val_kov = prn_get_kind_of_value(prn);
    if (can_we_cast_kovs(kovs_of_terms[1], val_kov) == NEVER_MATCH) {
        LOG("Property value given as $u not $u\n", kovs_of_terms[1], val_kov);
        quote_kov(4, kovs_of_terms[1]);
        quote_kov(5, val_kov);
        quote_property(6, prn);
        tcp_problem(_P_(BelievedImpossible), tck,          the A-parser won't generate these now
            "that tries to set the value of the '%6' property to %4 - which "
            "must be wrong because this property has to be %5.");
        return NEVER_MATCH;
    }
    if (kov_has_properties(kovs_of_terms[0]) == FALSE) {
        LOG("Property value for impossible domain $u\n", kovs_of_terms[0]);
        quote_kov(4, kovs_of_terms[0]);
        quote_property(5, prn);
        tcp_problem(_P_(C9PropForBadKOV), tck,
            "that tries to set the property '%5' for %4. Values of that kind "
            "are not allowed to have properties. (Some kinds of value are, "
            "some aren't - see the Kinds index for details. It's a matter "
            "of what is practical in terms of how much memory is needed.)");
        return NEVER_MATCH;
```

```
    }
    return ALWAYS_MATCH;
}
```

The function PROPERTY_SETTING_KBP_typecheck is called from 5/bp.

§10.

```
int PROPERTY_SAME_KBP_typecheck(binary_predicate *bp,
        kind_of_value **kovs_of_terms, kind_of_value **kovs_required, tc_problem_kit *tck) {
    return DECLINE_TO_MATCH;
}
```

The function PROPERTY_SAME_KBP_typecheck is called from 5/bp.

§11. Assertion.

```
int PROPERTY_SETTING_KBP_assert(binary_predicate *bp,
        world_object *wo0, specification *spec0,
        world_object *wo1, specification *spec1) {
    property_name *prn = bp->set_property;
    specification *spec_owner = spec0;
    specification *val_set = spec1;
    world_object *owner_if_wo = wo0;
    world_object *val_if_wo = wo1;
    inference_subject infs;
    if (owner_if_wo) infs = inference_subject_wo(owner_if_wo);
    else infs = inference_subject_spec(spec_owner);
    inference *i = create_property_inference(infs, prn, val_set);
    i->word_ref1 = val_set->word_ref1; i->word_ref2 = val_set->word_ref2;
    join_inference(i, infs);
    if (val_if_wo) {
        inf_set_object_reference(i, val_if_wo);
        if (prn == other_side_PRN) {
            i = create_inference(ISAROOM_INF, CERTAIN_CE);
            join_inference(i, inference_subject_wo(val_if_wo));
        }
    }
    return TRUE;
}
int PROPERTY_SAME_KBP_assert(binary_predicate *bp,
        world_object *wo0, specification *spec0,
        world_object *wo1, specification *spec1) {
    return FALSE;
}
```

The function PROPERTY_SETTING_KBP_assert is called from 5/bp.
The function PROPERTY_SAME_KBP_assert is called from 5/bp.

§**12. Compilation.** We need do nothing special: these relations can be compiled from their schemas.

```
int PROPERTY_SETTING_KBP_compile(int task, binary_predicate *bp, annotated_i6_schema *asch)
{
    return FALSE;
}
int PROPERTY_SAME_KBP_compile(int task, binary_predicate *bp, annotated_i6_schema *asch) {
    return FALSE;
}
```

The function PROPERTY_SETTING_KBP_compile is called from 5/bp.
The function PROPERTY_SAME_KBP_compile is called from 5/bp.

§**13.**

```
property_name *bp_get_same_as_property(binary_predicate *bp) {
    return bp->same_property;
}
int bp_sets_a_property(binary_predicate *bp) {
    if ((bp->set_property) || (bp->property_pending_w1 >= 0)) return TRUE;
    return FALSE;
}
```

The function bp_get_same_as_property is called from 5/conj.
The function bp_sets_a_property is called from 8/mass.

§14. Problem message text.

```
int PROPERTY_SETTING_KBP_describe_for_problems(binary_predicate *bp, char *problem_buffer) {
    return FALSE;
}
int PROPERTY_SAME_KBP_describe_for_problems(binary_predicate *bp, char *problem_buffer) {
    return FALSE;
}
```

The function PROPERTY_SETTING_KBP_describe_for_problems is called from 5/bp.
The function PROPERTY_SAME_KBP_describe_for_problems is called from 5/bp.

Inferences 9/inf

Purpose

To manage the individual pieces of information gathered, with varying degrees of certainty, from assertion sentences. This is mostly information about which objects have what properties.

9/inf.§6 Inference subjects; §7 Comparing inferences; §8-10 Joining an inference to what is known about an object

Definitions

¶1. NI reads a natural language description of a world. As it runs through, it collects together the assertions made in this description (such as "On the table is a hat"), which are sometimes vague (where is this table?), sometimes imply further facts (the hat cannot be a room, and the table must be a supporter) and are sometimes contradictory (if, for instance, "The hat is in the hatbox" has also been read).

NI must form a model world consistent with the assertions made, in such a way that the designer would not be surprised by what it has done. To avoid surprises, Occam's Razor is applied: the simplest and most likely explanation (likely from the standpoint of interactive fiction) should be assumed.

Information being gathered is not stored in the raw form of assertion sentences like "The portcullis is a door" but in instances of a data structure called an `inference`.

Each inference is associated with a single specific object, even when it also concerns a second object: for instance, the knowledge that going east from room A takes one to room B is an inference associated with A.

¶2. Inferences come in the following types:

```
define ISAROOM_INF 1                    is O a room?
define ISWORN_INF 2                     is O initially worn by someone?
define CONTAINSTHINGS_INF 3             does O contain things?
define PARENTAGE_INF 4                  where is O in the object tree?
define PARENTAGE_HERE_INF 5             described vaguely as "here"?
define FOUNDIN_INF 6                    for backdrop things in many places
define PARTOF_INF 7                     is O a part of another object?
define ARBITRARY_INF 8                  records an arbitrary various-various relation
define VALUEARBITRARY_INF 9             ditto but where one term is a value
define MATCHING_INF 10                  matching property values: used for equivalence relations
define DIRECTION_INF 11                 where do map connections from O lead?
define PROPERTY_INF 12                  what are the properties of O?
define BLINDPROPERTY_INF 13             deletes inferences attached to certain I6 properties
define GLOBAL_INF 14                    what is the initial value of this global variable?
```

¶3. Not all information is positive, or certain. The likelihood of an inference being true is measured on a five-point scale, using the following values:

```
define IMPOSSIBLE_CE -2
define UNLIKELY_CE -1
define UNKNOWN_CE 0
define LIKELY_CE 1
define CERTAIN_CE 2
```

¶4. When a given sentence is being parsed, there is a prevailing mood of certainty or uncertainty about the information implied by it, and this is stored in the following global variable:

```
int prevailing_mood = UNKNOWN_CE;
```

¶5. This is the data structure used to store a single inference. Inferences are generally organised in linked lists, hence the `next` field. When references are not given (for instance the second object, or the property name) they will be NULL in the case of structure pointers, or `-1` in the case of word references.

```
typedef struct inference {
    int inference_type;                                   see above
    int certainty;                                        see above
    struct parse_node *inferred_from;                     from what sentence was this drawn?
    struct world_object *obj_ref1;                        first object referred to
    struct world_object *obj_ref2;                        second object, if any
    struct property_name *pname;                          property referred to, if any
    struct specification *val_ref1;                       first value referred to, if any
    struct specification *val_ref2;                       second value referred to, if any
    struct binary_predicate *bp_ref;                      relationship asserted, for ARBITRARY infs
    int word_ref1, word_ref2;                             property value described here in source
    int indices[2];                                       used when constructing V-to-V relation arrays
    struct inference *next;                               next in list of inferences on same subject
} inference;
```

```
define POSITIVE_KNOWLEDGE_LOOP(inf, wo, type)
    for (inf = wo->knowledge; inf; inf = inf->next)
        if ((inf->inference_type == type) && (inf->certainty > 0))
define KNOWLEDGE_LOOP(inf, wo, type)
    for (inf = wo->knowledge; inf; inf = inf->next)
        if (inf->inference_type == type)
define GENERAL_POSITIVE_KNOWLEDGE_LOOP(inf, infs, type)
    for (inf = *(infs_get_list_head(infs)); inf; inf = inf->next)
        if ((inf->inference_type == type) && (inf->certainty > 0))
define GENERAL_KNOWLEDGE_LOOP(inf, infs, type)
    for (inf = *(infs_get_list_head(infs)); inf; inf = inf->next)
        if (inf->inference_type == type)
```

The structure inference is private to this section.

¶6. An inference subject is a convenient holder for the different concepts in Inform about which inferences can be drawn (at present, objects, values, KOVs, BPs).

```
define WO_INFS 1
define VAL_INFS 2
define KOV_INFS 3
define BP_INFS 4

typedef struct inference_subject {
    int inference_subject_type;                    one of the *_INFS constants above
    struct general_pointer subject;                pointer to specific subject
} inference_subject;
```

The structure inference_subject is private to this section.

---

§**1.** The following routine coins a newly minted inference which is not yet attached to any object: since every inference concerns some object, it will not stay unattached for long.

§**2.** The `prevailing_mood` is an air of certainty or doubt which hangs over the current sentence being read, and applies to all of the inferences drawn from it. If nothing has been said about likelihood, the sentence is assumed to be factually certain.

```
inference *create_inference(int type, int certitude) {
    inference *new_i;
    new_i = CREATE(inference);
    new_i->next = NULL;
    new_i->inference_type = type;
    new_i->obj_ref1 = NULL;
    new_i->obj_ref2 = NULL;
    new_i->bp_ref = NULL;
    new_i->word_ref1 = -1;
    new_i->word_ref2 = -1;
    new_i->pname = NULL;
    new_i->certainty = certitude;
    if (certitude == UNKNOWN_CE) new_i->certainty = CERTAIN_CE;
    new_i->inferred_from = current_sentence;
    new_i->val_ref1 = NULL;
    new_i->val_ref2 = NULL;
    new_i->indices[0] = -1; new_i->indices[1] = -1;
    return new_i;
}
parse_node *inf_inferred_from(inference *i) {
    return i->inferred_from;
}
int inf_get_certainty(inference *i) {
    return i->certainty;
}
void inf_set_certainty(inference *i, int certitude) {
    i->certainty = certitude;
}
void inf_reverse_certainty(inference *i) {
    i->certainty = -i->certainty;
```

```
}
void inf_set_object_reference(inference *i, world_object *wo) {
    i->obj_ref1 = wo;
    i->obj_ref2 = NULL;
}
void inf_set_object_references(inference *i,
    world_object *wo1, world_object *wo2) {
    i->obj_ref1 = wo1;
    i->obj_ref2 = wo2;
}
void inf_get_all_references(inference *i, specification **spec1, world_object **wo1, specification
**spec2, world_object **wo2, int **indices) {
    *wo1 = i->obj_ref1; *wo2 = i->obj_ref2; *spec1 = i->val_ref1; *spec2 = i->val_ref2;
    *indices = i->indices;
}
void inf_set_all_references(inference *i, specification *spec1, world_object *wo1, specification
*spec2, world_object *wo2) {
    i->obj_ref1 = wo1; i->obj_ref2 = wo2; i->val_ref1 = spec1; i->val_ref2 = spec2;
}
world_object *inf_get_object_reference(inference *i) {
    return i->obj_ref1;
}
void inf_get_object_references(inference *i,
    world_object **ref1, world_object **ref2) {
    *ref1 = i->obj_ref1;
    *ref2 = i->obj_ref2;
}
binary_predicate *inf_get_bp(inference *i) {
    return i->bp_ref;
}
void inf_set_bp(inference *i, binary_predicate *bp) {
    i->bp_ref = bp;
}
```

The function create_inference is called from 6/asp, 9/kind, 9/spabp, 9/mapbp, 9/exrel and 9/vpbp.
The function inf_inferred_from is called from 2/prob3, 9/model and 9/cot.
The function inf_get_certainty is called from 8/imp, 9/model, 9/cot and 9/inpw.
The function inf_set_certainty is called from 9/spabp.
The function inf_reverse_certainty is called from 9/prop.
The function inf_set_object_reference is called from 6/asp, 9/spabp and 9/vpbp.
The function inf_set_object_references is called from 9/spabp, 9/mapbp and 9/exrel.
The function inf_get_all_references is called from 9/cot.
The function inf_set_all_references is called from 9/exrel.
The function inf_get_object_reference is called from 9/model, 9/cot and 9/inpw.
The function inf_get_object_references is called from 9/model and 9/cot.
The function inf_get_bp is called from 9/cot.
The function inf_set_bp is called from 9/exrel.

§**3.** This utility routine which looks for the "opposite" property in the linked list of inferences belonging to an object. (This is a property of directions.) Crude, but not time-sensitive, and there seems little point in writing this any better.

```
world_object *get_value_of_opposite_property(world_object *wo) {
    inference *i;
    KNOWLEDGE_LOOP(i, wo, PROPERTY_INF) {
        if (strcmp(lw_array[inf_get_property_name(i)->word_ref1].lw_text, "opposite") == 0)
            return i->obj_ref1;
    }
    return NULL;
}
```

The function get_value_of_opposite_property is called from 8/relv and 9/cot.

§**4.**

```
inference *create_global_inference(inference_subject infs, specification *val) {
    PROTECTED_MODEL_PROCEDURE;
    inference *i = create_inference(GLOBAL_INF, prevailing_mood);
    if (prevailing_mood == UNKNOWN_CE) i->certainty = infs_default_certainty(infs);
    i->val_ref1 = val;
    return i;
}
```

The function create_global_inference is called from 6/equal.

§**5.** Many inferences simply store property values, so we have a convenient routine to create those. But note that in this case, a statement of no explicit likelihood about a kind is taken to be merely "likely", not "certain": thus "The wooden box has capacity 2" is `CERTAIN_CE`, while "A container has capacity 2" is merely `LIKELY_CE`. That difference permits the user to type "A container with capacity 4" without a contradiction arising. Had the original sentence been "A container always has capacity 2", a clash would occur which would cause a Problem to be issued.

```
inference *create_property_inference(inference_subject infs,
    property_name *prn, specification *val) {
    PROTECTED_MODEL_PROCEDURE;
    inference *i = create_inference(PROPERTY_INF, prevailing_mood);
    if (prevailing_mood == UNKNOWN_CE) i->certainty = infs_default_certainty(infs);
    i->pname = prn;
    i->val_ref1 = val;
    if (prn == NULL) internal_error("null property inference");
    return i;
}

void inf_set_property_name(inference *i, property_name *prn) {
    i->pname = prn;
}

property_name *inf_get_property_name(inference *i) {
    return i->pname;
}

specification *inf_get_literal_value(inference *i) {
    return i->val_ref1;
}
```

```
void inf_make_blind_property(inference *i) {
    i->inference_type = BLINDPROPERTY_INF;
}

void log_inference_type(int it) {
    switch(it) {
        case CONTAINSTHINGS_INF: LOG("CONTAINSTHINGS_INF"); break;
        case ISWORN_INF: LOG("ISWORN_INF"); break;
        case PARTOF_INF: LOG("PARTOF_INF"); break;
        case ARBITRARY_INF: LOG("ARBITRARY_INF"); break;
        case VALUEARBITRARY_INF: LOG("VALUEARBITRARY_INF"); break;
        case MATCHING_INF: LOG("MATCHING_INF"); break;
        case ISAROOM_INF: LOG("ISAROOM_INF"); break;
        case PARENTAGE_INF: LOG("PARENTAGE_INF"); break;
        case PARENTAGE_HERE_INF: LOG("PARENTAGE_HERE_INF"); break;
        case FOUNDIN_INF: LOG("FOUNDIN_INF"); break;
        case DIRECTION_INF: LOG("DIRECTION_INF"); break;
        case PROPERTY_INF: LOG("PROPERTY_INF"); break;
        case GLOBAL_INF: LOG("GLOBAL_INF"); break;
        default: LOG("<unknown-inference-type>"); break;
    }
}

void log_inference(inference *in) {
    if (in == NULL) { LOG("<null-inference>"); return; }
    log_inference_type(in->inference_type);
    LOG(" -");
    switch(in->certainty) {
        case IMPOSSIBLE_CE: LOG(" Impossible "); break;
        case UNLIKELY_CE: LOG(" Unlikely "); break;
        case UNKNOWN_CE: LOG(" <No information> "); break;
        case LIKELY_CE: LOG(" Likely "); break;
        case CERTAIN_CE: LOG(" Certain "); break;
        default: LOG("<unknown-certainty>"); break;
    }
    if (in->pname) LOG("($W)", in->pname->word_ref1, in->pname->word_ref2);
    if (in->word_ref1 != -1) LOG("- $W", in->word_ref1, in->word_ref2);
    if (in->obj_ref1) LOG("- 1:$O", in->obj_ref1);
    if (in->obj_ref2) LOG("- 2:$O", in->obj_ref2);
    if (in->val_ref1) LOG("- literal:$S", in->val_ref1);
}
```

The function create_property_inference is called from 9/qty, 9/spabp, 9/exrel, 9/prop, 9/madj and 9/vpbp.

The function inf_set_property_name is called from 9/exrel.

The function inf_get_property_name is called from 2/prob3, 8/imp, 9/pp, 9/model, 9/cot and 9/inpw.

The function inf_get_literal_value is called from 9/qty, 9/model and 9/cot.

The function inf_make_blind_property is called from 9/model.

The function log_inference_type is called from 2/dl.

The function log_inference is called from 2/dl.

§6. Inference subjects.

```
void log_inference_subject(inference_subject *infs) {
    switch(infs->inference_subject_type) {
        case WO_INFS: LOG("WO_INFS:$O",
            RETRIEVE_POINTER_world_object(infs->subject)); break;
        case VAL_INFS: LOG("VAL_INFS:$Q",
            RETRIEVE_POINTER_quantity(infs->subject)); break;
        case KOV_INFS: LOG("KOV_INFS:$u",
            RETRIEVE_POINTER_kind_of_value(infs->subject)); break;
        case BP_INFS: LOG("BP_INFS:$2",
            RETRIEVE_POINTER_binary_predicate(infs->subject)); break;
        default: LOG("<bad-inference-subject>"); break;
    }
}

inference_subject inference_subject_wo(world_object *wo) {
    inference_subject infs;
    infs.inference_subject_type = WO_INFS;
    infs.subject = STORE_POINTER_world_object(wo);
    return infs;
}

inference_subject inference_subject_spec(specification *spec) {
    inference_subject infs;

    if (spec_is_generic_CONSTANT(spec)) {
        kind_of_value *kov = spec_get_kind_of_value(spec);
        infs.inference_subject_type = KOV_INFS;
        infs.subject = STORE_POINTER_kind_of_value(kov);
    } else {
        quantity *q = spec_get_constant_quantity_if_any(spec);
        if (q == NULL) {
            issue_bad_owner_problem();
            infs.inference_subject_type = VAL_INFS;
            infs.subject = STORE_POINTER_quantity(max_score_quantity);
        } else {
            infs.inference_subject_type = VAL_INFS;
            infs.subject = STORE_POINTER_quantity(q);
        }
    }
    return infs;
}

inference_subject inference_subject_q(quantity *q) {
    inference_subject infs;
    if (q == NULL) internal_error("can't form inference subject from this Q");
    infs.inference_subject_type = VAL_INFS;
    infs.subject = STORE_POINTER_quantity(q);
    return infs;
}

inference_subject inference_subject_kov(kind_of_value *kov) {
    inference_subject infs;
    if (kov == NULL) internal_error("can't form inference subject from null KOV");
    infs.inference_subject_type = KOV_INFS;
    infs.subject = STORE_POINTER_kind_of_value(kov);
```

```
    return infs;
}

inference_subject inference_subject_bp(binary_predicate *bp) {
    inference_subject infs;
    if (bp == NULL) internal_error("can't form inference subject from null BP");
    infs.inference_subject_type = BP_INFS;
    infs.subject = STORE_POINTER_binary_predicate(bp);
    return infs;
}

int infs_default_certainty(inference_subject infs) {
    switch(infs.inference_subject_type) {
        case WO_INFS: {
            world_object *wo = RETRIEVE_POINTER_world_object(infs.subject);
            if ((wo) && (wo->kind_flag)) return LIKELY_CE;
        }
        case VAL_INFS:
            return CERTAIN_CE;
        case KOV_INFS:
            return LIKELY_CE;
        case BP_INFS:
            return CERTAIN_CE;
    }
    return CERTAIN_CE;
}

inference **infs_get_list_head(inference_subject infs) {
    switch(infs.inference_subject_type) {
        case WO_INFS: {
            world_object *wo = RETRIEVE_POINTER_world_object(infs.subject);
            return &(wo->knowledge);
        }
        case VAL_INFS: {
            quantity *q = RETRIEVE_POINTER_quantity(infs.subject);
            return qty_get_knowledge(q);
        }
        case KOV_INFS: {
            kind_of_value *kov = RETRIEVE_POINTER_kind_of_value(infs.subject);
            return kov_get_knowledge(kov);
        }
        case BP_INFS: {
            binary_predicate *bp = RETRIEVE_POINTER_binary_predicate(infs.subject);
            return bp_get_inferences(bp);
        }
    }
    return NULL;
}

int compile_inferred_value(OUTPUT_STREAM, inference_subject infs, property_name *prn) {
    inference *inf = *(infs_get_list_head(infs));
    while (inf) {
        if (prn_is_either_or(prn)) {
            if ((inf->inference_type == PROPERTY_INF) && (inf->pname == prn)) {
                if (inf->certainty > 0) { WRITE("1"); return TRUE; }
                else { WRITE("0"); return TRUE; }
```

```
            }
            if ((inf->inference_type == PROPERTY_INF) &&
                (prn_either_or_get_negation(prn) == inf->pname)) {
                if (inf->certainty > 0) { WRITE("0"); return TRUE; }
                else { WRITE("1"); return TRUE; }
            }
        } else {
            if ((inf->inference_type == PROPERTY_INF) && (inf->pname == prn) &&
                (inf->certainty > 0)) {
                if (inf_get_object_reference(inf) != NULL)
                    WRITE("%s",
                        wo_get_I6_representation(inf_get_object_reference(inf)));
                else {
                    specification *val = inf_get_literal_value(inf);
                    current_sentence = inf_inferred_from(inf);
                    if (val) {
                        BEGIN_COMPILATION_MODE;
                        COMPILATION_MODE_EXIT(DEREFERENCE_POINTERS_CMODE);
                        spec_compile(OUT, val);
                        END_COMPILATION_MODE;
                    } else
                        sentence_problem(_P_(BelievedImpossible),
                            "a property value seems to be missing",
                            "since this is not an either/or property.");
                }
                return TRUE;
            }
        }
        inf = inf->next;
    }
    return FALSE;
}
```

The function log_inference_subject is called from 2/dl.

The function inference_subject_wo is called from 6/asp, 9/qty, 9/kind, 9/spabp, 9/mapbp, 9/exrel, 9/prop, 9/madj and 9/vpbp.

The function inference_subject_spec is called from 9/qty, 9/prop and 9/vpbp.

The function inference_subject_q is called from 6/equal, 9/qty and 9/prop.

The function inference_subject_kov is called from 9/prop.

The function inference_subject_bp is called from 9/exrel and 9/cot.

The function compile_inferred_value is called from 9/prop.

§**7. Comparing inferences.** The following routine is a little like `strcmp`, the standard C routine for comparing strings, in that it compares two inferences and returns a value useful for sorting algorithms: 0 if equal, positive if `i1 < i2`, negative if `i2 < i1`. As we shall see, inferences are insertion-sorted into their linked lists as they are made, but this comparison is also used to watch out for contradictions which arise: the magnitude of the number returned here is used as an indication of how similar are the questions for which the inferences provide answers. (Once it reaches level 5, we know that the two inferences may be clashing with each other, or may be reiterating each other, or simply providing further certainty.)

There's a lot of crude conversion of pointers to integers here, but it doesn't much matter what ordering is used so long as a definite and unchanging ordering is used.

```
int compare_inferences(inference *i1, inference *i2) {
    pointer_sized_int c;
    property_name *pr1, *pr2;
    if (i1 == i2) return 0;
    if (i1 == NULL) return 1;
    if (i2 == NULL) return -1;
    c = i1->inference_type - i2->inference_type;
    if (c > 0) return 2; if (c < 0) return -2;
    c = ((pointer_sized_int) i1->bp_ref) - ((pointer_sized_int) i2->bp_ref);
    if (c > 0) return 3; if (c < 0) return -3;
    pr1 = i1->pname; pr2 = i2->pname;
    if ((pr1) && (prn_is_either_or(pr1)) &&
        (pr2) && (prn_is_either_or(pr2)) &&
        ((pr1 == prn_either_or_get_negation(pr2)) ||
        (pr2 == prn_either_or_get_negation(pr1)))) pr2 = pr1;
    c = ((pointer_sized_int) pr1) - ((pointer_sized_int) pr2);
    if (c > 0) return 3; if (c < 0) return -3;
    c = ((pointer_sized_int) i1->obj_ref2) - ((pointer_sized_int) i2->obj_ref2);
    if (c > 0) return 4; if (c < 0) return -4;
    if (i1->inference_type == VALUEARBITRARY_INF) {
        c = ((pointer_sized_int) i1->val_ref2) - ((pointer_sized_int) i2->val_ref2);
        if (c > 0) return 4; if (c < 0) return -4;
    }
    c = ((pointer_sized_int) i1->obj_ref1) - ((pointer_sized_int) i2->obj_ref1);
    if (c > 0) return 5; if (c < 0) return -5;
    if (i1->inference_type == VALUEARBITRARY_INF) {
        c = ((pointer_sized_int) i1->val_ref1) - ((pointer_sized_int) i2->val_ref1);
        if (c > 0) return 5; if (c < 0) return -5;
    }
    c = ((pointer_sized_int) i1) - ((pointer_sized_int) i2);
    if (c > 0) return 6; if (c < 0) return -6;
    return 0;
}
```

§8. **Joining an inference to what is known about an object.** As the above will have made clear, it is not altogether easy to join a new inference to the linked list of inferences which belong to an object: we can't simply put it at the end of the list. The following routine looks simple enough, but was a difficult routine to get right.

Note that this is not the only place where contradictions are unearthed: higher-level problems are picked up when we try to construct an object tree which fits the known facts. At this stage, for instance, no objection will be made to inferences which make something both a supporter and a container.

```
void join_inference(inference *i, inference_subject infs) {
    PROTECTED_MODEL_PROCEDURE;
    inference *list;
    inference *prev = NULL;
    inference **list_head;
    int c, d, j, icl, lcl, contradiction_flag;
    if (i == NULL) internal_error("joining null inference");
    list_head = infs_get_list_head(infs);
    if (list_head == NULL) internal_error("joining inference to null subject");
    list = *list_head;
    if (list == NULL) { *list_head = i; i->next = NULL; goto ReportIt; }
    while (list != NULL) {
        c = compare_inferences(i, list);
        d = c; if (d < 0) d = -d;
        icl = i->certainty; if (icl < 0) icl = -icl;
        lcl = list->certainty; if (lcl < 0) lcl = -lcl;
        if (d >= 5) {
            Same inference type, property name, direction
            if (icl != lcl) {
                Then we take the more certain to be true
                if (lcl > icl) {
                    report_inference(i, infs, "discarded (we already know better)");
                    return;
                }
                Cut out previous inference
                i->next = list->next;
                if (prev == NULL) *list_head = i; else prev->next = i;
                report_inference(i, infs, "replaced existing equally certain one");
                return;
            }
            ⟨JI - Do these inferences contradict? 9⟩;
            if ((contradiction_flag) && (icl == 2)) {
            Contradictions are not permitted at "certainty" level
                report_inference(i, infs, "contradiction");
                report_inference(list, infs, "with");
                if (i->inference_type == PROPERTY_INF)
                two_sentences_problem(_P_(C9PropertyContradiction),
                    list->inferred_from,
                    "this looks like a contradiction",
                    "because the same property seems to be being set in each "
                    "of these sentences, but with a different outcome.");
                else
                two_sentences_problem(_P_(C9Contradiction),
```

```
                    list->inferred_from,
                    "this looks like a contradiction",
                    "which might be because I have misunderstood what was "
                    "meant to be the subject of one or both of those sentences.");
                return;
            }
            if ((contradiction_flag) && (icl == 1) &&
                (infs_default_certainty(infs) == LIKELY_CE)) {
                Cut out previous inference
                i->next = list->next;
                if (prev == NULL) *list_head = i; else prev->next = i;
                report_inference(i, infs, "replaced existing also only likely one");
                return;
            }
            And otherwise the new information is redundant
            report_inference(i, infs, "redundant");
            return;
        }
        if (c<0) {
            if (prev == NULL) *list_head = i; else prev->next = i;
            i->next = list;
            goto ReportIt;
        }
        prev = list; list = list->next;
    }
    prev->next = i;
    ReportIt:
    report_inference(i, infs, "drawn");
}
```

The function join_inference is called from 6/equal, 6/asp, 9/qty, 9/kind, 9/spabp, 9/mapbp, 9/exrel, 9/prop, 9/madj and 9/vpbp.

§**9.** The following code is used when the new inference, `i`, is dangerously similar (similarity `d` equal to 5 or 6) to an existing one, `list`. It must set `contradiction_flag` to `TRUE` if they in fact conflict. Most inferences essentially convey true/false information, so that they conflict if and only if they have certainties of opposite sign (thus, "usually" contradicts "seldom" and "impossible" contradicts "certain"). However, map direction and property inferences have values attached, and these conflict only if those values are different: moreover, the certainties having opposite sign now makes them uncontradictory after all. The following, after all, do not conflict:

The capacity of the box is 10. The capacity of the box is not 12.

⟨JI - Do these inferences contradict? 9⟩ ≡

```
{ int list_c = list->certainty, inf_c = i->certainty;
    contradiction_flag = FALSE;
    if ((i->inference_type == DIRECTION_INF) && (d == 5))
        contradiction_flag = TRUE;
    if (i->inference_type == GLOBAL_INF) contradiction_flag = TRUE;
    if (i->inference_type == PARENTAGE_INF) {
        if (list->obj_ref1 != i->obj_ref1)
            contradiction_flag = TRUE;
    }
```

```
    if (i->inference_type == PROPERTY_INF) {
        if (d == 5) contradiction_flag = TRUE;
        if ((d == 6) && (i->obj_ref1 == NULL)) {
            if ((i->val_ref1) && (list->val_ref1) &&
                (species_is(i->val_ref1, CONSTANT_SPC)) &&
                (species_is(list->val_ref1, CONSTANT_SPC))) {
                kind_of_value *kov_i, *kov_list;
                if ((i->pname) && (prn_condition_of_which_object(i->pname))) {
                    if (spec_get_constant_quantity_if_any(i->val_ref1) !=
                        spec_get_constant_quantity_if_any(list->val_ref1))
                        contradiction_flag = TRUE;
                } else {
                    kov_i = spec_get_kind_of_value(i->val_ref1);
                    kov_list = spec_get_kind_of_value(list->val_ref1);
                    if (kov_compare(kov_i, kov_list) == FALSE)
                        contradiction_flag = TRUE;
                    else {
                        if (is_kova(kov_i, OBJECT_TY)) {
                            if (OBJECT_spec_to_world_object(i->val_ref1) !=
                                OBJECT_spec_to_world_object(list->val_ref1))
                                contradiction_flag = TRUE;
                        }
                    }
                }
            }
            if ((i->word_ref1 >= 0) && (list->word_ref1 >= 0)) {
                if ((i->word_ref2-i->word_ref1) !=
                    (list->word_ref2-list->word_ref1))
                    contradiction_flag = TRUE;
                else {
                    for (j=0; j<=i->word_ref2-i->word_ref1; j++)
                        if (strcmp(lw_array[j+i->word_ref1].lw_text,
                            lw_array[j+list->word_ref1].lw_text) != 0)
                            contradiction_flag = TRUE;
                }
            }
        }
    }
    if ((i->pname) && (list->pname) && (i->pname != list->pname))
        inf_c = -inf_c;
    if (inf_c == -list_c)
        contradiction_flag = (!contradiction_flag);
}
```

This code is used in §8.

§**10.** We keep the debugging log file more than usually well informed about what goes on with inferences, as there is obviously great potential for mystifying bugs if inferences are incorrectly ignored, etc.

```
void report_inference(inference *i, inference_subject infs, char *what_happened) {
    LOGIF(INFERENCES, ":::: %s: $j - $I\n", what_happened, &infs, i);
}
```

Make Model World 9/model

Purpose

Once the assertions have all been read and reduced to inferences, and all the world objects are created, we then have to work out their likely kinds. Mostly this is easy: if we were told that X is on Y, for instance, we can immediately see that Y needs to be a supporter. But we go through a fairly thorough process in order to make sure that every piece of information is considered, even in easy cases, so that any contradiction is caught and reported back. "Creating a program often means that you have to create a small universe" (Donald Knuth).

Template interpreter commands

```
7   {-callv:make_model_world}
22   {-callv:Inform_name_the_objects}
23   {-callv:log_brief_picture}
```

---

§**1. The object tree.** This is the NI representation of the tree structure which objects will eventually have when the final story file is compiled. In other words, `child`, `sibling` and `parent` mean what they mean in Inform 6 terms: they define which objects begin the game contained within which others.

The following routine is the equivalent of the Inform 6 statement "move `wo1` to `wo2`", and moves `wo1` and its children to become the final `sibling` of the `child` of `wo2` (or the only `child` if there are none already). The tree is grown entirely from its root by repeated uses of this one operation. We perform a check that it remains well-founded at all times out of sheerest paranoia, and because time is not critical here.

```
void graft_object(world_object *wo1, world_object *wo2) {
    world_object *cw; int error_found, max_loop;
    LOGIF(OBJECT_TREE, "Grafting $O to be child of $O\n", wo1, wo2);
    if (wo1 == NULL) internal_error("wo1 is null in tree graft");
    if (wo2 == NULL) internal_error("wo2 is null in tree graft");
    if (wo2->object_tree_child == NULL) wo2->object_tree_child = wo1;
    else {
        wo2 = wo2->object_tree_child;
        while (wo2->object_tree_sibling != NULL) wo2 = wo2->object_tree_sibling;
        wo2->object_tree_sibling = wo1;
    }
    cw = wo1->object_tree_parent; max_loop = no_wos_now_existing + 1;
    error_found = FALSE;
    while ((cw != NULL) && (error_found == FALSE)) {
        if (cw == wo1) { error_found = TRUE; break; }
        cw = cw->object_tree_parent;
        max_loop--;
        if (max_loop == 0) break;
    }
```

```
    if (error_found) {
        foundation_problem(_P_(BelievedImpossible), wo1);
        wo1->object_tree_parent->object_tree_child = NULL;
        wo1->object_tree_parent = NULL;
        wo1->object_tree_sibling = NULL;
        wo1->object_tree_child = NULL;
    }
}
```

§**2.** The tree is built in two stages. First, the `parent` of every object is determined (see below); then the following routine is called to build a tree structure conforming to that information, which will of course leave the `parent` fields unchanged, but will arrange different objects with the same parent as siblings.

```
world_object *wo_progenitor(world_object *wo) {
    if (wo == NULL) return NULL;
    if (wo->object_tree_parent) return wo->object_tree_parent;
    return wo->part_of;
}
void build_object_tree(void) {
    world_object *wo, *wo2;
    LOOP_OVER(wo, world_object) {
        int k;
        for (wo2 = wo_progenitor(wo), k=0;
            (wo2) && (k<10000);
            wo2 = wo_progenitor(wo2), k++) {
            if (wo2 == wo) {
                foundation_problem(_P_(C9IllFounded),
                    wo);
                break;
            }
        }
    }
    LOOP_OVER(wo, world_object)
        if (wo->object_tree_parent) graft_object(wo, wo->object_tree_parent);
    if (dl_this(OBJECT_TREE_DA)) log_object_tree();
}
```

§**3. A standard error.** This is a notorious Inform 6 limitation which we shall need to own up to. (It is produced in two different ways below, so is worth its own routine.)

```
void cant_cont_and_supp(parse_node *s1, parse_node *s2, world_object *wo) {
    contradiction_problem(_P_(C9CantContainAndSupport),
        s1, s2, wo,
        "cannot both contain things and support things",
        "which is what you're implying here. If you need both, "
        "the easiest way is to make it either a supporter with "
        "a container attached or vice versa. For instance: "
        "'A desk is here. On the desk is a newspaper. "
        "An openable container called the drawer is part "
        "of the desk. In the drawer is a stapler.'");
}
```

§4. **Contradictions between instance and kind.**

```
void check_compatible_with_kind(world_object *k, world_object *wo,
    inference *wo_inf, property_name *pname, int sign) {
    inference *inf;
    if ((k == NULL) || (k == kind_kind)) return;
    KNOWLEDGE_LOOP(inf, k, PROPERTY_INF) {
        if ((inf_get_property_name(inf) == pname)
            && (sign*inf_get_certainty(inf) == IMPOSSIBLE_CE)) {
            Clash:
            LOG("Checking wo $O compatible with kind $O for property $Y\n",
                wo, k, pname);
            contradiction_problem(_P_(C9InstanceContradiction),
                inf_inferred_from(wo_inf), inf_inferred_from(inf), wo,
                "therefore has to have both states of an either/or property "
                "at once",
                "which is impossible. When a kind's definition says that "
                "something is 'always' true, there is no way to override "
                "that for particular things of the kind.");
            return;
        }
    }
    if (prn_either_or_get_negation(pname)) {
        KNOWLEDGE_LOOP(inf, k, PROPERTY_INF) {
            if ((inf_get_property_name(inf) == prn_either_or_get_negation(pname))
                && (sign*inf_get_certainty(inf) == CERTAIN_CE)) {
                goto Clash;
            }
        }
    }
    check_compatible_with_kind(k->kind, wo, wo_inf, pname, sign);
}
```

§5. **Determining scenery.** This is not quite as easy to do as it looks.

```
int wo_is_scenery(world_object *wo) {
    inference *inf;
    if (wo == NULL) return FALSE;
    property_name *pname = find_property_translating("scenery");
    POSITIVE_KNOWLEDGE_LOOP(inf, wo, PROPERTY_INF) {
        if (inf_get_property_name(inf) == pname) return TRUE;
    }
    if ((wo->kind) && (wo->kind != kind_kind)) return wo_is_scenery(wo->kind);
    return FALSE;
}
```

§6. What goes where.

```
int wo_in_wo(world_object *wo1, world_object *wo2) {
    if ((wo1 == NULL) || (wo2 == NULL)) return FALSE;
    while ((wo1 = wo1->object_tree_parent) != NULL) {
        if (wo1 == wo2) return TRUE;
    }
    return FALSE;
}
```

The function wo_in_wo is called from 7/spec.

§7. Turning inferences into a spatial model. At this point all objects are, of course, created, and they have kinds associated with them if the source text has said explicitly what kind they have: but that is not good enough. It often happens that the source implicitly specifies a kind, and we need to take note. We also need to work out what is contained inside what, and the two tasks are related: if X is on Y, then Y must be a supporter, for instance. The five kinds we need to sort out here are room, container, supporter, door and thing. Doors, moreover, come in two varieties: those accessible from both sides, and those accessible one way only.

Once implications have been considered, all inferences are finally drawn. Inferences will still be needed even after the routine has completed, because they are used when compiling the I6 objects which correspond to the world objects in the tree.

```
/****/ void make_model_world (void) {
    int i, j;
    world_object *wo;
    property_name *prn;
    inference *inf;
    if (wo_yourself == NULL)
        internal_error("The Standard Rules have not defined 'yourself'");
    start_room = NULL;
    start_object = NULL;
    LOOP_OVER(wo, world_object) {
        wo->here_flag = FALSE;
        POSITIVE_KNOWLEDGE_LOOP(inf, wo, PARENTAGE_HERE_INF) {
            wo->here_flag = TRUE;
        }
    }
    LOOP_OVER(wo, world_object)
        if (wo->here_flag == FALSE)
            wo_position(wo);
    LOOP_OVER(wo, world_object)
        if (wo->here_flag)
            wo_position(wo);
    ⟨MMW - Issue error if there is no start room 15⟩;
    ⟨MMW - Issue error if any name contains a comma 16⟩;
    ⟨MMW - Issue error if any kind has contents 17⟩;
    ⟨MMW - Police the map connections 18⟩;
    ⟨MMW - Police the properties 19⟩;
    ⟨MMW - Police 1 to 1 relations 20⟩;
    ⟨MMW - Work out exits for index mapping later 21⟩;
    build_object_tree();
```

```
    model_world_constructed = TRUE;
}
void wo_position(world_object *wo) {
    world_object *k;
    inference *inf;
    ⟨MMW - Room, supporter, container or thing? 8⟩;
    ⟨MMW - Set room and door flags 9⟩;
    ⟨MMW - Is this the start room? 10⟩;
    ⟨MMW - Turn descriptive text into an inference 11⟩;
    consider_implications(wo);
    ⟨MMW - Set description and initially carried flags 13⟩;
    ⟨MMW - Work out parent object 14⟩;
}
```

§**8.** In the following code, we weigh the likelihood of `wo` being a room, supporter or container, and give up and make it a thing if there is no evidence to the contrary. We first find what the geography of the world suggests (the "geography choice"), then what the source text explicitly states (the "designer choice"), and try to reconcile the two, producing problem messages where they clash.

⟨MMW - Room, supporter, container or thing? 8⟩ ≡

```
    if (wo->kind_flag == FALSE) {
        inference *room_inference = NULL, *supporter_inference = NULL,
            *container_inference = NULL, *geography_inference = NULL;
        world_object *designers_choice, *geography_choice;
        int room_certainty, container_certainty, supporter_certainty,
            geography_certainty;
        room_certainty = UNKNOWN_CE;
        container_certainty = UNKNOWN_CE;
        supporter_certainty = UNKNOWN_CE;
        KNOWLEDGE_LOOP(inf, wo, ISAROOM_INF) {
            room_certainty = inf_get_certainty(inf);
            room_inference = inf;
        }
        KNOWLEDGE_LOOP(inf, wo, CONTAINSTHINGS_INF) {
            container_certainty = inf_get_certainty(inf);
            container_inference = inf;
        }
        geography_choice = NULL;
        geography_certainty = UNKNOWN_CE;
        geography_inference = NULL;
        if (room_certainty > geography_certainty) {
            geography_choice = kind_room;
            geography_certainty = room_certainty;
            geography_inference = room_inference;
        }
        if (container_certainty > geography_certainty) {
            geography_choice = kind_container;
            geography_certainty = container_certainty;
            geography_inference = container_inference;
        }
        if (supporter_certainty > geography_certainty) {
```

```
    geography_choice = kind_supporter;
    geography_certainty = supporter_certainty;
    geography_inference = supporter_inference;
}
if (room_certainty > UNKNOWN_CE) {
    geography_choice = kind_room;
    geography_certainty = room_certainty;
}
if (wo->word_ref1 >= 0) current_sentence = wo->creating_sentence;
if ((room_certainty == CERTAIN_CE) &&
    (supporter_certainty == CERTAIN_CE))
    contradiction_problem(_P_(BelievedImpossible),      other problems catch this earlier
        inf_inferred_from(room_inference),
        inf_inferred_from(supporter_inference), wo,
        "and a room cannot also be a supporter",
        "which is what you are implying. Because a map "
        "connection leads from it, I think it must be a room, "
        "and because something is on it, I think it must be a "
        "supporter. (Perhaps it's just that you called the "
        "location in question something like 'Plateau' and "
        "then wrote 'the bindweed is on the Plateau' instead "
        "of 'in the Plateau'?).");
if ((container_certainty == CERTAIN_CE) &&
    (supporter_certainty == CERTAIN_CE)) {
    cant_cont_and_supp(inf_inferred_from(container_inference),
        inf_inferred_from(supporter_inference), wo);
}
designers_choice = kind_thing;
k = wo->kind; if (k == NULL) designers_choice = NULL;
while ((k != NULL) && (k != kind_kind)) {
    if (k == kind_room) designers_choice = k;
    if (k == kind_container) designers_choice = k;
    if (k == kind_supporter) designers_choice = k;
    if (k == kind_door) designers_choice = k;
    if (k == kind_person) designers_choice = k;
    k = k->kind;
}
if ((designers_choice == kind_person) && (geography_choice != NULL)) {
    contradiction_problem(_P_(C9PersonContaining),
        wo->kind_set_at,
        inf_inferred_from(geography_inference), wo,
        "cannot contain or support things like something inanimate",
        "which is what you are implying. Instead, people must "
        "carry or wear them: so 'The briefcase is in Daphne.' "
        "is disallowed, but 'The briefcase is carried by Daphne.' "
        "is fine, or indeed 'Daphne carries the briefcase.'");
}
if ((designers_choice == NULL) && (geography_choice != NULL)) {
    LOGIF(KIND_CHANGES, "Deducing kind of $O is $O\n",
        wo, geography_choice);
    assert_kind_of_object(wo, geography_choice);
    goto NoProblem;
}
```

```
    if ((designers_choice == kind_thing) &&
        (geography_certainty == CERTAIN_CE) &&
        ((geography_choice == kind_container) ||
            (geography_choice == kind_supporter))) {
        assert_kind_of_object(wo, geography_choice);
        goto NoProblem;
    }
    if ((geography_certainty == CERTAIN_CE) &&
        (geography_choice != designers_choice)) {
        parse_node *decider = wo->creating_sentence;
        if (wo->kind_set_at != NULL) decider = wo->kind_set_at;
        if ((designers_choice == kind_door) &&
            (geography_choice == kind_room)) goto NoProblem;
        if ((designers_choice != kind_supporter) &&
            (geography_choice == kind_container)) goto NoProblem;
        if (((designers_choice == kind_supporter) &&
            (geography_choice == kind_container)) ||
            ((designers_choice == kind_container) &&
            (geography_choice == kind_supporter))) {
            cant_cont_and_supp(decider,
                inf_inferred_from(geography_inference), wo);
        } else {
            contradiction_problem(_P_(C9BothRoomAndSupporter),
                decider,
                inf_inferred_from(geography_inference), wo,
                "has a kind incompatible with its situation",
                "and this is a contradiction.");
        }
        LOG("Designer choice: $O; Geography choice: $O.\n",
            designers_choice, geography_choice);
    }
    NoProblem:
    if (wo->kind == NULL) assert_kind_of_object(wo, kind_thing);
}
```

This code is used in §7.

§**9.** We need to know in a quick and easy fashion whether something is a door or not, and similarly for rooms. We cannot simply see if it has kind "door", because its kind might be (say) "revolving door", a putative kind of door. This means we need to recurse back through its kind hierarchy; which is a good opportunity to check that that hierarchy is well-founded, something we ought to do for safety's sake.

⟨MMW - Set room and door flags 9⟩ ≡

```
    wo->room_flag = FALSE;
    if (wo->kind_flag == FALSE) {
        int safety_limit = 30;
        k = wo->kind;
        while ((k != NULL) && (k != kind_kind)) {
            if (k == kind_room) wo->room_flag = TRUE;
            if (k == kind_door) wo->door_flag = TRUE;
            k = k->kind;
            safety_limit--;
            if (safety_limit == 0) {
```

```
            LOG("Object $O has kind hierarchy:\n", wo);
            k = wo->kind;
            for (safety_limit = 0; safety_limit<30; safety_limit++) {
                LOG(" -> kind $O\n", k);
                k = k->kind;
            }
            internal_error("The kind hierarchy is not well-founded.");
        }
    }
}
```

This code is used in §7.

§**10.** If no start room has explicitly been appointed by the source text, the first-created room is elected unopposed.

⟨MMW - Is this the start room? 10⟩ ≡

```
if ((wo->room_flag) && (start_room == NULL)) start_room = wo;
```

This code is used in §7.

§**11.** A lone string as a sentence is a description for a room, but an initial description for an object. Only now can we know which, since we have only just decided whether `wo` is a room or not. We therefore draw the necessary inference.

⟨MMW - Turn descriptive text into an inference 11⟩ ≡

```
if (wo->quoted_appearance != NULL) {
    property_name *pn;
    char *inf_name = NULL;
    int thing_flag = FALSE;
    current_sentence = wo->quoted_appearance_sentence;
    k = wo->kind;
    while ((k != NULL) && (k != kind_kind)) {
        if (k == kind_thing) thing_flag = TRUE;
        k = k->kind;
    }
    if (thing_flag) {
        if (wo_is_scenery(wo)) {
            inference *inf; int dont_infer = FALSE;
            KNOWLEDGE_LOOP(inf, wo, PROPERTY_INF) {
                property_name *prn = inf_get_property_name(inf);
                if (((prn) && (inf_get_certainty(inf) > 0)) &&
                    (strcmp(prn_get_i6_identifier(prn), "description")==0)) {
                    ⟨Produce a problem for doubly described scenery 12⟩;
                    dont_infer = TRUE;
                }
            }
            if (dont_infer == FALSE) inf_name = "description";
        } else inf_name = "initial";
    } else inf_name = "description";
    if (inf_name) {
        pn = find_property_translating(inf_name);
        if (pn == NULL) internal_error("It appears that "
            "no I7 properties have been defined translating "
            "to I6 'description' and 'initial'");
```

```
            assert_property_value_explicitly(pn, wo, NULL, wo->quoted_appearance);
        }
    }
```

This code is used in §7.

§**12.**

⟨Produce a problem for doubly described scenery 12⟩ ≡

```
    object_problem(_P_(C9SceneryDoublyDescribed),
        wo,
        "is scenery, which means that it cannot sensibly have any 'initial "
        "appearance' property - being scenery, it isn't announced when the "
        "player first sees it. That means the quoted text about it has to "
        "be read as its 'description' instead, seen when the player examines "
        "it. But the source text writes out its description, too",
        "which means we have a subtle little contradiction here.");
```

This code is used in §11.

§**13.** Now that all inferences are drawn, we can find out whether `wo` provides a description, and whether it is to be initially carried.

⟨MMW - Set description and initially carried flags 13⟩ ≡

```
    wo->defines_description = FALSE;
    wo->initially_carried = FALSE;
    KNOWLEDGE_LOOP(inf, wo, PROPERTY_INF) {
        property_name *prn = inf_get_property_name(inf);
        if (prn == NULL)
            internal_error("Inference with null property");
        if (inf_get_certainty(inf) > 0) {
            if (strcmp(prn_get_i6_identifier(prn), "description")==0)
                wo->defines_description = TRUE;
            if (strcmp(prn_get_i6_identifier(prn), "initially_carried")==0)
                wo->initially_carried = TRUE;
            if (strcmp(prn_get_i6_identifier(prn), "privately_named")==0)
                wo->nameless = TRUE;
            if ((prn_is_either_or(prn)) &&
                (prn_either_or_get_negation(prn)) &&
                (strcmp(prn_get_i6_identifier(prn_either_or_get_negation(prn)),
                    "privately_named")==0))
                wo->nameless = FALSE;
        }
        if (inf_get_certainty(inf) < 0) {
            if (strcmp(prn_get_i6_identifier(prn), "privately_named")==0)
                wo->nameless = FALSE;
            if ((prn_is_either_or(prn)) &&
                (prn_either_or_get_negation(prn)) &&
                (strcmp(prn_get_i6_identifier(prn_either_or_get_negation(prn)),
                    "privately_named")==0))
                wo->nameless = TRUE;
        }
    }
```

This code is used in §7.

§**14.** This is where we set the `parent` field. We need to split off into a special case for "part of" containment, which is handled by a different tree structure altogether, since it is represented rather differently by the I6 library. Another point of interest is that we can finally interpret the word "here": its meaning ("the most recently created room") is clear enough, but was not possible to decide until we knew what was, and what was not, a room.

⟨MMW - Work out parent object 14⟩ ≡

```
if (wo->kind_flag == FALSE) {
    int component_part = FALSE;
    parse_node *parent_setting_sentence = NULL;
    wo->object_tree_parent = NULL;
    POSITIVE_KNOWLEDGE_LOOP(inf, wo, ISWORN_INF) {
        wo->world_index_description = "worn";
    }
    POSITIVE_KNOWLEDGE_LOOP(inf, wo, PARTOF_INF) {
        component_part = TRUE;
    }
    POSITIVE_KNOWLEDGE_LOOP(inf, wo, PARENTAGE_HERE_INF) {
        parent_setting_sentence = inf_inferred_from(inf);
    }
    POSITIVE_KNOWLEDGE_LOOP(inf, wo, PARENTAGE_INF) {
        world_object *supposed_parent = inf_get_object_reference(inf);
        current_sentence = inf_inferred_from(inf);
        if ((wo->object_tree_parent) || (wo->here_flag)) {
            contradiction_problem(_P_(C9DuplicateParentage),
                parent_setting_sentence, current_sentence,
                wo,
                "can only be given its position once",
                "in a single assertion sentence.");
        }
        parent_setting_sentence = current_sentence;
        wo->object_tree_parent = supposed_parent;
    }
    POSITIVE_KNOWLEDGE_LOOP(inf, wo, PARENTAGE_HERE_INF) {
        world_object *wo2 = inf_get_object_reference(inf);
        if ((wo2 == NULL) || (wo2->room_flag == FALSE)) {
            parse_node *sent, *here_sentence = inf_inferred_from(inf);
            wo2 = NULL;
            LOG("IOS for $O needed at $T\n", wo, here_sentence);
            for (TREE_START(sent);
                (sent) && (sent != here_sentence);
                TREE_NEXT(sent)) {
                world_object *sub = pn_get_interpretation_of_subject(sent);
                if ((sub) && (sub->room_flag)) {
                    wo2 = sub;
                    LOG("IOS is $O at $T\n", sub, sent);
                }
            }
            LOG("IOS for $O resolved to $O\n", wo, wo2);
            current_sentence = here_sentence;
            if (wo2 == NULL) {
                object_problem(_P_(C9NoHere),
```

```
                wo,
                "was described as being 'here', but here "
                "doesn't seem to be any location at this point "
                "in the story",
                "so there's nowhere you can call 'here'.");
            goto GiveUpOnParentage;
        }
    }
    wo->object_tree_parent = wo2;
    LOGIF(OBJECT_TREE, "Deducing 'here' as parent of $O means $O\n",
        wo, wo->object_tree_parent);
}
if (component_part) {
    if ((wo->room_flag) || (wo->door_flag)) {
        object_problem(_P_(C9RoomOrDoorAsPart),
            wo,
            "was set up as being part of something else, "
            "which doors and rooms are not allowed to be",
            "because they are part of the fixed map of the world - "
            "if they were parts of something else, they might move "
            "around. (Of course, it's easy to make a door look "
            "as if it's part of something to the player - describing "
            "it as part of a wall, or bulkhead, or cottage, say - "
            "and if there really is an entrance that needs to move "
            "around - say, the hatchway on a tank - it's probably "
            "best to make it an enterable container.)");
        goto GiveUpOnParentage;
    }
    world_object *wo2 = wo->object_tree_parent;
    if (wo2 != NULL) {
        if (wo2->first_part == NULL) wo2->first_part = wo;
        else {
            wo2 = wo2->first_part;
            while ((wo2 != NULL) && (wo2->next_part != NULL))
                wo2 = wo2->next_part;
            wo2->next_part = wo;
        }
        wo->part_of = wo->object_tree_parent;
        wo->world_index_description = "part";
        wo->object_tree_parent = NULL;
    }
}
if (wo->initially_carried) {
    if (wo->object_tree_parent == NULL) {
        wo->object_tree_parent = wo_yourself;
        wo->world_index_description = "carried";
    } else object_problem(_P_(C9InitiallyCarriedToo),
        wo,
        "was described as being 'initially carried', so should "
        "not also be said to be in, on or part of something else",
        "which is what seems to have happened here.");
}
if ((wo->world_index_description == NULL) && (wo->object_tree_parent)) {
```

```
            if (wo_of_kind(wo->object_tree_parent, kind_supporter))
                wo->world_index_description = "on";
            if (wo_of_kind(wo->object_tree_parent, kind_person))
                wo->world_index_description = "carried";
        }
        GiveUpOnParentage: ;
    }
```

This code is used in §7.

§**15.** If there is still no start room, then there were no rooms at all!

⟨MMW - Issue error if there is no start room 15⟩ ≡

```
    if (existing_story_file) {
        int rc = 0;
        LOOP_OVER(wo, world_object) if (wo->room_flag) rc++;
        if (rc > 0) {
            unlocated_problem(_P_(C9RoomInIgnoredSource),
                "This is supposed to be a source text which only contains "
                "release instructions to bind up an existing story file "
                "(for instance, one produced using Inform 6). That's because "
                "the instruction 'Release along with an existing story file' "
                "is present. So the source text must not contain rooms or "
                "other game design - these would be ignored.");
            return;
        }
        if (for_release == FALSE) {
            unlocated_problem(_P_(C9UnreleasedRelease),
                "This is supposed to be a source text which only contains "
                "release instructions to bind up an existing story file "
                "(for instance, one produced using Inform 6). That's because "
                "the instruction 'Release along with an existing story file' "
                "is present. So the only way to build the project is to use "
                "the Release option - not, for instance, Go or Replay, because "
                "it would make no sense to translate the source text into "
                "something to play. (Of course, you can play the released "
                "story file using an interpreter such as Zoom or Windows "
                "Frotz, etc.: just not here, within Inform.)");
            return;
        }
    } else {
        if (start_room == NULL) {
            unlocated_problem(_P_(C9NoStartRoom),
                "There doesn't seem to be any location in this story, so there's "
                "nowhere for the player to begin. This may be because I have "
                "misunderstood what was meant to be a room and what wasn't: I "
                "only know something is a room if you tell me explicitly ('The "
                "Observatory is a room') or if you imply it by giving map "
                "directions ('East of the Observatory is the Planetarium').");
            return;
        }
    }
```

This code is used in §7.

§**16.** This was added in beta-testing when it turned out that mistakes in coding tended to produce strange results rather than problem messages.

⟨MMW - Issue error if any name contains a comma 16⟩ ≡

```
LOOP_OVER(wo, world_object)
    if (wo->word_ref1 >= 0) {
        int w1 = wo->word_ref1, w2 = wo->word_ref2;
        LOGIF(OBJECT_TREE, "$O has creator text <$W>\n", wo, w1, w2);
        for (j=w1; ((j>=0) && (j<=w2)); j++) {
            LOGIF(OBJECT_TREE, "Word %d <$W> ex <%s>\n", j, j, j,
                vocab_get_exemplar(lw_array[j].lw_identity, FALSE));
            if [[word j == COMMA]] {
                object_creation_problem(_P_(C9CommaInName),
                    wo,
                    "has a comma in its name",
                    "which is forbidden. Perhaps you used a comma in "
                    "punctuating a sentence? Inform generally doesn't "
                    "like this because it reserves commas for specific "
                    "purposes such as dividing rules or 'if' phrases.");
                break;
            }
            if (vocab_test_flags(j, TEXT_MC)) {
                object_creation_problem(_P_(BelievedImpossible),
                    wo,
                    "has some double-quoted text in its name",
                    "which is forbidden. Perhaps something odd happened "
                    "to do with punctuation between sentences? Or perhaps "
                    "you really do need the item to be described with "
                    "literal quotation marks on screen when the player "
                    "sees it. If so, try giving it a printed name: perhaps "
                    "'The printed name of Moby Dick is \"'Moby Dick'\".'");
                break;
            }
        }
    }
```

This code is used in §7.

§**17.** It should be difficult, and may even be impossible, to get the assertion maker to allow you to incur the following error message. But it seems worth checking just to be on the safe side.

⟨MMW - Issue error if any kind has contents 17⟩ ≡

```
LOOP_OVER(wo, world_object)
    if ((wo->kind_flag == FALSE) && (wo->object_tree_parent != NULL) &&
        (wo->object_tree_parent->kind_flag == TRUE))
            object_problem(_P_(BelievedImpossible),
                wo, "seems to be inside a kind",
                "but needs to be in a room or another thing.");
```

This code is used in §7.

§**18.** The following code does little if the source is correct: it mostly checks that various mapping impossibilities do not occur. In the process, we record up to three different *destinations* of directions from each given object `wo`.

The one constructive thing we do is to set the `parent` of a one-sided door to the room it leads from. We do this for one-sided, but not two-sided, doors owing to the asymmetry between them in the I6 library: see the DM4 for details.

⟨MMW - Police the map connections 18⟩ ≡

```
LOOP_OVER(wo, world_object) {
    wo->map_connection_a = NULL;
    wo->map_connection_b = NULL;
    wo->map_connection_c = NULL;
    POSITIVE_KNOWLEDGE_LOOP(inf, wo, DIRECTION_INF) {
        world_object *ref1, *ref2;
        inf_get_object_references(inf, &ref1, &ref2);
        if (ref1) {
            if (wo->map_connection_a == NULL) {
                wo->map_connection_a = ref1;
                wo->map_direction_a = ref2;
                goto RecordedConnection;
            }
            if (wo->map_connection_a == ref1)
                goto RecordedConnection;
            if (wo->map_connection_b == NULL) {
                wo->map_connection_b = ref1;
                wo->map_direction_b = ref2;
                goto RecordedConnection;
            }
            if (wo->map_connection_b == ref1)
                goto RecordedConnection;
            if (wo->map_connection_c == NULL) {
                wo->map_connection_c = ref1;
                wo->map_direction_c = ref2;
            }
            RecordedConnection:
            if ((wo->room_flag) && (ref1->door_flag == FALSE) &&
                (ref1->room_flag == FALSE))
                contradiction_problem(_P_(C9BadMapCell),
                    ref1->creating_sentence,
                    inf_inferred_from(inf), ref1,
                    "appears to be something which can be reached via a map "
                    "connection, but it seems to be neither a room nor a door",
                    "and these are the only possibilities allowed in the "
                    "Inform world model.");
        }
    }
    if (wo->door_flag) {
        if (wo->map_connection_a == NULL) {
            object_problem(_P_(C9DoorUnconnected),
                wo,
                "seems to be a door with no way in or out",
                "so either you didn't mean it to be a door "
                "or you haven't specified what's on each "
```

```
                "side. You could do this by writing something "
                "like 'The blue door is east of the Library "
                "and west of the Conservatory'.");
            goto EnoughComplaining;
        }
        if (wo->map_connection_c != NULL) {
            object_problem(_P_(C9DoorOverconnected),
                wo,
                "seems to be a door with three ways out",
                "but you can only have one or two sides to a "
                "door in Inform: a one-sided door means a door "
                "which is only touchable and usable from one "
                "side, and an example might be a window through "
                "which one falls to the ground below. If you "
                "really need a three-sided cavity, best to make "
                "it a room in its own right.");
            goto EnoughComplaining;
        }
        if (((wo->map_connection_a != NULL) &&
            (wo->map_connection_a->room_flag == FALSE))
            ||
            ((wo->map_connection_b != NULL) &&
            (wo->map_connection_b->room_flag == FALSE))) {
            object_problem(_P_(C9DoorToNonRoom),
                wo,
                "seems to be a door opening on something not a room",
                "but a door must connect one or two rooms "
                "(and in particular is not allowed to connect "
                "to another door).");
            goto EnoughComplaining;
        }
        if (wo->object_tree_parent == NULL) {
            if (wo->map_connection_b == NULL)
                wo->object_tree_parent = wo->map_connection_a;
        } else {
            if ((wo->object_tree_parent != wo->map_connection_a)
                && (wo->object_tree_parent != wo->map_connection_b)) {
                object_problem(_P_(C9DoorInThirdRoom),
                    wo,
                    "seems to be a door which is present in a "
                    "room to which it is not connected",
                    "but this is not allowed. A door must be "
                    "in one or both of the rooms it is between, "
                    "but not in a third place altogether.");
                goto EnoughComplaining;
            }
        }
    }
    EnoughComplaining: ;
}
LOOP_OVER(wo, world_object) if (wo->room_flag) {
    POSITIVE_KNOWLEDGE_LOOP(inf, wo, DIRECTION_INF) {
        world_object *goes_to, *via_direction;
```

```
            inf_get_object_references(inf, &goes_to, &via_direction);
            if ((goes_to) && (goes_to->door_flag) &&
                (wo != goes_to->map_connection_a) &&
                (wo != goes_to->map_connection_b)) {
                quote_object(1, wo);
                quote_object(2, goes_to);
                quote_object(3, goes_to->map_connection_a);
                quote_object(4, goes_to->map_connection_b);
                handmade_problem(_P_(C9RoomMissingDoor));
                issue_problem_segment(
                    "%1, a room, seems to have a map connection which goes "
                    "through %2, a door: but that doesn't seem physically "
                    "possible, since the rooms on each side of %2 have "
                    "been established as %3 and %4.");
                issue_problem_end();
            }
        }
    }
    LOOP_OVER(wo, world_object) if (wo->room_flag) {
        POSITIVE_KNOWLEDGE_LOOP(inf, wo, PARTOF_INF) {
            object_problem(_P_(BelievedImpossible),                    picked up elsewhere now
                wo,
                "seems to be a room which has parts",
                "but this is not allowed. A room can contain things, "
                "but not support or incorporate them.");
        }
    }
```

This code is used in §7.

§**19.** Here we check whether inferences have been made about inapplicable properties, or have been made in the wrong way.

⟨MMW - Police the properties 19⟩ ≡

```
    LOOP_OVER(wo, world_object) {
        KNOWLEDGE_LOOP(inf, wo, PROPERTY_INF) {
            property_name *prn = inf_get_property_name(inf);
            if (prn_is_either_or(prn)) {
                int abc = inf_get_certainty(inf), sign = 1;
                if (abc < 0) { abc = -abc; sign = -1; }
                if (abc == 2)
                check_compatible_with_kind(wo->kind, wo, inf, prn, sign);
            }
        }
    }
    LOOP_OVER(wo, world_object) {
        KNOWLEDGE_LOOP(inf, wo, PROPERTY_INF) {
            property_name *prn = inf_get_property_name(inf);
            world_object *ref1, *ref2;
            inf_get_object_references(inf, &ref1, &ref2);
            int treated_as_either_or = TRUE;
            if ((prn != specification_PRN) &&
                (prn != plural_named_PRN) &&
```

```
            (prn != proper_named_PRN)) {
            int bad_case = FALSE;
            if (wo->kind_flag) {
                if (does_kind_provide(wo, prn) == FALSE) bad_case = TRUE;
            } else {
                if ((does_object_provide(wo, prn) == FALSE) &&
                    (does_kind_provide(wo->kind, prn) == FALSE)) bad_case = TRUE;
            }
            if (bad_case) {
                inference_problem(_P_(C9PropertyNotPermitted),
                    wo, inf, "is not allowed to exist",
                    "because you haven't said it is. What properties "
                    "something can have depends on what kind of thing "
                    "it is: see the Index for details.");
            }
        }
        if (prn_is_value_property(prn)) {
            kind_of_value *kov_needed = prn_get_kind_of_value(prn);
            if (kovko_get_kind(kov_needed)) {
                if ((ref1) &&
                    (wo_of_kind(ref1, kovko_get_kind(kov_needed)) == FALSE)) {
                    LOG("Investigation found kov $u, prop $Y, obj $O\n",
                        kov_needed, prn, ref1);
                    quote_object(1, wo);
                    quote_property(2, prn);
                    quote_object(3, kovko_get_kind(kov_needed));
                    quote_object(4, ref1);
                    quote_object(5, ref1->kind);
                    handmade_problem(_P_(BelievedImpossible));          caught elsewhere
                    issue_problem_segment(
                        "The property '%2' of %1 seems to be set here to %4, "
                        "but this is the wrong kind of object to go into the "
                        "property: %5 not %3.");
                    issue_problem_end();
                }
            }
        }
        if ((ref2) || (inf_get_literal_value(inf)))
            treated_as_either_or = FALSE;
        if ((prn_is_either_or(prn)) && (treated_as_either_or == FALSE))
            inference_problem(_P_(BelievedImpossible),
                wo, inf, "can't have a value attached",
                "because it's a yes or no question. In the same "
                "way that something can be 'opaque' or not, but it "
                "can't be 'printed name' or not; whereas it can be "
                "'with printed name \"marble\"' but it can't be "
                "'with opaque'.");
    }
}
LOOP_OVER(wo, world_object) {
    KNOWLEDGE_LOOP(inf, wo, PROPERTY_INF) {
        property_name *prn = inf_get_property_name(inf);
        if (prn_is_either_or(prn)) {
```

```
                int abc = inf_get_certainty(inf), sign = 1;
                if (abc < 0) { abc = -abc; sign = -1; }
                if (abc == 2)
                check_compatible_with_kind(wo->kind, wo, inf,
                    inf_get_property_name(inf), sign);
            }
        }
    }
```

This code is used in §7.

§**20.** We now check 1-to-1 relations to see if the initial conditions have violated the 1-to-1-ness. Because of the way these relations are implemented using a property, it seems in fact to be impossible to violate the left-hand count – a contradiction problem is reported at an earlier stage. But in case the implementation is ever changed, it seems prudent to leave this checking in.

⟨MMW - Police 1 to 1 relations 20⟩ ≡

```
LOOP_OVER(prn, property_name) if (prn_stores_1to1_relation(prn)) {
    LOOP_OVER(wo, world_object) {
        wo->leftc = 0; wo->rightc = 0;
    }
    LOOP_OVER(wo, world_object) {
        inference *inf1 = NULL;
        KNOWLEDGE_LOOP(inf, wo, PROPERTY_INF) {
            if ((inf_get_property_name(inf) == prn) &&
                (inf_get_certainty(inf) == CERTAIN_CE)) {
                specification *val = inf_get_literal_value(inf);
                world_object *wo2 = OBJECT_spec_to_world_object(val);
                wo->leftc++;
                if (wo2) {
                    if (wo2->rightc < 2) wo2->right_infs[wo2->rightc] = inf;
                    wo2->rightc++;
                }
                if (wo->leftc == 1) inf1 = inf;
                if (wo->leftc == 2) {
                    contradiction_problem(_P_(BelievedImpossible),
                        inf_inferred_from(inf1), inf_inferred_from(inf),
                        wo, "can only relate to one other thing in this way",
                        "since the relation in question is one-to-one.");
                }
            }
        }
    }
    LOOP_OVER(wo, world_object) {
        if (wo->rightc >= 2) {
            contradiction_problem(_P_(C9Relation1to1Right),
                inf_inferred_from(wo->right_infs[0]),
                inf_inferred_from(wo->right_infs[1]),
                wo, "can only relate to one other thing in this way",
                "since the relation in question is one-to-one.");
        }
    }
}
```

This code is used in §7.

§**21.** Each room has an array showing the destinations in the ordinary cardinal directions (the "registered" ones), and we fill that in now. It is used only for the index map.

⟨MMW - Work out exits for index mapping later 21⟩ ≡

```
LOOP_OVER(wo, world_object) {
    POSITIVE_KNOWLEDGE_LOOP(inf, wo, DIRECTION_INF) {
        world_object *ref1, *ref2;
        inf_get_object_references(inf, &ref1, &ref2);
        if (ref1)
            for (i=0; i<registered_directions; i++)
                if ((ref2->direction_index >= 0) &&
                    (ref2->direction_index < MAX_DIRECTIONS))
                    wo->exits[ref2->direction_index] = ref1;
    }
}
```

This code is used in §7.

§22. Naming for I6 source code purposes.

```
void Inform_name_the_objects(void) {
    world_object *wo;
    LOOP_OVER(wo, world_object) {
        inference *inf; char id[32];
        if (wo->word_ref1 >= 0) {
            isn_compose_identifier(id,
                (wo->kind_flag)?'K':'O', wo->allocation_id,
                wo->word_ref1, wo->word_ref2);
            wo_set_I6_representation(wo, id);
        } else {
            if (wo == kind_kind)
                wo_set_I6_representation(wo, "K0_kind");
            else {
                sprintf(id, "X%d", wo->allocation_id);
                wo_set_I6_representation(wo, id);
            }
        }
        wo->direction_property[0] = 0;
        KNOWLEDGE_LOOP(inf, wo, PROPERTY_INF) {
            if (inf_get_property_name(inf) == Inform_property_PRN) {
                char bigbuff[MAX_STRING_LENGTH]; char *bb;
                inf_make_blind_property(inf);
                sprintf(bigbuff, "%s", lw_array[inf->word_ref1].lw_rawtext);
                bb = bigbuff; if (bb[0]=='\"') bb++;
                if (bb[strlen(bb)-1] == '\"') bb[strlen(bb)-1] = 0;
                bb[31] = 0;
                strcpy(wo->direction_property, bb);
            }
        }
    }
    parse_node *p;
```

```
    for (TREE_START(p); p; TREE_NEXT(p))
        if ((pn_get_node_type(p) == SENTENCE_NT) &&
            (pn_int_annotation(p, category_of_I6_translation_ANNOT) == WORLD_OBJECT_I6TR)) {
            world_object *wo =
                parse_world_object(p->down->next->word_ref1, p->down->next->word_ref2, FALSE);
            if (wo == NULL) {
                LOG("Tree is:\n$T", p);
                sentence_problem(_P_(C9BadObjectTranslation),
                    "there is no such object or kind",
                    "so its name will never be translated into I6 in any event.");
                return;
            }
            wo_set_I6_representation(wo, lw_array[p->down->next->next->word_ref1].lw_text);
        }
}
```

The function Inform_name_the_objects is invoked by a command in a `.i6t` template file.

§23. Rough results to the debugging log.

```
void log_brief_picture(void) {
    int i = 0;
    world_object *wo;
    property_name *pname;
    if (dl_this(BRIEF_PICTURE_DA) == FALSE) return;
    LOOP_OVER(wo, world_object) {
        log_world_object(wo);
        i++; if (i%4 == 0) LOG("\n"); else LOG("   ");
    }
    LOG("\n\n");
    LOOP_OVER(pname, property_name) {
        log_property_name(pname);
        LOG("\n");
    }
}
```

The function log_brief_picture is invoked by a command in a `.i6t` template file.

Compile Object Tree 9/cot

Purpose

To write out the I6 object or class schemas as needed, looking through the inferences as needed to find the property values required, and to declare the objects in foundational order, as I6 (unlike I7) requires.

9/cot.§19 Initial time and place; §20 Runtime storage for relations, where needed

---

Template interpreter commands

```
0   {-callv:compile_object_tree}
17   {-routine:I7_Kind_Name}
19   {-array:InitialSituation}
```

```
define INSTANCE_COUNT(wo, k) kind_instance_counts[(wo)->allocation_id*max_kind_instance_count
+ (k)->allocation_id]

world_object *objects_in_I6_order = NULL;
world_object *last_object_in_I6_order = NULL;

int *kind_instance_counts = NULL;
int max_kind_instance_count = 0;
int no_directions = 0;

void compile_object_tree(OUTPUT_STREAM) {
    world_object *wo, *k;
    max_kind_instance_count = 0;
    LOOP_OVER(wo, world_object)
        if (max_kind_instance_count <= wo->allocation_id)
            max_kind_instance_count = wo->allocation_id+1;
    kind_instance_counts =
        I7_calloc(max_kind_instance_count*max_kind_instance_count, sizeof(int),
            INSTANCE_COUNTING_MREASON);

    LOOP_OVER(k, world_object)
        LOOP_OVER(wo, world_object)
            INSTANCE_COUNT(wo, k) = -1;

    LOOP_OVER(k, world_object)
        if (k->kind_flag) {
            int ix_count = 0;
            LOOP_OVER(wo, world_object)
                if (k == kind_kind) {
                    if (wo->kind_flag == TRUE) {
                        INSTANCE_COUNT(wo, k) = ix_count++;
                    }
                } else {
                    if ((wo->kind_flag == FALSE) && (wo_of_kind(wo, k))) {
                        INSTANCE_COUNT(wo, k) = ix_count++;
                    }
                }
            if (k == kind_direction) no_directions = ix_count;
```

```
        }
    WRITE("Array KindHierarchy --> ");
    LOOP_OVER(k, world_object)
        if (k->kind_flag) {
            WRITE("%s (%d) ", wo_get_I6_representation(k), INSTANCE_COUNT(k->kind, kind_kind));
        }
    WRITE(";\n");
    if (no_directions > MAX_DIRECTIONS)
        no_directions = MAX_DIRECTIONS;
    WRITE("Constant No_Directions = %d;\n", no_directions);
    WRITE("Array Map_Storage -->\n");
    WRITE("! ");
    LOOP_OVER(wo, world_object)
        if ((wo_of_kind(wo, kind_direction)) && (wo->kind_flag == FALSE))
            WRITE(" %s", wo_get_I6_representation(wo));
    WRITE("\n");
    if (existing_story_file) WRITE(" 0 0 0 0");                    as there are no rooms then
    int words_used = 0;
    LOOP_OVER(wo, world_object)
        if ((wo_of_kind(wo, kind_room)) && (wo->kind_flag == FALSE)) {
            int i;
            inference *inf;
            world_object *to[MAX_DIRECTIONS];
            for (i=0; i<MAX_DIRECTIONS; i++) to[i] = NULL;
            POSITIVE_KNOWLEDGE_LOOP(inf, wo, DIRECTION_INF) {
                world_object *ref1, *ref2;
                inf_get_object_references(inf, &ref1, &ref2);
                if (ref1) to[INSTANCE_COUNT(ref2, kind_direction)] = ref1;
            }
            for (i=0; i<no_directions; i++) {
                if (to[i])
                    WRITE(" %s", wo_get_I6_representation(to[i]));
                else
                    WRITE(" 0");
                words_used++;
            }
            WRITE(" ! Exits from: %s\n", wo_get_I6_representation(wo));
        }
    WRITE(";\n\n");
    note_VM_usage("map", -1, -1, "map of rooms and doors", words_used, 0, FALSE);
    compute_equivalence_partitions();
    LOOP_OVER(wo, world_object) if (wo->kind_flag) compile_object(OUT, wo);
    compile_value_properties(OUT);
    LOOP_OVER(wo, world_object)
        if ((wo->kind_flag == FALSE) && (wo->object_tree_parent == NULL))
            add_to_object_sequence(wo, 0);
    for (wo = objects_in_I6_order; wo; wo = wo->next_in_I6_order)
        compile_object(OUT, wo);
    place_instance_starts(OUT);
}
void add_to_object_sequence(world_object *wo, int depth) {
```

```
    if (objects_in_I6_order == NULL) { objects_in_I6_order = wo; }
    else last_object_in_I6_order->next_in_I6_order = wo;
    wo->next_in_I6_order = NULL;
    last_object_in_I6_order = wo;
    wo->I6_definition_depth = depth;

    if (wo->object_tree_child)
        add_to_object_sequence(wo->object_tree_child, depth+1);
    if (wo->object_tree_sibling)
        add_to_object_sequence(wo->object_tree_sibling, depth);
}
```

The function compile_object_tree is invoked by a command in a `.i6t` template file.

§**1.** And here is the actual object definition. The `depth` parameter represents the depth in the tree: 0 means "has no parent" and so on. Recall from the DM4 that an I6 object/class declaration consists of a mandatory header line followed by a series of one or more `class`, `with` or `has` clauses, and that there can be any number of each of these kinds of clause: we will make heavy use of the ability to have more than one `with` clause, in particular.

The most surprising feature of the declarations produced by this routine is that they do not (normally) include `before`, `after` or `life` properties. Nothing actually prevents such things from working (and explicit I6 inclusions can be used to paste them in), but action handling is now the province of the rules engine at run-time, and is *not* handled in an object-oriented way any longer.

```
void compile_object(OUTPUT_STREAM, world_object *wo) {
    int i, j, w1, w2, door_sides, short_named, plural_named, articled, door_to_d;
    world_object *k, *wo2;
    inference *inf;
    property_name *prn, *switched_prn = NULL;
    int depth = wo->I6_definition_depth;
    int words_used = 0;

    LOGIF(OBJECT_COMPILATION, "Compiling object definition for $O\n", wo);

    ⟨CO - Compile object header 2⟩;
    ⟨CO - Compile class clause 3⟩;

    compile_inclusions_for_wo(OUT, wo);                    Any pasted-in inclusions

    ⟨CO - Compile the name property 4⟩;
    ⟨CO - Compile the parse name property 6⟩;
    ⟨CO - Compile path-finding workspace 5⟩;

    ⟨CO - Compile component part information 7⟩;
    ⟨CO - Compile found in information for backdrops 8⟩;
    ⟨CO - Compile grammatical attributes 9⟩;

    ⟨CO - Compile inferred properties and attributes 10⟩;

    ⟨CO - Compile article property 11⟩;
    ⟨CO - Compile short name property 12⟩;
    ⟨CO - Compile list together property 13⟩;
    ⟨CO - Compile door-related properties 14⟩;
    ⟨CO - Compile empty description for room if necessary 15⟩;

    compile_permitted_but_unspecified_properties(OUT, wo);
    ⟨CO - Compile action bitmap property 16⟩;

    WRITE(";\n\n");
    wo->rough_array_memory_used = words_used;
    words_used = 0;
    for (k = wo; k && (k!=kind_kind) && (k->kind_flag); k=k->kind)
```

```
        words_used += k->rough_array_memory_used;
    if ((wo->kind_flag) && (wo != kind_kind))
        note_VM_usage("object", wo->word_ref1, wo->word_ref2, NULL, 16+words_used, 0, TRUE);
    LOGIF(OBJECT_COMPILATION, "Compilation of $O complete\n", wo);
}
```

§**2.** For the header syntax, see the DM4. Note that we use the `->` method to indicate parentage for all objects other than directions, where we explicitly specify that they must belong to I6's special "Compass" object. Note also that the "hardwired" short name is intentionally made blank: we always use I6's `short_name` property instead.

⟨CO - Compile object header 2⟩ ≡

```
    if (wo->kind_flag) WRITE("Class "); else WRITE("Object ");
    if ((wo->kind_flag == FALSE) && (wo->kind != kind_direction))
        for (i=0; i<depth; i++) WRITE("-> ");
    WRITE("%s ", wo_get_I6_representation(wo));
    if ((strcmp(wo_get_I6_representation(wo), "selfobj")==0) &&
        (wo->object_tree_parent)) {
        start_object = wo->object_tree_parent;
        start_room = start_object;
        while ((start_room) && (start_room->object_tree_parent))
            start_room = start_room->object_tree_parent;
        if ((start_room) && (start_room->room_flag == FALSE)) {
            object_problem(_P_(C9StartsOutsideRooms),
                start_object,
                "seems to be where the player is supposed to begin",
                "but (so far as I know) it is not a room, nor is it "
                "ultimately contained inside a room.");
        }
    }
    if (wo->kind_flag == FALSE) WRITE("\"\"");
    if (wo->kind == kind_direction) WRITE(" Compass");
    WRITE("\n");
```

This code is used in §1.

§**3.** We specify the class of an instance in the old-fashioned way, so that the same code can be used for both Object and Class schemas.

⟨CO - Compile class clause 3⟩ ≡

```
/*  if (!((strcmp(wo_get_I6_representation(wo), "selfobj")==0) &&
        (qty_initial_value(player_quantity) != wo_yourself) &&
        (qty_initial_value(player_quantity) != NULL))) { */
        if ((wo->kind != NULL) && (wo != kind_kind)) {
            WRITE("class %s\n", wo_get_I6_representation(wo->kind));
            words_used++;
        }
        if (wo->kind_flag == FALSE) place_instance_link(OUT, wo, wo->kind);
```

This code is used in §1.

§**4.** The name property requires special care, partly over I6 eccentricities such as the way that single-letter dictionary words can be misinterpreted as characters (hence the double slash below), but also because something called "your ..." in the source text – "your nose", say – needs to be altered to "my ..." for purposes of parsing during play.

Note that `name` is additive in I6 terms, meaning that its values accumulate from class down to instance: but we prevent this, by only compiling `name` properties for instance objects directly. The practical consequence is that we have to imitate this inheritance when it comes to single-word grammar for things. Recall that a sentence like "Understand "cube" as the block" formally creates a grammar line which ought to be parsed as part of some elaborate `parse_name` property: but that for efficiency's sake, we notice that "cube" is only one word and so put it into the `name` property instead. And we need to perform the same trick for the kinds we inherit from.

⟨CO - Compile the name property 4⟩ ≡

```
if ((wo->kind_flag == FALSE) &&
    (wo != kind_direction) && (wo_is_nameless(wo) == FALSE)) {
    int pw1, pw2;
    words_used++;
    WRITE(" with name ");
    if (wo->word_ref1 >= 0) {
        w1 = wo->word_ref1;
        w2 = wo->word_ref2;
    } else {
        w1 = wo->kind->word_ref1;
        w2 = wo->kind->word_ref2;
    }
    if (wo->creator_plural != NULL) {
        pw1 = wo->creator_plural->word_ref1;
        pw2 = wo->creator_plural->word_ref2;
    } else {
        pw1 = wo->kind->creator_plural->word_ref1;
        pw2 = wo->kind->creator_plural->word_ref2;
    }
    LOGIF(PLURALS,
        "Object $O: <$W>\n", wo, w1, w2);
    LOGIF(PLURALS,
        "Object $O: wo->creator_plural %d, wo->kind->creator_plural %d, pw = <$W>\n",
        wo, (wo->creator_plural)?1:0, (wo->kind->creator_plural)?1:0, pw1, pw2);
    for (j=w1; j<=w2; j++) {
        char *p = lw_array[j].lw_text;
        if ((j==w1) && [[word j == your]]) p = "my";
        isn_compile_dictionary_word(OUT, p, FALSE);
        WRITE(" ");
        words_used++;
    }
    if (wo->creator_plural == NULL)                    see test case C9PluralsFromKind
        for (j=pw1; j<=pw2; j++) {
            int k, additional = TRUE;
            for (k=w1; k<=w2; k++)
                if (compare_word(j, lw_array[k].lw_identity)) additional = FALSE;
            if (additional) isn_compile_dictionary_word(OUT, lw_array[j].lw_text, TRUE);
            WRITE(" ");
            words_used++;
        }
```

```
    wo2 = wo;
    while ((wo2) && (wo2 != kind_kind)) {
        if (wo2->understand_as_this_object)
            gv_take_out_one_word_grammar(OUT, wo2->understand_as_this_object);
        wo2 = wo2->kind;
    }
    WRITE("\n");
}
```

This code is used in §1.

§**5.** Workspace for the path-finding algorithm:

⟨CO - Compile path-finding workspace 5⟩ ≡

```
    WRITE(" with vector 0\n");
    if (wo->room_flag) WRITE(" with room_index -1\n");
    if (wo == kind_room) words_used += 2;
```

This code is used in §1.

§**6.** We attach numbered parse name routines as properties for any object where grammar has specified a need. (By default, this will not happen.)

⟨CO - Compile the parse name property 6⟩ ≡ {

```
    gpr_compile_parse_name_property(OUT, wo, wo->understand_as_this_object);
}
```

This code is used in §1.

§**7.** Component parts are represented in I6 in a rather clumsy way, using the `add_to_scope` property. We create this as needed, and we also set a property not normally found in the I6 library, `component_parent`, to make it possible to see from an object what it is a part of. This is done in order to make it easier to produce responses like "The handle is attached to the door" in reply to commands like "take handle" at run-time.

⟨CO - Compile component part information 7⟩ ≡

```
    if (wo->part_of != NULL)
        WRITE(" with component_parent %s\n",
            wo_get_I6_representation(wo->part_of));
    if (wo->first_part != NULL)
        WRITE(" with component_child %s\n",
            wo_get_I6_representation(wo->first_part));
    if (wo->next_part != NULL)
        WRITE(" with component_sibling %s\n",
            wo_get_I6_representation(wo->next_part));
```

This code is used in §1.

§**8.** The I6 `found_in` property is used both for two-sided doors and for backdrop objects, but we handle two-sided doors separately (see below).

⟨CO - Compile found in information for backdrops 8⟩ ≡ {

```
    int found_in_compiled = FALSE;
    if (wo->kind == kind_region)
        WRITE(" with regional_found_in [; if (TestRegionalContainment(location, %s)) rtrue;
],",
            wo_get_I6_representation(wo));
    if (wo == kind_region) words_used += 2;
    i = FALSE;
    POSITIVE_KNOWLEDGE_LOOP(inf, wo, FOUNDIN_INF) {
        if (inf_get_object_reference(inf) == NULL) {
            WRITE(" with found_in FoundEverywhere,");
            i = TRUE; found_in_compiled = TRUE;
        } else {
            if (wo_of_kind(inf_get_object_reference(inf), kind_region)) i = 2;
        }
    }
    if (i == 2) {
        found_in_compiled = TRUE;
        WRITE(" with found_in [; ");
        POSITIVE_KNOWLEDGE_LOOP(inf, wo, FOUNDIN_INF) {
            if (wo_of_kind(inf_get_object_reference(inf), kind_region))
                WRITE("if (TestRegionalContainment(location, %s)) rtrue; ",
                    wo_get_I6_representation(inf_get_object_reference(inf)));
            else
                WRITE("if (location == %s) rtrue; ",
                    wo_get_I6_representation(inf_get_object_reference(inf)));
        }
        WRITE("],");
    }
    if (i == FALSE) {
        i = FALSE;
        POSITIVE_KNOWLEDGE_LOOP(inf, wo, FOUNDIN_INF) {
            if (i == FALSE) WRITE(" with found_in");
            i = TRUE; found_in_compiled = TRUE;
            WRITE(" %s", wo_get_I6_representation(inf_get_object_reference(inf)));
        }
    }
    if (i) WRITE("\n");
    if ((wo->kind_flag == FALSE) && (wo_of_kind(wo, kind_backdrop)) &&
        (found_in_compiled == FALSE)) {
        WRITE(" with found_in [; rfalse; ],\n  has absent\n");
    }
    if (wo == kind_backdrop) words_used += 2;
}
```

This code is used in §1.

§**9.** This is where we hand on useful grammatical information to the I6 library.

⟨CO - Compile grammatical attributes 9⟩ ≡

```
if (wo->pluralname) WRITE("  has pluralname\n");
if (wo->kind_flag == FALSE) {
    if (wo->propername) WRITE("  has proper\n");
}
```

This code is used in §1.

§**10.** The bulk of properties and attributes are set here.

⟨CO - Compile inferred properties and attributes 10⟩ ≡

```
inf = wo->knowledge;
switched_prn = NULL;
door_sides = 0;
short_named = FALSE; articled = FALSE; door_to_d = FALSE;
plural_named = FALSE;
POSITIVE_KNOWLEDGE_LOOP(inf, wo, ISWORN_INF) {
    WRITE("  has worn clothing\n");
    break;
}
KNOWLEDGE_LOOP(inf, wo, PROPERTY_INF) {
    current_sentence = inf_inferred_from(inf);
    prn = inf_get_property_name(inf);
    if ((prn) && (prn != specification_PRN)) {
        prn_set_last_initialised_for(prn, wo);
        if (prn_is_either_or(prn)) {
            property_name *prnbar;
            prnbar = prn_either_or_get_negation(prn);
            if (!((strcmp(prn_get_i6_identifier(prn), "privately_named")==0) ||
                (prnbar && (strcmp(prn_get_i6_identifier(prnbar), "privately_named")==0))))
{
                if (prnbar) prn_set_last_initialised_for(prnbar, wo);
                if (inf_get_certainty(inf) < 0)
                    write_attribute_name(OUT, prn, FALSE);
                else write_attribute_name(OUT, prn, TRUE);
                WRITE("\n");
            }
        } else {
            if (inf_get_certainty(inf) > 0) {
                if (switched_prn != prn) {
                    switched_prn = prn;
                    WRITE(" with %s ",
                        prn_get_i6_identifier(prn));
                    if (strcmp("short_name", prn_get_i6_identifier(prn)) == 0)
                        short_named = TRUE;
                    if (strcmp("plural", prn_get_i6_identifier(prn)) == 0)
                        plural_named = TRUE;
                    if (strcmp("article", prn_get_i6_identifier(prn)) == 0)
                        articled = TRUE;
                    if (strcmp("door_to", prn_get_i6_identifier(prn)) == 0)
                        door_to_d = TRUE;
                    words_used += 2;
```

```
                    }
                    LOGIF(OBJECT_COMPILATION, "$I\n", inf);
                    if (inf_get_object_reference(inf) != NULL)
                        WRITE("%s\n",
                            wo_get_I6_representation(inf_get_object_reference(inf)));
                    else {
                        specification *val = inf_get_literal_value(inf);
                        current_sentence = inf_inferred_from(inf);
                        if (val) {
                            BEGIN_COMPILATION_MODE;
                            COMPILATION_MODE_EXIT(DEREFERENCE_POINTERS_CMODE);
                            spec_compile(OUT, val);
                            END_COMPILATION_MODE;
                            WRITE("\n");
                        } else
                            sentence_problem(_P_(BelievedImpossible),
                                "a property value seems to be missing",
                                "since this is not an either/or property.");
                    }
                }
            }
        }
    }
    wo->defines_printed_name = short_named;
    wo->defines_plural_name = plural_named;
    if (wo->kind_flag == FALSE) {
        WRITE("  with IK_0 %d\n", INSTANCE_COUNT(wo->kind, kind_kind));
        LOOP_OVER(k, world_object)
            if ((k->kind_flag) && (k != kind_kind) && (wo_of_kind(wo, k)))
                WRITE("  with IK_%d %d\n",
                    k->allocation_id, INSTANCE_COUNT(wo, k));
    } else {
        k = wo;
        while ((k) && (k != kind_kind)) {
            words_used += 4;
            k = k->kind;
        }
    }
    equivalence_relations_add_properties(OUT, wo);
```

This code is used in §1.

§**11.** If we believe that the object's name ought to take the definite article, but have not already set an article as a result of specific instructions in the source text, then we do so here.

⟨CO - Compile article property 11⟩ ≡

```
    if ((articled == FALSE) &&
        ((wo->article_is_the) ||
            ((wo->named_after) && (wo->named_after->article_is_the))))
        WRITE(" with article \"the\"\n");
```

This code is used in §1.

§**12.** The short name is always coded in I6 as a property, so that inheritance will work properly as between kinds and things. Note that it is important here to preserve the cases of the original source text description, so that "Mr Beebe" will not be flattened to "mr beebe"; but that we take care to reduce the case of "Your nose" (etc.) to "your nose".

⟨CO - Compile short name property 12⟩ ≡

```
if ((wo->kind_flag == FALSE) && (wo->defines_printed_name == FALSE)) {
    world_object *k = wo->kind;
    int inherits_printed_name = FALSE;
    while ((k) && (k != kind_kind)) {
        if (k->defines_printed_name) inherits_printed_name = TRUE;
        k = k->kind;
    }
    if (inherits_printed_name) wo->defines_printed_name = TRUE;
    else {
        int begins_with_lower_case = TRUE;
        if (wo->word_ref1 >= 0) {
            w1 = wo->word_ref1;
            w2 = wo->word_ref2;
        } else {
            w1 = wo->kind->word_ref1;
            w2 = wo->kind->word_ref2;
        }
        if ((wo->named_after) && (wo->named_after->defines_printed_name)) {
            WRITE("  with short_name [; print (name) %s, \"'s ",
                wo->named_after->wo_I6_identifier);
            for (j=wo->named_after_w1; j<=wo->named_after_w2; j++) {
                isn_compile_string(OUT, lw_array[j].lw_rawtext, 0);
                if (j<wo->named_after_w2) WRITE(" ");
            }
            WRITE("\"; rtrue; ],\n");
        } else {
            WRITE("  with short_name \"");
            if (w1 >= 0) {
                for (j=w1; j<=w2; j++) {
                    if ((j==w1) && (wo->room_flag == FALSE) &&
                        [[word j == your]]) WRITE("your");
                    else {
                        char *p = lw_array[j].lw_rawtext;
                        if (j == w1) {
                            begins_with_lower_case = FALSE;
                            if (islower(p[0])) begins_with_lower_case = TRUE;
                        }
                        isn_compile_string(OUT, p, 0);
                    }
                    if (j<w2) WRITE(" ");
                }
            } else WRITE("object");
            WRITE("\"\n");
        }
        if ((wo->named_after) && (wo->named_after->defines_printed_name)) {
            WRITE("  with cap_short_name [; "
                "PrintOrRun(%s, cap_short_name, true); print \"'s ",
                wo->named_after->wo_I6_identifier);
```

```
            for (j=wo->named_after_w1; j<=wo->named_after_w2; j++) {
                isn_compile_string(OUT, lw_array[j].lw_rawtext, 0);
                if (j<wo->named_after_w2) WRITE(" ");
            }
            WRITE("\"; rtrue; ],\n");
        } else {
            if ((wo->propername) && (begins_with_lower_case)) {
                WRITE("  with cap_short_name \"");
                if (w1 >= 0) {
                    for (j=w1; j<=w2; j++) {
                        if ((j==w1) && (wo->room_flag == FALSE) &&
                            [[word j == your]]) WRITE("Your");
                        else {
                            char *p = lw_array[j].lw_rawtext;
                            if (j == w1) isn_compile_string(OUT, p, ISN_CAPITALISE);
                            else isn_compile_string(OUT, p, 0);
                        }
                        if (j<w2) WRITE(" ");
                    }
                } else WRITE("Object");
                WRITE("\"\n");
            }
        }
    }
}
if ((wo->kind_flag) && (wo->defines_plural_name == FALSE) &&
    (wo_of_kind(wo, kind_room) == FALSE) && (wo != kind_thing) &&
    (wo->creator_plural)) {
    WRITE("  with plural \"");
    w1 = wo->creator_plural->word_ref1;
    w2 = wo->creator_plural->word_ref2;
    if (w1 >= 0) {
        for (j=w1; j<=w2; j++) {
            if ((j==w1) && [[word j == your]]) WRITE("your");
            else isn_compile_string(OUT, lw_array[j].lw_rawtext, 0);
            if (j<w2) WRITE(" ");
        }
    } else WRITE("object");
    WRITE("\"\n");
}
```

This code is used in §1.

§**13.** Perhaps surprisingly, every Object schema contains a `list_together` property. This is so that arbitrary tricks can be pulled at run-time with the grouping together activity: it imposes a small overhead, but is worth it.

⟨CO - Compile list together property 13⟩ ≡

```
if (wo->kind_flag == FALSE) WRITE("  with list_together 0,\n");
```

This code is used in §1.

§**14.** See the DM4 for details of how to compile one and two-sided doors in I6. Alternatively, take it on trust that there is nothing surprising here.

⟨CO - Compile door-related properties 14⟩ ≡

```
if (wo->door_flag) {
    door_sides = 0;
    if (wo->map_connection_a != NULL) door_sides++;
    if (wo->map_connection_b != NULL) door_sides++;
    if (door_sides == 2) {
        if (door_to_d) {
            object_problem(_P_(C9BothWaysDoor),
                wo, "seems to be a door whose connections have been "
                "given in both of the alternative ways at once",
                "by directly giving its map connections (the normal "
                "way to set up a two-sided door) and also by saying "
                "what is through it (the normal way to set up a one-sided "
                "door). As a door can't be both one- and two-sided at "
                "once, I'm going to object to this.");
        }
        WRITE(" with found_in %s %s,\n",
            wo_get_I6_representation(wo->map_connection_a),
            wo_get_I6_representation(wo->map_connection_b));
        WRITE(" with door_dir [ loc; loc = location;\n"
            "        if (loc == thedark) loc = real_location;\n"
            "        if (loc == %s) return %s; return %s; ],\n",
            wo_get_I6_representation(wo->map_connection_a),
            wo_get_I6_representation(wo->map_direction_b),
            wo_get_I6_representation(wo->map_direction_a));
        WRITE(" with door_to [ loc; loc = location;\n"
            "        if (loc == thedark) loc = real_location;\n"
            "        if (loc == %s) return %s; return %s; ],\n",
            wo_get_I6_representation(wo->map_connection_a),
            wo_get_I6_representation(wo->map_connection_b),
            wo_get_I6_representation(wo->map_connection_a));
    }
    if (door_sides == 1) {
        world_object *backwards = get_value_of_opposite_property(wo->map_direction_a);
        if (backwards)
            WRITE(" with door_dir %s,\n",
                wo_get_I6_representation(backwards));
    }
}
if (wo == kind_door) words_used += 14;
```

This code is used in §1.

§**15.** In I6, a room must have a description, unlike in I7. We give it a dummy description if necessary (i.e., not if it has one already, and not if it inherits one from its kind anyway).

⟨CO - Compile empty description for room if necessary 15⟩ ≡

```
if ((wo->room_flag) && (wo->defines_description == FALSE)) {
    k = wo->kind;
    while ((k != NULL) && (k != kind_kind)) {
        if (k->defines_description) goto DescriptionProvided;
        k = k->kind;
    }
    WRITE(" with description NULL,\n");
    DescriptionProvided: ;
}
if (wo->room_flag) WRITE(" has mark_as_room,\n");
if (wo_of_kind(wo, kind_thing)) WRITE(" has mark_as_thing,\n");
```

This code is used in §1.

§**16.** The action bitmap is an array of bits attached to each object, one for each action, which records whether that action has yet applied successfully to that object. This is used at run-time to handle past tense conditions such as "the jewels have been taken". Note that we give the bitmap in the class definition associated with "thing" to ensure that it will be inherited by all I6 objects of this class, i.e., all I6 objects corresponding to I7 things.

⟨CO - Compile action bitmap property 16⟩ ≡

```
if (((wo->room_flag == FALSE) && (wo->kind_flag == FALSE))
    || (wo == kind_thing)) {
    act_compile_action_bitmap_property(OUT);
}
```

This code is used in §1.

§**17.** Code for a table of names of kinds.

```
void compile_I7_Kind_Name_routine(OUTPUT_STREAM) {
    world_object *wo;
    WRITE("[ I7_Kind_Name k;\n"); INDENT;
    LOOP_OVER(wo, world_object) if ((wo->kind_flag) && (wo != kind_kind)) {
        int w1, w2;
        w1 = wo->word_ref1;
        w2 = wo->word_ref2;
        WRITE("if (k == %s) print \"", wo_get_I6_representation(wo));
        print_raw_text_to_file(w1, w2, OUT);
        WRITE("\";\n");
    }
    OUTDENT; WRITE("];\n");
}
```

The function compile_I7_Kind_Name_routine is invoked by a command in a `.i6t` template file.

§**18.** And code for equivalence classes.

```
void compute_equivalence_partitions(void) {
    int domain_size = 0;
    world_object *wo, *wo2;
    binary_predicate *bp;
    LOOP_OVER(wo, world_object)
        if (wo->allocation_id >= domain_size)
            domain_size = wo->allocation_id+1;
    equivalence_relations_make_singleton_partitions(domain_size);
    LOOP_OVER(wo, world_object) {
        inference *i;
        POSITIVE_KNOWLEDGE_LOOP(i, wo, MATCHING_INF) {
            wo2 = inf_get_object_reference(i);
            bp = inf_get_bp(i);
            equivalence_relation_merge_classes(bp, domain_size, wo->allocation_id, wo2->allocation_id);
        }
    }
}
void place_instance_link(OUTPUT_STREAM, world_object *wo, world_object *k) {
    world_object *wo2; int found = FALSE;
    WRITE("  with IK%d_Link ", k->allocation_id);
    k->instance_list_length++;
    for (wo2 = wo->next_in_I6_order; wo2; wo2 = wo2->next_in_I6_order)
        if ((wo2->kind_flag == FALSE) && (wo_of_kind(wo2, k))) {
            if ((wo2 == wo_yourself) &&
                (qty_initial_value(player_quantity) != NULL) &&
                (qty_initial_value(player_quantity) != wo_yourself))
                continue;
            WRITE("%s", wo_get_I6_representation(wo2));
            found = TRUE;
            break;
        }
    if (found == FALSE) WRITE("nothing");
    WRITE(",\n");
    if (k->kind != kind_kind) place_instance_link(OUT, wo, k->kind);
}
void place_instance_starts(OUTPUT_STREAM) {
    world_object *wo, *wo2; int found;
    LOOP_OVER(wo, world_object)
        if (wo->kind_flag) {
            WRITE("Constant IK%d_First = ", wo->allocation_id);
            found = FALSE;
            for (wo2 = objects_in_I6_order; wo2; wo2 = wo2->next_in_I6_order)
                if ((wo2->kind_flag == FALSE) && (wo_of_kind(wo2, wo))) {
                    if ((wo2 == wo_yourself) &&
                        (qty_initial_value(player_quantity) != NULL) &&
                        (qty_initial_value(player_quantity) != wo_yourself))
                        continue;
                    WRITE("%s", wo_get_I6_representation(wo2));
                    found = TRUE;
                    break;
                }
```

```
            if (found == FALSE) WRITE("nothing");
            WRITE(";\n");
        }
}
```

§**19. Initial time and place.** Well... and also the initial state of randomness in the world, for the sake of convenience.

```
void compile_InitialSituation_array(OUTPUT_STREAM) {
    WRITE("Array InitialSituation --> ");
    compile_quantity_initial(OUT, player_quantity);
    WRITE(" %s", wo_get_I6_representation(start_object));
    WRITE(" %s ", wo_get_I6_representation(start_room));
    compile_quantity_initial(OUT, time_of_day_quantity);
    WRITE(";\n\n");
}
```

The function compile_InitialSituation_array is invoked by a command in a `.i6t` template file.

§**20. Runtime storage for relations, where needed.**

```
void begin_collating_vpairs(binary_predicate *bp, int left_count, int right_count) {
    inference *inf;
    GENERAL_POSITIVE_KNOWLEDGE_LOOP(inf, inference_subject_bp(bp), VALUEARBITRARY_INF) {
        world_object *left_wo, *right_wo;
        specification *left_spec, *right_spec;
        int *indices;
        inf_get_all_references(inf, &left_spec, &left_wo, &right_spec, &right_wo, &indices);
        if (left_wo) indices[0] = left_wo->relation_indices[0];
        else {
            quantity *q = spec_get_constant_quantity_if_any(left_spec);
            if (q == NULL) internal_error("Generated VA with non-quantity value");
            indices[0] = qty_get_value_index(q);
        }
        if (right_wo) indices[1] = right_wo->relation_indices[1];
        else {
            quantity *q = spec_get_constant_quantity_if_any(right_spec);
            if (q == NULL) internal_error("Generated VA with non-quantity value");
            indices[1] = qty_get_value_index(q);
        }
        if ((indices[0] < 0) || (indices[0] >= left_count))
            internal_error("Left count goes wrong in VA");
        if ((indices[1] < 0) || (indices[1] >= right_count))
            internal_error("Right count goes wrong in VA");
    }
}
void send_vpairs_in_row(binary_predicate *bp, int i, char *handle) {
    inference *inf;
    GENERAL_POSITIVE_KNOWLEDGE_LOOP(inf, inference_subject_bp(bp), VALUEARBITRARY_INF) {
        world_object *left_wo, *right_wo; specification *left_spec, *right_spec;
        int *indices;
        inf_get_all_references(inf, &left_spec, &left_wo, &right_spec, &right_wo, &indices);
        if (indices[0] == i) send_related_index(bp, indices[1], handle);
```

```
    }
}

void begin_collating_wopairs(binary_predicate *bp, int left_count, int right_count) {
}

void send_wopairs_in_row(binary_predicate *bp, world_object *wo, char *handle) {
    inference *inf;
    POSITIVE_KNOWLEDGE_LOOP(inf, wo, ARBITRARY_INF)
        if (inf_get_bp(inf) == bp)
            send_related_index(bp, inf_get_object_reference(inf)->relation_indices[1], handle);
}
```

The function begin_collating_vpairs is called from 5/rel.

The function send_vpairs_in_row is called from 5/rel.

The function begin_collating_wopairs is called from 5/rel.

The function send_wopairs_in_row is called from 5/rel.

Index Physical World 9/inpw

Purpose

To index the kind hierarchy and the object tree.

9/inpw.§2-4 Indexing of kinds

Template interpreter commands

```
1   {-callv:index_page_Kinds}
1   {-callv:index_page_World}
2   {-callv:add_showme_details}
```

¶1. Known bugs:

Contents index omits things created before all headings.

```
void index_backdrops_foundin(world_object *wo, int depth, int details,
    char *bef, char *aft) {
    world_object *wo2;
    int discoveries = 0;
    LOOP_OVER(wo2, world_object) {
        if ((wo2->kind == kind_backdrop)
            && (wo2->kind_flag == FALSE)) {
            inference *inf;
            POSITIVE_KNOWLEDGE_LOOP(inf, wo2, FOUNDIN_INF) {
                if (inf_get_object_reference(inf) == wo) {
                    discoveries++;
                    if (discoveries == 1) INDEX("%s", bef);
                    index_ot(wo2, depth+1, details);
                }
            }
        }
    }
    if (discoveries > 0) INDEX("%s", aft);
}
void index_ot(world_object *wo, int depth, int details) {
    int c, shaded = FALSE;
    world_object *wo2;
    wo->index_appearances++;
    if (depth == 1000) internal_error("Whoops!");
    if (tabulating_kinds_index) index_KOV_begin_chart_row();
    if (details) {
        open_html_paragraph(ifl, depth, "");
        index_anchor(wo_get_I6_representation(wo));
    } else {
        open_html_paragraph(ifl, depth, "tight");
        if (wo->world_index_description != NULL) INDEX("<i>%s</i> ",
            wo->world_index_description);
```

```
}
if (wo->kind_flag) {
    c=0;
    LOOP_OVER(wo2, world_object) {
        world_object *wo3 = wo2;
        if (wo2->kind_flag) continue;
        while (wo3 != kind_kind) {
            if (wo3->kind == wo) { c++; break; }
            wo3 = wo3->kind;
        }
    }
    wo->instance_count = c;
}
if ((tabulating_kinds_index) && (wo->instance_count == 0)) {
    INDEX("<font color=\"%s\">", KINDS_INDEX_SHADE); shaded = TRUE;
}
if (wo->word_ref1 < 0) {
    INDEX("nameless");
} else {
    if ((details) || (wo->room_flag)) INDEX("<b>");
    print_raw_text_to_file(wo->word_ref1, wo->word_ref2, ifl);
    if ((details) || (wo->room_flag))
        INDEX("</b>");
    if (details) {
        if (wo->kind != kind_kind) {
            INDEX(", a kind of ");
            print_raw_text_to_file(wo->kind->word_ref1, wo->kind->word_ref2, ifl);
        }
        if (wo->creator_plural != NULL) {
            INDEX(" (<i>plural</i> ");
            print_raw_text_to_file(wo->creator_plural->word_ref1,
                wo->creator_plural->word_ref2, ifl);
            INDEX(")");
        }
    }
}
if ((tabulating_kinds_index) && (wo->instance_count == 0))
    INDEX("</font>");
if (wo->kind_flag) {
    if ((details == FALSE) && (wo->instance_count != 0)) INDEX(" [%d]", wo->instance_count);
} else {
    if ((wo->kind != NULL) && (wo->kind->word_ref1 >= 0)
        && (wo->kind != kind_thing) && (wo->kind != kind_room)) {
        INDEX(" - <i>");
        print_raw_text_to_file(wo->kind->word_ref1, wo->kind->word_ref2, ifl);
        INDEX("</i>");
    }
    if (wo == start_room) {
        INDEX(" - <i>room where play begins</i>");
        index_doc_link("ROOMPLAYBEGINS");
    }
}
if (wo->creating_sentence != NULL)
```

```
        index_link(lw_array[wo->creating_sentence->word_ref1].lw_source);
    if (details == FALSE) {
        if (wo->kind_flag) index_below_link(wo_get_I6_representation(wo));
        if (wo->wo_documentation_symbol != NULL)
            index_doc_link(wo->wo_documentation_symbol);
        if (tabulating_kinds_index)
            index_KOV_end_chart_row(shaded, wo, -1, "tick", "tick", "tick");
        else INDEX("<br>\n");
    } else {
        if (wo->wo_documentation_symbol != NULL)
            index_doc_link(wo->wo_documentation_symbol);
        INDEX("<br>\n");
        index_kind(wo);
    }
    if (wo->kind_flag) {
        LOOP_OVER(wo2, world_object) {
            if (wo2 == wo) continue;
            if (wo2->kind_flag == FALSE) continue;
            if (wo2 == kind_kind) continue;
            if (wo2->kind == wo) {
                index_ot(wo2, depth+1, details);
            }
        }
    } else {
        if (wo->first_part != NULL) {
            world_object *wo2 = wo->first_part;
            while (wo2 != NULL) {
                index_ot(wo2, depth+1, details);
                wo2 = wo2->next_part;
            }
        }
        if (wo->object_tree_child)
            index_ot(wo->object_tree_child, depth+1, details);
        if (wo->room_flag) {
            world_object *wo2;
            LOOP_OVER(wo2, world_object) {
                if ((wo2->door_flag) && (wo2->object_tree_parent != wo)) {
                    if (wo2->map_connection_a == wo)
                        index_ot(wo2, depth+1, details);
                    if (wo2->map_connection_b == wo)
                        index_ot(wo2, depth+1, details);
                }
            }
            index_backdrops_foundin(wo, depth, details, "", "");
            if ((wo == start_room) && (wo_yourself->index_appearances == 0))
                index_ot(wo_yourself, depth+1, details);
        }
        if (wo->object_tree_sibling)
            index_ot(wo->object_tree_sibling, depth, details);
    }
    if ((details) && (depth > 1)) INDEX("<p>");
}
```

The function index_ot is called from 9/map.

§**1.** The following routine creates both the Kinds and World index files, but it delegates all the real work elsewhere.

```
void index_page_Kinds(void) {
    kinds_documentation();
    index_kinds_of_value(1);
    index_anchor("ARITHMETIC");
    index_dimensional_rules();
    INDEX("<p><hr>");
    index_anchor("KDETAILS");
    index_kinds_of_value(2);
}
void index_object_kinds(int pass) {
    world_object *wo;
    if (pass == 1) {
        LOOP_OVER(wo, world_object) {
            if ((wo->object_tree_parent == NULL) &&
                (wo != kind_kind) && (wo->kind == kind_kind)) {
                index_ot(wo, 2, FALSE);
            }
        }
    }
    if (pass == 2) {
        LOOP_OVER(wo, world_object) {
            if ((wo->object_tree_parent == NULL) &&
                (wo != kind_kind) && (wo->kind == kind_kind)) {
                index_ot(wo, 2, TRUE);
            }
        }
    }
}
void index_page_World(void) {
    int out_of_play_count = 0, unruly = FALSE;
    world_object *wo, *wo2;
    if (existing_story_file) return;
    index_world_map(NULL);
    write_eps_file();
    add_region_key();
    index_backdrops_foundin(NULL, 0, FALSE,
        "<p><b>Present everywhere:</b><br>", "<p><hr><p>");
    index_anchor("MDETAILS");
    LOOP_OVER(wo, world_object) if (wo_of_kind(wo, kind_region)) {
        int subheaded = FALSE;
        wo->index_appearances = 1;
        LOOP_OVER(wo2, world_object) {
            if ((wo2->room_flag) && (wo2->in_region == wo)) {
                if (subheaded == FALSE) {
                    subheaded = TRUE;
                    if (unruly) INDEX("<p><hr><p>");
                    unruly = TRUE;
```

```
                    INDEX("<b>The <i>");
                    print_raw_text_to_file(wo->word_ref1, wo->word_ref2, ifl);
                    INDEX("</i> region");
                    if ((wo->object_tree_parent) &&
                        (wo_of_kind(wo->object_tree_parent, kind_region))) {
                        INDEX(" within the <i>");
                        print_raw_text_to_file(
                            wo->object_tree_parent->word_ref1,
                            wo->object_tree_parent->word_ref2, ifl);
                        INDEX("</i> region");
                    }
                    INDEX("</b>");
                    index_backdrops_foundin(wo, 0, FALSE, "<br>", "");
                    INDEX("<p>");
                }
                index_world_map(wo2);
                wo2->index_appearances = 1;
            }
        }
    }
    LOOP_OVER(wo, world_object) {
        if ((wo->room_flag) && (wo->index_appearances == 0)) {
            if (unruly) INDEX("<p><hr><p>");
            unruly = FALSE;
            index_world_map(wo);
        }
        if ((wo->kind_flag)
            || (wo->part_of != NULL)
            || (wo->kind == kind_direction))
            wo->index_appearances = 1;
    }
    out_of_play_count = 0;
    LOOP_OVER(wo, world_object)
        if (wo->index_appearances == 0)
            out_of_play_count++;
    if (out_of_play_count > 1)
        INDEX("<p><hr><p><b>Offstage (that is, initially not in any room):</b><br>");
    out_of_play_count = 0;
    LOOP_OVER(wo, world_object)
        if (wo->index_appearances == 0)
            if (out_of_play_count++ >= 0)
                index_ot(wo, 1, FALSE);
    index_common_nouns();
}
```

The function index_page_Kinds is invoked by a command in a `.i6t` template file.

The function index_object_kinds is called from 7/kix.

The function index_page_World is invoked by a command in a `.i6t` template file.

§**2. Indexing of kinds.** This is where the detailed description of a given kind – what properties it has, and so on – is generated.

```
void index_kind(world_object *wo) {
    inference *inf; int c, f;
    world_object *wo2;
    property_name *prn;
    KNOWLEDGE_LOOP(inf, wo, PROPERTY_INF) {
        if (inf_get_property_name(inf) == specification_PRN) {
            index_dequote(lw_array[inf->word_ref1].lw_rawtext);
            INDEX("<br>");
        }
    }
    LOOP_OVER(prn, property_name) prn_set_indexed_flag(prn, FALSE);
    for (c = CERTAIN_CE; c >= IMPOSSIBLE_CE; c--) {
        char *cert = "Text only put here to stop gcc -O2 wrongly reporting an error";
        if (c == UNKNOWN_CE) continue;
        switch(c) {
            case CERTAIN_CE:    cert = "Always"; break;
            case LIKELY_CE:     cert = "Usually"; break;
            case UNLIKELY_CE:   cert = "Usually not"; break;
            case IMPOSSIBLE_CE: cert = "Never"; break;
        }
        index_provided(wo, TRUE, c, cert);
    }
    index_provided(wo, FALSE, UNKNOWN_CE, "Can have");
    f = FALSE;
    LOOP_OVER(wo2, world_object) {
        if (wo2 == kind_kind) continue;
        if ((wo2->kind_flag == FALSE) && (wo2->kind == wo)) {
            if (f) INDEX(", "); f = TRUE;
            index_name_object(wo2, "<i>", "</i>");
        }
    }
}
void add_showme_details(OUTPUT_STREAM) {
    world_object *wo;
    property_name *prn;
    LOOP_OVER(wo, world_object) {
        int todo = FALSE;
        if (wo == kind_kind) continue;
        LOOP_OVER(prn, property_name)
            if (showme_provided(OUT, wo, prn, FALSE)) {
                todo = TRUE;
                break;
            }
        if (todo == FALSE) continue;
        if (wo->kind_flag) {
            WRITE("if (view ofclass %s) { na = 0;\n",
                wo_get_I6_representation(wo));
        } else {
            WRITE("if (view == %s) { na = 0;\n",
```

```
                wo_get_I6_representation(wo));
        }
        INDENT;
        LOOP_OVER(prn, property_name)
            if (prn_is_value_property(prn) == FALSE)
                showme_provided(OUT, wo, prn, TRUE);
        WRITE("if (na>0) print \"^\";\n");
        LOOP_OVER(prn, property_name)
            if (prn_is_value_property(prn))
                showme_provided(OUT, wo, prn, TRUE);
        OUTDENT; WRITE("}\n");
    }
}
```

The function add_showme_details is invoked by a command in a `.i6t` template file.

§**3.** The following lists off the properties of the kind, with the given state of being boolean, and the given certainty levels:

```
void index_provided(world_object *wo, int bool, int c, char *cert) {
    int f = TRUE, cp, print_me, named_negation, provision_flag,
        parent_provision_flag,
        parent_provision_sign = 1, parent_certainty_level = UNKNOWN_CE;
    property_name *prn;
    LOOP_OVER(prn, property_name) {
        property_name *print_prn = prn;
        if (prn_is_used_by_i6_library_only(prn)) continue;
        if (prn_get_indexed_flag(prn)) continue;
        print_me = FALSE; named_negation = FALSE;
        if ((prn_is_either_or(prn)) && (prn_either_or_get_negation(prn)))
            named_negation = TRUE;
        parent_provision_flag = FALSE;
        if (wo->kind != kind_kind) {
            parent_provision_flag = does_kind_provide(wo->kind, prn);
            parent_provision_sign = provision_sign;
            parent_certainty_level = UNKNOWN_CE;
            if (provision_inference != NULL)
                parent_certainty_level = inf_get_certainty(provision_inference);
        }
        provision_flag = does_kind_provide(wo, prn);
        if ((provision_inference == NULL) ||
            (parent_certainty_level == inf_get_certainty(provision_inference)))
            if ((provision_flag == parent_provision_flag) &&
                (provision_sign == parent_provision_sign)) continue;
        if (provision_flag) {
            if ((bool) && (prn_is_either_or(prn))) {
                cp = UNLIKELY_CE*provision_sign;
                if (provision_inference != NULL)
                    cp = inf_get_certainty(provision_inference)*provision_sign;
                if (c == cp) print_me = TRUE;
                if ((c > 0) && (c == -cp) && (named_negation)) {
                    print_me = TRUE;
                    print_prn = prn_either_or_get_negation(prn);
                }
```

```
                if ((c < 0) && (c == cp) && (named_negation))
                    print_me = FALSE;
            }
            if ((bool == FALSE) && (prn_is_value_property(prn)))
                print_me = TRUE;
            if ((named_negation) && (provision_sign < 0)) print_me = FALSE;
        }
        if (print_me) {
            if (f) { INDEX("<i>%s</i> ", cert); f = FALSE; }
            else INDEX(", ");
            print_raw_text_to_file(print_prn->word_ref1, print_prn->word_ref2, ifl);
            prn_set_indexed_flag(print_prn, TRUE);
            if (named_negation) {
                property_name *prnbar = prn_either_or_get_negation(print_prn);
                INDEX(" <i>not</i> ");
                print_raw_text_to_file(prnbar->word_ref1, prnbar->word_ref2, ifl);
                prn_set_indexed_flag(prnbar, TRUE);
            }
            if ((prn_is_value_property(print_prn)) &&
                (prn_get_kind_of_value(print_prn) != NULL)) {
                INDEX(" (<i>");
                index_kov(prn_get_kind_of_value(print_prn));
                INDEX("</i>)");
            }
        }
    }
    if (f == FALSE) INDEX(".<br>");
}

int showme_provided(OUTPUT_STREAM, world_object *wo, property_name *prn, int comp) {
    int provision_flag, parent_provision_flag;
    if (prn_is_used_by_i6_library_only(prn)) return FALSE;
    parent_provision_flag = FALSE;
    if (wo->kind != kind_kind) {
        parent_provision_flag = does_kind_provide(wo->kind, prn);
    }
    provision_flag = does_kind_provide(wo, prn);
    if (provision_flag == parent_provision_flag) return FALSE;
    if (provision_flag) {
        if (prn_is_value_property(prn)) {
            kind_of_value *kov = prn_get_kind_of_value(prn);
            if (kov != NULL) {
                if (comp == FALSE) return TRUE;
                WRITE("if (view provides %s) print \"",
                    prn_get_i6_identifier(prn));
                print_raw_text_to_file(prn->word_ref1, prn->word_ref2, OUT);
                WRITE(": \", (%s) ValueProperty(view, %s), \"^\";\n",
                    kov_get_name_of_printing_rule(kov),
                    prn_get_i6_identifier(prn));
                return TRUE;
            }
        } else {
            if (prn_is_ephemeral(prn) == FALSE) {
                if (comp == FALSE) return TRUE;
```

```
                WRITE("if (view ");
                compile_has_property(OUT, prn);
                WRITE(") { if (na++ > 0) print \"; \"; print \"");
                print_raw_text_to_file(prn->word_ref1, prn->word_ref2, OUT);
                WRITE("\"; }\n");
                return TRUE;
            }
        }
    }
    return FALSE;
}
```

§**4.** The following routine looks at each kind, takes the first word of its name, prefixes "kind" and looks to see if there is a documentation symbol of that name: if there is, it attaches it to the relevant field of the world object. That will be useful when we come to index the kind.

```
void kinds_documentation(void) {
    world_object *wo;
    LOOP_OVER(wo, world_object) if ((wo->kind_flag) && (wo->word_ref1 >= 0)) {
        char temp[128];
        sprintf(temp, "kind_%s", lw_array[wo->word_ref1].lw_text);
        wo->wo_documentation_symbol = dref_validate_if_possible(temp);
    }
}
```

Spatial Map 9/map

Purpose

To fit the map of the rooms in the game into a cubical grid, preserving distances and angles where possible, and then render it as HTML using a grid of icons for the World Index and also as an EPS file.

9/map.¶3-0 Components of the index map; §2 Map reading; §3-9 Phase I: Positioning the rooms in space; §10-14 Phase II: Building the grids; §15-23 Phase III: Plotting

---

Definitions

¶1. The EPS map-maker is really a miniature interpreted programming language in its own right, and here we define that language's data types and variables.

The "mapping parameters" amount to being variables. The following structure defines the type and current value for each variable: see the Inform documentation for details. But note that variables of the same name are held by many different objects in the map, and their values inherited by sub-objects.

```
define INT_MDT 1                                   an integer
define BOOL_MDT 2                                  true or false
define TEXT_MDT 3                                  quoted text
define COL_MDT 4                                   an HTML-safe colour
define FONT_MDT 5                                  the name of a font
define OFF_MDT 6                                   a positional offset in an (x, y) grid

typedef struct plotting_parameter {
    int specified;                                 is it explicitly specified at this scope?
    char *name;                                    name (used only in global scope)
    int parameter_data_type;                       one of the above types (used only in global scope)
    char *string_value;                            string value, if appropriate to this type;
    int numeric_value;                             or numeric value, if appropriate to this type
} plotting_parameter;
```

The structure plotting_parameter is private to this section.

¶2. A set of variables associated with any map object is called a "scope". As implied above, the global scope is special: it contains the default settings passed down to all lower scopes.

define NO_MAP_PARAMETERS 35

```
typedef struct map_parameter_scope {
    struct map_parameter_scope *wider_scope;                    that is, the scope above this
    struct plotting_parameter values[NO_MAP_PARAMETERS];
} map_parameter_scope;

map_parameter_scope global_map_scope = {
    NULL,
    {
        { TRUE, "font", FONT_MDT, "Helvetica", 0 },
        { TRUE, "minimum-map-width", INT_MDT, NULL, 72*5 },
        { TRUE, "title", TEXT_MDT, "Map", 0 },
        { TRUE, "title-size", INT_MDT, NULL, 24 },
        { TRUE, "title-font", FONT_MDT, "<font>", 0 },
        { TRUE, "title-colour", COL_MDT, "000000", 0 },
        { TRUE, "map-outline", BOOL_MDT, NULL, 1 },
        { TRUE, "border-size", INT_MDT, NULL, 12 },
        { TRUE, "vertical-spacing", INT_MDT, NULL, 6 },
        { TRUE, "monochrome", BOOL_MDT, NULL, 0 },
        { TRUE, "annotation-size", INT_MDT, NULL, 8 },
        { TRUE, "annotation-length", INT_MDT, NULL, 8 },
        { TRUE, "annotation-font", FONT_MDT, "<font>", 0 },
        { TRUE, "subtitle", TEXT_MDT, "Map", 0 },
        { TRUE, "subtitle-size", INT_MDT, NULL, 16 },
        { TRUE, "subtitle-font", FONT_MDT, "<font>", 0 },
        { TRUE, "subtitle-colour", COL_MDT, "000000", 0 },
        { TRUE, "grid-size", INT_MDT, NULL, 72 },
        { TRUE, "route-stiffness", INT_MDT, NULL, 100 },
        { TRUE, "route-thickness", INT_MDT, NULL, 1 },
        { TRUE, "route-colour", COL_MDT, "000000", 0 },
        { TRUE, "room-offset", OFF_MDT, NULL, 0 },
        { TRUE, "room-size", INT_MDT, NULL, 36 },
        { TRUE, "room-colour", COL_MDT, "DDDDDD", 0 },
        { TRUE, "room-name", TEXT_MDT, "", 0 },
        { TRUE, "room-name-size", INT_MDT, NULL, 12 },
        { TRUE, "room-name-font", FONT_MDT, "<font>", 0 },
        { TRUE, "room-name-colour", COL_MDT, "000000", 0 },
        { TRUE, "room-name-length", INT_MDT, NULL, 5 },
        { TRUE, "room-name-offset", OFF_MDT, NULL, 0 },
        { TRUE, "room-outline", BOOL_MDT, NULL, 1 },
        { TRUE, "room-outline-colour", COL_MDT, "000000", 0 },
        { TRUE, "room-outline-thickness", INT_MDT, NULL, 1 },
        { TRUE, "room-shape", TEXT_MDT, "square", 0 }
    }
};
```

The structure map_parameter_scope is private to this section.

¶3. Components of the index map. A structure used in building the index map, but not during compilation of the story file. Components are separate areas of map not connected by regular exits. Note that components are three-dimensional and occupy subsets of a grid of integer-valued (x, y, z) positions, where the position of the origin is undefined (and not relevant, since only the location of one room relative to another is important).

The extreme values below specify that the component lies inside the cube with opposing corners (x_0, y_0, z_0) and (x_1, y_1, z_1).

```
typedef struct map_component {
    int extreme_x0, extreme_x1, extreme_y0, extreme_y1, extreme_z0, extreme_z1;
    int x_offset, y_offset, z_offset;                    of the bottom corner from the origin
    int size;                                            number of rooms in this component
    int accuracy;                                        current penalty score for bad placement of rooms
    MEMORY_MANAGEMENT
} map_component;
```

The structure map_component is private to this section.

---

§**1.**

```
int map_directions_as_if[MAX_DIRECTIONS];
void find_spatial_offsets(int exit, int *exit_x, int *exit_y, int *exit_z, char **clue) {
    *exit_x=0; *exit_y=0; *exit_z=0; *clue = "";
    if (exit < 0) return;
    exit = map_directions_as_if[exit];
    if (exit >= 12) return;
    switch(exit) {
        case 0: *exit_x=0; *exit_y=1; *exit_z=0; *clue="n"; break;
        case 1: *exit_x=1; *exit_y=1; *exit_z=0; *clue="ne"; break;
        case 2: *exit_x=-1; *exit_y=1; *exit_z=0; *clue="nw"; break;
        case 3: *exit_x=0; *exit_y=-1; *exit_z=0; *clue="s"; break;
        case 4: *exit_x=1; *exit_y=-1; *exit_z=0; *clue="se"; break;
        case 5: *exit_x=-1; *exit_y=-1; *exit_z=0; *clue="sw"; break;
        case 6: *exit_x=1; *exit_y=0; *exit_z=0; *clue="e"; break;
        case 7: *exit_x=-1; *exit_y=0; *exit_z=0; *clue="w"; break;
        case 8: *exit_x=0; *exit_y=0; *exit_z=1; *clue="u"; break;
        case 9: *exit_x=0; *exit_y=0; *exit_z=-1; *clue="d"; break;
        case 10: *clue="in"; break;
        case 11: *clue="out"; break;
    }
}
void find_map_offsets(int exit, int *mx, int *my) {
    *mx = 0; *my = 0;
    if (exit < 0) return;
    exit = map_directions_as_if[exit];
    if (exit >= 12) return;
    switch(exit) {
        case 0: *mx = 2; *my = 0; break;
        case 1: *mx = 4; *my = 0; break;
        case 2: *mx = 0; *my = 0; break;
        case 3: *mx = 2; *my = 4; break;
```

```
        case 4: *mx = 4; *my = 4; break;
        case 5: *mx = 0; *my = 4; break;
        case 6: *mx = 4; *my = 2; break;
        case 7: *mx = 0; *my = 2; break;
        case 8: *mx = 1; *my = 0; break;
        case 9: *mx = 3; *my = 4; break;
        case 10: *mx = 4; *my = 3; break;
        case 11: *mx = 0; *my = 1; break;
    }
}

int direction_is_lateral(int exit) {
    int ex, ey, ez; char *clue;
    find_spatial_offsets(exit, &ex, &ey, &ez, &clue);
    if ((ex == 0) && (ey == 0)) return FALSE;
    return TRUE;
}

int direction_is_along_lattice(int exit) {
    int ex, ey, ez; char *clue;
    find_spatial_offsets(exit, &ex, &ey, &ez, &clue);
    if ((ex == 0) && (ey == 0) && (ez == 0)) return FALSE;
    return TRUE;
}

int direction_is_mappable(int exit) {
    if (exit < 0) return FALSE;
    exit = map_directions_as_if[exit];
    if (exit >= 12) return FALSE;
    return TRUE;
}
```

§**2. Map reading.** The map is read using the `room_exit` routine below, which works out what room the exit leads to (perhaps through a door). It returns either the world-object on the other side, or else `NULL`, and also sets `door_status` to 0 (no door), 1 (standard 2-sided door) or 2 (a 1-sided door, effectively blocking the exit). In all cases except the boring one (`NULL`, no door), the text `exit_title` is set to a suitable element for use as a tool-tip inside an HTML `title` element.

```
int door_status;
char exit_title[256];

void add_words_utf8(int w1, int w2, char *str) {
    int i, j=0, k, charcode;
    for (i=w1; i<=w2; i++) {
        char *p = lw_array[i].lw_rawtext;
        if [[word i == STROKE]] continue;
        if ((i>w1)
            && ((p[1] != 0) || (is_punctuation(p[0]) == FALSE) || (p[0] == '('))
            && ([[word i == OPENBRACKET]] == FALSE))
            str[j++] = ' ';
        for (k=0; p[k]; k++) {
            switch(p[k]) {
                case ' ':
                case '\x0a':
                case '\x0d':
                case '\t':
```

```
                case NEWLINE_IN_STRING:
                    str[j++] = ' ';
                    break;
                default:
                    charcode = (int) (((unsigned char *)p)[k]);
                    if (charcode >= 128) {
                        str[j++] = 0xC0 + (charcode >> 6);
                        str[j++] = 0x80 + (charcode & 0x3f);
                    } else str[j++] = charcode;
                    break;
            }
        }
    }
    str[j++] = 0;
}
void add_wo_name(world_object *wo, char *str) {
    if (wo->word_ref1 >= 0)
        add_words_utf8(wo->word_ref1,
            wo->word_ref2, str+strlen(str));
    else {
        sprintf(str+strlen(str), "nameless ");
        add_words_utf8(wo->kind->word_ref1,
            wo->kind->word_ref2, str+strlen(str));
    }
}
void copy_map_dirname(char *string, int dirnum) {
    world_object *wo;
    LOOP_OVER(wo, world_object)
        if (INSTANCE_COUNT(wo, kind_direction) == dirnum)
            print_text_to_string(wo->word_ref1, wo->word_ref2, string);
}
world_object *room_exit(world_object *wo, int dir_num, int write_tooltip) {
    world_object *wo2, *wo3, *door_object = NULL;
    door_status = 0;
    exit_title[0] = 0;
    if (wo == NULL) return NULL;
    if (wo->room_flag == FALSE) return NULL;
    if (dir_num < 0) return NULL;
    wo2 = wo->exits[dir_num];
    if (wo2 == NULL) return NULL;
    wo3 = NULL;
    if (wo2->room_flag) wo3 = wo2;
    if (wo2->door_flag) {
        door_status = 1;
        door_object = wo2;
        if (wo2->map_connection_a == wo)
            wo3 = wo2->map_connection_b;
        if (wo2->map_connection_b == wo)
            wo3 = wo2->map_connection_a;
        if (wo3 == NULL) {
            door_status = 2;
            goto MakeToolTip;
        }
```

```
    }
    if (wo3 != NULL) {
        MakeToolTip:
        if (write_tooltip) {
            sprintf(exit_title, "title=\"");
            copy_map_dirname(exit_title+strlen(exit_title), dir_num);
            if (door_status == 1) {
                sprintf(exit_title+strlen(exit_title), " through ");
                add_wo_name(door_object, exit_title);
            }
            if (door_status == 2) {
                sprintf(exit_title+strlen(exit_title), " exit blocked by ");
                add_wo_name(door_object, exit_title);
            } else {
                sprintf(exit_title+strlen(exit_title), " to ");
                add_wo_name(wo3, exit_title);
            }
            sprintf(exit_title+strlen(exit_title), "\"");
        }
    }
    return wo3;
}
```

§3. **Phase I: Positioning the rooms in space.** We assign (x, y, z) coordinates to each room, and calculate the extremal values of each of x, y, z. This is potentially a research-level problem in graph theory or aesthetics, but not the simplistic way we're going to use it isn't.

The first task is to partition the map into components, that is, equivalence classes under the relation $R \sim S$ if either an exit leads from R to S or vice versa. For these purposes, in and out do not count.

```
void set_component_of(world_object *wo, map_component *mc) {
    int i;
    wo->component = mc;
    for (i=0; i<no_directions; i++) {
        if (direction_is_along_lattice(i) == FALSE) continue;
        world_object *to = room_exit(wo, i, FALSE);
        if ((to != NULL) && (to->component != mc)) set_component_of(to, mc);
    }
}
map_component *get_component_of(world_object *wo) {
    world_object *wo2;
    map_component *mc;
    int i, changed;
    if (wo->component != NULL) return wo->component;
    mc = CREATE(map_component);
    mc->extreme_x0 = 0;
    mc->extreme_x1 = 0;
    mc->extreme_y0 = 0;
    mc->extreme_y1 = 0;
    mc->extreme_z0 = 0;
    mc->extreme_z1 = 0;
    mc->x_offset = 0;
```

```
    mc->y_offset = 0;
    mc->z_offset = 0;
    wo->component = mc; changed = 1; mc->size = 1;
    while (changed > 0) {
        changed = 0;
        LOOP_OVER(wo2, world_object) if (wo2->room_flag) {
            if (wo2->component == mc) {
                if ((wo2->lock_exit_to) && (wo2->lock_exit_to->component != mc)) {
                    changed++;
                    wo2->lock_exit_to->component = mc;
                    mc->size++;
                }
                for (i=0; i<no_directions; i++)
                    if (direction_is_along_lattice(i))
                        if ((room_exit(wo2, i, FALSE) != NULL)
                            && (room_exit(wo2, i, FALSE)->component != mc)) {
                            changed++;
                            room_exit(wo2, i, FALSE)->component = mc;
                            mc->size++;
                        }
            }
            if (wo2->component == NULL) {
                if ((wo2->lock_exit_to) && (wo2->lock_exit_to->component == mc)) {
                    changed++;
                    wo2->component = mc;
                    mc->size++;
                }
                for (i=0; i<no_directions; i++)
                    if (direction_is_along_lattice(i))
                        if ((room_exit(wo2, i, FALSE) != NULL)
                            && (room_exit(wo2, i, FALSE)->component == mc)) {
                            changed++;
                            wo2->component = mc;
                            mc->size++;
                        }
            }
        }
    }
    return mc;
}

void establish_components(void) {
    world_object *wo;
    if (start_room == NULL) internal_error("No start room when mapping");
    get_component_of(start_room);
    LOOP_OVER(wo, world_object) if (wo->room_flag) get_component_of(wo);
}
```

§**4.** In establishing the layout, we will score penalty points for each exit which appears wrongly laid out. The score is based primarily on the angular divergence in direction between the known exit direction and the grid direction, and only secondarily on distance anomalies, because when the user typed "the Field is east of the River" no particular distance was implied. In the following calculation of the penalty associated with an exit, we can also set the `move_flag` to have a movement carried out so that the destination is in the "right" place for this exit: but we remember what we did in order that we can undo if necessary.

```
world_object *saved_to; int saved_grid_x, saved_grid_y, saved_grid_z;
void undo_room_move(void) {
    saved_to->grid_x = saved_grid_x;
    saved_to->grid_y = saved_grid_y;
    saved_to->grid_z = saved_grid_z;
}
int locking_mode_engaged = 0;
int exit_penalty(world_object *wo, int exit, int move_flag) {
    int dx, dy, dz, exit_x, exit_y, exit_z, penalty = 0; char *clue;
    float distance, ux, uy, uz;
    world_object *to = room_exit(wo, exit, FALSE);
    if (to == NULL) return 0;
    if (to->room_flag == FALSE) return 0;
    if (locking_mode_engaged > 0) {
        if ((wo->lock_exit_to == to) || (to->lock_exit_to == wo)) return 0;
        if (locking_mode_engaged == 2) {
            world_object *wo2;
            if (to->lock_exit_to) return 0;
            LOOP_OVER(wo2, world_object)
                if (wo2->lock_exit_to == to) return 0;
        }
    }
    find_spatial_offsets(exit, &exit_x, &exit_y, &exit_z, &clue);
    if ((exit_x == 0) && (exit_y == 0) && (exit_z == 0)) return 0;
    dx = to->grid_x - wo->grid_x;
    dy = to->grid_y - wo->grid_y;
    dz = to->grid_z - wo->grid_z;
    if ((dx == 0) && (dy == 0) && (dz == 0)) {
        if (wo == to) return 0;
        penalty = 10000;
    }
    if ((dx == exit_x) && (dy == exit_y) && (dz == exit_z)) return 0;
    if (penalty == 0) {
        distance = sqrt(dx*dx + dy*dy + dz*dz);
        ux = dx/distance; uy = dy/distance; uz = dz/distance;
        penalty += (int)
        (
        100*((exit_x-ux)*(exit_x-ux) + (exit_y-uy)*(exit_y-uy)
            + (exit_z-uz)*(exit_z-uz)) +
        (exit_x-dx)*(exit_x-dx) + (exit_y-dy)*(exit_y-dy) +
        (exit_z-dz)*(exit_z-dz)
        );
    }
    if ((move_flag) && (penalty > 0)) {
        saved_to = to;
```

```
        saved_grid_x = to->grid_x;
        saved_grid_y = to->grid_y;
        saved_grid_z = to->grid_z;
        to->grid_x = wo->grid_x + exit_x;
        to->grid_y = wo->grid_y + exit_y;
        to->grid_z = wo->grid_z + exit_z;
        LOGIF(SPATIAL_MAP, "Moving $O to offset %d %d %d from $O\n",
            to, exit_x, exit_y, exit_z, wo);
    }
    return penalty;
}
```

§**5.** The penalty for a room is the sum of its exit penalties; for a component, it's the sum of its room penalties. We also calculate the smallest squarely oriented cuboid which contains the component.

```
int room_penalty(world_object *wo, int move_flag) {
    int i, total_penalty = 0;
    for (i=0; i<no_directions; i++)
        if (direction_is_along_lattice(i)) {
            int badness = exit_penalty(wo, i, move_flag);
            total_penalty += badness;
        }
    return total_penalty;
}
int component_penalty(map_component *mc, int move_flag) {
    world_object *wo;
    mc->accuracy = 0;
    LOOP_OVER(wo, world_object) {
        if (mc != wo->component) continue;
        mc->accuracy += room_penalty(wo, move_flag);
        if (wo->grid_x < mc->extreme_x0) mc->extreme_x0 = wo->grid_x;
        if (wo->grid_x > mc->extreme_x1) mc->extreme_x1 = wo->grid_x;
        if (wo->grid_y < mc->extreme_y0) mc->extreme_y0 = wo->grid_y;
        if (wo->grid_y > mc->extreme_y1) mc->extreme_y1 = wo->grid_y;
        if (wo->grid_z < mc->extreme_z0) mc->extreme_z0 = wo->grid_z;
        if (wo->grid_z > mc->extreme_z1) mc->extreme_z1 = wo->grid_z;
    }
    return mc->accuracy;
}
```

§**6.** Locking.

```
int move_anything_locked_to(world_object *wo) {
    int chain = 0, exit_x, exit_y, exit_z; char *clue;
    while (wo->lock_exit_to) {
        world_object *wo2 = wo->lock_exit_to;
        wo->grid_x = wo2->grid_x;
        wo->grid_y = wo2->grid_y;
        wo->grid_z = wo2->grid_z;
        find_spatial_offsets(wo->lock_exit, &exit_x, &exit_y, &exit_z, &clue);
        wo->grid_x += exit_x;
        wo->grid_y += exit_y;
        wo = wo2;
        chain++;
        if (chain >= 1000) break;
    }
    return chain;
}
void lock_positions_in_component(map_component *mc) {
    world_object *wo;
    int changes = 0;
    LOOP_OVER(wo, world_object) {
        if (mc != wo->component) continue;
        changes += move_anything_locked_to(wo);
    }
    LOGIF(SPATIAL_MAP,
        "\nComponent locking produced aggregate chain length %d.\n",
        changes);
    LOOP_OVER(wo, world_object) {
        if (mc != wo->component) continue;
        LOGIF(SPATIAL_MAP, "$O is now at (%d,%d,%d)\n",
            wo, wo->grid_x, wo->grid_y, wo->grid_z);
    }
}
```

§**7.** In each component, we repeatedly run an algorithm which searches all exits, counting those which lead to rooms not at the coordinates you would expect. (In the sense that you'd expect the room north from $(0, 0, 0)$ to be at $(0, 1, 0)$, and so on.) For each such exit, the destination room is moved so that it is at the expected coordinates.

If we ran this algorithm indefinitely it would very likely lock up, continually moving rooms back and forth but never solving the problem. So we stop once passes through the whole component are no longer improving the situation, in the sense of reducing the number of "bad" exits.

Imagining the exit penalties as being heat to be dissipated, we speak of this process as a rapid cooling:

```
void cool_component(map_component *mc) {
    int last_off = 1000000, next_off = 0;
    while (TRUE) {
        next_off = component_penalty(mc, TRUE);
        mc->accuracy = next_off;
        if (next_off >= last_off) return;
        last_off = next_off;
```

```
    }
}
```

§**8.** This produces surprisingly adequate results and for many maps (those conformable to cubical grids with symmetrical unit length exits) will be correct. It is guaranteed to terminate, but the final step might increase the penalty. We finish by seeing if small local changes can improve things (looking only one move ahead), and here the penalty is absolutely required to reduce. This we imagine as diffusing heat, hence the name of:

```
void diffuse_component(map_component *mc) {
    world_object *wo;
    int f = TRUE, penalty = mc->accuracy;
    LOGIF(SPATIAL_MAP,
        "\nDiffusion time again, sports fans, and the penalty is %d.\n",
        penalty);
    while (f) {
        f = FALSE;
        LOOP_OVER(wo, world_object) {
            int i;
            if (mc != wo->component) continue;
            LOGIF(SPATIAL_MAP, "Attempting to diffuse $O\n", wo);
            for (i=0; i<no_directions; i++)
                if (direction_is_along_lattice(i))
                    if (exit_penalty(wo, i, TRUE)) {
                        LOGIF(SPATIAL_MAP,
                            "Attempting to diffuse along exit %d from $O\n", i, wo);
                        if ((component_penalty(mc, FALSE)) >= penalty) {
                            undo_room_move();
                            LOGIF(SPATIAL_MAP, "No, that just made things no better\n");
                        } else {
                            f = TRUE;
                            penalty = component_penalty(mc, FALSE);
                            LOGIF(SPATIAL_MAP, "Well, that dropped the penalty to %d\n",
                                penalty);
                        }
                    }
        }
    }
    component_penalty(mc, FALSE);
}
void desperation_moves(map_component *mc) {
    world_object *wo, *wo2;
    AndTryAgain:
    LOOP_OVER(wo, world_object) if (mc == wo->component) {
        LOOP_OVER(wo2, world_object) if ((wo != wo2) && (mc == wo2->component)) {
            if ((wo->grid_x == wo2->grid_x) &&
                (wo->grid_y == wo2->grid_y) &&
                (wo->grid_z == wo2->grid_z)) {
                wo->grid_x++;
                LOGIF(SPATIAL_MAP,
                    "Avoiding collision with $O by shifting $O to the east\n",
                    wo2, wo);
```

```
                goto AndTryAgain;
            }
        }
    }
}
```

§**9.** Having cooled and diffused each component, we now treat them as rigid bodies, but still have to establish their spatial relationship to each other. We do this by laying them out along the x axis so that they can be read left to right in the final HTML display, with the start room always being in the leftmost component: indeed, the start room is always at $(0, 0, 0)$. We ensure that the components do not overlap by the crude method of making their bounding cuboids disjoint, even though this will often mean that there is wasted space on the page. (Thus we do not, for instance, use the trick adopted by the British Ordnance Survey in mapping the outlying island of St Kilda on an inset square of what would otherwise be empty ocean on OS18 *Sound of Harris*, despite its being separated by about 60km from the position shown.)

Finally we calculate the extremal values of x, y and z.

```
int extreme_x0 = 0, extreme_x1 = 0,
    extreme_y0 = 0, extreme_y1 = 0,
    extreme_z0 = 0, extreme_z1 = 0;

int establish_grid_positions(void) {
    map_component *mc, *previous_mc;
    world_object *wo;
    int total_accuracy = 0;

    establish_components();

    LOOP_OVER(mc, map_component) {
        for (locking_mode_engaged=0; locking_mode_engaged<=2; locking_mode_engaged++) {
            lock_positions_in_component(mc);
            cool_component(mc);
        }
        lock_positions_in_component(mc);
        diffuse_component(mc);
        lock_positions_in_component(mc);
        desperation_moves(mc);
        total_accuracy += mc->accuracy;
    }

    previous_mc = NULL;
    LOOP_OVER(mc, map_component) {
        if (previous_mc == NULL) {
            mc->x_offset = 0;
            mc->y_offset = 0;
            mc->z_offset = 0;
        } else {
            mc->x_offset =
                previous_mc->x_offset +
                    previous_mc->extreme_x1 - mc->extreme_x0 + 1;
            mc->y_offset = 0;
            mc->z_offset = 0;
        }
        previous_mc = mc;
    }
```

```
    LOOP_OVER(wo, world_object) if (wo->room_flag) {
        wo->grid_x += wo->component->x_offset;
        wo->grid_y += wo->component->y_offset;
        wo->grid_z += wo->component->z_offset;
        if (wo->grid_x < extreme_x0) extreme_x0 = wo->grid_x;
        if (wo->grid_x > extreme_x1) extreme_x1 = wo->grid_x;
        if (wo->grid_y < extreme_y0) extreme_y0 = wo->grid_y;
        if (wo->grid_y > extreme_y1) extreme_y1 = wo->grid_y;
        if (wo->grid_z < extreme_z0) extreme_z0 = wo->grid_z;
        if (wo->grid_z > extreme_z1) extreme_z1 = wo->grid_z;
    }
    LOOP_OVER(mc, map_component) {
        LOGIF(SPATIAL_MAP, "Map component %d: size %d, "
            "extent (%d...%d, %d...%d, %d...%d), offset (%d,%d,%d)\n",
            mc->allocation_id,
            mc->size,
            mc->extreme_x0, mc->extreme_x1,
            mc->extreme_y0, mc->extreme_y1,
            mc->extreme_z0, mc->extreme_z1,
            mc->x_offset, mc->y_offset, mc->z_offset);
    }
    return total_accuracy;
}
```

§**10. Phase II: Building the grids.** Two three-dimensional arrays are used to store the final spatial representation of the map. Because these are dynamically allocated, we access them by calculating our own array indices. The `map_grid` tells us which world object can be found at (x, y, z), while the `icon_grid` is 25 times larger since it splits each room cell into a 5 by 5 subgrid of icons. Bitmaps stored in the 16 icon cells around the perimeter of the 5 by 5 subgrid tell us which exits to mark (and since we map only 12 kinds of exit, this means that four of them are unused). The central 3 by 3 part of the subgrid is not used: nothing can be plotted there since that's where the room icon goes, which is made not with an image tag but using the HTML table routine below. We will often use the wasteful coordinate system (x, y, z, i_1, i_2) to mean the icon at (i_1, i_2) (with $0 \le i_1, i_2 \le 4$) associated with the room cell at (x, y, z).

The extremal values are the largest and smallest coordinates found anywhere in the model world, and are inclusive. Thus if all six continue to hold the value zero, there can only be one room in the game.

```
world_object **map_grid;
int *icon_grid, *exit_grid;

int posn_3d(int x, int y, int z) {
    return (x-extreme_x0)
        + (y-extreme_y0)*(extreme_x1-extreme_x0+1)
        + (z-extreme_z0)*(extreme_x1-extreme_x0+1)*(extreme_y1-extreme_y0+1);
}

int icon_posn_3d(int x, int y, int z, int i1, int i2) {
    return 25*(posn_3d(x,y,z)) + 5*i1 + i2;
}

int calculate_map_grid(void) {
    world_object *wo;
    int x, overlie = 0;
```

```
    int size_needed =
        (extreme_x1-extreme_x0+1)*
        (extreme_y1-extreme_y0+1)*
        (extreme_z1-extreme_z0+1);
    map_grid = (world_object **)
        (I7_calloc(size_needed, sizeof(world_object *), MAP_INDEX_MREASON));
    for (x=0; x<size_needed; x++) map_grid[x] = NULL;
    icon_grid = (int *)
        (I7_calloc(25*size_needed, sizeof(int), MAP_INDEX_MREASON));
    for (x=0; x<25*size_needed; x++) icon_grid[x] = 0;
    exit_grid = (int *)
        (I7_calloc(25*size_needed, sizeof(int), MAP_INDEX_MREASON));
    for (x=0; x<25*size_needed; x++) exit_grid[x] = -1;
    LOOP_OVER(wo, world_object) if (wo->room_flag) {
        int index = posn_3d(wo->grid_x, wo->grid_y, wo->grid_z);
        if (map_grid[index] != NULL) overlie++;
        map_grid[index] = wo;
    }
    return overlie;
}
```

§**11.** We next define constants needed for the icon bitmap. The information we extract from the map exits is recorded in the low four bits as follows:

```
define EXIT_MAPBIT 1                    An exit leads this way
define DOOR1_MAPBIT 2                   Into a 1-sided door
define DOOR2_MAPBIT 4                   Into a 2-sided door
define ADJACENT_MAPBIT 8                Into the room adjacent in space
define ALIGNED_MAPBIT 16                Into a room in correct direction
define CONNECTIVE_BITMAP 7              AND to test "any connection"
define CORNERDOT_BITMAP 7               Set to 7 to enable corner dots
```

§**12.** The higher bits are used for the nuances which improve the map when several rooms are plotted together. On "pass 2", when individual 1 by 1 room maps are plotted, these nuances distort the picture, so we AND them out.

```
define MEET_MAPBIT 32                   This door should meet the adjacent one
define CROSSDOOR_MAPBIT 64              There's a door on the diagonal athwart
define CROSSDOT_MAPBIT 128              There's a plain exit on ...
define LONGEW_MAPBIT 256
define LONGNS_MAPBIT 512
define OCCUPIED_MAPBIT 1024
define PASS2_BITMAP 15                  AND to remove all plotting nuances
```

§**13.** The following code calculates the low four bits of the icon bitmap grid. Note that the main map grid must already be finished before this stage can even begin.

```
void fill_in_gridsquare(world_object *wo, int exit, int i1, int i2) {
    world_object *o = room_exit(wo, exit, FALSE);
    int bitmap = 0;
    if ((o == NULL) && (door_status < 2)) return;
    if (o != NULL) bitmap |= EXIT_MAPBIT;
    if (door_status == 2) bitmap |= DOOR1_MAPBIT;
    if (door_status == 1) bitmap |= DOOR2_MAPBIT;
    if (o != NULL) {
        int ex, ey, ez, vx, vy, vz, sf; char *clue;
        find_spatial_offsets(exit, &ex, &ey, &ez, &clue);
        if ((ex != 0) || (ey != 0) || (ez != 0)) {
            vx = o->grid_x - wo->grid_x;
            vy = o->grid_y - wo->grid_y;
            vz = o->grid_z - wo->grid_z;
            for (sf=1; sf<10; sf++)
                if ((vx == sf*ex) && (vy == sf*ey) && (vz == sf*ez)) {
                    if (sf == 1) bitmap |= ADJACENT_MAPBIT;
                    else bitmap |= ALIGNED_MAPBIT;
                }
        }
    }
    icon_grid[icon_posn_3d(wo->grid_x, wo->grid_y, wo->grid_z, i1, i2)]
        = bitmap;
    exit_grid[icon_posn_3d(wo->grid_x, wo->grid_y, wo->grid_z, i1, i2)]
        = exit;
}

void calculate_icon_grid(void) {
    world_object *wo;
    LOOP_OVER(wo, world_object)
        if (wo->room_flag) {
            int i;
            for (i=0; i<no_directions; i++) if (direction_is_mappable(i)) {
                int mx, my;
                find_map_offsets(i, &mx, &my);
                fill_in_gridsquare(wo, i, mx, my);
            }
        }
}
```

§**14.** Once the bottom four bits are filled in across the icon bitmap grid, a process called "pair correction" fills in the nuance bits for all adjacent icons representing the same exit. Thus the east side icon of one room may need to be married up with the west side icon of the adjacent room, and so on. The four by four cornices diagonally in between rooms require special care. To plot a northeast exit blocked by a 2-sided door, for instance, requires all four icons to be plotted, but we need to be careful in case the two icons not occupied by the exit are needed for something else (if a northwest exit crossed over it, for instance).

```
void correct_diagonal(int x, int y, int z, int dx) {
    int pos_00, pos_01, pos_10, pos_11, from, to;
    if (dx == 1) {
        pos_00 = icon_posn_3d(x, y, z, 4, 0);
        pos_01 = icon_posn_3d(x, y+1, z, 4, 4);
        pos_10 = icon_posn_3d(x+1, y, z, 0, 0);
        pos_11 = icon_posn_3d(x+1, y+1, z, 0, 4);
    } else {
        pos_10 = icon_posn_3d(x, y, z, 4, 0);
        pos_11 = icon_posn_3d(x, y+1, z, 4, 4);
        pos_00 = icon_posn_3d(x+1, y, z, 0, 0);
        pos_01 = icon_posn_3d(x+1, y+1, z, 0, 4);
    }
    from = icon_grid[pos_00]; to = icon_grid[pos_11];
    if ((from == to) && (from != 0) && (from & ADJACENT_MAPBIT)) {
        if ((icon_grid[pos_01] == 0) && (icon_grid[pos_10] == 0)) {
            if (from & DOOR2_MAPBIT) {
                icon_grid[pos_00] |= MEET_MAPBIT;
                icon_grid[pos_11] |= MEET_MAPBIT;
                icon_grid[pos_01] = CROSSDOOR_MAPBIT;
                icon_grid[pos_10] = CROSSDOOR_MAPBIT;
            }
        } else {
            if ((from & DOOR2_MAPBIT) && ((from & MEET_MAPBIT) == 0)) {
                icon_grid[pos_00]  = DOOR2_MAPBIT;
                icon_grid[pos_11]  = EXIT_MAPBIT;
            }
        }
    }
    from = icon_grid[pos_00]; to = icon_grid[pos_11];
    if ((from & CONNECTIVE_BITMAP) && (from == to) &&
        (icon_grid[pos_01] == 0) && (icon_grid[pos_10] == 0)) {
        icon_grid[pos_01] = CROSSDOT_MAPBIT;
        icon_grid[pos_10] = CROSSDOT_MAPBIT;
    }
}
void correct_pair(int x, int y, int z, int dx, int dy) {
    int from, to;
    int from_i1 = 0, from_i2 = 0, to_i1 = 0, to_i2 = 0;
    if (dx < 0) { correct_pair(x+dx, y, z, -dx, dy); return; }
    if (dy < 0) { correct_pair(x, y+dy, z, dx, -dy); return; }
    if (x < extreme_x0) return;
    if (x > extreme_x1) return;
    if (y < extreme_y0) return;
    if (y > extreme_y1) return;
```

```
    if (x+dx < extreme_x0) return;
    if (x+dx > extreme_x1) return;
    if (y+dy < extreme_y0) return;
    if (y+dy > extreme_y1) return;
    if ((dx == 1) && (dy == 0)) {
        from_i1 = 4; from_i2 = 2;
        to_i1 = 0; to_i2 = 2;
    }
    if ((dx == 0) && (dy == 1)) {
        from_i1 = 2; from_i2 = 0;
        to_i1 = 2; to_i2 = 4;
    }
    if ((dx == 1) && (dy == 1)) {
        from_i1 = 4; from_i2 = 0;
        to_i1 = 0; to_i2 = 4;
    }
    int k = icon_posn_3d(x, y, z, from_i1, from_i2);
    if (k < 0) {
        LOG("IP %d, %d, %d, %d, %d = %d\n", x, y, z, from_i1, from_i2, k);
        internal_error("icon_posn_3d(<0)");
    }
    from = icon_grid[k];
    from = icon_grid[icon_posn_3d(x, y, z, from_i1, from_i2)];
    to = icon_grid[icon_posn_3d(x+dx, y+dy, z, to_i1, to_i2)];
    if (dx+dy == 1) {
        if ((from == to) && (from != 0) && (from & ADJACENT_MAPBIT)) {
            icon_grid[icon_posn_3d(x, y, z, from_i1, from_i2)] |= MEET_MAPBIT;
            icon_grid[icon_posn_3d(x+dx, y+dy, z, to_i1, to_i2)] |= MEET_MAPBIT;
        }
    } else {
        correct_diagonal(x, y, z, 1);
        correct_diagonal(x, y, z, -1);
    }
    if ((dx+dy == 1) && (from != 0) && (from & ALIGNED_MAPBIT)) {
        while (TRUE) {
            x += dx; y += dy;
            if (icon_grid[icon_posn_3d(x, y, z, 2, 2)] == 0) {
                if (dx!=0) {
                    icon_grid[icon_posn_3d(x, y, z, 0, 2)] = EXIT_MAPBIT;
                    icon_grid[icon_posn_3d(x, y, z, 2, 2)] = LONGEW_MAPBIT;
                    icon_grid[icon_posn_3d(x, y, z, 4, 2)] = EXIT_MAPBIT;
                } else {
                    icon_grid[icon_posn_3d(x, y, z, 2, 0)] = EXIT_MAPBIT;
                    icon_grid[icon_posn_3d(x, y, z, 2, 2)] = LONGNS_MAPBIT;
                    icon_grid[icon_posn_3d(x, y, z, 2, 4)] = EXIT_MAPBIT;
                }
            } else break;
        }
    }
}
void pair_correction(void) {
    int x, y, z;
```

```
    world_object *wo;
    LOOP_OVER(wo, world_object)
        icon_grid[icon_posn_3d(wo->grid_x, wo->grid_y,
            wo->grid_z, 2, 2)] = OCCUPIED_MAPBIT;
    LOOP_OVER(wo, world_object) {
        x = wo->grid_x; y = wo->grid_y; z = wo->grid_z;
        correct_pair(x, y, z, -1, -1);
        correct_pair(x, y, z, -1, 0);
        correct_pair(x, y, z, -1, 1);
        correct_pair(x, y, z,  0, -1);
        correct_pair(x, y, z,  0, 1);
        correct_pair(x, y, z,  1, -1);
        correct_pair(x, y, z,  1, 0);
        correct_pair(x, y, z,  1, 1);
    }
}
```

§**15. Phase III: Plotting.** The icon bitmap grid can be plotted with the following routines, which convert the bitmap to a systematic family of icon names and compile the necessary HTML tag. These icons are associated with exits for the most part, so tool-tips are compiled using the `room_exit` routine. (The notional return value of which we ignore here: it's the side-effect of setting the titling text that we want.)

```
void plot_map_icon(char *icon_name, char *tool_tip) {
    INDEX("<img border=0 src=inform:/map_icons/%s.png %s>",
        icon_name, tool_tip);
}
int mapping_pass;
void plot_map_cell(int x, int y, int z, int i1, int i2, int faux_exit) {
    char icon_name[32]; char *tool_tip = "";
    int bitmap = icon_grid[icon_posn_3d(x, y, z, i1, i2)];
    if (mapping_pass == 2)
        bitmap &= PASS2_BITMAP;
    if (bitmap == 0) {
        if ((i1 == 1) || (i1 == 3)) sprintf(icon_name, "blank_ns");
        else {
            if ((i2 == 1) || (i2 == 3)) sprintf(icon_name, "blank_ew");
            else sprintf(icon_name, "blank_square");
        }
    } else {
        char *addendum = "";
        char *clue = "error";
        if (bitmap & DOOR2_MAPBIT) {
            addendum = "_door";
            if (bitmap & MEET_MAPBIT) addendum = "_door_meet";
        }
        if (bitmap & DOOR1_MAPBIT) addendum = "_door_blocked";
        if (bitmap & CROSSDOOR_MAPBIT) addendum = "_corner_door";
        if (bitmap & CROSSDOT_MAPBIT) addendum = "_dot";
        int exit_x, exit_y, exit_z, exit;
        exit = exit_grid[icon_posn_3d(x, y, z, i1, i2)];
        find_spatial_offsets(exit, &exit_x, &exit_y, &exit_z, &clue);
```

```
        if (*clue == 0) find_spatial_offsets(faux_exit, &exit_x, &exit_y, &exit_z, &clue);
        sprintf(icon_name, "%s_arrow%s", clue, addendum);
        room_exit(map_grid[posn_3d(x, y, z)], exit, TRUE);
        tool_tip = exit_title;
    }
    plot_map_icon(icon_name, tool_tip);
}
```

§**16.** The following routine renders the square icons for the rooms themselves, which are bordered and coloured single-cell tables. The start room is coloured differently from the rest; at some stage I hope to make dark rooms, well, darker. The letters inside are up to two capitalised initials, so East Ballroom becomes EB.

```
define ROOM_SQUARE_SIZE 27
define ROOM_INTERIOR_WIDTH 27
define ROOM_INTERIOR_HEIGHT 25
define ROOM_BORDER_SIZE 1
define LINK_SIZE 13
define BLANK_COLOUR "ffffff"
define ROOM_BORDER_COLOUR "000000"

void index_room_square(world_object *wo, int pass) {
    int i, c; char *bc, *ic;
    if ((wo != NULL) && (wo->room_flag)) {
        bc = ROOM_BORDER_COLOUR;
        ic = wo->world_index_colour;
    } else {
        bc = BLANK_COLOUR;
        ic = BLANK_COLOUR;
    }
    if ((pass == 2) && (wo != NULL))
        INDEX("<a name=wo_%d>", wo->allocation_id);
    INDEX("<table border=\"%d\" cellpadding=\"0\" cellspacing=\"0\" ",
        ROOM_BORDER_SIZE);
    INDEX("bordercolor=\"#%s\" width=\"%d\" height=\"%d\"",
        bc, ROOM_INTERIOR_WIDTH, ROOM_INTERIOR_HEIGHT);
    if (wo != NULL) {
        INDEX(" title=\"");
        if (wo->word_ref1 >= 0)
        print_raw_text_to_file(wo->word_ref1, wo->word_ref2, ifl);
        else INDEX("nameless room");
        INDEX("\"");
    }
    INDEX("><tr>");
    INDEX("<td valign=\"middle\" align=\"center\" bgcolor=\"#%s\"><font size=2>",
    ic);
    if ((pass == 1) && (wo != NULL)) {
        INDEX("<a href=#wo_%d><font color=\"#000000\">", wo->allocation_id);
        if (wo == start_room) INDEX("<b>");
    }
    if ((wo != NULL) && (wo->room_flag) && (wo->word_ref1 >= 0)) {
        for (i=wo->word_ref1, c=0; i<=wo->word_ref2; i++, c++) {
            if (c==2) break;
```

```
            char d = *(lw_array[i].lw_rawtext);
            html_char_out(ifl, toupper(d));
        }
    }
    if ((pass == 1) && (wo != NULL)) {
        if (wo == start_room) INDEX("</b>");
        INDEX("</font></a>");
    }
    INDEX("</font></td></tr></table>\n");
}
```

§**17.** Key to the regions:

```
void add_region_key(void) {
    world_object *reg; int rcount = 0;
    LOOP_OVER(reg, world_object)
        if (wo_of_kind(reg, kind_region))
            rcount += add_key_for(reg);
    if (rcount > 0) rcount += add_key_for(NULL);
    if (rcount > 0) INDEX("<hr><p>");
}

int add_key_for(world_object *reg) {
    int count = 0, rcount = 0;
    world_object *wo;
    LOOP_OVER(wo, world_object) {
        if ((wo_of_kind(wo, kind_room)) && (wo->in_region == reg)) {
            if (count++ == 0) {
                begin_plain_html_table(ifl);
                INDEX("<tr><td width=\"40\" valign=\"middle\" align=\"left\">");
                index_room_square(wo, 1);
                INDEX("</td><td valign=\"middle\" align=\"left\">");
                INDEX("<font %s>", DEFAULT_HTML_FONT);
                INDEX("<b>");
                if (reg) print_raw_text_to_file(reg->word_ref1, reg->word_ref2, ifl);
                else INDEX("<i>Not in any region</i>");
                INDEX("</b>: ");
                rcount++;
            } else {
                INDEX(", ");
            }
            print_raw_text_to_file(wo->word_ref1, wo->word_ref2, ifl);
        }
    }
    if (count > 0) { end_html_row(ifl); end_html_table(ifl); INDEX("<p>"); }
    return rcount;
}
```

The function add_region_key is called from 9/inpw.

§**18.** The map is composed of exit icons and room squares, and we have seen routines to make both elements, so it remains only to assemble them in an HTML framework sufficiently inflexible to force them into adjacencies. Our main routine plots a map of a rectangular X-Y area at a given fixed Z coordinate. The pass is 1 for the main mapping, 2 for single-room-only mapping lower down on the index page.

```
int map_tables_begun = 2;
void begin_map_table(int width, int height, int border) {
    int i;
    INDEX("\n");
    for (i=0; i<map_tables_begun; i++) INDEX("  ");
    if (width >= 0) begin_html_table(ifl, NULL, FALSE, border, 0, 0, height, width);
    else begin_html_table(ifl, NULL, FALSE, border, 0, 0, 0, 0);
    map_tables_begun++;
}
void end_map_table(void) {
    int i;
    map_tables_begun--;
    INDEX("\n");
    for (i=0; i<map_tables_begun; i++) INDEX("  ");
    end_html_table(ifl);
    INDEX("\n");
}
void plot_map_level(int x0, int x1, int y0, int y1, int z, int pass) {
    int do_top_row = TRUE, do_bottom_row = TRUE, i1, x, y;
    mapping_pass = pass;
    INDEX("\n\n");
    begin_map_table(-1, -1, 0);
    for (y=y1; y>=y0; y--) {
        do_top_row = FALSE; do_bottom_row = FALSE;
        for (x=x0; x<=x1; x++)
            for (i1=0; i1<=4; i1++) {
                if (icon_grid[icon_posn_3d(x, y, z, i1, 0)] != 0)
                    do_top_row = TRUE;
                if (icon_grid[icon_posn_3d(x, y, z, i1, 4)] != 0)
                    do_bottom_row = TRUE;
            }
        if (y > y0) do_bottom_row = TRUE;
        if (y < y1) do_top_row = TRUE;
        if (do_top_row) {
            INDEX("<tr>");
            for (x=x0; x<=x1; x++) {
                INDEX("<td>");
                begin_map_table(53, 13, 0);
                INDEX("<tr>");
                INDEX("<td>");
                plot_map_cell(x, y, z, 0, 0, 2);
                if (icon_grid[icon_posn_3d(x, y, z, 0, 0)] & CORNERDOT_BITMAP)
                    plot_map_icon("s_dot", ""); else plot_map_icon("ns_spacer", "");
                plot_map_cell(x, y, z, 1, 0, 8);
                plot_map_cell(x, y, z, 2, 0, 0);
                plot_map_cell(x, y, z, 3, 0, -1);
```

```
        if (icon_grid[icon_posn_3d(x, y, z, 4, 0)] & CORNERDOT_BITMAP)
            plot_map_icon("s_dot", ""); else plot_map_icon("ns_spacer", "");
        plot_map_cell(x, y, z, 4, 0, 1);
        INDEX("</td>");
        INDEX("</tr>");
        end_map_table();
        INDEX("</td>");
    }
    INDEX("</tr>");
}
INDEX("<tr>");
for (x=x0; x<=x1; x++) {
    INDEX("<td>");
    begin_map_table(53, 27, 0);
    INDEX("<tr>");
    INDEX("<td>");
    begin_map_table(-1, -1, 0);
    INDEX("<tr>");
    INDEX("<td>");
    if (icon_grid[icon_posn_3d(x, y, z, 0, 0)] & CORNERDOT_BITMAP)
        plot_map_icon("e_dot", ""); else plot_map_icon("ew_spacer", "");
    INDEX("<br>");
    plot_map_cell(x, y, z, 0, 1, 11);
    INDEX("<br>");
    plot_map_cell(x, y, z, 0, 2, 7);
    INDEX("<br>");
    plot_map_cell(x, y, z, 0, 3, -1);
    INDEX("<br>");
    if (icon_grid[icon_posn_3d(x, y, z, 0, 4)] & CORNERDOT_BITMAP)
        plot_map_icon("e_dot", ""); else plot_map_icon("ew_spacer", "");
    INDEX("</td>");
    INDEX("</tr>");
    end_map_table();
    INDEX("</td>");
    INDEX("<td>");
    if (icon_grid[icon_posn_3d(x, y, z, 2, 2)] & LONGEW_MAPBIT)
        plot_map_icon("long_ew", "");
    else {
        if (icon_grid[icon_posn_3d(x, y, z, 2, 2)] & LONGNS_MAPBIT)
            plot_map_icon("long_ns", "");
        else index_room_square(map_grid[posn_3d(x,y,z)], pass);
    }
    INDEX("</td>");
    INDEX("<td>");
    begin_map_table(13, 27, 0);
    INDEX("<tr>");
    INDEX("<td>");
    if (icon_grid[icon_posn_3d(x, y, z, 4, 0)] & CORNERDOT_BITMAP)
        plot_map_icon("w_dot", ""); else plot_map_icon("ew_spacer", "");
    INDEX("<br>");
    plot_map_cell(x, y, z, 4, 1, -1);
    INDEX("<br>");
    plot_map_cell(x, y, z, 4, 2, 6);
```

```
            INDEX("<br>");
            plot_map_cell(x, y, z, 4, 3, 10);
            INDEX("<br>");
            if (icon_grid[icon_posn_3d(x, y, z, 4, 4)] & CORNERDOT_BITMAP)
                plot_map_icon("w_dot", ""); else plot_map_icon("ew_spacer", "");
            INDEX("</td>");
            INDEX("</tr>");
            end_map_table();
            INDEX("</td>");
            INDEX("</tr>");
            end_map_table();
            INDEX("</td>");
        }
        INDEX("</tr>");
        if (do_bottom_row) {
            INDEX("<tr>");
            for (x=x0; x<=x1; x++) {
                INDEX("<td>");
                begin_map_table(53, 13, 0);
                INDEX("<tr>");
                INDEX("<td>");
                plot_map_cell(x, y, z, 0, 4, 5);
                if (icon_grid[icon_posn_3d(x, y, z, 0, 4)] & CORNERDOT_BITMAP)
                    plot_map_icon("n_dot", ""); else plot_map_icon("ns_spacer", "");
                plot_map_cell(x, y, z, 1, 4, -1);
                plot_map_cell(x, y, z, 2, 4, 3);
                plot_map_cell(x, y, z, 3, 4, 9);
                if (icon_grid[icon_posn_3d(x, y, z, 4, 4)] & CORNERDOT_BITMAP)
                    plot_map_icon("n_dot", ""); else plot_map_icon("ns_spacer", "");
                plot_map_cell(x, y, z, 4, 4, 4);
                INDEX("</td>");
                INDEX("</tr>");
                end_map_table();
                INDEX("</td>");
            }
            INDEX("</tr>");
        }
    }
    end_map_table();
}
```

§**19.** Lastly, the main routine which creates the maps. Note that we check to see if there is more than one room in the world: if there isn't, we don't bother with a full map, but we still calculate as far as the icon grid in order to be sure that the little 1 by 1 map for it will be all right.

```
void index_world_map(world_object *wo) {
    if (wo == NULL) {
        int rc = 0, regc = 0;
        int z, z_offset, blank_z, ztop, zbottom, acc, overlie;
        char *some_map_colours[] = {
            "33cc33",                                   green
            "6666cc",                                   blue
            "cc3333",                                   red
            "669900",                                   greenish
            "990066",                                   purple
            "006699",                                   light blue
            "339900",                                   greenish
            "990033",                                   purple
            "003399",                                   light blue
            "22bb22",                                   nearly green
            "5555bb",                                   nearly blue
            "bb2222",                                   nearly red
            "77aa11",                                   off-greenish
            "aa1177",                                   off-purple
            "1177aa",                                   off-light blue
            "44aa11",                                   off–othergreenish
            "1144aa",                                   off-other-light-blue
            "999999"                                    grey default
        };
        traverse_for_map_parameters(1);
        acc = establish_grid_positions();
        blank_z = 1; ztop = extreme_z1-1; zbottom = extreme_z0+1;
        while (blank_z != 1000000) {
            blank_z = 1000000;
            for (z=ztop; z>=zbottom; z--) {
                int occupied = FALSE;
                LOOP_OVER(wo, world_object)
                    if (wo->room_flag)
                        if (wo->grid_z == z) occupied = TRUE;
                if (occupied == FALSE) blank_z = z;
            }
            if (blank_z != 1000000) {
                LOOP_OVER(wo, world_object) if (wo->room_flag) {
                    if (wo->grid_z > blank_z)
                        wo->grid_z--;
                }
            }
            ztop--;
        }
        overlie = calculate_map_grid();
        calculate_icon_grid();
        pair_correction();
        LOOP_OVER(wo, world_object) if (wo->room_flag) rc++;
```

```
if (rc < 2) {
    LOOP_OVER(wo, world_object)
        if (wo->world_index_colour == NULL)
            wo->world_index_colour = "c0c0c0";
    return;
}
LOOP_OVER(wo, world_object) {
    if ((wo->room_flag) && (wo->in_region)) {
        if (wo->in_region->world_index_colour == NULL) {
            int ix = regc++;
            wo->in_region->world_index_colour = some_map_colours[ix % 17];
            /* LOG("Chose region colour %d as %s\n", regc,
                wo->in_region->world_index_colour); */
        }
        wo->world_index_colour = wo->in_region->world_index_colour;
    }
}
LOOP_OVER(wo, world_object)
    if (wo->world_index_colour == NULL)
        wo->world_index_colour = "c0c0c0";
for (z=extreme_z1; z>=extreme_z0; z--) {
    int y_max = -100000, y_min = 100000;
    LOOP_OVER(wo, world_object) if (wo->room_flag) {
        if (wo->grid_z == z) {
            if (wo->grid_y < y_min) y_min = wo->grid_y;
            if (wo->grid_y > y_max) y_max = wo->grid_y;
        }
    }
    if (y_max < y_min) continue;
    LOGIF(SPATIAL_MAP, "Level %d has (%d, %d)\n", z, y_min, y_max);
    switch(extreme_z1 - extreme_z0) {
        case 0: break;
        case 1: if (z == extreme_z0)
            INDEX("<i>Map of lower level</i><p>\n");
            if (z == extreme_z1)
            INDEX("<i>Map of upper level</i><p>\n");
            break;
        default:
            z_offset = z-start_room->grid_z;
            switch(z_offset) {
            case 0: INDEX("<i>Map of starting level</i><p>\n"); break;
            case 1: INDEX("<i>Map of first level up</i><p>\n"); break;
            case -1: INDEX("<i>Map of first level down</i><p>\n"); break;
            case 2: INDEX("<i>Map of second level up</i><p>\n"); break;
            case -2: INDEX("<i>Map of second level down</i><p>\n"); break;
            case 3: INDEX("<i>Map of third level up</i><p>\n"); break;
            case -3: INDEX("<i>Map of third level down</i><p>\n"); break;
            default:
                if (z_offset > 0)
                    INDEX("<i>Map of %dth level up</i><p>\n", z_offset);
                if (z_offset < 0)
                    INDEX("<i>Map of %dth level down</i><p>\n", z_offset);
                break;
```

```
                }
                break;
            }
            plot_map_level(extreme_x0, extreme_x1, y_min, y_max, z, 1);
            INDEX("<p>");
        }
        INDEX("<hr><p>");
    } else {
        INDEX("\n\n");
        begin_plain_html_table(ifl);
        first_html_column(ifl, 0);
        plot_map_level(wo->grid_x, wo->grid_x, wo->grid_y, wo->grid_y, wo->grid_z, 2);
        next_html_column(ifl, 0);
        index_ot(wo, 1, FALSE);
        end_html_row(ifl);
        end_html_table(ifl);
        INDEX("<p>");
    }
}
```

The function index_world_map is called from 9/inpw.

§**20.** And now here we go with the EPS map file.

```
void prepare_map_parameter_scope(map_parameter_scope *scope) {
    int s;
    scope->wider_scope = &global_map_scope;
    for (s=0; s<NO_MAP_PARAMETERS; s++) {
        scope->values[s].specified = FALSE;
        scope->values[s].name = NULL;
        scope->values[s].string_value = NULL;
        scope->values[s].numeric_value = 0;
    }
}
int get_map_variable_index(char *name) {
    int s;
    for (s=0; s<NO_MAP_PARAMETERS; s++)
        if (strcmp(name, global_map_scope.values[s].name) == 0)
            return s;
    LOG("Tried to look up <%s>\n", name);
    internal_error("looked up non-existent map variable");
    return 0;
}
int get_map_variable_index_forgivingly(char *name) {
    int s;
    for (s=0; s<NO_MAP_PARAMETERS; s++)
        if ((global_map_scope.values[s].name) &&
            (strcmp(name, global_map_scope.values[s].name) == 0))
            return s;
    return -1;
}
char *get_string_mp(char *name, map_parameter_scope *scope) {
    int s = get_map_variable_index(name);
```

```
    char *p;
    if (scope == NULL) scope = &global_map_scope;
    while (scope->values[s].specified == FALSE) {
        scope = scope->wider_scope;
        if (scope == NULL)
            internal_error("scope exhausted in looking up map parameter");
    }
    p = scope->values[s].string_value;
    if (strcmp(p, "<font>") == 0) return get_string_mp("font", NULL);
    return p;
}

void put_string_mp(char *name, map_parameter_scope *scope, char *val) {
    int s = get_map_variable_index(name);
    if (scope == NULL) scope = &global_map_scope;
    scope->values[s].specified = TRUE;
    scope->values[s].string_value = val;
}

int get_int_mp(char *name, map_parameter_scope *scope) {
    int s = get_map_variable_index(name);
    if (scope == NULL) scope = &global_map_scope;
    while (scope->values[s].specified == FALSE) {
        scope = scope->wider_scope;
        if (scope == NULL)
            internal_error("scope exhausted in looking up map parameter");
    }
    return scope->values[s].numeric_value;
}

void put_int_mp(char *name, map_parameter_scope *scope, int val) {
    int s = get_map_variable_index(name);
    if (scope == NULL) scope = &global_map_scope;
    scope->values[s].specified = TRUE;
    scope->values[s].numeric_value = val;
}

typedef struct colour_translation {
    char *chip_name;
    char *html_colour;
} colour_translation;

colour_translation table_of_translations[] = {
    { "Alice Blue", "F0F8FF" },
    { "Antique White", "FAEBD7" },
    { "Aqua", "00FFFF" },
    { "Aquamarine", "7FFFD4" },
    { "Azure", "F0FFFF" },
    { "Beige", "F5F5DC" },
    { "Bisque", "FFE4C4" },
    { "Black", "000000" },
    { "Blanched Almond", "FFEBCD" },
    { "Blue", "0000FF" },
    { "Blue Violet", "8A2BE2" },
    { "Brown", "A52A2A" },
    { "Burly Wood", "DEB887" },
    { "Cadet Blue", "5F9EA0" },
```

```
{ "Chartreuse", "7FFF00" },
{ "Chocolate", "D2691E" },
{ "Coral", "FF7F50" },
{ "Cornflower Blue", "6495ED" },
{ "Cornsilk", "FFF8DC" },
{ "Crimson", "DC143C" },
{ "Cyan", "00FFFF" },
{ "Dark Blue", "00008B" },
{ "Dark Cyan", "008B8B" },
{ "Dark Golden Rod", "B8860B" },
{ "Dark Gray", "A9A9A9" },
{ "Dark Green", "006400" },
{ "Dark Khaki", "BDB76B" },
{ "Dark Magenta", "8B008B" },
{ "Dark Olive Green", "556B2F" },
{ "Dark Orange", "FF8C00" },
{ "Dark Orchid", "9932CC" },
{ "Dark Red", "8B0000" },
{ "Dark Salmon", "E9967A" },
{ "Dark Sea Green", "8FBC8F" },
{ "Dark Slate Blue", "483D8B" },
{ "Dark Slate Gray", "2F4F4F" },
{ "Dark Turquoise", "00CED1" },
{ "Dark Violet", "9400D3" },
{ "Deep Pink", "FF1493" },
{ "Deep Sky Blue", "00BFFF" },
{ "Dim Gray", "696969" },
{ "Dodger Blue", "1E90FF" },
{ "Feldspar", "D19275" },
{ "Fire Brick", "B22222" },
{ "Floral White", "FFFAF0" },
{ "Forest Green", "228B22" },
{ "Fuchsia", "FF00FF" },
{ "Gainsboro", "DCDCDC" },
{ "Ghost White", "F8F8FF" },
{ "Gold", "FFD700" },
{ "Golden Rod", "DAA520" },
{ "Gray", "808080" },
{ "Green", "008000" },
{ "Green Yellow", "ADFF2F" },
{ "Honey Dew", "F0FFF0" },
{ "Hot Pink", "FF69B4" },
{ "Indian Red", "CD5C5C" },
{ "Indigo", "4B0082" },
{ "Ivory", "FFFFF0" },
{ "Khaki", "F0E68C" },
{ "Lavender", "E6E6FA" },
{ "Lavender Blush", "FFF0F5" },
{ "Lawn Green", "7CFC00" },
{ "Lemon Chiffon", "FFFACD" },
{ "Light Blue", "ADD8E6" },
{ "Light Coral", "F08080" },
{ "Light Cyan", "E0FFFF" },
```

```
{ "Light Golden Rod Yellow", "FAFAD2" },
{ "Light Grey", "D3D3D3" },
{ "Light Green", "90EE90" },
{ "Light Pink", "FFB6C1" },
{ "Light Salmon", "FFA07A" },
{ "Light Sea Green", "20B2AA" },
{ "Light Sky Blue", "87CEFA" },
{ "Light Slate Blue", "8470FF" },
{ "Light Slate Gray", "778899" },
{ "Light Steel Blue", "B0C4DE" },
{ "Light Yellow", "FFFFE0" },
{ "Lime", "00FF00" },
{ "Lime Green", "32CD32" },
{ "Linen", "FAF0E6" },
{ "Magenta", "FF00FF" },
{ "Maroon", "800000" },
{ "Medium Aquamarine", "66CDAA" },
{ "Medium Blue", "0000CD" },
{ "Medium Orchid", "BA55D3" },
{ "Medium Purple", "9370D8" },
{ "Medium Sea Green", "3CB371" },
{ "Medium Slate Blue", "7B68EE" },
{ "Medium Spring Green", "00FA9A" },
{ "Medium Turquoise", "48D1CC" },
{ "Medium Violet Red", "C71585" },
{ "Midnight Blue", "191970" },
{ "Mint Cream", "F5FFFA" },
{ "Misty Rose", "FFE4E1" },
{ "Moccasin", "FFE4B5" },
{ "Navajo White", "FFDEAD" },
{ "Navy", "000080" },
{ "Old Lace", "FDF5E6" },
{ "Olive", "808000" },
{ "Olive Drab", "6B8E23" },
{ "Orange", "FFA500" },
{ "Orange Red", "FF4500" },
{ "Orchid", "DA70D6" },
{ "Pale Golden Rod", "EEE8AA" },
{ "Pale Green", "98FB98" },
{ "Pale Turquoise", "AFEEEE" },
{ "Pale Violet Red", "D87093" },
{ "Papaya Whip", "FFEFD5" },
{ "Peach Puff", "FFDAB9" },
{ "Peru", "CD853F" },
{ "Pink", "FFC0CB" },
{ "Plum", "DDA0DD" },
{ "Powder Blue", "B0E0E6" },
{ "Purple", "800080" },
{ "Red", "FF0000" },
{ "Rosy Brown", "BC8F8F" },
{ "Royal Blue", "4169E1" },
{ "Saddle Brown", "8B4513" },
{ "Salmon", "FA8072" },
```

```
    { "Sandy Brown", "F4A460" },
    { "Sea Green", "2E8B57" },
    { "Sea Shell", "FFF5EE" },
    { "Sienna", "A0522D" },
    { "Silver", "C0C0C0" },
    { "Sky Blue", "87CEEB" },
    { "Slate Blue", "6A5ACD" },
    { "Slate Gray", "708090" },
    { "Snow", "FFFAFA" },
    { "Spring Green", "00FF7F" },
    { "Steel Blue", "4682B4" },
    { "Tan", "D2B48C" },
    { "Teal", "008080" },
    { "Thistle", "D8BFD8" },
    { "Tomato", "FF6347" },
    { "Turquoise", "40E0D0" },
    { "Violet", "EE82EE" },
    { "Violet Red", "D02090" },
    { "Wheat", "F5DEB3" },
    { "White", "FFFFFF" },
    { "White Smoke", "F5F5F5" },
    { "Yellow", "FFFF00" },
    { "Yellow Green", "9ACD32" },
    { "", "" }
};

char *translate_colour_name(char *original) {
    int j;
    for (j=0; strcmp(table_of_translations[j].chip_name, ""); j++)
        if (strcmp(table_of_translations[j].chip_name, original) == 0)
            return table_of_translations[j].html_colour;
    return NULL;
}

int translate_offset(char *original) {
    char offs[32];
    int j, k=-1, xbit, ybit;
    if (strlen(original) >= 30) return 100000000;
    strcpy(offs, original);
    for (j=0; offs[j]; j++) {
        int okay = FALSE;
        if (isdigit(offs[j])) okay = TRUE;
        if (offs[j] == '&') { okay = TRUE; offs[j] = 0; k=j+1; }
        if (offs[j] == '-') okay = TRUE;
        if (okay == FALSE) return 100000000;
    }
    if (k<0) return 100000000;
    xbit = atoi(offs); ybit = atoi(offs+k);
    return xbit+ybit*10000;
}

int changed_global_room_colour = FALSE;

map_parameter_scope *scope_of_chunk_of_level(int ln);

int wo_in_region(world_object *wo, world_object *reg) {
    if ((wo == NULL) || (reg == NULL)) return FALSE;
```

```
    if (wo->in_region == reg) return TRUE;
    return wo_in_region(wo->in_region, reg);
}
void do_map_put(char *v, map_parameter_scope *scope, world_object *scope_wo,
    char *put_string, int put_integer) {
    if (scope_wo) {
        if (scope_wo->room_flag) scope = &(scope_wo->local_map_parameters);
        else {
            if (scope_wo->kind_flag) {
                world_object *wo2;
                LOOP_OVER(wo2, world_object)
                    if ((wo2->room_flag) && (wo_of_kind(wo2, scope_wo)))
                        do_map_put(v, NULL, wo2, put_string, put_integer);
                return;
            }
            if (wo_of_kind(scope_wo, kind_region)) {
                world_object *wo2;
                LOGIF(SPATIAL_MAP,
                    "Making map vars change over region $O\n", scope_wo);
                LOOP_OVER(wo2, world_object)
                    if ((wo2->room_flag) && (wo_in_region(wo2, scope_wo))) {
                        LOGIF(SPATIAL_MAP, "Affects $O\n", wo2);
                        do_map_put(v, NULL, wo2, put_string, put_integer);
                    }
                return;
            }
            internal_error("no scope object in map variable put");
        }
    }
    if (put_string) put_string_mp(v, scope, put_string);
    else put_int_mp(v, scope, put_integer);
}
```

The function prepare_map_parameter_scope is called from 9/wo.
The structure colour_translation is private to this section.

§21.

define MAX_RUBRICS 100

```
int no_rubrics = 0;
typedef struct rubric_holder {
    char *annotation;
    int point_size;
    char *font;
    char *colour;
    int at_offset;
    struct world_object *offset_from;
} rubric_holder;
rubric_holder the_rubrics[MAX_RUBRICS];
int write_eps_to_desktop = FALSE;
int get_direction_number(int wn) {
    if [[word wn == north]] return 0;
```

```
    if [[word wn == northeast]] return 1;
    if [[word wn == northwest]] return 2;
    if [[word wn == south]] return 3;
    if [[word wn == southeast]] return 4;
    if [[word wn == southwest]] return 5;
    if [[word wn == east]] return 6;
    if [[word wn == west]] return 7;
    return -1;
}
void new_map_parameter(int pass, parse_node *p) {
    int w1 = p->word_ref1, w2 = p->word_ref2, i, nw1, nw2, ow1, ow2, dw1, dw2;
    if (is_list_divided(w1, w2, LOOK_FOR_AND)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        new_map_parameter(pass, new_nounphrase_articled(lw1, lw2));
        new_map_parameter(pass, new_nounphrase_articled(rw1, rw2));
        return;
    }
    ... mapped ¡direction¿ of ...
    if [[w1, w2 == ... mapped ... : i]] {
        if (pass == 1) {
            world_object *dobj;
            nw1 = w1; nw2 = i-1; ow1 = i+1; ow2 = w2;
            if [[ow1, ow2 == as ... --> ow1, ow2]] {
                world_object *wo = parse_world_object(nw1, nw2, FALSE);
                world_object *wo2 = parse_world_object(ow1, ow2, FALSE);
                if ((wo == NULL) || (wo2 == NULL) ||
                    (wo_of_kind(wo, kind_direction) == FALSE) ||
                    (wo_of_kind(wo2, kind_direction) == FALSE))
                    map_problem(_P_(C9MapDirectionClue),
                        p, "You can only say 'Index map with D mapped as E.' "
                        "when D and E are directions.");
                else
                    map_directions_as_if[wo->direction_index] = wo2->direction_index;
                return;
            }
            int j1 = is_word_intermediate(of_V, ow1, ow2);
            int j2 = is_word_intermediate(from_V, ow1, ow2);
            dw1 = -1; dw2 = -1;
            if ((j1 >= 0) && ((j2 == -1) || (j1 < j2)))
                [[ow1, ow2 == ... of ... : j1 --> dw1, dw2 ... ow1, ow2]];
            if ((j2 >= 0) && ((j1 == -1) || (j2 < j1)))
                [[ow1, ow2 == ... from ... : j1 --> dw1, dw2 ... ow1, ow2]];
            if (dw1 >= 0) {
                dobj = parse_world_object(dw1, dw2, FALSE);
                if ((dobj == NULL) || (wo_of_kind(dobj, kind_direction) == FALSE)) {
                    LOG("DW: $W; OW: $W\n", dw1, dw2, ow1, ow2);
                    map_problem(_P_(C9MapPlacementDirection),
                        p, "The direction given as a hint for map placement wasn't "
                        "one that I know of.");
                    return;
                }
            } else if [[ow1, ow2 == above ... --> ow1, ow2]] dobj = wo_up_direction;
            else if [[ow1, ow2 == below ... --> ow1, ow2]] dobj = wo_down_direction;
```

```
            else {
                map_problem(_P_(C9MapPlacement),
                    p, "The map placement hint should either have the form 'Index map with
X "
                    "mapped east of Y' or 'Index map with X mapped above/below Y'.");
                return;
            }
            world_object *wo = parse_world_object(nw1, nw2, FALSE);
            world_object *wo2 = parse_world_object(ow1, ow2, FALSE);
            int exit = dobj->direction_index, ex, ey, ez; char *clue;
            if ((wo == NULL) || (wo->room_flag == FALSE)) {
                map_problem(_P_(C9MapFromNonRoom),
                    p, "The first-named thing must be a room "
                    "(beware ambiguities!).");
                return;
            }
            if ((wo2 == NULL) || (wo2->room_flag == FALSE)) {
                map_problem(_P_(C9MapToNonRoom),
                    p, "The second-named thing must be a room "
                    "(beware ambiguities!).");
                return;
            }
            find_spatial_offsets(exit, &ex, &ey, &ez, &clue);
            if ((ex == 0) && (ey == 0)) {
                map_problem(_P_(C9MapNonLateral),
                    p, "The direction given as a hint for map placement must be "
                    "a lateral direction (not up, down, above, below, inside "
                    "or outside).");
                return;
            }
            LOGIF(SPATIAL_MAP,
                "Mapping clue: put $O next to $O via exit %d\n", wo, wo2, exit);
            wo->lock_exit = exit;
            wo->lock_exit_to = wo2;
        }
        return;
    }
    if [[w1, w2 == eps file]] {
        write_eps_to_desktop = TRUE;
        LOGIF(SPATIAL_MAP, "EPS requested\n");
        return;
    }
    if (pass == 1) return;
    rubric ...
    if ([[w1, w2 == rubric ...]] && (vocab_test_flags(w1+1, TEXT_MC))) {
        dequote_word(w1+1);
        the_rubrics[no_rubrics].annotation = lw_array[w1+1].lw_text;
        the_rubrics[no_rubrics].point_size = 12;
        the_rubrics[no_rubrics].font = "<font>";
        the_rubrics[no_rubrics].colour = "000000";
        the_rubrics[no_rubrics].at_offset = 10001;
        the_rubrics[no_rubrics].offset_from = NULL;
```

```
i = w1+2;
while (i<=w2) {
    LOGIF(SPATIAL_MAP, "Now working on <$W>\n", i, w2);
    if ([[i, w2 == size ...]] &&
        (vocab_test_flags(i+1, NUMBER_MC))) {
        the_rubrics[no_rubrics].point_size =
            vocab_get_literal_number_value(lw_array[i+1].lw_identity);
        i+=2; continue;
    }
    if ([[i, w2 == font ...]] &&
        (vocab_test_flags(i+1, TEXT_MC))) {
        dequote_word(i+1);
        the_rubrics[no_rubrics].font = lw_array[i+1].lw_text;
        i+=2; continue;
    }
    if ([[i, w2 == colour ...]] &&
        (vocab_test_flags(i+1, TEXT_MC))) {
        char *thec;
        dequote_word(i+1);
        thec = translate_colour_name(lw_array[i+1].lw_text);
        if (thec == NULL) {
            map_problem(_P_(C9MapUnknownColour),
                p, "There's no such map colour.");
            return;
        }
        the_rubrics[no_rubrics].colour = thec;
        i+=2; continue;
    }
    if [[i, w2 == at ...]] {
        int theat;
        theat = translate_offset(lw_array[i+1].lw_text);
        if (theat == 100000000) {
            map_problem(_P_(C9MapUnknownOffset),
                p, "There's no such offset.");
            return;
        }
        the_rubrics[no_rubrics].at_offset = theat;
        i+=2;
        if [[i, w2 == from ...]] {
            world_object *wo = parse_world_object(i+1, w2, FALSE);
            i = w2+1;
            if (wo == NULL) {
                map_problem(_P_(C9MapUnknownOffsetBase),
                    p, "There's no such room to be offset from.");
                return;
            }
            the_rubrics[no_rubrics].offset_from = wo;
        }
        continue;
    }
    map_problem(_P_(C9MapBadRubric),
        p, "Unfortunately the details of that rubric seem to be "
        "in error (a lame message, but an accurate one).");
```

```
        return;
    }
    no_rubrics++;
    return;
}
```

... [of ...] set to ...

```
if ([[w1, w2 == ... set ... : i]] && [[i, w2 == set to ...]]) {
    int sw1 = w1, sw2 = i-1, vw1 = i+2, vw2 = w2;
    map_parameter_scope *scope = NULL;
    world_object *scope_wo = NULL;
    int index_of_parameter = -1;
    int i = is_word_intermediate(of_V, sw1, sw2);
    int data_type_wanted = INT_MDT;
    char *i_wanted_a = "";
    int put_integer = 0; char *put_string;
    if (i>=0) {
        int ofw1 = i+1, ofw2 = sw2;
        sw2 = i-1;
        [[ofw1, ofw2 == the ... --> ofw1, ofw2]];
        if [[ofw1, ofw2 == first room]] {
            if (start_room == NULL) return;
            scope = &(start_room->local_map_parameters);
        } else if ([[ofw1, ofw2 == level ###]] &&
            (vocab_test_flags(ofw1+1, NUMBER_MC))) {
            int ln = vocab_get_literal_number_value(lw_array[ofw1+1].lw_identity);
            scope = scope_of_chunk_of_level(ln);
            if (scope == NULL) {
                map_problem(_P_(C9MapLevelMisnamed),
                    p, "Layers of the map must be called 'level N', where "
                    "N is a number, and level 0 is the one which contains "
                    "the first room.");
                return;
            }
        } else {
            world_object *wo = parse_world_object(ofw1, ofw2, FALSE);
            if (wo) LOGIF(SPATIAL_MAP, "Setting for world object $O\n", wo);
            if ((wo) && (wo->kind_flag) && (wo_of_kind(wo, kind_room))) {
                scope_wo = wo;
            } else if ((wo) && (wo_of_kind(wo, kind_region))) {
                scope_wo = wo;
            } else if ((wo == NULL) || (wo->room_flag == FALSE)) {
                map_problem(_P_(C9MapSettingOfUnknown),
                    p, "The parameter has to be 'of' either 'the first room' "
                    "or a specific named room (beware ambiguities!) or "
                    "a level such as 'level 0' (the first room is by "
                    "definition on level 0), or a region, or a kind of room.");
                return;
            }
            scope_wo = wo;
        }
    }
    LOGIF(SPATIAL_MAP, "Setting is <$W>\n", sw1, sw1);
    index_of_parameter = -1;
```

```
if ((sw1 == sw2) &&
    ((i = get_map_variable_index_forgivingly(lw_array[sw1].lw_text))>=0))
    index_of_parameter = i;
if (index_of_parameter == -1) {
    map_problem(_P_(C9MapSettingUnknown),
        p, "The parameter has to be one of the fixed named set given in "
        "the documentation, like 'room-name'. All parameters are one "
        "word, but many are hyphenated. (Also, note that 'colour' has the "
        "Canadian/English spelling, not the American one 'color'.)");
    return;
}
if (vw1 != vw2) {
    map_problem(_P_(C9MapSettingTooLong),
        p, "The value supplied has to be a single item, a number, a word "
        "or some text in double-quotes: this looks too long to be right.");
    return;
}
if ((scope == NULL) && (strcmp(lw_array[sw1].lw_text, "room-colour") == 0))
    changed_global_room_colour = TRUE;
data_type_wanted =
    global_map_scope.values[index_of_parameter].parameter_data_type;
put_integer = 0; put_string = NULL;
switch(data_type_wanted) {
    case INT_MDT:
        if (vocab_test_flags(vw1, NUMBER_MC)) {
            put_integer = vocab_get_literal_number_value(lw_array[vw1].lw_identity);
            goto PutValues;
        }
        i_wanted_a = "an integer";
        break;
    case OFF_MDT: {
        int j, k=-1, xbit, ybit; char offs[32];
        if (strlen(lw_array[vw1].lw_text) < 30) {
            strcpy(offs, lw_array[vw1].lw_text);
            LOGIF(SPATIAL_MAP, "Offset is <%s>\n", offs);
            for (j=0; offs[j]; j++) {
                int okay = FALSE;
                if (isdigit(offs[j])) okay = TRUE;
                if (offs[j] == '&') { okay = TRUE; offs[j] = 0; k=j+1; }
                if (offs[j] == '-') okay = TRUE;
                if (okay == FALSE) goto GoneBad;
            }
            if (k<0) goto GoneBad;
            xbit = atoi(offs); ybit = atoi(offs+k);
            put_integer = xbit+ybit*10000;
            goto PutValues;
        }
        GoneBad: i_wanted_a = "an offset in the form 34&-450";
        break;
    }
    case BOOL_MDT:
        if [[word vw1 == on]] {
```

```
                put_integer = TRUE;
                goto PutValues;
            }
            if [[word vw1 == off]] {
                put_integer = FALSE;
                goto PutValues;
            }
            i_wanted_a = "'on' or 'off'";
            break;
        case TEXT_MDT:
        case COL_MDT:
        case FONT_MDT:
            if (vocab_test_flags(vw1, TEXT_MC)) {
                dequote_word(vw1);
                if (data_type_wanted == COL_MDT) {
                    int j;
                    for (j=0; strcmp(table_of_translations[j].chip_name, ""); j++) {
                        if (strcmp(table_of_translations[j].chip_name,
                            lw_array[vw1].lw_text) == 0) {
                            LOGIF(SPATIAL_MAP, "Translated to colour <%s>\n",
                                table_of_translations[j].html_colour);
                            put_string = table_of_translations[j].html_colour;
                            goto PutValues;
                        }
                    }
                    break;
                }
                put_string = lw_array[vw1].lw_text;
                goto PutValues;
            }
            switch(data_type_wanted) {
                case TEXT_MDT:
                    i_wanted_a = "some text in double-quotes";
                    break;
                case COL_MDT:
                    i_wanted_a = "a colour name in double-quotes";
                    break;
                case FONT_MDT:
                    i_wanted_a = "a font name in double-quotes";
                    break;
            }
            break;
        default: internal_error("Unexpected map parameter data type");
    }
    map_problem_wanted_but(_P_(C9MapSettingTypeFailed),
        p, i_wanted_a, vw1);
    return;
    PutValues:
        do_map_put(lw_array[sw1].lw_text, scope, scope_wo, put_string, put_integer);
    return;
}
map_problem(_P_(C9MapHintUnknown),
    p, "The general form for this is 'Index map with ...' and then a "
```

```
        "list of clues, such as 'the Ballroom mapped east of the Terrace', "
        "or 'room-size of the Ballroom set to 100'.");
    return;
}
```

The structure rubric_holder is private to this section.

§22. Pass 1 for before HTML mapping, pass 2 for before EPS mapping

```
void traverse_for_map_parameters(int pass) {
    parse_node *p, *prevp;
    if (pass == 1) {
        int i;
        for (i=0; i<no_directions; i++)
            map_directions_as_if[i] = i;
    }
    for (prevp=NULL, TREE_START(p); p; prevp=p, TREE_NEXT(p)) {
        if ((pn_get_node_type(p) == SENTENCE_NT)
            && (p->down)
            && (pn_int_annotation(p->down, verb_id_ANNOT) == MAP_PARAMETER_VB)
            && (p->down->next) && (p->down->next->next))
            new_map_parameter(pass,
                new_nounphrase_articled(p->down->next->next->word_ref1,
                p->down->next->next->word_ref2));
    }
}
void plot_text_at(OUTPUT_STREAM, char *text_to_plot, world_object *wo, int abbrev_to,
    char *font, int x, int y, int pointsize, int centre_h, int centre_v) {
    char textual[1024];
    if (text_to_plot) {
        if (strlen(text_to_plot)>1024) internal_error("Too much legend text");
        strcpy(textual, text_to_plot);
    } else if ((wo) && (wo->word_ref1 >= 0)) {
        print_raw_text_to_string(
            wo->word_ref1, wo->word_ref2,
            textual);
    } else return;
    if (abbrev_to > 127) abbrev_to = 127;
    while (strlen(textual) > abbrev_to) {
        int j;
        for (j=strlen(textual)-1; j>=0; j--)
            if (islower(textual[j]) == ' ') goto RemoveOne;
        for (j=strlen(textual)-1; j>=0; j--)
            if ((textual[j] == 'a') || (textual[j] == 'e') ||
                (textual[j] == 'i') || (textual[j] == 'o') ||
                (textual[j] == 'u')) goto RemoveOne;
        for (j=strlen(textual)-1; j>=0; j--)
            if (islower(textual[j])) goto RemoveOne;
        for (j=strlen(textual)-1; j>=0; j--)
            if (isupper(textual[j]) == FALSE) goto RemoveOne;
        textual[abbrev_to] = 0;
        break;
        RemoveOne: ;
```

```
        for (; textual[j]; j++) textual[j] = textual[j+1];
    }
    WRITE("/%s findfont %d scalefont setfont\n", font, pointsize);
    WRITE("newpath (%s)\n", textual);
    if (centre_h) WRITE("dup stringwidth add 2 div %d exch sub %% = X centre-offset\n", x);
    else WRITE("%d %% = X\n", x);
    if (centre_v) WRITE("%d %d 2 div sub %% = Y centre-offset\n", y, pointsize);
    else WRITE("%d %% = Y\n", y);
    WRITE("moveto show\n");
}
int hex_to_int(char hex) {
    switch(hex) {
        case '0': return 0;
        case '1': return 1;
        case '2': return 2;
        case '3': return 3;
        case '4': return 4;
        case '5': return 5;
        case '6': return 6;
        case '7': return 7;
        case '8': return 8;
        case '9': return 9;
        case 'a': case 'A': return 10;
        case 'b': case 'B': return 11;
        case 'c': case 'C': return 12;
        case 'd': case 'D': return 13;
        case 'e': case 'E': return 14;
        case 'f': case 'F': return 15;
        default: internal_error("Improper character in HTML colour");
    }
    return 0;
}
void choose_colour_beam(OUTPUT_STREAM, char hex1, char hex2) {
    int k = hex_to_int(hex1)*16 + hex_to_int(hex2);
    WRITE("%.6f ", ((float) k)/255.0);
}
void choose_colour(OUTPUT_STREAM, char *htmlcolour) {
    if (strlen(htmlcolour) != 6) internal_error("Improper HTML colour");
    choose_colour_beam(OUT, htmlcolour[0], htmlcolour[1]);
    choose_colour_beam(OUT, htmlcolour[2], htmlcolour[3]);
    choose_colour_beam(OUT, htmlcolour[4], htmlcolour[5]);
    WRITE("setrgbcolor %% From HTML colour %s\n", htmlcolour);
}
typedef struct eps_map_chunk {
    int width;
    int actual_height;
    int height;
    char titling[128];
    int titling_point_size;
    int map_level;
    int y_max;
    int y_min;
```

```
    int contains_rooms;
    int contains_titling;
    int eps_origin;
    struct map_parameter_scope map_parameters;
} eps_map_chunk;

int no_eps_map_chunks = 0;
eps_map_chunk the_eps_map_chunks[30];

map_parameter_scope *scope_of_chunk_of_level(int ln) {
    int i;
    for (i=0; i<no_eps_map_chunks; i++)
        if ((the_eps_map_chunks[i].contains_rooms)
            && (the_eps_map_chunks[i].map_level - start_room->grid_z == ln))
                return &(the_eps_map_chunks[i].map_parameters);
    return NULL;
}

void bezier_curve(OUTPUT_STREAM, int stiffness0, int stiffness1,
    int x0, int y0, int exit0, int x1, int y1, int exit1) {
    int cx1, cy1, cx2, cy2, ex, ey, ez; char *clue;
    find_spatial_offsets(exit0, &ex, &ey, &ez, &clue);
    cx1 = x0+ex*stiffness0/100; cy1 = y0+ey*stiffness0/100;
    find_spatial_offsets(exit1, &ex, &ey, &ez, &clue);
    cx2 = x1+ex*stiffness1/100; cy2 = y1+ey*stiffness1/100;
    WRITE("%d %d moveto %% start of Bezier curve\n", x0, y0);
    WRITE("%d %d %d %d %d %d curveto %% control points 1, 2 and end\n",
        cx1, cy1, cx2, cy2, x1, y1);
    WRITE("stroke\n");
}

void faraway_arrow(OUTPUT_STREAM, int boxsize, int exit, int x0, int y0, world_object *wo2)
{
    int ex = 0, ey = 0, ez = 0, scaled = 1; char *clue;
    WRITE("[2 1] 0 setdash %% dashed line for abnormal exit\n");
    WRITE("%d %d moveto %% room centre\n", x0, y0);
    find_spatial_offsets(exit, &ex, &ey, &ez, &clue);
    switch(exit) {
        case 8: ex=2; ey=3; scaled = 2; break;
        case 9: ex=-2; ey=-3; scaled = 2; break;
        case 10: ex=3; ey=2; scaled = 2;  break;
        case 11: ex=-3; ey=-2; scaled = 2; break;
    }
    WRITE("%d %d rlineto %% arrow out\n",
        ex*boxsize/scaled, ey*boxsize/scaled);
    WRITE("stroke\n");
    WRITE("[] 0 setdash %% back to normal solid lines\n");
    if (wo2->word_ref1 >= 0)
    plot_text_at(OUT, NULL, wo2,
        get_int_mp("annotation-length", NULL),
        get_string_mp("annotation-font", NULL),
        x0+ex*boxsize*6/scaled/5, y0+ey*boxsize*6/scaled/5,
        get_int_mp("annotation-size", NULL),
        TRUE, TRUE);
}

void circular_path(OUTPUT_STREAM, int x0, int y0, int radius) {
```

```
    WRITE("%d %d moveto %% rightmost point\n", x0+radius, y0);
    WRITE("%d %d %d %d %d arc %% full circle traced anticlockwise\n",
        x0, y0, radius, 0, 360);
    WRITE("closepath\n");
}

void rectangular_path(OUTPUT_STREAM, int x0, int y0, int x1, int y1) {
    WRITE("%d %d moveto %% bottom left corner\n", x0, y0);
    WRITE("%d %d lineto %% bottom side\n", x1, y0);
    WRITE("%d %d lineto %% right side\n", x1, y1);
    WRITE("%d %d lineto %% top side\n", x0, y1);
    WRITE("closepath\n");
}

void make_room_path(OUTPUT_STREAM, int bx, int by, int boxsize, char *shape) {
    if (strcmp(shape, "square") == 0) {
        rectangular_path(OUT, bx-boxsize, by-boxsize, bx+boxsize, by+boxsize);
        return;
    }
    if (strcmp(shape, "rectangle") == 0) {
        rectangular_path(OUT, bx-2*boxsize, by-boxsize, bx+2*boxsize, by+boxsize);
        return;
    }
    if (strcmp(shape, "circle") == 0) {
        circular_path(OUT, bx, by, boxsize);
        return;
    }
}

void eps_map(OUTPUT_STREAM) {
    world_object *wo;
    int z, i, blh, blw, total_chunk_height, max_chunk_width,
        border = get_int_mp("border-size", NULL),
        vskip = get_int_mp("vertical-spacing", NULL);

    the_eps_map_chunks[0].width = get_int_mp("minimum-map-width", NULL);
    the_eps_map_chunks[0].actual_height = 0;
    the_eps_map_chunks[0].titling_point_size = get_int_mp("title-size", NULL);
    strcpy(the_eps_map_chunks[0].titling, "Map");
    the_eps_map_chunks[0].contains_titling = TRUE;
    the_eps_map_chunks[0].contains_rooms = FALSE;
    prepare_map_parameter_scope(&(the_eps_map_chunks[0].map_parameters));
    put_string_mp("title", &(the_eps_map_chunks[0].map_parameters),
        the_eps_map_chunks[0].titling);
    no_eps_map_chunks = 1;

    for (z=extreme_z1; z>=extreme_z0; z--) {
        eps_map_chunk *emc = &(the_eps_map_chunks[no_eps_map_chunks]);
        no_eps_map_chunks++;

        emc->contains_rooms = TRUE;
        emc->map_level = z;
        emc->y_max = -100000, emc->y_min = 100000;

        LOOP_OVER(wo, world_object) if (wo->room_flag) {
            if (wo->grid_z == z) {
                if (wo->grid_y < emc->y_min) emc->y_min = wo->grid_y;
                if (wo->grid_y > emc->y_max) emc->y_max = wo->grid_y;
            }
```

```
    }
    strcpy(emc->titling, "");
    switch(extreme_z1 - extreme_z0) {
        case 0: break;
        case 1:
            if (z == extreme_z0)
                sprintf(emc->titling, "Map of lower level");
            if (z == extreme_z1)
                sprintf(emc->titling, "Map of upper level");
            break;
        default:
            switch(z-start_room->grid_z) {
                case 0: sprintf(emc->titling, "Map of starting level"); break;
                case 1: sprintf(emc->titling, "Map of first level up"); break;
                case -1: sprintf(emc->titling, "Map of first level down"); break;
                case 2: sprintf(emc->titling, "Map of second level up"); break;
                case -2: sprintf(emc->titling, "Map of second level down"); break;
                case 3: sprintf(emc->titling, "Map of third level up"); break;
                case -3: sprintf(emc->titling, "Map of third level down"); break;
                default:
                    if (z > 0)
                        sprintf(emc->titling, "Map of %dth level up", z);
                    if (z < 0)
                        sprintf(emc->titling, "Map of %dth level down", z);
                    break;
            }
            break;
    }
    if (strcmp(emc->titling, "") == 0) emc->contains_titling = FALSE;
    else emc->contains_titling = TRUE;
    prepare_map_parameter_scope(&(emc->map_parameters));
    put_string_mp("subtitle", &(emc->map_parameters), emc->titling);
    LOOP_OVER(wo, world_object) if (wo->room_flag) {
        if (wo->grid_z == z) {
            wo->local_map_parameters.wider_scope = &(emc->map_parameters);
        }
    }
}
traverse_for_map_parameters(2);
if (changed_global_room_colour == FALSE)
    LOOP_OVER(wo, world_object)
        if (wo->room_flag)
            put_string_mp("room-colour",
                &(wo->local_map_parameters), wo->world_index_colour);
total_chunk_height = 0;
max_chunk_width = 0;
for (i=no_eps_map_chunks-1; i>=0; i--) {
    eps_map_chunk *emc = &(the_eps_map_chunks[i]);
    int mapunit = get_int_mp("grid-size", &(emc->map_parameters));
    if (i == 0)
        emc->titling_point_size =
            get_int_mp("title-size", &(emc->map_parameters));
```

```
    else emc->titling_point_size =
            get_int_mp("subtitle-size", &(emc->map_parameters));
    emc->width = (extreme_x1-extreme_x0+2)*mapunit;
    if (i == 0) emc->actual_height = 0;
    else emc->actual_height = (emc->y_max-emc->y_min+1)*mapunit;
    emc->eps_origin = total_chunk_height + border;
    emc->height = emc->actual_height + vskip;
    if (emc->contains_rooms) emc->height += vskip;
    if (emc->contains_titling) emc->height += emc->titling_point_size+vskip;
    total_chunk_height += emc->height;
    if (max_chunk_width < emc->width) max_chunk_width = emc->width;
}
blh = total_chunk_height;
blw = max_chunk_width;
WRITE("%%!PS-Adobe EPSF-3.0\n");
WRITE("%%%%BoundingBox: 0 0 %d %d\n", blw+2*border, blh+2*border);
WRITE("%%%%IncludeFont: %s\n", get_string_mp("title-font", NULL));
WRITE("/%s findfont %d scalefont setfont\n",
    get_string_mp("title-font", NULL), get_int_mp("title-size", NULL));
if (get_int_mp("map-outline", NULL)) {
    WRITE("newpath %% Ruled outline outer box of map\n");
    WRITE("%d %d moveto %% bottom left corner\n", border, border);
    WRITE("%d %d lineto %% bottom side\n", border+blw, border);
    WRITE("%d %d lineto %% right side\n", border+blw, border+blh);
    WRITE("%d %d lineto %% top side\n", border, border+blh);
    WRITE("closepath\n");
    WRITE("stroke\n");
}
for (i=0; i<no_eps_map_chunks; i++) {
    eps_map_chunk *emc = &(the_eps_map_chunks[i]);
    if (i<no_eps_map_chunks-1) {
        if (get_int_mp("map-outline", NULL)) {
            WRITE("newpath %% Ruled outline strut of map\n");
            WRITE("%d %d moveto %% LHS\n", border, emc->eps_origin);
            WRITE("%d %d lineto %% RHS\n", border+blw, emc->eps_origin);
            WRITE("closepath\n");
            WRITE("stroke\n");
        }
    }
    if (emc->contains_titling) {
        int y = emc->eps_origin + vskip + emc->actual_height;
        if (emc->contains_rooms) {
            if (get_int_mp("monochrome", &(emc->map_parameters)))
                WRITE("0 setgray %% black for subtitle\n");
            else choose_colour(OUT,
                get_string_mp("subtitle-colour", &(emc->map_parameters)));
            plot_text_at(OUT,
                get_string_mp("subtitle", &(emc->map_parameters)),
                NULL, 128,
                get_string_mp("subtitle-font", &(emc->map_parameters)),
                border*2, y+vskip,
                get_int_mp("subtitle-size", &(emc->map_parameters)),
```

```
            FALSE, FALSE);
    } else {
        if (get_int_mp("monochrome", &(emc->map_parameters)))
            WRITE("0 setgray %% black for title\n");
        else choose_colour(OUT,
            get_string_mp("title-colour", &(emc->map_parameters)));
        plot_text_at(OUT,
            get_string_mp("title", NULL),
            NULL, 128,
            get_string_mp("title-font", &(emc->map_parameters)),
            border*2, y+2*vskip,
            get_int_mp("title-size", &(emc->map_parameters)),
            FALSE, TRUE);
    }
}
if (emc->contains_rooms) {
    LOOP_OVER(wo, world_object)
    if ((wo->room_flag) && (wo->grid_z == emc->map_level)) {
        int bx = wo->grid_x-extreme_x0;
        int by = wo->grid_y-emc->y_min;
        int mapunit = get_int_mp("grid-size", &(emc->map_parameters));
        int offs = get_int_mp("room-offset", &(wo->local_map_parameters));
        int xpart = offs%10000, ypart = offs/10000;
        while (xpart > 5000) xpart-=10000;
        while (xpart < -5000) xpart+=10000;
        bx = (bx)*mapunit + border + mapunit/2;
        by = (by)*mapunit + emc->eps_origin + vskip + mapunit/2;
        bx += xpart*mapunit/100;
        by += ypart*mapunit/100;
        wo->eps_x = bx;
        wo->eps_y = by;
    }
    LOOP_OVER(wo, world_object)
    if ((wo->room_flag) && (wo->grid_z == emc->map_level)) {
        int bx = wo->eps_x;
        int by = wo->eps_y;
        int mapunit = get_int_mp("grid-size", &(emc->map_parameters));
        int boxsize = get_int_mp("room-size", &(wo->local_map_parameters))/2;
        int stiffnessx =
            get_int_mp("route-stiffness", &(wo->local_map_parameters));
        int exit, exit2;
        WRITE("currentlinewidth %% Stack for later\n");
        WRITE("%d setlinewidth\n",
            get_int_mp("route-thickness", &(wo->local_map_parameters)));
        for (exit=0; exit<10; exit++) {
            world_object *wo2 = room_exit(wo, exit, FALSE);
            if (wo2) {
                int plotted_exit = FALSE;
                int stiffnessy =
                    get_int_mp("route-stiffness", &(wo2->local_map_parameters));
                if (get_int_mp("monochrome", &(emc->map_parameters)))
                    WRITE("0 setgray %% black for route lines\n");
```

```
            else choose_colour(OUT,
                get_string_mp("route-colour", &(emc->map_parameters)));
            if (wo2->room_flag) {
                if (wo2->grid_z == emc->map_level) {
                    for (exit2=0; exit2<10; exit2++)
                        if (room_exit(wo2, exit2, FALSE) == wo) {
                            if (wo->allocation_id <= wo2->allocation_id)
                            bezier_curve(OUT,
                                stiffnessx*mapunit,
                                stiffnessy*mapunit,
                                bx, by, exit,
                                wo2->eps_x, wo2->eps_y, exit2);
                            plotted_exit = TRUE;
                        }
                }
                if (plotted_exit == FALSE) {
                    plotted_exit = TRUE;
                    faraway_arrow(OUT, boxsize, exit, bx, by, wo2);
                }
            }
        }
    }
    WRITE("setlinewidth\n");
}
LOOP_OVER(wo, world_object)
if ((wo->room_flag) && (wo->grid_z == emc->map_level)) {
    int bx = wo->eps_x;
    int by = wo->eps_y;
    int boxsize = get_int_mp("room-size", &(wo->local_map_parameters))/2;
    char *legend;
    int mapunit = get_int_mp("grid-size", &(emc->map_parameters));
    int offs = get_int_mp("room-name-offset", &(wo->local_map_parameters));
    int xpart = offs%10000, ypart = offs/10000;
    while (xpart > 5000) xpart-=10000;
    while (xpart < -5000) xpart+=10000;
    WRITE("newpath %% Shaded box for ");
    if (wo->word_ref1 >= 0)
        print_raw_text_to_file(wo->word_ref1, wo->word_ref2, OUT);
        else WRITE("nameless room");
    WRITE(" (%d,%d,%d)\n", wo->grid_x, wo->grid_y, wo->grid_z);
    if (get_int_mp("monochrome", &(wo->local_map_parameters)))
        WRITE("0.75 setgray %% light gray for interior\n");
    else choose_colour(OUT,
        get_string_mp("room-colour", &(wo->local_map_parameters)));
    make_room_path(OUT, bx, by, boxsize,
        get_string_mp("room-shape", &(wo->local_map_parameters)));
    WRITE("fill\n\n");
    if (get_int_mp("room-outline", &(wo->local_map_parameters))) {
        WRITE("currentlinewidth %% Stack for later\n");
        WRITE("%d setlinewidth\n",
            get_int_mp("room-outline-thickness",
                &(wo->local_map_parameters)));
```

```
                WRITE("newpath %% Room outline\n");
                if (get_int_mp("monochrome", &(emc->map_parameters)))
                    WRITE("0 setgray %% black for room outline\n");
                else choose_colour(OUT,
                    get_string_mp("room-outline-colour",
                        &(wo->local_map_parameters)));
                make_room_path(OUT, bx, by, boxsize,
                    get_string_mp("room-shape", &(wo->local_map_parameters)));
                WRITE("stroke\nsetlinewidth\n");
            }
            bx += xpart*mapunit/100;
            by += ypart*mapunit/100;
            if (get_int_mp("monochrome", &(emc->map_parameters)))
                WRITE("0 setgray %% black for room name\n");
            else choose_colour(OUT,
                get_string_mp("room-name-colour", &(wo->local_map_parameters)));
            legend = get_string_mp("room-name", &(wo->local_map_parameters));
            if (strcmp(legend, "") == 0) {
                plot_text_at(OUT, NULL, wo,
                    get_int_mp("room-name-length", &(wo->local_map_parameters)),
                    get_string_mp("room-name-font", &(wo->local_map_parameters)),
                    bx, by, get_int_mp("room-name-size",
                        &(wo->local_map_parameters)),
                    TRUE, TRUE);
            } else {
                plot_text_at(OUT, legend, NULL,
                    get_int_mp("room-name-length", &(wo->local_map_parameters)),
                    get_string_mp("room-name-font", &(wo->local_map_parameters)),
                    bx, by, get_int_mp("room-name-size",
                        &(wo->local_map_parameters)),
                    TRUE, TRUE);
            }
        }
    }
}
for (i=0; i<no_rubrics; i++) {
    int bx = 0, by = 0;
    int xpart = the_rubrics[i].at_offset%10000,
        ypart = the_rubrics[i].at_offset/10000;
    int mapunit = get_int_mp("grid-size", NULL);
    while (xpart > 5000) xpart-=10000;
    while (xpart < -5000) xpart+=10000;
    if (get_int_mp("monochrome", NULL)) WRITE("0 setgray %% black for rubric\n");
    else choose_colour(OUT, the_rubrics[i].colour);
    if (the_rubrics[i].offset_from) {
        bx = the_rubrics[i].offset_from->eps_x;
        by = the_rubrics[i].offset_from->eps_y;
    }
    bx += xpart*mapunit/100;
    by += ypart*mapunit/100;
    plot_text_at(OUT, the_rubrics[i].annotation, NULL, 128,
        the_rubrics[i].font,
```

```
            bx, by, the_rubrics[i].point_size,
            TRUE, TRUE);
    }
}
```

The structure eps_map_chunk is private to this section.

§23.

```
void write_eps_file(void) {
    STREAM EPS_struct; STREAM *EPS = &EPS_struct;
    char *fn = build_filename(EPSMAP_LEAFNAME);
    if (write_eps_to_desktop) fn = filename_of_epsfile;
    if (STREAM_OPEN_TO_FILE(EPS, fn, ISO_ENC) == FALSE)
        fatal_error2("Can't open EPS map file", fn);
    eps_map(EPS);
    STREAM_CLOSE(EPS);
}
```

The function write_eps_file is called from 9/inpw.

Temporal Map 9/tmap

Purpose

Parallel to the World index of space is the Scenes index of time, and in this section we render it as HTML.

Template interpreter commands

```
1  {-callv:index_page_Scenes}
```

---

§**1.** The mapping of time is on the one hand simpler than the mapping of space since there is only one dimension, but on the other hand more complex since scenes can be multiply present at the same instant of time (whereas rooms cannot be multiply present at the same point in space). We resolve this with a notation which takes a little bit of on-screen explanation, but seems natural enough to learn in practice.

```
void index_scene_end(scene *sc, int end) {
    int count = 0;
    scene_connector *scon;
    if ((end == 1) && (sc->no_ends > 2) &&
        (sc->anchor_condition[1]==NULL) && (sc->anchor_scene[1]==NULL))
        return;
    open_html_paragraph(ifl, 1, "hanging");
    INDEX("<i>%s ", (end==0)?"Begins":"Ends");
    if (end >= 2) {
        print_raw_text_to_file(sc->end_names_w1[end], sc->end_names_w2[end], ifl);
        INDEX(" ");
    }
    INDEX("when:</i> ");
    if ((end==0) && (sc->start_of_play)) {
        INDEX("<b>play begins</b>");
        count++;
    }
    if (sc->anchor_condition[end]) {
        if (count > 0) INDEX("<br><i>or when:</i> ");
        print_raw_text_to_file(
            sc->anchor_condition[end]->word_ref1,
            sc->anchor_condition[end]->word_ref2, ifl);
        index_link(lw_array[sc->anchor_condition_set[end]->word_ref1].lw_source);
        count++;
    }
    for (scon = sc->anchor_scene[end]; scon; scon=scon->next) {
        if (count > 0) INDEX("<br><i>or when:</i> ");
        INDEX("<b>");
        print_raw_text_to_file(
            scon->connect_to->q->word_ref1,
            scon->connect_to->q->word_ref2, ifl);
        INDEX("</b> <i>%s</i>", (scon->end==0)?"begins":"ends");
        if (scon->end >= 2) {
            INDEX(" ");
            print_raw_text_to_file(scon->connect_to->end_names_w1[scon->end],
```

```
                scon->connect_to->end_names_w2[scon->end], ifl);
        }
        index_link(lw_array[scon->where_said->word_ref1].lw_source);
        count++;
    }
    if (count == 0) {
        if ((end == 1) && (sc->no_ends > 2)) INDEX("see below");
        else INDEX("<b>never</b>");
    }
    INDEX("</p>");
    if (rb_is_empty(sc->end_rulebook[end], NULL) == FALSE) {
        open_html_paragraph(ifl, 1, "hanging");
        INDEX("<i>What happens:</i></p>");
        rb_index(sc->end_rulebook[end], "", NULL, NULL);
    }
}

int scene_icons = 0;

void index_from_scene(scene *sc, int depth, int end, scene *sc_from, scene **sorted, int nr)
{
    scene *sc2; scene_connector *scon;
    int i, j = 0;
    open_html_paragraph(ifl, depth+1, "tight");
    scene_icons++;
    switch(end) {
        case 0: INDEX("<img border=0 src=inform:/scene_icons/Simul.png>  "); break;
        case 1: INDEX("<img border=0 src=inform:/scene_icons/Segue.png>  "); break;
        case -1: break;
        case -2: INDEX("<img border=0 src=inform:/scene_icons/WNever.png>  "); break;
        default: INDEX("<img border=0 src=inform:/scene_icons/Segue.png>  [ends ");
            print_raw_text_to_file(
                sc_from->end_names_w1[end], sc_from->end_names_w2[end],
                ifl);
            INDEX("] "); break;
    }
    if (sc->indexed == FALSE) {
        if (sc->anchor_condition[0])
            INDEX("<img border=0 src=inform:/scene_icons/WhenC.png>  ");
        if (sc->start_of_play)
            INDEX("<img border=0 src=inform:/scene_icons/WPB.png>  ");
        for (i=1, j=0; i<sc->no_ends; i++) {
            if (sc->anchor_scene[i]) j++;
            if (sc->anchor_condition[i]) j++;
        }
    }
    if (sc->indexed) INDEX("<i>");
    print_raw_text_to_file(sc->q->word_ref1, sc->q->word_ref2, ifl);
    if (sc->indexed) INDEX("</i>");
    else index_below_link_numbered(sc->allocation_id);
    if (sc->indexed == FALSE) {
        if (j==0) INDEX("  <img border=0 src=inform:/scene_icons/ENever.png>");
    }
    INDEX("</p>");
    if (sc->indexed) return;
```

```
    sc->indexed = TRUE;
    for (i=0; i<nr; i++) {
        sc2 = sorted[i];
        for (scon = sc2->anchor_scene[0]; scon; scon=scon->next)
            if ((scon->connect_to == sc) && (scon->end >= 1))
                index_from_scene(sc2, depth + 1, scon->end, sc, sorted, nr);
    }
    for (i=0; i<nr; i++) {
        sc2 = sorted[i];
        for (scon = sc2->anchor_scene[0]; scon; scon=scon->next)
            if ((scon->connect_to == sc) && (scon->end == 0))
                index_from_scene(sc2, depth, scon->end, sc, sorted, nr);
    }
}
void index_page_Scenes(void) {
    int nr = NUMBER_CREATED(scene);
    scene **sorted = I7_calloc(nr, sizeof(scene *), INDEX_SORTING_MREASON);
    ⟨Sort the scenes 2⟩;
    ⟨Tabulate the scenes 4⟩;
    ⟨Give details of each scene in turn 5⟩;
    I7_free(sorted, INDEX_SORTING_MREASON);
    index_scene_rulebooks();
}
```

The function index_page_Scenes is invoked by a command in a `.i6t` template file.

§**2.** As usual, we sort with the C library's `qsort`.

⟨Sort the scenes 2⟩ ≡

```
    int i = 0;
    scene *sc;
    LOOP_OVER(sc, scene) sorted[i++] = sc;
    qsort(sorted, nr, sizeof(scene *), compare_scenes);
```

This code is used in §1.

§**3.** The following means the scenes are sorted in alphabetical order.

```
int compare_scenes(const void *ent1, const void *ent2) {
    const scene *sc1 = *((const scene **) ent1);
    const scene *sc2 = *((const scene **) ent2);
    return rangecmp(sc1->q->word_ref1, sc1->q->word_ref2, sc2->q->word_ref1, sc2->q->word_ref2);
}
```

§4.

⟨Tabulate the scenes 4⟩ ≡

```
scene *sc; int i;
for (i=0; i<nr; i++) {
    sc = sorted[i];
    if (sc->start_of_play)
        index_from_scene(sc, 0, -1, NULL, sorted, nr);
}
for (i=0; i<nr; i++) {
    sc = sorted[i];
    if (sc->anchor_condition[0])
        index_from_scene(sc, 0, -1, NULL, sorted, nr);
}
for (i=0; i<nr; i++) {
    sc = sorted[i];
    if (sc->indexed == FALSE)
        index_from_scene(sc, 0, -2, NULL, sorted, nr);
}
if (scene_icons > 0) {
    INDEX("<p>Legend: <img border=0 src=inform:/scene_icons/WPB.png> "
    "<i>Begins when play begins</i>; "
    "<img border=0 src=inform:/scene_icons/WhenC.png> "
    "<i>can begin whenever some condition holds</i>; "
    "<img border=0 src=inform:/scene_icons/Segue.png> "
    "<i>follows when a previous scene ends</i>; "
    "<img border=0 src=inform:/scene_icons/Simul.png> "
    "<i>begins simultaneously</i>; "
    "<img border=0 src=inform:/scene_icons/WNever.png> "
    "<i>never begins</i>; "
    "<img border=0 src=inform:/scene_icons/ENever.png> "
    "<i>never ends</i>.");
}
INDEX("<p><hr>");
```

This code is used in §1.

§5.

⟨Give details of each scene in turn 5⟩ ≡

```
INDEX("<p>");
index_anchor("SDETAILS");
scene *sc; int i;
for (i=0; i<nr; i++) {
    sc = sorted[i];
    rulebook *rb;
    open_html_paragraph(ifl, 1, "hanging");
    index_anchor_numbered(sc->allocation_id);
    INDEX("<b>The <i>");
    print_raw_text_to_file(sc->q->word_ref1, sc->q->word_ref2, ifl);
    INDEX("</i> scene</b>");
    index_link(lw_array[sc->scene_declared_at->word_ref1].lw_source);
    INDEX("</p>");
    index_scene_end(sc, 0);
```

```
        int rbc = 0;
        LOOP_OVER(rb, rulebook) {
            if (rb_is_empty(rb, sc) == FALSE) {
                if (rbc++ == 0) {
                    open_html_paragraph(ifl, 1, "hanging");
                    INDEX("<i>During this scene:</i></p>");
                }
                open_html_paragraph(ifl, 2, "hanging");
                INDEX("<i>");
                print_raw_text_to_file(rb->word_ref1, rb->word_ref2, ifl);
                INDEX("</i></p>");
                rb_index(rb, "", sc, NULL);
            }
        }
        int j;
        for (j=1; j<sc->no_ends; j++) index_scene_end(sc, j);
        INDEX("<p><hr><p>");
    }
```

This code is used in §1.

10 An Intangible Miscellany

`10/qnbp`: *Quasinumeric Relations.w* To define the binary predicates corresponding to numerical comparisons.

`10/str`: *String Constants.w* In this section we compile I6 values for the specifications `VALUE/TEXT` and `VALUE/QUOT`.

`10/list`: *List Constants.w* In this section we compile I6 arrays for constant lists arising from braced literals like "1,2,3".

`10/tab`: *Tables.w* To manage and compile tables, which are two-dimensional arrays with associative look-up facilities provided at run-time.

`10/eqns`: *Equations.w* To manage and compile equations, which relate numerical quantities.

`10/libp`: *Listed-In Relations.w* To define the binary predicates corresponding to table columns, and which determine whether a given value is listed in that column.

`10/ifid`: *Interactive Fiction ID.w* Computes and makes available the IFID (Interactive Fiction ID) number for an Inform-generated work of IF, in compliance with the Treaty of Babel.

`10/bib`: *Bibliographic Data.w* To manage the special quantities providing bibliographic data on the work of IF being generated (title, author's name and so forth); to write the iFiction record for the work of IF compiled, its release instructions and its picture manifest, if any.

`10/fig`: *Figures.w* To register the names associated with picture resource numbers, which are defined to allow the final story file to display pictures, and to produce the thumbnail index of figures.

`10/sfx`: *Sound Effects.w* To register the names associated with sound resource numbers, which are defined to allow the final story file to play sound effects, and to produce the index of sound effects.

`10/exf`: *External Files.w* To register the names associated with external files, and build the small I6 arrays associated with each.

`10/isin`: *Inform 6 Inclusions.w* To include Inform 6 code almost verbatim in the output of NI, as instructed by low-level Inform 7 sentences.

Quasinumeric Relations 10/qnbp

Purpose

To define the binary predicates corresponding to numerical comparisons.

Definitions

¶1. The inequality relations $<$, $>$, $\leq$, and $\geq$, which can be applied not only to numbers but also to units (height, length and so on).

It might seem redundant to define both `a_gt_b_predicate` (which makes the numerical test $a > b$) and also `a_lt_b_predicate` (which tests $a < b$). Why not define only one, and get the other meaning free as its reversal? The answer is that is more convenient not to, because it allows us to give both of them names.

There is no numerical equality relation $=$ as such: numbers use the same equality BP as everything else.

```
binary_predicate *a_gt_b_predicate = NULL;
binary_predicate *a_lt_b_predicate = NULL;
binary_predicate *a_ge_b_predicate = NULL;
binary_predicate *a_le_b_predicate = NULL;
```

§1. Initial stock.

These relations are all hard-wired in.

```
void QUASINUMERIC_KBP_create_initial_stock(void) {
    bp_term_details number_term = bptd_new(NULL, kova(NUMBER_TY));
    a_gt_b_predicate =
        make_pair_of_BPs(QUASINUMERIC_KBP,
            number_term, number_term,
            "greater-than", NULL, -1, NULL, NULL, sch_new("*1 > *2"),
            numerically_greater_than_V);
    a_lt_b_predicate =
        make_pair_of_BPs(QUASINUMERIC_KBP,
            number_term, number_term,
            "less-than", NULL, -1, NULL, NULL, sch_new("*1 < *2"),
            numerically_less_than_V);
    a_ge_b_predicate =
        make_pair_of_BPs(QUASINUMERIC_KBP,
            number_term, number_term,
            "at-least", NULL, -1, NULL, NULL, sch_new("*1 >= *2"),
            numerically_greater_than_or_equal_to_V);
    a_le_b_predicate =
        make_pair_of_BPs(QUASINUMERIC_KBP,
            number_term, number_term,
            "at-most", NULL, -1, NULL, NULL, sch_new("*1 <= *2"),
            numerically_less_than_or_equal_to_V);
}
```

The function QUASINUMERIC_KBP_create_initial_stock is called from 5/bp.

§2. **Second stock.** There is none – this is a family of relations which is all built in.

```
void QUASINUMERIC_KBP_create_second_stock(void) {
}
```

The function QUASINUMERIC_KBP_create_second_stock is called from 5/bp.

§3. **Typechecking.**

```
int QUASINUMERIC_KBP_typecheck(binary_predicate *bp,
        kind_of_value **kovs_of_terms, kind_of_value **kovs_required, tc_problem_kit *tck) {
    if ((can_we_cast_kovs(kovs_of_terms[0], kovs_of_terms[1]) == NEVER_MATCH) &&
        (can_we_cast_kovs(kovs_of_terms[1], kovs_of_terms[0]) == NEVER_MATCH)) {
        if (tck->log_to_I6_text)
            LOG("Unable to apply inequality of $u and $u\n", kovs_of_terms[0], kovs_of_terms[1]);
        quote_kov(4, kovs_of_terms[0]);
        quote_kov(5, kovs_of_terms[1]);
        tcp_problem(_P_(C10InequalityFailed), tck,
            "that would mean comparing two kinds of value which cannot mix - "
            "%4 and %5 - so this must be incorrect.");
        return NEVER_MATCH;
    }
    return ALWAYS_MATCH;
}
```

The function QUASINUMERIC_KBP_typecheck is called from 5/bp.

§4. **Assertion.** These relations cannot be asserted.

```
int QUASINUMERIC_KBP_assert(binary_predicate *bp,
        world_object *wo0, specification *spec0,
        world_object *wo1, specification *spec1) {
    return FALSE;
}
```

The function QUASINUMERIC_KBP_assert is called from 5/bp.

§5. **Compilation.** We need do nothing special: these relations can be compiled from their schemas.

```
int QUASINUMERIC_KBP_compile(int task, binary_predicate *bp, annotated_i6_schema *asch) {
    return FALSE;
}
```

The function QUASINUMERIC_KBP_compile is called from 5/bp.

§6. **Problem message text.**

```
int QUASINUMERIC_KBP_describe_for_problems(binary_predicate *bp, char *problem_buffer) {
    return FALSE;
}
```

The function QUASINUMERIC_KBP_describe_for_problems is called from 5/bp.

String Constants 10/str

Purpose

In this section we compile I6 values for the specifications `VALUE/TEXT` and `VALUE/QUOT`.

10/str.¶2-0 Modes used in compiling literal text

Template interpreter commands

```
9   {-callv:compile_string_constants}
10   {-callv:compile_text_routines}
10   {-callv:allow_no_further_text_subs}
```

Definitions

¶1. Literal text constants need to be stored up, because we will want to choose their ordering. In the Z-machine, strings are referred to by their packed addresses, and I6 always compiles them in order of creation. If we are careful, we can arrange the I6 output of NI such that they are created in alphabetical order: then a simple unsigned comparison of packed addresses at run-time is equivalent to an alphabetical comparison of the text. (We make great use of this when sorting tables.) We can also ensure that two literal texts with the same contents are indeed equal: I6 would have compiled two citations of the same text to two different copies with different packed addresses, so that a numerical comparison of addresses would have failed.

The "literal type" alluded to below is one of `TEXT_TY` for plain text, `QUOT_TY` for raw text used in I6-style boxed quotations and `TEXT_ROUTINE_TY` for text with substitutions (which will need to be printed by a routine).

```
define RED_NODE 1
define BLACK_NODE 2

typedef struct literal_text {
    int word_ref1;                                  position in the source of quoted text
    int literal_type;                               one of the three type values above
    int bibliographic_conventions;                  mostly for apostrophes
    int node_colour;                                red or black: see above
    struct literal_text *left_node;                 within their red-black tree
    struct literal_text *right_node;
    MEMORY_MANAGEMENT
} literal_text;

literal_text *root_of_literal_text = NULL;
literal_text *z_node = NULL;
```

The structure literal_text is private to this section.

¶2. Modes used in compiling literal text. When NI compiles a piece of literal text, it is usually aiming to write this in a notation which I6 will read, which will involve I6 character escapes to render certain exotica: but sometimes it is writing the XML iFiction record instead, which takes a more literal approach to literal text. Hence:

```
int divert_constant_text_bibliographically = FALSE;      Compile literal text more literally
int encode_constant_text_bibliographically = FALSE;      Compile literal text semi-literally
```

¶3. Text containing substitutions, such as "You pick up [the noun] thoughtfully.", are compiled as routines rather than Z-machine strings. Each is stored in one of the following structures. Unlike literal text, a text routine might lead to problem messages when eventually compiled, so it is useful to record the current sentence when a text routine is created: this means a problem can be reported at the right place.

```
typedef struct text_routine {
    int word_ref1, word_ref2;                    text when expanded just prior to compilation
    int wn;                                      position of unexpanded text in the source of quoted text
    struct parse_node *sentence_using_this;      where this occurs in source
    int local_names_existed_at_usage_time;       remember in case of problems
    struct ph_stack_frame *parked_stack_frame;   for cases where possible
    MEMORY_MANAGEMENT
} text_routine;
```

The structure text_routine is private to this section.

¶4. We are only allowed to create new ones until the following is set:

```
int no_further_text_subs = FALSE;
```

¶5. The following global variable records whether we are currently compiling a text routine, rather than some other routine, or free-standing objects.

```
int compiling_text_routines_mode = FALSE;    used for better problem messages
```

---

§**1.** It might be thought that a string constant could be compiled to I6 simply by printing it, but there are three obstacles. Firstly, we need to escape the various escape characters. Secondly, the double-quoted strings used in `box` quotation commands require special handling. And thirdly, we want to achieve a sneaky effect: to ensure that two copies of the same text (other than used for box quotations) always have the same value in the Z-machine, and that their numerical values coincide with alphabetical order when compared as unsigned integers.

Each new string is therefore insertion-sorted into a list in alphabetical order, and given a unique ID number, `X`; then what we compile is the name of an I6 constant, `SC_X`. These constants must be defined later, right at the end of the compilation when we know that all necessary strings have been created and that the list is complete. We achieve the alphabetical ordering by defining the constants in list order, which is not the order of their `X` suffixes.

Quotation texts are instead compiled to routines, also called `SC_X`, which carry out I6 `box` statements.

Insertion-sort has a rather poor running time. At present this does not seem to pose any problem, but if profiling suggests that it is becoming wasteful when I7 source texts become longer, we will probably have to replace it with a red-black tree or similar.

```
literal_text *lt_new(int w1, int colour) {
    literal_text *x = CREATE(literal_text);
    x->left_node = NULL;
    x->right_node = NULL;
    x->node_colour = colour;
    x->word_ref1 = w1;
    x->literal_type = TEXT_TY;
    x->bibliographic_conventions = FALSE;
    return x;
```

```
}
int lt_cmp(int w1, literal_text *lt) {
    if (lt == NULL) return 1;
    if (lt->word_ref1 < 0) return 1;
    return strcmp(lw_array[w1].lw_text, lw_array[lt->word_ref1].lw_text);
}
literal_text *compile_literal_text(OUTPUT_STREAM, int w1) {
    if (divert_constant_text_bibliographically) {
        compile_bibliographic_text(OUT, lw_array[w1].lw_text);
        return NULL;
    }
    if (strcmp(lw_array[w1].lw_text, "\"\"") == 0) {
        WRITE("EMPTY_TEXT_VALUE");
        return NULL;
    }
    if (z_node == NULL) ⟨Initialise the red-black tree 2⟩;
    ⟨Search for the text as a key in the red-black tree 3⟩;
}
```

The function compile_literal_text is called from 7/vasp and 10/bib.

§**2.** For an account of the theory of red-black trees, see Sedgewick, *Algorithms* (2nd edn, chap. 15).

⟨Initialise the red-black tree 2⟩ ≡

```
    z_node = lt_new(-1, BLACK_NODE); z_node->left_node = z_node; z_node->right_node = z_node;
    root_of_literal_text = lt_new(-1, BLACK_NODE);
    root_of_literal_text->left_node = z_node;
    root_of_literal_text->right_node = z_node;
```

This code is used in §1.

§**3.**

⟨Search for the text as a key in the red-black tree 3⟩ ≡

```
    literal_text *x = root_of_literal_text, *p = x, *g = p, *gg = g;
    int went_left = FALSE;                          redundant assignment to appease gcc -O2
    do {
        gg = g; g = p; p = x;
        int sgn = lt_cmp(w1, x);
        if (sgn == 0) ⟨Locate this as the new node 4⟩;
        if (sgn < 0) { went_left = TRUE; x = x->left_node; }
        if (sgn > 0) { went_left = FALSE; x = x->right_node; }
        if ((x->left_node->node_colour == RED_NODE) &&
            (x->right_node->node_colour == RED_NODE))
            ⟨Perform a split 5⟩;
    } while (x != z_node);
    x = lt_new(w1, RED_NODE);
    x->left_node = z_node; x->right_node = z_node;
    if (went_left == TRUE) p->left_node = x; else p->right_node = x;
    literal_text *new_x = x;
    ⟨Perform a split 5⟩;
    x = new_x;
    ⟨Locate this as the new node 4⟩;
```

This code is used in §1.

§4.

⟨Locate this as the new node 4⟩ ≡

```
    if (encode_constant_text_bibliographically) x->bibliographic_conventions = TRUE;
    WRITE("SC_%d", x->allocation_id);
    return x;
```

This code is used in §3.

§5.

⟨Perform a split 5⟩ ≡

```
    x->node_colour = RED_NODE;
    x->left_node->node_colour = BLACK_NODE;
    x->right_node->node_colour = BLACK_NODE;
    if (p->node_colour == RED_NODE) ⟨Rotations will be needed 6⟩;
    root_of_literal_text->right_node->node_colour = BLACK_NODE;
```

This code is used in §3.

§6.

⟨Rotations will be needed 6⟩ ≡

```
    g->node_colour = RED_NODE;
    int left_of_g = FALSE, left_of_p = FALSE;
    if (lt_cmp(w1, g) < 0) left_of_g = TRUE;
    if (lt_cmp(w1, p) < 0) left_of_p = TRUE;
    if (left_of_g != left_of_p) p = rotate(w1, g);
    x = rotate(w1, gg);
    x->node_colour = BLACK_NODE;
```

This code is used in §5.

§7.

```
literal_text *rotate(int w1, literal_text *y) {
    literal_text *c, *gc;
    if (lt_cmp(w1, y) < 0) c = y->left_node; else c = y->right_node;
    if (lt_cmp(w1, c) < 0) {
        gc = c->left_node; c->left_node = gc->right_node; gc->right_node = c;
    } else {
        gc = c->right_node; c->right_node = gc->left_node; gc->left_node = c;
    }
    if (lt_cmp(w1, y) < 0) y->left_node = gc; else y->right_node = gc;
    return gc;
}
```

§**8.**

```
int extent_of_runtime_quotations_array = 1;                    start at 1 to avoid 0 length
void compile_quotation(OUTPUT_STREAM, int w1) {
    literal_text *lt = compile_literal_text(OUT, w1);
    lt->literal_type = QUOT_TY;
    extent_of_runtime_quotations_array++;
}
```

The function compile_quotation is called from 7/vasp.

§**9.** The above gradually piled up the need for `SC_X` constants/routines, as compilation went on: now comes the reckoning, when we have to declare all of these.

See the DM4 for details of the escape characters used in I6 string constants, which use the at symbol – making the code below slightly unreadable in raw `CWEB` format, because the at-symbol is also the escape character in CWEB: thus, in raw (untypeset) format, a double-at below means a single-at. (If you are reading this in the typeset PDF, then this will have been absorbed and you will see the actual text printed, without the doubling for `CWEB` purposes.) For the avoidance of doubt: in I6, one prints a literal tilde with the sequence of characters "at, at, 1, 2, 6", and that is what is compiled below.

```
void compile_string_constants(OUTPUT_STREAM) {
    traverse_lts(OUT, root_of_literal_text);
}
void traverse_lts(OUTPUT_STREAM, literal_text *lt) {
    if (lt->left_node != z_node) traverse_lts(OUT, lt->left_node);
    if (lt->word_ref1 >= 0) {
        if (lt->literal_type == TEXT_TY) {
            int options;
            WRITE("Constant SC_%d = \"", lt->allocation_id);
            options = ISN_DEQUOTE + ISN_EXPAND_APOSTROPHES;
            if (lt->bibliographic_conventions)
                options += ISN_RECOGNISE_APOSTROPHE_SUBSTITUTION;
            isn_compile_string(OUT, lw_array[lt->word_ref1].lw_text, options);
            WRITE("\";\n");
        } else {
            char *p = lw_array[lt->word_ref1].lw_text;
            WRITE("[ SC_%d;\n", lt->allocation_id); INDENT;
            WRITE("box\n\""); INDENT;
            p++;
            while (*p) {
                if ((*p == '\n') || (*p == '\t') || (*p == '^') || (*p == NEWLINE_IN_STRING))
                    WRITE("\"\n\"");
                else WRITE("%c", *p);
                p++;
            }
            WRITE(";\n");
            OUTDENT; OUTDENT; WRITE("];\n");
        }
    }
    if (lt->right_node != z_node) traverse_lts(OUT, lt->right_node);
}
```

The function compile_string_constants is invoked by a command in a `.i6t` template file.

§10.

```
int text_routine_cue(int wn, ph_stack_frame *phsf) {
    text_routine *tr = CREATE(text_routine);
    if (no_further_text_subs) internal_error("Too late for further text substitutions");
    tr->word_ref1 = -1;
    tr->word_ref2 = -1;
    tr->wn = wn;
    tr->sentence_using_this = current_sentence;
    tr->local_names_existed_at_usage_time = FALSE;
    tr->parked_stack_frame = phsf;
    if ((phrase_being_compiled) &&
        (phsf_contains_local_quantities(current_stack_frame())))
        tr->local_names_existed_at_usage_time = TRUE;
    return tr->allocation_id;
}

text_routine *tr_under_way = NULL;

void append_text_routine_proviso(void) {
    int w1 = tr_under_way->word_ref1;
    int w2 = tr_under_way->word_ref2;
    if (it_is_not_worth_adding) return;
    if (tr_under_way->local_names_existed_at_usage_time) {
        quote_words(9, w1, w2);
        issue_problem_segment(                                    test with XLocalInSubstitution
            " %PIt may be worth adding that this problem arose in text "
            "which both contains substitutions and is also being used as "
            "a value - being put into a variable, or used as one of the "
            "ingredients in a phrase other than 'say'. Because that means "
            "it needs to be used in places outside its immediate context, "
            "it is not allowed to refer to any 'let' values or phrase "
            "options - those are temporary things, long gone by the time it "
            "would need to be printed.");
    }
}

void compile_text_routines(OUTPUT_STREAM) {
    text_routine *tr;
    int lw1, lw2, k;

    compiling_text_routines_mode = TRUE;

    LOOP_OVER(tr, text_routine) {
        ph_stack_frame *phsf;
        tr_under_way = tr;

        phsf = phsf_create_nonphrase_stack_frame();
        if (tr->parked_stack_frame)
            phsf_copy_locals(phsf, tr->parked_stack_frame);

        OUT = begin_compiling_phrase(OUT);
        if (tr->parked_stack_frame)
            phsf_compile_local_retrieval(OUT, tr->parked_stack_frame);
        begin_code_blocks();
        INDENT;

        lw1 = lexer_wordcount;
        feed_into_lexer(" say ", FALSE, FALSE);
```

```
        splice_words(tr->wn, tr->wn);
        lw2 = lexer_wordcount - 1;
        feed_into_lexer(" . ", FALSE, FALSE);
        current_sentence = tr->sentence_using_this;
        tr->word_ref1 = lw1;
        tr->word_ref2 = lw2;
        for (k=lw1; k<=lw2; k++)
            if [[word k == FULLSTOP/SEMICOLON]] {
                sentence_problem(_P_(BelievedImpossible),          now caught in the S-parser
                    "a substitution contains a '.' or ';'",
                    "which suggests that a close square bracket ']' may "
                    "have gone astray.");
                goto CutLosses;
            }
        LOGIF(COMPILE_TEXT_ROUTINES,
            "Compiling text routine %d\n$W (=%d, %d)\nWith current sentence:\n$P",
            tr->allocation_id, lw1, lw2, lw1, lw2, current_sentence);
        BEGIN_COMPILATION_MODE;
        COMPILATION_MODE_EXIT(IMPLY_NEWLINES_IN_SAY_CMODE);
        compile_To_phrase(OUT, lw1, lw2);
        END_COMPILATION_MODE;
        WRITE("\n");
        CutLosses:
        WRITE("rtrue;\n");
        OUTDENT; WRITE("];\n\n");
        OUT = write_routine_header();
        WRITE("[ text_routine_%d", tr->allocation_id);
        copy_compiled_phrase();
        end_code_blocks();
        phsf_remove_nonphrase_stack_frame();
        finish_this_session_of_parsing();
    }
    compiling_text_routines_mode = FALSE;
}
void allow_no_further_text_subs(void) {
    no_further_text_subs = TRUE;
}
```

The function text_routine_cue is called from 7/vasp.

The function append_text_routine_proviso is called from 2/prob2.

The function compile_text_routines is invoked by a command in a `.i6t` template file.

The function allow_no_further_text_subs is invoked by a command in a `.i6t` template file.

List Constants 10/list

Purpose

In this section we compile I6 arrays for constant lists arising from braced literals like "1,2,3".

Template interpreter commands

```
2   {-callv:compile_list_constants}
2   {-array:ConstantListPointers}
```

Definitions

```
typedef struct literal_list {
    int word_ref1, word_ref2;                  /* position in the source of quoted text */
    struct kind_of_value *entry_kov;           /* i.e., of the entries, not the list */
    int make_flag;                             /* build on initialisation from array */
    int no_entries;
    int compiled_already;
    MEMORY_MANAGEMENT
} literal_list;
```

The structure literal_list is private to this section.

---

§**1.**

```
kind_of_value *kov_list_of(kind_of_value *kov) {
    return kovcon(LIST_OF_TY, kov);
}
```

The function kov_list_of is called from 7/tc.

§2.

```
define FAIL_PLOV
    { if (compile_mode) internal_error("tried to compile malformed constant list");
        else return NULL;
    }

kind_of_value *parse_list_of_values(OUTPUT_STREAM, int w1, int w2, int compile_mode) {
    kind_of_value *kov = NULL, *ekov;
    literal_list *ll;
    int e1 = w1, e2, ne1, ecount = 0, blevel = 0;
    if (w1<0) FAIL_PLOV;
    if (w2<w1) {
        kov = kova(NO_ENTRIES_TY);
    } else {
        LOG("%s list of values $W\n", (compile_mode)?"Compiling":"Parsing", w1, w2);
        while (e1 <= w2) {
            e2 = e1;
            ne1 = -1;
            while (e2 < w2) {
                if [[word e2 == OPENBRACE]] blevel++;
                if [[word e2 == CLOSEBRACE]] blevel--;
                if ([[word e2 == COMMA]] && (blevel == 0)) { ne1 = e2+1; e2--; break; }
                e2++;
            }
            if (e2 < e1) FAIL_PLOV;
            specification *spec = parse_expression(e1, e2, VALUE_EXPCON);
            if (species_is(spec, CONSTANT_SPC) == FALSE) FAIL_PLOV;
            ekov = spec_evaluates_to(spec);
            if (is_kova(ekov, TEXT_ROUTINE_TY)) ekov = kova(TEXT_TY);
            LOG("Parsed list entry $W as $u\n", e1, e2, ekov);
            if (ekov == NULL) FAIL_PLOV;
            ecount++;
            if (compile_mode) {
                WRITE("(");
                BEGIN_COMPILATION_MODE;
                COMPILATION_MODE_EXIT(DEREFERENCE_POINTERS_CMODE);
                spec_compile(OUT, spec);
                END_COMPILATION_MODE;
                WRITE(") ");
                if (ne1 >= 0) e1 = ne1; else e1 = w2+1;
            } else {
                if (ne1 >= 0) e1 = ne1; else e1 = w2+1;
                if (kov == NULL) {
                    kov = ekov;
                    continue;
                }
                if (can_we_cast_kovs(ekov, kov) == ALWAYS_MATCH) continue;
                if (can_we_cast_kovs(kov, ekov) == ALWAYS_MATCH) {
                    kov = ekov;
                    continue;
                }
                world_object *ek, *kk;
                ek = kovko_get_kind(ekov); kk = kovko_get_kind(kov);
```

```
                if ((ek) && (kk)) {
                    while (ek) {
                        if (wo_of_kind(kk, ek)) break;
                        ek = ek->kind;
                        if (ek == kind_kind) ek = NULL;
                    }
                    if (ek) kov = kovko(ek);
                    else kov = kova(OBJECT_TY);
                    continue;
                }
                LOG("Entry kov mismatch entry %d: expected $u, found $u\n",
                    ecount, kov, ekov);
                FAIL_PLOV;
            }
        }
    }
    if (compile_mode) return NULL;
    LOOP_OVER(ll, literal_list)
        if (ll->word_ref1 == w1)
            return kov_list_of(kov);
    ll = CREATE(literal_list);
    ll->word_ref1 = w1; ll->word_ref2 = w2; ll->entry_kov = kov;
    ll->no_entries = ecount;
    ll->make_flag = FALSE;
    ll->compiled_already = FALSE;
    ensure_basic_heap_present();
    return kov_list_of(kov);
}
void compile_literal_list(OUTPUT_STREAM, int incipit) {
    literal_list *ll;
    LOOP_OVER(ll, literal_list)
        if (ll->word_ref1 == incipit+1) {
            if (current_stack_frame()) {
                WRITE("ConstantListPointers-->%d", ll->allocation_id);
                ll->make_flag = TRUE;
            } else {
                WRITE("LIST_CONST_%d", incipit+1);
            }
        }
}
void compile_list_constant_allocations(OUTPUT_STREAM) {
    literal_list *ll;
    LOOP_OVER(ll, literal_list)
        if (ll->make_flag) {
            WRITE("pv = ConstantListPointers-->%d; ConstantListPointers-->%d = ",
                ll->allocation_id, ll->allocation_id);
            compile_heap_allocation(OUT, kov_list_of(ll->entry_kov), 1, TRUE);
            WRITE(";\n");
        }
}
void compile_list_constants(OUTPUT_STREAM) {
    literal_list *ll;
    LOOP_OVER(ll, literal_list)
```

```
        if (ll->compiled_already == FALSE) {
            WRITE("Array LIST_CONST_%d --> 0 %d %d ",
                ll->word_ref1, kov_I6_ID(ll->entry_kov), ll->no_entries);
            parse_list_of_values(OUT, ll->word_ref1, ll->word_ref2, TRUE);
            WRITE(";\n");
            ll->compiled_already = TRUE;
        }
}

void compile_ConstantListPointers_array(OUTPUT_STREAM) {
    literal_list *ll;
    WRITE("Array ConstantListPointers --> \n");
    LOOP_OVER(ll, literal_list) {
        WRITE("LIST_CONST_%d ", ll->word_ref1);
    }
    WRITE(" 0 0;\n");
}
```

The function parse_list_of_values is called from 5/lit.

The function compile_literal_list is called from 7/vasp.

The function compile_list_constant_allocations is called from 9/rsdt.

The function compile_list_constants is invoked by a command in a `.i6t` template file.

The function compile_ConstantListPointers_array is invoked by a command in a `.i6t` template file.

Tables 10/tab

Purpose

To manage and compile tables, which are two-dimensional arrays with associative look-up facilities provided at run-time.

Template interpreter commands

```
3   {-callv:traverse_to_create_tables}
3   {-callv:traverse_to_stock_tables}
3   {-callv:check_tables_for_kind_clashes}
8   {-callv:compile_max_score}
11   {-callv:compile_tables}
11   {-callv:compile_print_table_names}
11   {-callv:index_tables}
```

Definitions

¶1. Arrays of static data are held in columns, which belong to tables. Column names are, in effect, types, since they must be consistently typed even across different tables: if Table A and Table B each have a column called "topmost point", then the contents must have a consistent type in both tables. (This enables the name alone to guarantee the type, and also allows for the tables to be linked.)

The predicate calculus engine often finds conditions equivalent to "if A is a C listed in T": it is efficient to rewrite this in a special predicate *listed-in-C*(A, T), and to do this we need to have a binary predicate associated with each possible table column C.

```
define MAX_COLUMNS_PER_TABLE 20

typedef struct table_column {
    int word_ref1, word_ref2;                          name of column (without "entry" suffix)
    int column_id;                                column ID number (uniquely associated with name)
    struct kind_of_value *kov_stored_in_column;        what kind of value is stored in this column
    struct table *table_from_which_data_type_inferred;                    usually the earliest use
    struct binary_predicate *listed_in_predicate;                                      see above
    MEMORY_MANAGEMENT
} table_column;

typedef struct table {
    int table_no_w1, table_no_w2;                                        the table number (if any)
    int table_name_w1, table_name_w2;                                     the table name (if any)
    struct parse_node *table_created_at;                                  where created in source
    int no_columns;                                                                must be at least 1
    int blank_rows;                          number of entirely blank rows to be appended (may be 0)
    int blank_rows_for_each_thing;                                        add one blank for each thing
    int fill_in_blanks;                                 if set, fill any blank entries with default values
    int first_column_by_definition;                      if set, first column defines new value names
    int preserve_row_order_at_run_time;                                    if set, don't sort this table
    struct table_column *column_type[MAX_COLUMNS_PER_TABLE];
    struct parse_node *column_data[MAX_COLUMNS_PER_TABLE];
```

```
    struct table *continuation_of;                    /* if continuation of earlier table */
    struct table *amendment_of;                       /* if amendment of earlier table */
    char tab_I6_identifier[32];                       /* an I6 identifier */
    int approximate_array_space_needed;               /* at run-time, in words */
    MEMORY_MANAGEMENT
} table;
```

The structure table_column is private to this section.

The structure table is private to this section.

---

§**1.** Tables of data are created in two special passes through the source text: the first finds their names and registers them with the parser, while the second (much later) works out their columns and contents. At some point we also try to find ambiguity problems which might bite us later on.

§**2.** During the main assertions traverse, we simply skip over tables.

```
sentence_handler TABLE_SH_handler = { TABLE_NT, -1, 0, NULL };
```

§**3.** Because we use specialist traverses later on, when we can be certain that all named values are known.

```
void traverse_to_create_tables(void) {
    parse_node *p;
    for (TREE_START(p); p; TREE_NEXT(p))
        if (pn_get_node_type(p) == TABLE_NT)
            new_table(NULL, p);
}
void traverse_to_stock_tables(void) {
    table *t;
    LOOP_OVER(t, table) {
        current_sentence = t->table_created_at;
        examine_table(t);
    }
}
void check_tables_for_kind_clashes(void) {
    table *t;
    LOOP_OVER(t, table) {
        if (t->table_name_w1 >= 0) {
            world_object *k =
                parse_world_object(t->table_name_w1, t->table_name_w2, TRUE);
            if (k) {
                quote_source(1, new_nounphrase_worldly(t->table_name_w1, t->table_name_w2, FALSE));
                handmade_problem(_P_(C10TableCoincidesWithKind));
                issue_problem_segment(
                    "The table name %1 will have to be disallowed "
                    "as it is also the name of a kind, or of the plural "
                    "of a kind. For instance, writing 'Table of Rooms' is "
                    "disallowed - it could lead to great confusion.");
                issue_problem_end();
                return;
            }
        }
    }
}
```

The function traverse_to_create_tables is invoked by a command in a .i6t template file.
The function traverse_to_stock_tables is invoked by a command in a .i6t template file.
The function check_tables_for_kind_clashes is invoked by a command in a .i6t template file.

§**4. Table columns.** These associate column names with value specific types, in such a way that if one table has a column called "score" which is a number, then every table with a "score" column must also store numbers in it: this is to make relational links type-safe. (It's also necessary to make column names independently parsable from table names.)

```
table_column *new_table_column(int w1, int w2) {
    table_column *tc;
    meaning_list *ml;
    int i, kov1, kov2;
    kind_of_value *kov = NULL;
    if [[w1, w2 == ... OPENBRACKET ... CLOSEBRACKET : i --> w1, w2 ... kov1, kov2]] {
        specification *spec = parse_expression(kov1, kov2, TYPE_EXPCON);
        if (spec_is_generic_CONSTANT(spec))
            kov = spec_get_kind_of_value(spec);
        else {
            quote_source(1, new_nounphrase_worldly(kov1, kov2, FALSE));
            handmade_problem(_P_(C10TableColumnBrackets));
            issue_problem_segment(
                "Brackets are only allowed in table column names when giving "
                "the kind of value which will be stored in the column. So "
                "'poems (indexed text)' is legal, but not 'poems (chiefly "
                "lyrical)'.");
            issue_problem_end();
            kov = NULL;
        }
    }
    LOGIF(TABLE_CONSTRUCTION, "Table column name $W initial kov $u\n", w1, w2, kov);
    ml = SP_excerpt(TABLE_COLUMN_MC, w1, w2);
    if (ml) return RETRIEVE_POINTER_table_column(em_data(ml_meaning(ml)));
    tc = CREATE(table_column);
    tc->word_ref1 = w1; tc->word_ref2 = w2;
    register_excerpt_meaning(TABLE_COLUMN_MC, 0, w1, w2,
        STORE_POINTER_table_column(tc));
    register_reworded_meaning(TABLE_COLUMN_MC, 0, 0, 0, w1, w2, column_V,
        STORE_POINTER_table_column(tc));
    tc->kov_stored_in_column = kov;
    tc->table_from_which_data_type_inferred = NULL;
    tc->column_id = 100+tc->allocation_id;
    tc->listed_in_predicate = make_listed_in_predicate(tc);
    return tc;
}

void log_table_column(table_column *tc) {
    LOG("'$W'/", tc->word_ref1, tc->word_ref2);
    if (tc->kov_stored_in_column == NULL) LOG("typeless");
    else LOG("$u", tc->kov_stored_in_column);
}

void tc_set_kov(table_column *tc, table *t, kind_of_value *kov) {
    tc->kov_stored_in_column = kov;
    if (is_kova(kov, TEXT_ROUTINE_TY)) tc->kov_stored_in_column = kova(TEXT_TY);
```

```
    tc->table_from_which_data_type_inferred = t;
    LOGIF(TABLE_CONSTRUCTION, "Table column $C has type $u, according to $B\n",
        tc, tc->kov_stored_in_column, t);
    if (is_kova(kov, UNDERSTANDING_TY))
        supply_KOV_for_listed_in_tc(tc->listed_in_predicate, kova(SNIPPET_TY));
    else supply_KOV_for_listed_in_tc(tc->listed_in_predicate, kov);
}

binary_predicate *tc_get_listed_in_predicate(table_column *tc) {
    return tc->listed_in_predicate;
}

int tc_get_column_id(table_column *tc) {
    return tc->column_id;
}

kind_of_value *get_data_type_stored(table_column *tc) {
    return tc->kov_stored_in_column;
}
```

The function log_table_column is called from 2/dl.
The function tc_get_listed_in_predicate is called from 6/simp.
The function tc_get_column_id is called from 7/vasp and 10/libp.
The function get_data_type_stored is called from 6/simp and 7/stsp.

§5. Examining the entries of tables.

```
void examine_table(table *t) {
    int i;
    i = 0;
    if (t->first_column_by_definition) i = 1;
    for (; i<t->no_columns; i++) {
        parse_node *PN = t->column_data[i]->down;
        while (PN != NULL) {
            examine_table_cell(t, PN, i);
            PN = PN->next;
        }
        finish_this_session_of_parsing();
    }
    LOGIF(TABLE_CONSTRUCTION, "Table data is:\n");
    for (i=0; i<t->no_columns; i++) {
        kind_of_value *kov = t->column_type[i]->kov_stored_in_column;
        LOGIF(TABLE_CONSTRUCTION, "Column %d '$W' kov %s with data:\n$T",
            i, t->column_type[i]->word_ref1, t->column_type[i]->word_ref2,
            kov_get_NI_source_name(kov),
            t->column_data[i]);
        if ((is_kovcon(kov)) &&
            (is_kova(kovcon_get_base(kov), NO_ENTRIES_TY))) {
            table_problem(_P_(C10TableColumnEmptyLists),
                t, NULL, t->column_data[i],
                "In the table %1, the table column %3 seems to consist only "
                "of empty lists. This means that I can't tell what kind of value "
                "it should hold - are they to be lists of numbers, for "
                "instance, or lists of texts, or some other possibility? "
                "Either one of the entries must contain a non-empty list - "
                "so that I can deduce the answer by looking at what is in it - "
```

```
                "or else the column heading must say, e.g., by calling it "
                "'exceptions (list of texts)' with the kind of value in "
                "brackets after the name.",
                "TE15");
        }
    }
}
```

§**6.** Note that the data type of the contents of a table column is something which has to be deduced from observation of what is put into that column, and that it is possible for more than one table to use the same column name: where this happens, we attempt to reconcile the data type of the column to a type which is possible for all values in both tables, and this can in principle widen the type of a column.

```
void examine_table_cell(table *t, parse_node *P2, int col_count) {
    int i, k;
    [[i, k <-- P2]];
    if (i<0) return;
    current_sentence = P2;
    if [[i, k == DASHDASH]] {
        pn_set_evaluation(P2, new_UNKNOWN_spec());
        pn_annotate_int(P2, table_cell_unspecified_ANNOT, TRUE);
        return;
    }
    specification *spec = parse_expression(P2->word_ref1, P2->word_ref2, DESCRIPTIVE_TYPE_EXPCON);
    if (spec_is_UNKNOWN(spec)) {
        meaning_list *ml = SP_excerpt(QUANTITY_MC, P2->word_ref1, P2->word_ref2);
        if (ml) {
            quantity *q = RETRIEVE_POINTER_quantity(em_data(ml_meaning(ml)));
            if ((q) && (qty_is_a_variable(q))) {
                LOG("Error at: $X", pn_get_evaluation(P2));
                if (q == player_quantity) {
                table_problem(_P_(C10TablePlayerEntry),
                    t, NULL, P2,
                    "In the table %1, the entry %3 is the name of a value which "
                    "varies, not a constant%|, and can't be stored as a table entry. "
                    "This is because Inform allows the perspective of play "
                    "to change: 'player' is thus a person that varies, being the "
                    "one currently being played through. (You could always "
                    "write 'yourself' to mean the standard player-character "
                    "through which almost all Inform works are played almost "
                    "all of the time. If no changes of player are used in your "
                    "design, then this will work just as well.)",
                    "TE10");
                } else {
                table_problem(_P_(C10TableVariableEntry),
                    t, NULL, P2,
                    "In the table %1, the entry %3 is the name of a value which varies, "
                    "not a constant%|, and therefore can't be stored as a table entry.",
                    "TE10");
                }
                return;
            }
```

```
    }
}
pn_set_evaluation(P2, spec);
if ((species_is(pn_get_evaluation(P2), DESCRIPTION_SPC))
    && (number_of_adjectives_applied_to(pn_get_evaluation(P2)) == 1)
    && (spec_get_described_object(pn_get_evaluation(P2)) == NULL)
    && (spec_get_described_kind(pn_get_evaluation(P2)) == NULL)) {
    adjective_list_entry *fale = first_adjective_list_entry(pn_get_evaluation(P2));
    adjectival_phrase *aph = get_adjective_from_list_entry(fale);
    quantity *q = aph_has_ENUMERATIVE_meaning(aph);
    if (q) {
        specification *specv = new_QUANTITY_spec(q);
        LOGIF(TABLE_CONSTRUCTION, "Adjectival usage at table entry\n");
        pn_set_evaluation(P2, specv);
    }
}
LOGIF(TABLE_CONSTRUCTION, "Cell in col %d has type $S\n",
    col_count, pn_get_evaluation(P2));
if (spec_is_generic_CONSTANT(pn_get_evaluation(P2))) {
    pn_annotate_int(P2, table_cell_unspecified_ANNOT, TRUE);
}
if ((species_is(pn_get_evaluation(P2), DESCRIPTION_SPC)) &&
    (spec_get_described_kind(pn_get_evaluation(P2))) &&
    (number_of_adjectives_applied_to(pn_get_evaluation(P2)) == 0)) {
    int w1 = pn_get_evaluation(P2)->word_ref1,
        w2 = pn_get_evaluation(P2)->word_ref2;
    pn_set_evaluation(P2, new_generic_CONSTANT_type(kova(OBJECT_TY)));
    pn_get_evaluation(P2)->word_ref1 = w1;
    pn_get_evaluation(P2)->word_ref2 = w2;
    pn_annotate_int(P2, table_cell_unspecified_ANNOT, TRUE);
}
if ((species_is(pn_get_evaluation(P2), DESCRIPTION_SPC)) &&
    (number_of_adjectives_applied_to(pn_get_evaluation(P2)) > 0)) {
    LOG("Error at: $X", pn_get_evaluation(P2));
    table_problem(_P_(C10TableDescriptionEntry),
        t, NULL, P2,
        "In the table %1, the entry %3 is a general description of things "
        "with no definite value%|, and can't be stored as a table entry.",
        "TE10");
    return;
}
if (((species_is(pn_get_evaluation(P2), DESCRIPTION_SPC) == FALSE) &&
    (family_is(pn_get_evaluation(P2), VALUE_FMY) == FALSE)) ||
    (spec_is_phrasal(pn_get_evaluation(P2)))) {
    int w, linrow;
    int w1 = pn_get_evaluation(P2)->word_ref1,
        w2 = pn_get_evaluation(P2)->word_ref2;
    for (w=w1, linrow=0; w<=w2; w++) {
        kind_of_value *isal = is_a_literal(w, w);
        if (isal) linrow++; else linrow = 0;
        if (linrow > 1) goto UnknownEntryProblem;
        if ((isal == NULL) && ([[word w == or]] == FALSE)) {
            if (w1 >= 0) {
```

```
                specification *spec;
                [[w1, w2 == the ... --> w1, w2]];
                [[w1, w2 == action of ... --> w1, w2]];
                spec = parse_expression(w1, w2, CONDITION_EXPCON);
                LOG("Offending entry is <$W> with $S\n", w1, w2, spec);
                if (species_is(spec, TEST_ACTION_SPC)) {
                    coerce_TEST_ACTION_to_STORED_ACTION(spec);
                    pn_set_evaluation(P2, spec);
                    pn_get_evaluation(P2)->word_ref1 = w1;
                    pn_get_evaluation(P2)->word_ref2 = w2;
                    pn_annotate_int(P2, table_cell_allocated_ANNOT, TRUE);
                    goto Fixed;
                }
            }
            UnknownEntryProblem:
            LOG("Error at: $X", pn_get_evaluation(P2));
            table_problem(_P_(C10TableUnknownEntry),
                t, NULL, P2,
                "In the table %1, the entry %3 is not something "
                "I recognise%|, and should either be a value, or a "
                "kind (such as 'a number'), or a blank entry '--'.",
                "TE5");
            return;
        }
    }
    pn_set_evaluation(P2, new_generic_CONSTANT_type(kova(TEXT_TY)));
    pn_get_evaluation(P2)->word_ref1 = w1;
    pn_get_evaluation(P2)->word_ref2 = w2;
    Fixed: ;
}
if (t->column_type[col_count]->kov_stored_in_column == NULL) {
    tc_set_kov(t->column_type[col_count], t,
        spec_evaluates_to(pn_get_evaluation(P2)));
} else {
    kind_of_value *s1 = spec_evaluates_to(pn_get_evaluation(P2));
    kind_of_value *s2 = t->column_type[col_count]->kov_stored_in_column;
    if ((is_kovcon(s1)) && (is_kovcon(s2))) {
        if (is_kova(kovcon_get_base(s2), NO_ENTRIES_TY)) return;
        if (is_kova(kovcon_get_base(s1), NO_ENTRIES_TY)) {
            tc_set_kov(t->column_type[col_count], t, s2);
            return;
        }
    }
    if (is_kova(s2, UNDERSTANDING_TY)) s2 = kova(TEXT_TY);
    if (is_kova(s1, TEXT_ROUTINE_TY)) s1 = kova(TEXT_TY);
    if (is_kova(s2, TEXT_ROUTINE_TY)) s2 = kova(TEXT_TY);
    if (is_kova(s1, OBJECT_TY)) s1 = kova(OBJECT_TY);
    if (is_kova(s2, OBJECT_TY)) s2 = kova(OBJECT_TY);
    if (kov_compare(s1, s2) == FALSE) {
        kind_of_value *w = NULL;
        if (can_we_cast_kovs(s1, s2) == ALWAYS_MATCH)
            w = s2;
        if (can_we_cast_kovs(s2, s1) == ALWAYS_MATCH)
```

```
                w = s1;
            if (w == NULL) {
                LOG("Incompatible types: s1 = $u, s2 = $u\n", s1, s2);
                table_problem(_P_(C10TableIncompatibleEntry),
                    t, t->column_type[col_count], P2,
                    "In the table %1, the entry %3 is incompatible with others "
                    "in the column '%2'%|: the entries under a given column "
                    "name must be blanks or values of the same kind, and "
                    "this applies even to columns in different tables if they "
                    "have the same name.",
                    "TE6");
                return;
            }
            if (is_kova(t->column_type[col_count]->kov_stored_in_column,
                UNDERSTANDING_TY) == FALSE) {
                tc_set_kov(t->column_type[col_count], t, w);
            }
        }
    }
}
```

§**7. Creating individual tables.** The source text declaration of tables is not easy to parse, as tabs are significantly different from spaces or new-lines, for instance – so the ordinary rules about white space are suspended. Tabs divide entries in a row; new-lines divide rows in a paragraph; and the table is terminated by a paragraph break.

If a table is declared as

Table 12 - Chemical Elements

then it can be referred to elsewhere in the source either as "Table 12" or as "Table of Chemical Elements", so both excerpts are registered as meaningful. But it is legal to declare a table with only one of the two forms in any case.

```
table *new_table(table *t, parse_node *PN) {
    int w1 = PN->word_ref1, w2 = PN->word_ref2, i, j, k,
        row_count, col_count, tw1, tw2, connection;
    parse_node *P2;
    i = last_word_of_formatted_text(w1, w2, FALSE);
    if (i == w1) {
        sentence_problem(_P_(BelievedImpossible),          seems impossible, thanks to lexer fixes
            "this table does not strictly speaking start a paragraph",
            "and I'm afraid we need to speak strictly here. Even a comment "
            "coming before the start of the table is too much.");
        return NULL;
    }
    t = CREATE(table);
    t->table_created_at = current_sentence;
    t->blank_rows = 0;
    t->blank_rows_for_each_thing = FALSE;
    t->fill_in_blanks = FALSE;
    t->preserve_row_order_at_run_time = FALSE;
    t->first_column_by_definition = FALSE;
    tw1 = w1; tw2 = i;
```

```
connection = 0;
if [[tw1, tw2 == ... OPENBRACKET continued CLOSEBRACKET --> tw1, tw2]]
    connection = 1;
else if [[tw1, tw2 == ... OPENBRACKET amended CLOSEBRACKET --> tw1, tw2]]
    connection = 2;
j = is_word_intermediate(HYPHEN_V, tw1, tw2);
LOGIF(TABLE_CONSTRUCTION, "New table $W: j=%d: with text: $F\n",
    tw1, tw2, j, w1, w2);
t->table_no_w1 = -1; t->table_no_w2 = -1;
t->table_name_w1 = -1; t->table_name_w2 = -1;
if (j>=0) {
    t->table_no_w1 = tw1+1; t->table_no_w2 = j-1;
    t->table_name_w1 = j+1; t->table_name_w2 = tw2;
} else {
    if ([[tw1, tw2 == table ###]] && (vocab_test_flags(tw2, NUMBER_MC))) {
        t->table_no_w1 = tw1+1; t->table_no_w2 = tw2;
    } else if [[tw1, w2 == table of ...]] {
        t->table_name_w1 = tw1+2; t->table_name_w2 = tw2;
    } else {
        sentence_problem(_P_(C10TableMisnamed),
            "this does not start with a legal table name",
            "since a table is required either to have a number, or "
            "to be a table 'of' something (or both). For example: "
            "'Table 5', 'Table of Blue Meanies', and 'Table 2 - "
            "Submarine Hues' are all allowed, but 'Table concerning "
            "Pepperland' is not.");
        return NULL;
        t->table_no_w1 = tw1+1; t->table_no_w2 = tw2;
    }
}
LOGIF(TABLE_CONSTRUCTION, "Table no $W, name $W\n",
    t->table_no_w1, t->table_no_w2, t->table_name_w1, t->table_name_w2);
t->continuation_of = NULL;
t->amendment_of = NULL;
tw1 = t->table_name_w1;
tw2 = t->table_name_w2;
if (connection != 0) {
    table *t2, *connection_to = NULL;
    if (t->table_no_w1 >= 0) {
        LOOP_OVER(t2, table)
            if ((t2 != t) && (t2->table_no_w1 >= 0) &&
                (compare_word_range(t2->table_no_w1, t2->table_no_w2,
                t->table_no_w1, t->table_no_w2)))
                    connection_to = t2;
    }
    if ((connection_to == NULL) && (t->table_name_w1 >= 0)) {
        LOOP_OVER(t2, table)
            if ((t2 != t) && (t2->table_name_w1 >= 0) &&
                (compare_word_range(t2->table_name_w1, t2->table_name_w2,
                t->table_name_w1, t->table_name_w2)))
                    connection_to = t2;
    }
```

```
    if (connection_to == NULL) {
        sentence_problem(_P_(C10TableNotContinuation),
            "this does not seem to be a continuation or amendment",
            "since the name/number do not match any previously defined "
            "table. (Perhaps you've put the continuation or amendment "
            "before the original?)");
        return NULL;
    }
    if (connection == 1) t->continuation_of = connection_to;
    if (connection == 2) t->amendment_of = connection_to;
}
if ((t->continuation_of == NULL) && (t->amendment_of == NULL)) {
    table *t2, *duplicate_of = NULL;
    if (t->table_name_w1 < 0) {
        if (t->table_no_w1 >= 0)
            LOOP_OVER(t2, table)
                if ((t2 != t) && (t2->table_no_w1 >= 0) &&
                    (compare_word_range(t2->table_no_w1, t2->table_no_w2,
                    t->table_no_w1, t->table_no_w2)))
                        duplicate_of = t2;
    } else {
        LOOP_OVER(t2, table)
            if ((t2 != t) && (t2->table_name_w1 >= 0) &&
                (compare_word_range(t2->table_name_w1, t2->table_name_w2,
                t->table_name_w1, t->table_name_w2)))
                    duplicate_of = t2;
    }
    if (duplicate_of) {
        if (t->table_name_w1 >=0)
            quote_source(1, new_nounphrase_worldly(t->table_name_w1, t->table_name_w2, FALSE));
        else
            quote_source(1, new_nounphrase_worldly(t->table_no_w1, t->table_no_w2, FALSE));
        handmade_problem(_P_(C10TableNameDuplicate));
        issue_problem_segment(
            "The table name %1 will have to be disallowed as it is "
            "already the name of an existing table (of which it is not "
            "a continuation).");
        issue_problem_end();
        return NULL;
    }
    if (t->table_no_w1 >= 0) {
        LOGIF(TABLE_CONSTRUCTION, "Logging table name: table $W\n",
            t->table_no_w1, t->table_no_w2);
        register_reworded_meaning(TABLE_MC, 0, table_V, 0,
            t->table_no_w1, t->table_no_w2, 0, STORE_POINTER_table(t));
    }
    if (t->table_name_w1 >= 0) {
        specification *spec = parse_expression(t->table_name_w1, t->table_name_w2, TYPE_OR_VALUE_EXPCON);
        if (spec_is_UNKNOWN(spec) == FALSE) {
            quote_source(1, new_nounphrase_worldly(t->table_name_w1, t->table_name_w2, FALSE));
            handmade_problem(_P_(C10TableNameAmbiguous));
            issue_problem_segment(
                "The table name %1 will have to be disallowed as it is text "
```

```
                "which already has a meaning to Inform. For instance, "
                "creating the 'Table of Seven' would be disallowed because of "
                "the possible confusion with the number 'seven'.");
            issue_problem_end();
            return NULL;
        }
        LOGIF(TABLE_CONSTRUCTION, "Logging table name: table of $W\n",
            t->table_name_w1, t->table_name_w2);
        register_reworded_meaning(TABLE_MC, 0, table_V, of_V,
            t->table_name_w1, t->table_name_w2, 0, STORE_POINTER_table(t));
    }
}
t->approximate_array_space_needed = 0;
t->no_columns = 0;
t->blank_rows = 0;
t->blank_rows_for_each_thing = FALSE;
isn_compose_identifier(t->tab_I6_identifier, 'T', t->allocation_id,
    t->table_name_w1, t->table_name_w2);
if ([[w1, w2 == *** with ### blank row/rows]] &&
    (is_kova(is_a_literal(w2-2, w2-2), NUMBER_TY))) {
    t->blank_rows += last_literal_evaluated;
    PN->word_ref2 -= 4;
    w2 -= 4;
}
if [[w1, w2 == *** with blank row/rows for each thing]] {
    t->blank_rows_for_each_thing = TRUE;
    PN->word_ref2 -= 6;
    w2 -= 6;
}
LOGIF(TABLE_CONSTRUCTION, "Created: $B\n", t);
i++;
row_count = 0;
while (i <= w2) {
    col_count = 0;
    j = last_word_of_formatted_text(i, w2, FALSE);
    LOGIF(TABLE_CONSTRUCTION, "Row %d is $W\n", row_count, i, j);
    while (i <= j) {
        specification *col_header_meaning = NULL;
        k = last_word_of_formatted_text(i, j, TRUE);
        LOGIF(TABLE_CONSTRUCTION, "Row %d col %d is $W\n",
            row_count, col_count, i, k);
        switch(row_count) {
            case 0:
                if (col_count >= MAX_COLUMNS_PER_TABLE) {
                    current_sentence = PN;
                    table_problem(_P_(C10TableTooManyColumns),
                        t, NULL, PN,
                        "There are too many columns in the table %1 already, "
                        "so the column %3 cannot be accommodated.", "TE2");
                    return NULL;
                }
                if [[i, k == a/an/some/the]] {
```

```
                        current_sentence = PN;
                        LOG("Offending column meaning: $X", col_header_meaning);
                        table_problem(_P_(C10TableColumnArticle),
                            t, NULL, new_nounphrase_raw(i, k),
                            "In the table %1, the column name %3 cannot be used, "
                            "because there would be too much ambiguity arising "
                            "from its ordinary meaning as an article.", "TE13");
                        return NULL;
                    }
                    col_header_meaning = parse_expression(i, k, TYPE_EXPCON);
                    if ((spec_is_UNKNOWN(col_header_meaning) == FALSE) &&
                        (spec_is_CONSTANT_of_kova(col_header_meaning, PROPERTY_TY) == FALSE)
&&
                        (spec_is_actual_CONSTANT(col_header_meaning)) &&
                        (!([[i, k == topic]]))) {
                        current_sentence = PN;
                        LOG("Offending column meaning: $X", col_header_meaning);
                        table_problem(_P_(C10TableColumnAlready),
                            t, NULL, new_nounphrase_raw(i, k),
                            "In the table %1, the column name %3 cannot be used, "
                            "because it already means something else.", "TE12");
                        return NULL;
                    }
                    t->column_type[col_count] = new_table_column(i, k);
                    t->column_data[col_count] = new_nounphrase_raw(i, k);
                    if [[i, k == topic]]
                        tc_set_kov(t->column_type[col_count], t, kova(UNDERSTANDING_TY));
                    t->no_columns++;
                    break;
                default:
                    if (col_count >= t->no_columns) {
                        current_sentence = PN;
                        table_problem(_P_(C10TableRowFull),
                            t, NULL, PN,
                            "In the table %1, the entry %3 cannot be accommodated%|"
                            ", because its row is already full.", "TE3");
                        return NULL;
                    }
                    P2 = new_nounphrase_raw(i, k);
                    pn_annotate_int(P2, table_cell_unspecified_ANNOT, FALSE);
                    graft(P2, t->column_data[col_count]);
                    break;
            }
            col_count++;
            i = k+1;
        }
        while (col_count < t->no_columns) {
            P2 = new_nounphrase_raw(-1, -1);
            pn_annotate_int(P2, table_cell_unspecified_ANNOT, TRUE);
            graft(P2, t->column_data[col_count]);
            col_count++;
        }
        row_count++;
```

```
}
if (row_count < 2) {
    current_sentence = PN;
    table_problem(_P_(C10TableWithoutRows),
        t, NULL, PN, "The table %1 has no rows.", "TE8");
    return NULL;
}
if ((t->continuation_of) || (t->amendment_of)) {
    table *connected_to = t->continuation_of;
    if (connected_to == NULL) connected_to = t->amendment_of;
    int found_col[MAX_COLUMNS_PER_TABLE], take_from[MAX_COLUMNS_PER_TABLE];
    connected_to->blank_rows += t->blank_rows;
    for (i=0; i<connected_to->no_columns; i++)
        take_from[i] = -1;
    for (i=0; i<t->no_columns; i++)
        found_col[i] = -1;
    for (i=0; i<t->no_columns; i++)
        for (j=0; j<connected_to->no_columns; j++)
            if (t->column_type[i] == connected_to->column_type[j]) {
                found_col[i] = j;
                take_from[j] = i;
            }
    LOGIF(TABLE_CONSTRUCTION, "Making continuation table:\n");
    for (i=0; i<connected_to->no_columns; i++)
        LOGIF(TABLE_CONSTRUCTION, "%d ", take_from[i]);
    LOGIF(TABLE_CONSTRUCTION, "\n");
    for (i=0; i<t->no_columns; i++)
        LOGIF(TABLE_CONSTRUCTION, "%d ", found_col[i]);
    LOGIF(TABLE_CONSTRUCTION, "\n");
    for (i=0; i<t->no_columns; i++)
        if (found_col[i] == -1) {
            current_sentence = t->table_created_at;
            table_problem(_P_(C10TableContinuationAddsRows),
                t, NULL, PN,
                "The table %1 was said to be a continuation or amendment, "
                "but it contains columns not found in the original.", "TE9");
            return NULL;
        }
    if (t->continuation_of) {
        for (i=0; i<connected_to->no_columns; i++)
            if (take_from[i] >= 0) {
                for (PN = connected_to->column_data[i]->down;
                    PN && (PN->next); PN = PN->next) ;
                PN->next = t->column_data[take_from[i]]->down;
            } else {
                for (j=1; j<row_count; j++) {
                    P2 = new_nounphrase_raw(-1, -1);
                    pn_annotate_int(P2, table_cell_unspecified_ANNOT, TRUE);
                    graft(P2, connected_to->column_data[i]);
                }
            }
        DESTROY(t, table);
```

```
            return NULL;
        }
        if (t->amendment_of) {
            int mismatch = FALSE;
            if (connected_to->no_columns != connected_to->no_columns)
                mismatch = TRUE;
            for (i=0; i<connected_to->no_columns; i++)
                if (take_from[i] != i)
                    mismatch = TRUE;
            if (mismatch) {
                current_sentence = t->table_created_at;
                table_problem(_P_(C10TableAmendmentMisfit),
                    t, NULL, PN,
                    "The table %1 was said to be an amendment, "
                    "but its columns do not exactly match the original.", "TE9");
                return NULL;
            }
            LOGIF(TABLE_CONSTRUCTION, "Amendment table created pro tem\n");
        }
    }
    return t;
}
int tab_no_columns(table *t) {
    return t->no_columns;
}
int tab_no_rows(table *t) {
    parse_node *PN; int c=0;
    for (PN=t->column_data[0]->down; PN; PN=PN->next) c++;
    c += t->blank_rows;
    return c;
}
kind_of_value *table_column_data_type(table *t, int i) {
    if ((i<0) || (i>=t->no_columns))
        internal_error("tcdt for column out of range");
    return t->column_type[i]->kov_stored_in_column;
}
void log_table(table *t) {
    LOG("{%s}", t->tab_I6_identifier);
}
```

The function tab_no_columns is called from 9/rsdt.
The function tab_no_rows is called from 9/rsdt.
The function table_column_data_type is called from 9/rsdt.
The function log_table is called from 2/dl.

§8. **The maximum score and rankings table.** A special rule is that if a table is called "Rankings" and contains a column of numbers followed by a column of text, then it is used by the run-time scoring system. This can only happen if we declare it somehow in I6 code, which we do with the constant `RANKING_TABLE`. We also set the `MAX_SCORE` constant equal to the number in the bottom row of the table, which is assumed to be the score corresponding to successful completion and the highest rank.

```
void compile_max_score(OUTPUT_STREAM) {
    table *t;
    LOOP_OVER(t, table) {
        int tn1 = t->table_name_w1, tn2 = t->table_name_w2;
        if ([[tn1, tn2 == rankings]] &&
            (t->no_columns >= 2) &&
            (is_kova(t->column_type[0]->kov_stored_in_column, NUMBER_TY)) &&
            ((is_kova(t->column_type[1]->kov_stored_in_column, TEXT_TY)) ||
            (is_kova(t->column_type[1]->kov_stored_in_column, TEXT_ROUTINE_TY)))) {
            parse_node *PN = t->column_data[0]->down;
            while ((PN != NULL) && (PN->next != NULL)) PN = PN->next;
            WRITE("Constant RANKING_TABLE = %s;\n", t->tab_I6_identifier);
            if ((PN != NULL) && (max_score_quantity) &&
                (qty_has_initial_value_set(max_score_quantity) == FALSE))
                initialise_global_variable(max_score_quantity, pn_get_evaluation(PN), PN);
        }
    }
    if (qty_has_initial_value_set(max_score_quantity)) {
        WRITE("Constant MAX_SCORE = ");
        compile_quantity_initial(OUT, max_score_quantity);
        WRITE(";\n");
    }
}
```

The function compile_max_score is invoked by a command in a `.i6t` template file.

§9. Kinds defined by table.

```
sentence_handler DEFINED_BY_SH_handler =
    { SENTENCE_NT, DEFINED_BY_VB, 1, kind_defined_by_table };
void kind_defined_by_table(parse_node *pn) {
    specification *spec, *sts; table *t;
    int ltw1 = pn->down->next->next->word_ref1;
    int ltw2 = pn->down->next->next->word_ref2;
    int spw1 = pn->down->next->word_ref1;
    int spw2 = pn->down->next->word_ref2;
    parse_node *name_entry;
    LOGIF(TABLE_CONSTRUCTION, "I now want to define <$W> by table <$W>\n",
        spw1, spw2, ltw1, ltw2);
    sts = parse_expression(spw1, spw2, TYPE_EXPCON);
    spec = parse_expression(ltw1, ltw2, TYPE_EXPCON);
    if (spec_is_CONSTANT_of_kova(spec, TABLE_TY) == FALSE) {
        LOG("Error at: $X", spec);
        sentence_problem(_P_(C10TableUndefined),
            "you can only use 'defined by' in terms of a table",
            "which lists the value names in the first column.");
```

```
        return;
    }
    t = TABLE_spec_to_table(spec);
    if ((spec_is_UNKNOWN(sts)) && [[spw1, spw2 == kind/kinds of ...]]) {
        sts = parse_expression(spw1+2, spw2, TYPE_EXPCON);
    }
    LOGIF(TABLE_CONSTRUCTION, "I find that sts is $S\n", &sts);
    pcalc_prop *sprop = spec_get_proposition(sts);
    if ((sprop) && (prop_contains_quantifier(sprop))) {
        sentence_problem(_P_(C10TableOfQuantifiedKind),
            "you can't use 'defined by' a table while also talking about the "
            "number of things to be defined",
            "since that could too easily lead to contradictions. (So "
            "'Six doors are defined by the Table of Portals' is not allowed - "
            "suppose there are ten rows in the Table, making ten doors?)");
        return;
    }
    if (spec_is_generic_CONSTANT(sts)) {
        parse_node *data_entry;
        int i;
        kind_of_value *kov = spec_get_kind_of_value(sts);
        if (kov_is_built_in(kov)) {
            sentence_problem(_P_(C10TableOfBuiltInKind),
                "you can only use 'defined by' on new kinds of value",
                "since the built-in ones already have their instances set up.");
            return;
        }
        if (kov_is_uncertainly_defined(kov) == FALSE) {
            sentence_problem(_P_(C10TableOfExistingKind),
                "this is a kind of value which already has named values",
                "so it can't be defined by a table as well.");
            return;
        }
        tc_set_kov(t->column_type[0], t, kov);
        t->first_column_by_definition = TRUE;
        LOGIF(TABLE_CONSTRUCTION,
            "I now want to define kov $u by table <$W>\n",
            kov, ltw1, ltw2);
        for (name_entry = t->column_data[0]->down; name_entry;
            name_entry=name_entry->next) {
            LOGIF(TABLE_CONSTRUCTION, "So I want to create instance <$W>\n",
                name_entry->word_ref1, name_entry->word_ref2);
            specification *newname = parse_expression(name_entry->word_ref1, name_entry->word_ref2,
TYPE_OR_VALUE_EXPCON);
            if (spec_is_UNKNOWN(newname) == FALSE) {
                LOG("Offending SP: $X", newname);
                quote_source(1, current_sentence);
                quote_source(2, name_entry);
                quote_type_of(3, newname);
                quote_kov(4, kov);
                handmade_problem(_P_(C10TableCreatedClash));
                issue_problem_segment(
                    "You wrote %1, and one of the entries in the first column of "
```

```
                "that table is %2, which I ought to create as a new value of %4. "
                "But I can't do that: it already has a meaning (as %3).");
            issue_problem_end();
        }
        pcalc_prop *prop = prop_to_create_something(kov, name_entry->word_ref1, name_entry->word_ref2);
        prop_true_in_world_model(prop);
        pn_set_evaluation(name_entry, spec_copy(qty_get_initial_value(latest_quantity)));
    }
    for (i=1; i<t->no_columns; i++) {
        kind_of_value *st = NULL;
        parse_node *hunt_value;
        property_name *prn =
            value_property_name_ref(t->column_type[i]->word_ref1,
                t->column_type[i]->word_ref2);
        if (t->column_type[i]->kov_stored_in_column)
            st = t->column_type[i]->kov_stored_in_column;
        else {
            data_entry = t->column_data[i];
            hunt_value = data_entry->down;
            while ((hunt_value->next) && [[hunt_value == DASHDASH]])
                hunt_value = hunt_value->next;
            spec = parse_expression(hunt_value->word_ref1,
                hunt_value->word_ref2, TYPE_EXPCON);
            if (spec_is_generic_CONSTANT(spec)) st = spec_get_kind_of_value(spec);
            else st = spec_evaluates_to(spec);
            if (is_kova(st, OBJECT_TY)) st = kova(OBJECT_TY);
        }
        if (st == NULL) {
            LOG("Error at: $X", spec);
            table_problem(_P_(C10TableWithUnobviousColumn),
                t, t->column_type[i], NULL,
                "In the table %1, the column '%2' has no obvious kind "
                "of value%|, but this is obligatory for tables to "
                "define new values. Perhaps it uses a form of "
                "constant not yet established at the point in the "
                "source text where the '... defined by ...' sentence "
                "occurs?",
                "TE10");
            return;
        }
        if (is_kova(st, TEXT_ROUTINE_TY)) st = kova(TEXT_TY);
        prn_set_kind_of_value(prn, st);
        prop_true_in_world_model_about(prop_to_provide_property(prn),
            NULL, kov_as_spec(kov));
        pp_set_table_storage(latest_property_permission, t->allocation_id, i);
    }
    t->fill_in_blanks = TRUE;
    t->preserve_row_order_at_run_time = TRUE;
    return;
}
if (spec_object_exactly_described_if_any(sts)) {
    LOG("Error at: $X", sts);
    sentence_problem(_P_(C10TableDefiningObject),
```

```
        "you can only use 'defined by' to set up values and things",
        "as created with sentences like 'The tree species are "
        "defined by Table 1.' or 'Some men are defined by the Table "
        "of Eligible Bachelors.' - trying to define a single "
        "specific object, as here, is not allowed.");
    return;
}
if (species_is(sts, DESCRIPTION_SPC)) {
    world_object *wo = NULL; parse_node *data_entry; int i;
    parse_node *hunt_value;
    t = TABLE_spec_to_table(spec);
    for (name_entry = t->column_data[0]->down; name_entry;
        name_entry=name_entry->next) {
        if [[name_entry == DASHDASH]] {
            table_problem(_P_(C10TableWithBlankNames),
                t, NULL, name_entry,
                "In the table %1, one or more of the entries in the "
                "first column is marked as blank%|, and while that "
                "would normally be fine, here you are using to create "
                "named objects and values. That means entries in the "
                "the first column are used as new names, and a blank "
                "therefore makes no sense.",
                "TE11");
            return;
        }
        LOGIF(TABLE_CONSTRUCTION,
            "I now want to define instance <$W> as <$W>\nSubtree $T\n",
            name_entry->word_ref1, name_entry->word_ref2, ltw1, ltw2, pn->down->next);
        fix_articles_in_raw_NP(name_entry);
        make_assertion(name_entry, pn->down->next);
    }
    for (i=1; i<t->no_columns; i++) {
        kind_of_value *st = NULL;
        int w1, w2;
        property_name *prn =
            value_property_name_ref(t->column_type[i]->word_ref1,
                t->column_type[i]->word_ref2);
        if (t->column_type[i]->kov_stored_in_column)
            st = t->column_type[i]->kov_stored_in_column;
        else {
            data_entry = t->column_data[i];
            hunt_value = data_entry->down;
            while ((hunt_value->next) && [[hunt_value == DASHDASH]])
                hunt_value = hunt_value->next;
            spec = parse_expression(hunt_value->word_ref1,
                hunt_value->word_ref2, TYPE_EXPCON);
            if (spec_is_generic_CONSTANT(spec)) st = spec_get_kind_of_value(spec);
            else st = spec_evaluates_to(spec);
            if (is_kova(st, OBJECT_TY)) st = kova(OBJECT_TY);
        }
        if (st == NULL) {
            LOG("Error at: $X", sts);
            table_problem(_P_(C10TableWithUnobviousColumn2),
```

```
                t, t->column_type[i], NULL,
                "In the table %1, the column '%2' has no obvious kind "
                "of value%|, but this is obligatory for tables to "
                "define new things. Perhaps it uses a form of constant "
                "not yet established at the point in the source text "
                "where the '... defined by ...' sentence occurs?",
                "TE10");
            return;
        }
        if (is_kova(st, TEXT_ROUTINE_TY)) st = kova(TEXT_TY);
        prn_set_kind_of_value(prn, st);
        for (name_entry = t->column_data[0]->down,
            data_entry = t->column_data[i]->down; name_entry && data_entry;
            name_entry = name_entry->next, data_entry = data_entry->next) {
            spec = parse_expression(name_entry->word_ref1, name_entry->word_ref2,
                TYPE_EXPCON);
            if (spec_is_generic(spec)) {
                table_problem(_P_(C10TableEntryGeneric),
                    t, NULL, name_entry,
                    "In the table %1, the entry %3 is the name of a kind of value, "
                    "so it can't be the name of a new object.",
                    "TE10");
                return;
            }
            if (spec_get_described_kind(spec)) wo = spec_get_described_kind(spec);
            else wo = spec_object_exactly_described_if_any(spec);
            if (wo == NULL) {
                LOG("Error at: $X", sts);
                table_problem(_P_(C10TableDefiningNothing),
                    t, NULL, name_entry,
                    "In the table %1, the entry %3 seems not to have defined "
                    "a thing there, so perhaps the first column did not "
                    "consist of new names?",
                    "TE10");
                return;
            }
            prop_true_in_world_model_about(prop_to_provide_property(prn),
                wo->kind, NULL);
            w1 = data_entry->word_ref1;
            w2 = data_entry->word_ref2;
            if ((w1>=0) && (w2>=w1) &&
                ((w1!=w2) || ([[word w1 == DASHDASH]] == FALSE))) {
                pn_set_evaluation(data_entry,
                    parse_expression(w1, w2, TYPE_EXPCON));
                pn_set_node_type(data_entry, VALUE_NT);
                assert_property_value_from_property_subtree(prn, wo, NULL, data_entry);
            }
        }
    }
    return;
}
LOG("Error at: $X", sts);
sentence_problem(_P_(C10TableDefiningTheImpossible),
```

```
        "you can only use 'defined by' to set up values and things",
        "as created with sentences like 'The tree species are "
        "defined by Table 1.' or 'Some men are defined by the Table "
        "of Eligible Bachelors.'");
}
```

§10. Amending tables.

```
void amend_table(table *main_table, table *amendments) {
    parse_node *PN = NULL, *PN2, *PN3;
    int amend_row = 1;
    do {
        int col;
        for (PN3 = main_table->column_data[0]->down; PN3; PN3 = PN3->next)
            pn_annotate_int(PN3, row_amendable_ANNOT, TRUE);
        for (col = 0; col < main_table->no_columns; col++) {
            int i;
            for (i = 1, PN = amendments->column_data[col]->down;
                PN && (i < amend_row); i++, PN = PN->next) ;
            if (PN) {
                specification *amend_key = pn_get_evaluation(PN);
                LOG("On amend row %d, read $S: $T\n", amend_row, amend_key, PN);
                if (spec_is_actual_CONSTANT(amend_key) == FALSE) {
                    current_sentence = amendments->table_created_at;
                    sentence_problem(_P_(BelievedImpossible),                    caught earlier
                        "a key in the first column of an amendment table is not a constant",
                        "which makes the amendments impossible to carry through.");
                    return;
                }
                int match_row = -1;
                int matches = 0;
                int row;
                for (row = 1, PN2 = main_table->column_data[col]->down,
                    PN3 = main_table->column_data[0]->down;
                    PN2; row++, PN2 = PN2->next, PN3 = PN3->next) {
                    specification *main_key = pn_get_evaluation(PN2);
                    if (pn_int_annotation(PN3, row_amendable_ANNOT) == FALSE)
                        continue;
                    LOG("On row %d, read $S: $T\n", row, main_key, PN2);
                    if ((spec_is_actual_CONSTANT(main_key)) &&
                        (spec_compare_CONSTANT(amend_key, main_key))) {
                        matches++;
                        match_row = row;
                    } else pn_annotate_int(PN3, row_amendable_ANNOT, FALSE);
                }
                if (matches == 0) {
                    current_sentence = amendments->table_created_at;
                    sentence_problem(_P_(C10TableAmendmentMismatch),
                        "a row in the amendment table matched no equivalent "
                        "row of the original",
                        "which makes the amendments impossible to carry through.");
                    return;
                }
```

```
                if (matches > 1) {
                    if (col+1 < main_table->no_columns) continue;
                    current_sentence = amendments->table_created_at;
                    sentence_problem(_P_(BelievedImpossible),       with the current algorithm
                        "a row in the amendment table matched more than one "
                        "equivalent row in the original",
                        "which makes the amendments ambiguous to carry through.");
                    return;
                }
                splice_table_row(main_table, amendments, match_row, amend_row);
                break;
            }
        }
        amend_row++;
    } while (PN);
}

void splice_table_row(table *table_to, table *table_from, int row_to, int row_from) {
    int i;
    for (i=0; i<table_to->no_columns; i++) {
        parse_node *PN_TO, *PN_FROM;
        int row;
        for (row = 1, PN_TO = table_to->column_data[i]->down;
                PN_TO && (row < row_to); PN_TO = PN_TO->next, row++) ;
        for (row = 1, PN_FROM = table_from->column_data[i]->down;
                PN_FROM && (row < row_from); PN_FROM = PN_FROM->next, row++) ;
        if ((PN_TO) && (PN_FROM)) {
            pn_set_evaluation(PN_TO, pn_get_evaluation(PN_FROM));
            pn_annotate_int(PN_TO, table_cell_unspecified_ANNOT,
                pn_int_annotation(PN_FROM, table_cell_unspecified_ANNOT));
        } else internal_error("bad table row splice");
    }
}
```

§**11. Compiling tables.** At run time, the data in T is essentially stored as a table of column tables, one for each column. A column table begins with a word identifying the table column number (so that two columns both called "price" in different tables will have the same identifying value heading their column tables), together with special bits set to indicate that exotic types of data are stored inside. Thus if `T` has R rows and C columns, the column tables are found at `T-->1`, `T-->2`, ..., `T-->`C. Each column table `C` has its identifying number in `C-->1`, a pointer to its blank bits in `C-->2` and its data in `C-->3`, `C-->4`, ..., `C-->`$(R+2)$.

The contents of a cell are not just its value but also an indication of whether or not it is formally blank, which in effect makes it impossible to store in a single virtual machine word. In practice, though, it is inefficient to look up two parallel structures each time we want to access the cell. So a blank cell is represented as both the value `TABLE_NOVALUE`, chosen so that it is very unlikely ever to occur as a genuine table value, and also as a blank bit set in the blanks bitmap. This makes a negative check – that something is not blank – very quick, since only in the very rare case when the value does coincide with `TABLE_NOVALUE` do we need to check the blanks bitmap. A positive check necessarily takes longer, but this cannot be helped.

With some data types it is possible to prove that `TABLE_NOVALUE` cannot be a legal value, and then the bitmap can be dispensed with, and the column is flagged as not having one. This basically means enumerated types and also object numbers. (The latter are enumerated in the Z-machine but not in Glulx: however, the value chosen – currently `0xDEADCE11` – is such that it can never be

an object reference value in any story file smaller than 3.5 GB, and in practice I6 and I7 are not capable of generating story files above 2 GB in any case.)

Note that tables with only one entry are padded with dummy values `NULL` to prevent I6 from misreading the table declarations as requiring huge numbers of entries.

The following flags are also defined in `Tables.i6t` and must agree with the values given there.

```
define TB_COLUMN_SIGNED 0x4000
define TB_COLUMN_TOPIC 0x2000
define TB_COLUMN_DONTSORTME 0x1000
define TB_COLUMN_NOBLANKBITS 0x0800
define TB_COLUMN_CANEXCHANGE 0x0400
define TB_COLUMN_ALLOCATED 0x0200
```

```
void compile_tables(OUTPUT_STREAM) {
    int j, k, e, blanks_offset = 0, bit = 1, blb = 0, no_distinct_tables = 0;
    table *t; table_column *tc;
    BEGIN_COMPILATION_MODE;
    COMPILATION_MODE_EXIT(DEREFERENCE_POINTERS_CMODE);
    LOOP_OVER(t, table)
        if (t->amendment_of)
            amend_table(t->amendment_of, t);
    LOOP_OVER(t, table) {
        int words_used = 0;
        if (t->amendment_of) continue;
        if (t->blank_rows_for_each_thing)
            t->blank_rows += instance_count_for_kind(kind_thing);
        current_sentence = t->table_created_at;
        no_distinct_tables++;
        WRITE("Array %s ", t->tab_I6_identifier);
        if (t->no_columns == 1) {
            WRITE("--> 1 tab_%d_%d", t->allocation_id, 0);
        } else {
            WRITE("table ");
            for (j=0; j<t->no_columns; j++)
                WRITE(" tab_%d_%d", t->allocation_id, j);
        }
        WRITE(";\n");
        words_used += t->no_columns + 1;
        for (j=0; j<t->no_columns; j++) {
            int bits, blanks_offset_base, blank_bitmap_unnecessary;
            table_column *tc = t->column_type[j];
            parse_node *PN = t->column_data[j]->down;
            LOGIF(TABLE_CONSTRUCTION, "Compiling column: $C\n", tc);
            current_sentence = t->table_created_at;
            if (tc->kov_stored_in_column == NULL) {
                table_problem(_P_(C10TableKindlessColumn),
                    t, tc, NULL,
                    "In the table %1, the column called '%2' contains no values "
                    "and no indication of its kind%|, which means that I "
                    "am unable to know how to treat it later. If you would "
                    "like the column to be entirely blank of values at the "
                    "start of play, you should enter text describing its "
                    "kind as the top entry: 'a number', for instance, or "
```

```
        "'text'.", "TE1");
    tc_set_kov(tc, t, kova(NUMBER_TY));
}
blank_bitmap_unnecessary = FALSE;
bits = 0;
if (kov_can_exchange(tc->kov_stored_in_column))
    bits += TB_COLUMN_CANEXCHANGE;
if (kov_uses_signed_comparisons(tc->kov_stored_in_column))
    bits += TB_COLUMN_SIGNED;
if (kov_uses_pointer_values(tc->kov_stored_in_column))
    bits += TB_COLUMN_ALLOCATED;
if (is_kova(tc->kov_stored_in_column, UNDERSTANDING_TY))
    bits = TB_COLUMN_TOPIC;
if (kov_requires_blanks_bitmap(tc->kov_stored_in_column) == FALSE) {
    bits += TB_COLUMN_NOBLANKBITS;
    blank_bitmap_unnecessary = TRUE;
}
if (t->preserve_row_order_at_run_time) bits += TB_COLUMN_DONTSORTME;
WRITE("Array tab_%d_%d table $%04x ", t->allocation_id, j,
    t->column_type[j]->column_id + bits);
if (blank_bitmap_unnecessary) {
    WRITE("NULL ");
} else {
    WRITE("%d ", blanks_offset);
}
words_used += 2;
blanks_offset_base = blanks_offset;
e = 0;
while (PN != NULL) {
    current_sentence = PN;
    if (pn_int_annotation(PN, table_cell_unspecified_ANNOT)) {
        if (t->fill_in_blanks == FALSE) WRITE(" TABLE_NOVALUE");
        else {
            WRITE(" (");
            compile_default_value(OUT, tc->kov_stored_in_column,
                -1, -1, "table entry");
            WRITE(") ");
        }
    }
    else {
        WRITE(" (");
        if (is_kova(tc->kov_stored_in_column, UNDERSTANDING_TY))
            compile_understanding(OUT,
                pn_get_evaluation(PN)->word_ref1,
                pn_get_evaluation(PN)->word_ref2,
                TRUE);
        else
            spec_compile(OUT, pn_get_evaluation(PN));
        WRITE(") ");
    }
    PN = PN->next; e++; if ((e % 8) == 0) blanks_offset++;
    words_used++;
}
```

```
        for (k=0; k<t->blank_rows; k++) {
            WRITE(" TABLE_NOVALUE");
            e++; if ((e % 8) == 0) blanks_offset++;
            words_used++;
        }
        if ((e % 8) != 0) blanks_offset++;
        while (e++ < 1) { WRITE(" NULL"); words_used++; }
        WRITE(";\n");
        if (blank_bitmap_unnecessary) blanks_offset = blanks_offset_base;
    }
    t->approximate_array_space_needed = words_used;
}
WRITE("Array TB_Blanks -> ");
bit = 1; blanks_offset = 0; blb = 0;
LOOP_OVER(t, table) {
    WRITE("\n  ! For table %s\n  ", t->tab_I6_identifier);
    current_sentence = t->table_created_at;
    for (j=0; j<t->no_columns; j++) {
        table_column *tc = t->column_type[j];
        parse_node *PN = t->column_data[j]->down;
        if (kov_requires_blanks_bitmap(tc->kov_stored_in_column) == FALSE)
            continue;
        e = 0;
        while (PN != NULL) {
            if ((pn_int_annotation(PN, table_cell_unspecified_ANNOT))
                && (t->fill_in_blanks == FALSE))
                blb += bit;
            PN = PN->next;
            bit = bit*2;
            e++; if ((e % 8) == 0) {
                bit = 1; blanks_offset++; WRITE("$%02x ", blb); blb = 0;
            }
        }
        for (k=0; k<t->blank_rows; k++) {
            blb += bit;
            bit = bit*2;
            e++; if ((e % 8) == 0) {
                bit = 1; blanks_offset++; WRITE("$%02x ", blb); blb = 0;
            }
        }
        if ((e % 8) != 0) {
            bit = 1; blanks_offset++; WRITE("$%02x ", blb); blb = 0;
        }
        WRITE(" ! Column %d\n  ", j);
    }
}
WRITE("\n  ! End of table\n  NULL NULL;\n");
END_COMPILATION_MODE;
WRITE("Array TC_KOVs --> ");
LOOP_OVER(tc, table_column) {
    WRITE("%d %d %d ", tc->column_id, kov_I6_ID(tc->kov_stored_in_column),
        kov_I6_construct_ID(tc->kov_stored_in_column));
}
```

```
    WRITE("0 0;\n");
    WRITE("Array TableOfTables --> TheEmptyTable ");
    LOOP_OVER(t, table) {
        if (t->amendment_of) continue;
        WRITE("%s  ", t->tab_I6_identifier);
    }
    WRITE("0 0;\n");

    LOOP_OVER(t, table) {
        int w1, w2;
        table_get_identifying_name(t, &w1, &w2);
        note_VM_usage("table", w1, w2, NULL, t->approximate_array_space_needed, 0, FALSE);
    }
}

char *tab_get_I6_representation(table *t) {
    return t->tab_I6_identifier;
}

void compile_print_table_names(OUTPUT_STREAM) {
    table *t;
    WRITE("  TheEmptyTable: print \"(the empty table)\"; return;\n");
    LOOP_OVER(t, table) {
        int mw = t->table_no_w2;
        if (t->amendment_of) continue;
        if (t->table_name_w2 > mw) mw = t->table_name_w2;
        WRITE("  %s: print \"", t->tab_I6_identifier);
        print_raw_text_to_file(t->table_created_at->word_ref1, mw, OUT);
        WRITE("\"; return;\n");
    }
}

void index_tables(void) {
    int tc = 0; table *t;
    LOOP_OVER(t, table) { tc++; }
    if (tc == 0) return;
    INDEX("<p><hr><p><b>List of Tables</b> (<i>About tables</i>");
    index_doc_link("TABLES"); INDEX(")<p>\n");

    begin_plain_html_table(ifl);
    LOOP_OVER(t, table) {
        int mw = t->table_no_w2;
        if (t->amendment_of) continue;
        first_html_column(ifl, 0);
        if (t->table_name_w2 > mw) mw = t->table_name_w2;
        print_raw_text_to_file(t->table_created_at->word_ref1, mw, ifl);
        index_link(lw_array[t->table_created_at->word_ref1].lw_source);
        next_html_column(ifl, 0);
        INDEX("%d column%s, %d row%s",
            t->no_columns, (t->no_columns == 1)?"":"s",
            tab_no_rows(t), (tab_no_rows(t) == 1)?"":"s");
        if (t->blank_rows > 0) INDEX(" (%d blank)", t->blank_rows);
        end_html_row(ifl);
        INDEX("\n");
    }
    end_html_table(ifl);
    INDEX("<p>\n");
```

```
}
void table_get_identifying_name(table *t, int *tw1, int *tw2) {
    if (t->table_no_w1 >= 0) { *tw1 = t->table_no_w1; *tw2 = t->table_no_w2; }
    else { *tw1 = t->table_name_w1; *tw2 = t->table_name_w2; }
}
```

The function compile_tables is invoked by a command in a `.i6t` template file.

The function tab_get_I6_representation is called from 9/rsdt and 7/vasp.

The function compile_print_table_names is invoked by a command in a `.i6t` template file.

The function index_tables is invoked by a command in a `.i6t` template file.

The function table_get_identifying_name is called from 2/prob3.

Equations 10/eqns

Purpose

To manage and compile equations, which relate numerical quantities.

Template interpreter commands

```
1   {-callv:traverse_to_create_equations}
6   {-callv:traverse_to_stock_equations}
47   {-callv:compile_equations}
68   {-callv:index_equations}
```

Definitions

¶1. POWERS OF INTEGERS; TEST PROBLEMS

¶2. As with tables, equations are detected early on in Inform's run but not parsed for their contents until later, so we store several word ranges. Also as with tables, each can have a number, a name or both.

```
typedef struct equation {
    int word_ref1, word_ref2;                          the text of the actual equation
    int equation_no_w1, equation_no_w2;                the equation number (if any)
    int equation_name_w1, equation_name_w2;            the equation name (if any)
    int where_w1, where_w2;                            declaration of symbols
    int usage_w1, usage_w2;                            usage notes
    struct parse_node *equation_created_at;            where created in source
    int examined_already;
    struct equation_node *parsed_equation;             and the equation itself (when eventually parsed)
    char eqn_I6_identifier[32];                        an I6 identifier
    struct equation_symbol *symbol_list;               the symbols used
    MEMORY_MANAGEMENT
} equation;
```

The structure equation is private to this section.

¶3. Each equation is allowed to use one or more symbols. Some may correspond to local variables in surrounding code from time to time, but others will be constants, and it's better to think of these as placeholders in the syntax of the equation, not as storage objects like variables. For instance, in

$$E = mc^2$$

we have three symbols: E, m and c. (The 2 does not count.) There might be symbols called m in any number of other equations; if so each instance has its own `equation_symbol` structure.

```
typedef struct equation_symbol {
    char variable_name[16];              C string form of name of the var (not done with words)
    int variable_wn;                                          word number of this name
    struct kind_of_value *var_kov;                  if a variable – must be quasinumerical
    struct specification *var_const;                      if a symbol for a constant value
    int temp_constant;                          is this constant a substitution for one usage only?
    struct equation_symbol *next;                          in the list belonging to the equation
    int local_map;                  local variable number, when being solved in a given stack frame
    MEMORY_MANAGEMENT
} equation_symbol;
```

The structure equation_symbol is private to this section.

¶4. When parsed, the equation is stored as a tree of `equation_node` structures. As usual, the leaves represent symbols or else constants not given symbol status (such as the 2 in $E = mc^2$); the non-leaf nodes represent operations, identified with the same codes as used in "Dimensions" in Chapter 7. Note that the equals sign `=` is itself considered an operation here.Thus:

```
OPERATION_EQN =
    SYMBOL_EQN E
    OPERATION_EQN *
        SYMBOL_EQN m
        OPERATION_EQN ^
            SYMBOL_EQN c
            CONSTANT_EQN 2
```

```
define CONSTANT_EQN 1          a leaf, representing a quasinumerical constant not given a symbol
define SYMBOL_EQN 2                                                  a leaf, representing a symbol
define OPERATION_EQN 3                                       a non-leaf, representing an operation
```

¶5. However, because of the algorithm used to parse the text of the equation into this tree, we also need certain other kinds of node to exist during parsing only. They are syntactic gimmicks, and are forbidden in the final tree.

```
define OPEN_BRACKET_EQN 4
define CLOSE_BRACKET_EQN 5
define END_EQN 6                                  the end (left or right edge, really) of the equation
```

¶6. Another temporary trick in parsing is to distinguish between explicit multiplication, where the source text uses an asterisk `*`, and implicit, as between m and c^2 in $E = mc^2$. We distinguish these so that they can bind with different tightnesses, but both are represented just as `TIMES_OPERATION` nodes in the eventual tree.

```
define IMPLICIT_TIMES_OPERATION 100
```

¶7. And now the equation node structure:

```
define MAX_EQN_ARITY 2                                   at present all operations are at most binary

typedef struct equation_node {
    int eqn_type;                                                              one of the *_EQN values
    int eqn_operation;                                  one of the *_OPERATION values (see "Dimensions.w")
    int enode_arity;                                                                    0 for a leaf
    struct equation_node *enode_operands[MAX_EQN_ARITY];                             the operands
    struct specification *leaf_constant;                                              if e.g. "21"
    struct equation_symbol *leaf_symbol;                                              if e.g. "G"
    struct kind_of_value *enode_kov;                                  result of evaluating this node
    MEMORY_MANAGEMENT
} equation_node;
```

The structure equation_node is private to this section.

---

§**1. Traversing for equations.** Early in Inform's run, the following takes place:

```
sentence_handler EQUATION_SH_handler = { EQUATION_NT, -1, 0, NULL };

void traverse_to_create_equations(void) {
    parse_node *p;
    for (TREE_START(p); p; TREE_NEXT(p))
        if (pn_get_node_type(p) == EQUATION_NT)
            new_equation(p->word_ref1, p->word_ref2, FALSE);
}
```

The function traverse_to_create_equations is invoked by a command in a `.i6t` template file.

§**2.** The above catches all of the named expressions written out in the source text, but not the ones written "inline", in phrases like

> let F be given by F = ma;

Those equations are created by calling `new_equation` direct from the meaning list converter, when the phrase is parsed: such equations are called "anonymous", as they have no name. But in either case, an equation begins here:

```
equation *new_equation(int w1, int w2, int anonymous) {
    equation *eqn;
    LOOP_OVER(eqn, equation)
        if (eqn->equation_created_at == current_sentence)
            return eqn;
    eqn = CREATE(equation);
    eqn->equation_created_at = current_sentence;
    eqn->where_w1 = -1; eqn->where_w2 = -1;
    eqn->usage_w1 = -1; eqn->usage_w2 = -1;
    eqn->parsed_equation = NULL;
    eqn->symbol_list = NULL;
    eqn->examined_already = FALSE;
    isn_compose_identifier(eqn->eqn_I6_identifier, 'E', eqn->allocation_id,
        eqn->equation_name_w1, eqn->equation_name_w2);
    int num1 = -1, num2 = -1, name1 = -1, name2 = -1;
    if (anonymous == FALSE) {
```

```
    ⟨Parse the equation's number and/or name 3⟩;
    ⟨Register any names for this equation 4⟩;
    int i, v1, v2;
    if [[w1, w2 == ... where ... : i --> w1, w2 ... v1, v2]]
        eqn_set_wherewithal(eqn, v1, v2);
}
eqn->equation_no_w1 = num1; eqn->equation_no_w2 = num2;
eqn->equation_name_w1 = name1; eqn->equation_name_w2 = name2;
[[w1, w2 == ... COMMA --> w1, w2]];
eqn->word_ref1 = w1; eqn->word_ref2 = w2;
return eqn;
}
```

The function new_equation is called from 5/mlc.

§**3.** We take the word range (w_1, w_2) and shave off the first line, that is, all the words up to the first line break occurring between words. (Compare the syntax for a table declaration.) This becomes the word range (tw_1, tw_2). We know that this begins with the word "equation", or we wouldn't be here (because the sentence would not have been classed an EQUATION_NT).

⟨Parse the equation's number and/or name 3⟩ ≡

```
int i, j, tw1, tw2;
i = last_word_of_formatted_text(w1, w2, FALSE);
tw1 = w1; tw2 = i; w1 = i+1;
j = is_word_intermediate(HYPHEN_V, tw1, tw2);
if [[tw1, tw2 == ... HYPHEN ... : j --> num1, num2 ... name1, name2]] {
    num1++; if (num1 > num2) { num1 = -1; num2 = -1; }          skip the word "equation"
} else {
    num1 = tw1+1; num2 = tw2;
}
if ((num1 >= 0) &&
    (!([[num1, num2 == ###]] && (vocab_test_flags(num1, NUMBER_MC))))) {
    LOG("num = <$W>, name = <$W>\n", num1, num2, name1, name2);
    sentence_problem(_P_(C10EquationMisnumbered),
        "the top line of this equation declaration seems not to be a "
        "legal equation number or name",
        "and should read something like 'Equation 6', or 'Equation - "
        "Newton's Second Law', or 'Equation 41 - Coulomb's Law'.");
    return NULL;
}
```

This code is used in §2.

§**4.** An equation can be referred to by its number, or by its name. Thus

Equation 64 - Distribution of Cheese

could be referred to elsewhere in the text by any of three names:

equation 64, Distribution of Cheese, Distribution of Cheese equation

⟨Register any names for this equation 4⟩ ≡

```
if (num1 >= 0)
    register_reworded_meaning(EQUATION_MC, 0, equation_V, 0,
        num1, num2, 0, STORE_POINTER_equation(eqn));
if (name1 >= 0) {
    specification *spec = parse_expression(name1, name2, TYPE_OR_VALUE_EXPCON);
    if (spec_is_UNKNOWN(spec) == FALSE) {
        quote_source(1, new_nounphrase_worldly(name1, name2, FALSE));
        handmade_problem(_P_(C10EquationMisnamed));
        issue_problem_segment(
            "The equation name %1 will have to be disallowed as it is text "
            "which already has a meaning to Inform. For instance, creating "
            "an equation called 'Equation - 2 + 2' would be disallowed "
            "because Inform would read '2 + 2' as arithmetic, not a name.");
        issue_problem_end();
    } else {
        register_excerpt_meaning(EQUATION_MC, 0, name1, name2,
            STORE_POINTER_equation(eqn));
        register_reworded_meaning(EQUATION_MC, 0, 0, 0, name1, name2, equation_V,
            STORE_POINTER_equation(eqn));
    }
}
```

This code is used in §2.

§**5.** A "where" clause following an equation defines its symbols, as we shall see. That can be detected in the above parsing process where the equation is displayed, but for anonymous equations occurring inline, the S-parser has to discover it; and the meaning list converter then calls this routine:

```
void eqn_set_wherewithal(equation *eqn, int w1, int w2) {
    eqn->where_w1 = w1; eqn->where_w2 = w2;
}
```

The function eqn_set_wherewithal is called from 5/mlc.

§**6. Parsing equations.** So now it's later on. We can run through all the equations displayed in the source text:

```
void traverse_to_stock_equations(void) {
    equation *eqn;
    LOOP_OVER(eqn, equation) {
        current_sentence = eqn->equation_created_at;
        examine_equation(eqn);
    }
}
```

The function traverse_to_stock_equations is invoked by a command in a `.i6t` template file.

§**7.** And, as with creation, `examine_equation` is called explicitly in the meaning list converter when an equation is found inline. So in all cases, we call the following before we need to use the equation, which runs a three-stage process: parsing the "where..." clause to declare the symbols, then parsing the equation, then type-checking it.

```
void examine_equation(equation *eqn) {
    if (eqn->examined_already) return;
    eqn->examined_already = TRUE;
    if (eqn_declare_symbols(eqn) == FALSE) return;
    eqn->parsed_equation = eqn_parse(eqn);
    LOG("PARSED AS:\n");
    log_equation_node(eqn->parsed_equation);
    if (eqn->parsed_equation == NULL) return;
    if (eqn_typecheck(eqn) == FALSE) log_equation_node(eqn->parsed_equation);
}
```

The function examine_equation is called from 5/mlc.

§**8. Declaring symbols.** For a displayed equation, the following parses the "where..." text, which is expected to declare every symbol occurring in it.

```
int eqn_declare_symbols(equation *eqn) {
    if (eqn->where_w1 < 0) return TRUE;
    int result = eqn_declare_variables_inner(eqn, eqn->where_w1, eqn->where_w2, FALSE);
    equation_symbol *ev;
    for (ev = eqn->symbol_list; ev; ev = ev->next)
        if (ev->var_kov == NULL) {
            if (ev->next == NULL) {
                equation_symbol_problem(_P_(BelievedImpossible),
                    eqn, eqn->where_w1, eqn->where_w2,
                    "each symbol in a equation has to be declared with a kind of "
                    "value or else an actual value. So '...where N = 1701.' or "
                    "'...where N, M are numbers.' would be fine.");
                result = FALSE;
            } else {
                ev->var_kov = ev->next->var_kov;
                ev->var_const = ev->next->var_const;
            }
        }
    return result;
}
```

§**9.** But the following routine is also used with the "where" text supplied in a phrase like so:

> let F be given by Newton's Second Law, where m = 101kg;

In this context the "where" text sets explicit values for symbols occurring in the equation; these are temporary settings only and will not change the equation's behaviour elsewhere.

So the following is called in either permanent mode, when it declares symbols for an equation, or temporary mode, when it gives them temporary assignments. It returns `TRUE` if all went well, or `FALSE` if problem messages had to be issued.

The syntax does in fact require every symbol to be declared as an actual or generic value, and the case where `const_flag` is `NOT_APPLICABLE` arises only when the list is abbreviated thus:

> where F is a force, a = 9.801 m/ss, m1 and m2 are masses;

This is split into four clauses: "F is a force" (variable), "a = 9.801 m/ss" (constant), "m1" (neither) and "m2 are masses" (variable). In a `NOT_APPLICABLE` case, which is allowed only in permanent declarations, the symbol is given the same definition as the one following it – so m1 becomes defined as a mass, too.

```
int eqn_declare_variables_inner(equation *eqn, int w1, int w2, int temp) {
    if (is_list_divided(w1, w2, LOOK_FOR_AND)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2, result = TRUE;
        if (eqn_declare_variables_inner(eqn, lw1, lw2, temp) == FALSE) result = FALSE;
        if (eqn_declare_variables_inner(eqn, rw1, rw2, temp) == FALSE) result = FALSE;
        return result;
    }
    specification *spec = NULL;
    kind_of_value *kov = NULL;
    int x1 = -1, x2 = -1;                          word range of the declared value or kind of value
    int const_flag = FALSE;              declared as a constant (TRUE), a variable (FALSE) or neither?
    ⟨Identify the clause as having the syntax for a constant or variable symbol 10⟩;
    if (const_flag != NOT_APPLICABLE)
        ⟨Find the actual value, or kind of value, which the symbol is to match 11⟩;
    if (temp) ⟨Assign the given value to this symbol on a temporary basis 12⟩
    else eqn_add_symbol(eqn, w1, kov, spec);
    return TRUE;
}
```

§**10.**

⟨Identify the clause as having the syntax for a constant or variable symbol 10⟩ ≡

```
int i;
if [[w1, w2 == ... is ... : i --> w1, w2 ... x1, x2]] const_flag = FALSE;
else if [[w1, w2 == ... are ... : i --> w1, w2 ... x1, x2]] const_flag = FALSE;
else if [[w1, w2 == ... EQUALS ... : i --> w1, w2 ... x1, x2]] const_flag = TRUE;
else if (w1 == w2) const_flag = NOT_APPLICABLE;
else {
    sentence_problem(_P_(C10EquationSymbolMisdeclared),
        "the symbols here are not declared properly",
        "and should each be declared with a kind of value or else an "
        "actual value.");
    return FALSE;
}
if (equation_symbol_legal(w1, w2) == FALSE) {
    equation_symbol_problem(_P_(C10EquationSymbolMalformed), eqn, w1, w2,
        "a symbol in a equation has to be a sequence of one to ten "
        "letters optionally followed by a number from 0 to 99, so "
        "'G', 'm', 'pi' and 'KE1' are all legal symbol names. But "
        "this one is not.");
    return FALSE;
}
```

This code is used in §9.

§**11.** Symbols are allowed to be set equal to kinds of value, but only quasinumerical ones; or to quasinumerical constants; or to global variables which contain quasinumerical values. The latter are included to make it easier for extensions to set up sets of equations for, say, gravity, defining

> The acceleration due to gravity is an acceleration that varies.

and thus letting the extension's user decide how strong gravity is, but still using it in equations:

> let F be given by Newton's Second Law, where a = the acceleration due to gravity;

⟨Find the actual value, or kind of value, which the symbol is to match 11⟩ ≡

```
spec = parse_expression(x1, x2, TYPE_OR_VALUE_EXPCON);
if (spec) {
    if (spec_is_generic_CONSTANT(spec)) {
        if (temp) {
            equation_symbol_problem(_P_(C10EquationSymbolVague), eqn, w1, w2,
                "when an equation is named for use in a 'let' "
                "phrase, any variables listed under 'where...' have "
                "to be given definite values, not just vaguely said "
                "to have particular kinds. Otherwise, I can't do any "
                "calculation with them.");
            return FALSE;
        }
        kov = spec_get_kind_of_value(spec);
        spec = NULL;
    } else kov = spec_evaluates_to(spec);
}
if ((spec) && (spec_is_actual_CONSTANT(spec) == FALSE) &&
    (spec_is_global_variable(spec) == FALSE)) {
```

```
        equation_symbol_problem(_P_(C10EquationSymbolNonValue), eqn, w1, w2,
            "this has neither a kind of value nor an actual value.");
        return FALSE;
    }
    if ((spec == NULL) && (const_flag)) {
        equation_symbol_problem(_P_(C10EquationSymbolEqualsKOV), eqn, w1, w2,
            "'is' should be used, not '=', for a kind of value rather "
            "than an actual value.");
        return FALSE;
    }
    if ((kov) && (kov_quasinumerical(kov) == FALSE)) {
        equation_symbol_problem(_P_(C10EquationSymbolNonNumeric), eqn, w1, w2,
            "this has a kind of value on which arithmetic cannot be done, "
            "so it can have no place in an equation.");
        return FALSE;
    }
```

This code is used in §9.

§**12.** At this point we know the user means the variable named at word `w1` to have the temporary value `spec`, and we have to identify that as one of the symbols:

⟨Assign the given value to this symbol on a temporary basis 12⟩ ≡

```
    equation_symbol *ev;
    for (ev = eqn->symbol_list; ev; ev = ev->next)
        if (strcmp(lw_array[w1].lw_rawtext, ev->variable_name) == 0) {
            if (kov_compare(kov, ev->var_kov) == FALSE) {
                equation_symbol_problem(_P_(C10EquationSymbolBadSub), eqn, w1, w2,
                    "you're using 'where' to substitute something into this "
                    "symbol which has the wrong kind of value.");
            }
            ev->temp_constant = TRUE;
            ev->var_const = spec;
            return TRUE;
        }
    equation_symbol_problem(_P_(C10EquationSymbolSpurious), eqn, w1, w2,
        "when 'where' is used to supply values to plug into a "
        "named equation as part of a 'let' phrase, you can only "
        "supply values for symbols actually used in that equation. "
        "This one doesn't seem to occur there.");
    return FALSE;
```

This code is used in §9.

§**13.** We won't want those temporary assignments hanging around, so once the hurly-burly is done, the following is called:

```
void eqn_remove_temp_variables(equation *eqn) {
    equation_symbol *ev;
    for (ev = eqn->symbol_list; ev; ev = ev->next)
        if (ev->temp_constant) {
            ev->var_const = NULL;
            ev->temp_constant = FALSE;
        }
}
```

§**14.** As we saw, permanent symbol declarations cause `eqn_add_symbol` to be called. But what about the symbols for an inline equation, like this one?

let F be given by F = ma;

These are not explicitly declared. What happens is that any local variable on the current stack frame, whose name could plausibly be that of a symbol, is made into one. Sometimes the locals won't be symbols in the equation at all, but will just have short names and coincidentally hold quasinumeric values; that doesn't matter, because if they're not in the equation, they'll never be used.

```
void equation_declare_local_variables(equation *eqn) {
    declare_local_names_in_current_stack_frame(eqn);
}
```

which calls the following for each current local variable in turn:

```
void equation_declare_local(equation *eqn, int w1, int w2, kind_of_value *kov) {
    if (w1 != w2) return;
    if ((equation_symbol_legal(w1, w2)) && (kov_quasinumerical(kov)))
        eqn_add_symbol(eqn, w1, kov, NULL);
}
```

The function equation_declare_local_variables is called from 5/mlc.
The function equation_declare_local is called from 12/phsf.

§**15.** And that about wraps up symbol declaration, except for the routine which actually declares symbols:

```
void eqn_add_symbol(equation *eqn, int w1, kind_of_value *kov, specification *spec) {
    equation_symbol *ev = CREATE(equation_symbol);
    strcpy(ev->variable_name, lw_array[w1].lw_rawtext);
    ev->var_kov = kov;
    ev->var_const = spec;
    ev->next = NULL;
    if (eqn->symbol_list == NULL) eqn->symbol_list = ev;
    else {
        equation_symbol *f = eqn->symbol_list;
        while ((f) && (f->next)) f = f->next;
        f->next = ev;
    }
    ev->variable_wn = w1;
    ev->temp_constant = FALSE;
    LOG("Declared <%s> with KOV $u, value $S\n", ev->variable_name, kov, spec);
}
```

§**16.** This is where the criterion for being a valid symbol name is expressed: it parses the regular expression `[A-Za-z]?{1,8}\d?{0,2}`.

```
int equation_symbol_legal(int w1, int w2) {
    if (w2 == w1) {
        char *p = lw_array[w1].lw_rawtext;
        int j, letters = 0, digits = 0, name_legal = TRUE;
        for (j=0; p[j]; j++) {
            unsigned int c = p[j];
            if (isdigit(c)) digits++;
            else if (isalpha(c)) { letters++; if (digits > 0) name_legal = FALSE; }
            else name_legal = FALSE;
            if (j >= 13) break;
        }
        if ((letters > 10) || (digits > 2)) name_legal = FALSE;
        return name_legal;
    }
    return TRUE;
}
```

§**17. Equation nodes.** The parsed equation is a tree full of nodes, so we need routines to make and examine them.

```
equation_node *enode_new(int t) {
    equation_node *enode = CREATE(equation_node);
    enode->eqn_type = t;
    enode->eqn_operation = -1;
    enode->enode_arity = 0;
    enode->leaf_constant = NULL;
    enode->leaf_symbol = NULL;
    enode->enode_operands[0] = NULL;
    enode->enode_kov = NULL;
    return enode;
}
```

§**18.** This is how we make the three kinds of enode permitted in the final compiled equation. (The other kinds can be created using `enode_new` directly.)

```
equation_node *enode_new_op(int op) {
    equation_node *enode = enode_new(OPERATION_EQN);
    enode->eqn_operation = op;
    return enode;
}
equation_node *enode_new_symbol(equation_symbol *ev) {
    equation_node *enode = enode_new(SYMBOL_EQN);
    enode->leaf_symbol = ev;
    return enode;
}
equation_node *enode_new_constant(specification *spec) {
    equation_node *enode = enode_new(CONSTANT_EQN);
    enode->leaf_constant = spec;
    return enode;
}
```

§**19.** Being able to log nodes is useful, if only because it's always pretty to watch shift-reduce parsers in action.

```
void log_equation_node(equation_node *tok) {
    log_equation_node_inner(tok, 0);
}
void log_equation_node_inner(equation_node *tok, int d) {
    int i;
    for (i=0; i<d; i++) if (i+1<d) LOG("    "); else LOG("+---");
    if (tok == NULL) LOG("<NULL>\n");
    else if (tok->eqn_type == OPERATION_EQN) {
        switch (tok->eqn_operation) {
            case PLUS_OPERATION: LOG("<add>"); break;
            case MINUS_OPERATION: LOG("<subtract>"); break;
            case DIVIDE_OPERATION: LOG("<divide>"); break;
            case TIMES_OPERATION: LOG("<multiply>"); break;
            case IMPLICIT_TIMES_OPERATION: LOG("<implicitly multiply>"); break;
            case EQUALS_OPERATION: LOG("<set equal>"); break;
            case ROOT_OPERATION: LOG("<square root>"); break;
            case CUBEROOT_OPERATION: LOG("<cube root>"); break;
            case POWER_OPERATION: LOG("<to the power>"); break;
            case UNARY_MINUS_OPERATION: LOG("<unary subtraction>"); break;
            default: LOG("<op-%d>", tok->eqn_operation); break;
        }
        LOG("\n");
        for (i=0; i<tok->enode_arity; i++)
            log_equation_node_inner(tok->enode_operands[i], d+1);
    } else if (tok->eqn_type == SYMBOL_EQN) LOG("<symbol-%s>\n", tok->leaf_symbol->variable_name);
    else if (tok->eqn_type == CONSTANT_EQN) LOG("<constant-$S>\n", tok->leaf_constant);
    else if (tok->eqn_type == OPEN_BRACKET_EQN) LOG("<open-bracket>\n");
    else if (tok->eqn_type == CLOSE_BRACKET_EQN) LOG("<close-bracket>\n");
    else if (tok->eqn_type == END_EQN) LOG("<end>\n");
    else LOG("<bad-eqn>\n");
}
```

§**20. Tokenising equations.** We break up the word range (w_1, w_2) into tokens of equation matter. Word boundaries divide tokens, but so do operators like `+`, and boundaries can also occur in runs of alphanumerics if we spot symbol names: thus `mv^21` will be divided into tokens `m`, `v`, `^`, `21`.

The following routine sends each token in turn to the shift/reduce parser below, encoding each token as an enode. We return `NULL` if a problem message has to be issued, or else a pointer to the parsed tree if we succeed.

define `MAX_ENODES_IN_EXPRESSION 100`

```
equation_node *eqn_parse(equation *eqn) {
    int w1 = eqn->word_ref1, w2 = eqn->word_ref2;
LOG("Parsing equation %d <$W>\n", eqn->allocation_id, w1, w2);
    enode_sr_start();                                          start the shift-reduce parser
    equation_node *previous_token = NULL;
    int enode_count = 0;                                       number of tokens shipped so far
    int bl = 0;                                                bracket nesting level
    int wn = w1, i = 0; char *p = NULL;
```

```
    while ((wn <= w2) || (p)) {
        if (p == NULL) { i = 0; p = lw_array[wn++].lw_rawtext; }
        we are now at character i in string p, while wn is the next word
        equation_node *token = NULL;
        ⟨Break off a token from the current position 21⟩;
        ⟨Issue the token to the shift-reduce parser 25⟩;
        previous_token = token;
        if (p[i] == 0) p = NULL;
    }
    if (enode_sr_token(eqn, enode_new(END_EQN)) == FALSE)
        ⟨Equation fails in the shift-reduce parser 26⟩;
    equation_node *result = enode_sr_result();
    if (bl != 0) {
        equation_problem(_P_(C10EquationBracketMismatch), eqn, "",
            "this seems to use brackets in a mismatched way, since there "
            "are different numbers of left and right brackets '(' and ')'.");
        return NULL;
    }
    log_equation_node(result);
    return result;
}
```

§**21.** Note that symbol names can't begin with a digit.

⟨Break off a token from the current position 21⟩ ≡

```
    unsigned int c = p[i];
    if (isalpha(c)) ⟨Break off a symbol name as a token 22⟩
    else if (isdigit(c)) ⟨Break off a numeric constant as a token 23⟩
    else ⟨Break off an operator or a piece of punctuation as a token 24⟩;
```

This code is used in §20.

§**22.** Note that symbols are identified by recognition: without knowing the identities of the symbols, the syntax alone wouldn't tell us how to break them. We can only break `mc^2` as `m` followed by `c^2` if we know that `m` and `c` are symbols, rather than `mc`. (This is one reason why most programming languages don't allow implicit multiplication.)

⟨Break off a symbol name as a token 22⟩ ≡

```
    char text_of_symbol[16];
    int j;                                   the length of the symbol name we try to break off
    for (j=1; (j<15) && (isalnum(p[i+j-1])) && (token == NULL); j++) {
        copy the first j characters into a C string:
        int k; for (k=0; k<j; k++) text_of_symbol[k] = p[i+k];
        text_of_symbol[j] = 0;
        try to identify this as one of the declared symbols:
        equation_symbol *ev;
        for (ev = eqn->symbol_list; ev; ev = ev->next)
            if (strcmp(text_of_symbol, ev->variable_name) == 0) {
                token = enode_new_symbol(ev);
                i += j;
                break;
            }
    }
```

```
if (token == NULL) {
    equation_problem(_P_(C10EquationTokenUnrecognised), eqn, text_of_symbol,
        "the symbol '%3' is one that I don't recognise. It doesn't "
        "seem to be declared after the equation - for instance, "
        "by adding 'where %3 is a number'.");
    return NULL;
}
```

This code is used in §21.

§**23.** The following is reliable because a string of digits not starting with a 0 is always a valid number to Inform unless it overflows the virtual machine's capacity; and so is the number 0 itself.

⟨Break off a numeric constant as a token 23⟩ ≡

```
int j;                                    again the length of the number we try to break off
if ((p[i] == '0') && (isdigit(p[i+1]))) {
    equation_problem(_P_(C10EquationLeadingZero), eqn, "",
        "a number in an equation isn't allowed to begin with a "
        "'0' digit, so an equation like 'M = 007+Q' is against the rules.");
    return NULL;
}
copy the digits into a C string, flanked by spaces:
char text_of_number[16];
text_of_number[0] = ' ';
for (j=1; (j<14) && (isdigit(p[i])); j++) text_of_number[j] = p[i++];
text_of_number[j++] = ' ';
text_of_number[j] = 0;
now sneakily add this to the word stream, and let the S-parser read it:
feed_into_lexer(text_of_number, FALSE, FALSE);
specification *spec =
    parse_expression(lexer_wordcount-1, lexer_wordcount-1, TYPE_EXPCON);
this can only go wrong if there was an overflow, and a problem will have been issued for that:
if (spec_is_actual_CONSTANT(spec) == FALSE) return NULL;
token = enode_new_constant(spec);
```

This code is used in §21.

§**24.** Which leaves just the easiest case:

⟨Break off an operator or a piece of punctuation as a token 24⟩ ≡

```
switch (c) {
    case '=': token = enode_new_op(EQUALS_OPERATION); break;
    case '+': token = enode_new_op(PLUS_OPERATION); break;
    case '-':
        if ((previous_token == NULL) ||
            (previous_token->eqn_type == OPERATION_EQN) ||
            (previous_token->eqn_type == OPEN_BRACKET_EQN))
            token = enode_new_op(UNARY_MINUS_OPERATION);
        else
            token = enode_new_op(MINUS_OPERATION); break;
    case '/': token = enode_new_op(DIVIDE_OPERATION); break;
    case '*': token = enode_new_op(TIMES_OPERATION); break;
    case '^': token = enode_new_op(POWER_OPERATION); break;
    case '(': token = enode_new(OPEN_BRACKET_EQN); bl++; break;
```

```
        case ')': token = enode_new(CLOSE_BRACKET_EQN); bl--; break;
        default: {
            char symbol[2]; symbol[0] = c; symbol[1] = 0;
            equation_problem(_P_(C10EquationOperatorUnrecognised), eqn, symbol,
                "the symbol '%3' is one that I don't recognise. I was "
                "expecting an arithmetic sign, '+', '-', '*','/', or '^', "
                "or else '=' or a bracket '(' or ')'.");
            LOG("Bad operator '%c'\n", c); return NULL;
        }
    }
    i++;
```

This code is used in §21.

§**25.** So now we have our next token, and are ready to ship it. But if we detect an implicit multiplication, for instance between `m` and `c^2` in `E=mc^2`, we issue that as an `IMPLICIT_TIMES_OPERATION` enode in between.

⟨Issue the token to the shift-reduce parser 25⟩ ≡

```
    if (multiplication_is_implied(previous_token, token)) {
        if (enode_sr_token(eqn, enode_new_op(IMPLICIT_TIMES_OPERATION)) == FALSE)
            ⟨Equation fails in the shift-reduce parser 26⟩;
        enode_count++;
    }
    if (enode_sr_token(eqn, token) == FALSE)
        ⟨Equation fails in the shift-reduce parser 26⟩;
    enode_count++;
    if (enode_count >= MAX_ENODES_IN_EXPRESSION - 2) {
        equation_problem(_P_(C10EquationTooComplex), eqn, "",
            "this is too long and complex an equation.");
        return NULL;
    }
```

This code is used in §20.

§**26.** In case any of the enode insertions fail. It's tricky to generate good error messages and recover well when an operator-precedence grammar fails to match in a parser like this, so we'll fall back on this:

⟨Equation fails in the shift-reduce parser 26⟩ ≡

```
    equation_problem(_P_(C10EquationMispunctuated), eqn, "",
        "this seems to be wrongly punctuated, and doesn't make sense as a "
        "mathematical formula.");
    return NULL;
```

This code is used in §20,25,20,25,20,25.

§**27.** Lastly, here is when multiplication is implied:

```
int multiplication_is_implied(equation_node *previous_token, equation_node *token) {
    int lt, rt;
    if ((token == NULL) || (previous_token == NULL)) return FALSE;
    lt = previous_token->eqn_type; rt = token->eqn_type;
    if (((lt == SYMBOL_EQN) || (lt == CONSTANT_EQN) || (lt == CLOSE_BRACKET_EQN)) &&
        ((rt == SYMBOL_EQN) || (rt == CONSTANT_EQN) || (rt == OPEN_BRACKET_EQN)))
        return TRUE;
    return FALSE;
}
```

§**28. The shift-reduce parser.** This is a classic algorithm for expression-evaluator grammars; see for instance Aho, Sethi and Ullman, *Compilers*, §4.6 in the second edition. We use a pair of stacks. The SR stack holds possible attempts to understand what we have so far, given the tokens that have arrived; the emitter stack holds nodes which form pieces of the output tree as it is assembled. Nodes flow from our input, are usually "shifted" onto the SR stack for a while, are eventually "reduced" in clumps taken off this stack and "emitted", then go onto the emitter stack, and are finally removed as they are made into trees.

The flow is therefore always forwards; tokens can't slosh back and forth between the stacks. On each iteration, at least one token makes progress, so if there are N tokens of input (including both end marker tokens) then we take at worst $2N$ steps to finish. Each stack can't need more than N entries, and N is bounded above by `MAX_ENODES_IN_EXPRESSION` plus 2 (allowing for the end markers). So:

```
int SR_sp = 0;
equation_node *SR_stack[MAX_ENODES_IN_EXPRESSION+2];
int emitter_sp = 0;
equation_node *emitter_stack[MAX_ENODES_IN_EXPRESSION+2];
void log_sr_stacks(void) {
    int i;
    LOG("SR: ");
    for (i=0; i<SR_sp; i++) { LOG(" %d: ", i); log_equation_node(SR_stack[i]); }
    LOG("EMITTER: ");
    for (i=0; i<emitter_sp; i++) { LOG(" %d: ", i); log_equation_node(emitter_stack[i]); }
}
```

§**29.** The start and finish are as follows. At the start, the emitter stack is empty and the SR stack contains an `END_EQN` token, which represents the left-hand end of the expression. (Another such token, this time representing the right-hand end, will be sent by the routines above at the end of the stream. So there will be two `END_EQN` tokens in play.)

```
void enode_sr_start(void) {
    SR_stack[0] = enode_new(END_EQN);
    SR_sp = 1;
    emitter_sp = 0;
}
```

§**30.** If we have succeeded, the end state of the emitter stack contains a single node: the head of the tree we have grown to represent the expression.

```
equation_node *enode_sr_result(void) {
    return emitter_stack[0];
}
```

§**31.** So the following is the routine which iteratively deals with tokens as they arrive. As noted above, the loop however ominous always terminates.

For proofs and explanations see ASU, but the idea is simple enough: as we see the expression a little at a time, we collect possibilities of how to read it on the SR-stack, until we reach a point where it's possible to tell what was meant; we then reduce the SR-stack by taking the winning possibility off the top and moving it to the emitter stack. For instance, if we have read `4 + 5` then we don't know yet whether the `+` will add the 4 to the 5; if the next token is `+` or `END_EQN` then it will, but if the next token is `*` then it won't, because we're looking at something like `4 + 5 * 6`.

If the next token is of lower precedence than `+` then we "reduce" – telling the emitter about the addition, which we now understand – but if it's higher, as with `*`, then we "shift", meaning, we postpone worrying about the addition and start worrying about the multiplication instead; our new problem, working out what `*` applies to, sits on top of the addition problem on the SR-stack.

```
int enode_sr_token(equation *eqn, equation_node *tok) {
    int safety_cutout = 3*MAX_ENODES_IN_EXPRESSION;
    while (TRUE) {
        if (SR_sp <= 0) internal_error("SR stack empty");
        if ((SR_stack[SR_sp-1]->eqn_type == END_EQN) && (tok->eqn_type == END_EQN)) break;
        if ((enode_lt(SR_stack[SR_sp-1], tok)) || (enode_eq(SR_stack[SR_sp-1], tok)))
            ⟨Shift an enode onto the SR-stack 32⟩
        else if (enode_gt(SR_stack[SR_sp-1], tok))
            ⟨Reduce some enodes from the SR-stack to the emitter stack 33⟩
        else return FALSE;
        if (safety_cutout-- < 0) internal_error("SR parser deadlocked");
    }
    if ((emitter_sp != 1) || (SR_sp != 1)) return FALSE;
    return TRUE;
}
```

§**32.** After shifting, we return a signal of success, which asks for the next token to be sent.

⟨Shift an enode onto the SR-stack 32⟩ ≡

```
    SR_stack[SR_sp++] = tok;
    return TRUE;
```

This code is used in §31.

§**33.** The ASU book is a little vague about what happens if there is an underflow here, I think because it's possible to set up the grammar such that an underflow cannot occur. But I can see no obvious proof that it will never occur for us given syntactically incorrect input, so we will return `FALSE` on an underflow to be safe.

Note that we can never emit the bottom-most token on the SR stack: that's the left-hand end marker, so can never be validly part of any arithmetic. So an underflow occurs if that's all that's left, i.e., when the SR stack pointer is 1, not 0.

⟨Reduce some enodes from the SR-stack to the emitter stack 33⟩ ≡

```
do { if (SR_sp <= 1) return FALSE;
     if (enode_emit(SR_stack[--SR_sp]) == FALSE) return FALSE;
} while ((SR_sp >= 1) && (enode_lt(SR_stack[SR_sp-1], SR_stack[SR_sp]) == FALSE));
```

This code is used in §31.

§**34.** The key point is that if nodes arrive at the SR parser in their ordinary order of mathematical writing, then they "reduce" off the SR stack and onto the emitter stack in reverse Polish notation order. Thus the sequence `4 + 2 * 7` is emitted as `4 2 7 * +`. RPN has no need of brackets to clarify the sequence of operation, and it's very easy to build a tree from.

```
int enode_emit(equation_node *tok) {
    switch (tok->eqn_type) {
        case SYMBOL_EQN: case CONSTANT_EQN:
            emitter_stack[emitter_sp++] = tok;
            break;
        case OPERATION_EQN:
            if (tok->eqn_operation == IMPLICIT_TIMES_OPERATION)
                tok->eqn_operation = TIMES_OPERATION;
            if (arithmetic_op_is_unary(tok->eqn_operation)) tok->enode_arity = 1;
            else tok->enode_arity = 2;
            int i;
            for (i = tok->enode_arity - 1; i >= 0; i--) {
                emitter_sp--;
                if (emitter_sp < 0) return FALSE;
                tok->enode_operands[i] = emitter_stack[emitter_sp];
            }
            emitter_stack[emitter_sp++] = tok;
            break;
    }
    return TRUE;
}
```

§**35.** All we need now is to decide the order of precedence of our tokens, though this isn't as simple as it looks, because they are not all symmetrical left-right. That's obviously true of things like an open bracket `(`, which affects the stuff to the left very differently from the stuff to the right. But it is also true of operators. `+` may be associative mathematically, but in computing there's a difference between evaluating `a + (b+c)` and `(a+b) + c`.

All of this means there's no simple order relationship on the tokens, where $T < S$ if and only if $S > T$. We order them using a numerical score, but they get one score $f(T)$ if they appear on the left and another score $g(T)$ if they appear on the right:

```
int enode_lt(equation_node *tok1, equation_node *tok2) {
    int f_left = f_function(tok1), g_right = g_function(tok2);
    if (f_left < g_right) return TRUE; return FALSE;
}
int enode_eq(equation_node *tok1, equation_node *tok2) {
    int f_left = f_function(tok1), g_right = g_function(tok2);
    if (f_left == g_right) return TRUE; return FALSE;
}
int enode_gt(equation_node *tok1, equation_node *tok2) {
    int f_left = f_function(tok1), g_right = g_function(tok2);
    if (f_left > g_right) return TRUE; return FALSE;
}
```

§**36.** And here are those scorings. Note that for the binary operators, f scores are always slightly higher than g scores: that's what makes them left associative, that is, $a+b+c$ is read as $(a+b)+c$. The exception to this is raising to powers: `a^2^3` evaluates a^8, not a^6, because it is read as `a^(2^3)`.

Implicit multiplication has higher precedence than explicit. This is actually to give it higher precedence than division (which has to have the same precedence as explicit multiplication), and is so that `ab/cd` evaluates $(ab)/(cd)$ rather than $a \cdot (b/c) \cdot d$.

```
int f_function(equation_node *tok) {
    switch (tok->eqn_type) {
        case SYMBOL_EQN: case CONSTANT_EQN: return 14;
        case OPERATION_EQN:
            switch (tok->eqn_operation) {
                case EQUALS_OPERATION: return 2;
                case PLUS_OPERATION: case MINUS_OPERATION: return 4;
                case TIMES_OPERATION: case DIVIDE_OPERATION: return 6;
                case IMPLICIT_TIMES_OPERATION: return 8;
                case POWER_OPERATION: return 9;
                case UNARY_MINUS_OPERATION: return 1;
            }
            internal_error("unknown operator precedence");
        case OPEN_BRACKET_EQN: return 0;
        case CLOSE_BRACKET_EQN: return 14;
        case END_EQN: return 0;
    }
    internal_error("unknown f-value"); return 0;
}
```

§**37.** And symmetrically:

```
int g_function(equation_node *tok) {
    switch (tok->eqn_type) {
        case SYMBOL_EQN: case CONSTANT_EQN: return 13;
        case OPERATION_EQN:
            if (tok) switch (tok->eqn_operation) {
                case EQUALS_OPERATION: return 1;
                case PLUS_OPERATION: case MINUS_OPERATION: return 3;
                case TIMES_OPERATION: case DIVIDE_OPERATION: return 5;
                case IMPLICIT_TIMES_OPERATION: return 7;
                case POWER_OPERATION: return 10;
                case UNARY_MINUS_OPERATION: return 12;
            }
            internal_error("unknown operator precedence");
        case OPEN_BRACKET_EQN: return 13;
        case CLOSE_BRACKET_EQN: return 0;
        case END_EQN: return 0;
    }
    internal_error("unknown g-value"); return 0;
}
```

§**38. Typechecking equations.** The SR parser can generate trees for any syntactically valid equation, but it may be something using = inappropriately or not at all. We rule that out first: we want the top node in the tree to be the unique = operator.

```
int eqn_typecheck(equation *eqn) {
    switch (enode_count_equals(eqn->parsed_equation)) {
        case 0:
            equation_problem(_P_(C10EquationDoesntEquate), eqn, "",
                "this equation doesn't seem to contain an equals sign, and "
                "without '=' there is no equating anything with anything.");
            return FALSE;
        case 1:
            if (enode_is_equals(eqn->parsed_equation) == FALSE) {
                equation_problem(_P_(C10EquationEquatesBadly), eqn, "",
                    "the equals sign '=' here seems to be buried inside the "
                    "formula, not at the surface. For instance, 'F = ma' is "
                    "fine, but 'F(m=a)' would not make sense - the '=' would "
                    "be inside brackets.");
                return FALSE;
            }
            break;
        default:
            equation_problem(_P_(C10EquationEquatesMultiply), eqn, "",
                "this equation seems to contain more than one equals "
                "sign '='.");
            return FALSE;
    }
    return enode_typecheck(eqn, eqn->parsed_equation);
}
```

§**39.** A recursive count of instances down the tree from `tok`:

```
int enode_count_equals(equation_node *tok) {
    int c = 0, i;
    if (tok) {
        if (enode_is_equals(tok)) c++;
        for (i=0; i<tok->enode_arity; i++)
            c += enode_count_equals(tok->enode_operands[i]);
    }
    return c;
}
int enode_is_equals(equation_node *tok) {
    if (tok == NULL) return FALSE;
    if ((tok->eqn_type == OPERATION_EQN) && (tok->eqn_operation == EQUALS_OPERATION))
        return TRUE;
    return FALSE;
}
```

§**40.** Now we come to the real typechecking. The following is called, depth-first, at each node in the equation; it has to assign a KOV at every node, in such a way that all operations are dimensionally valid. We return `FALSE` if we are obliged to issue a problem message.

```
int enode_typecheck(equation *eqn, equation_node *tok) {
    int result = TRUE;
    if (tok == NULL) return result;
    int i;
    for (i=0; i<tok->enode_arity; i++)
        if (enode_typecheck(eqn, tok->enode_operands[i]) == FALSE)
            result = FALSE;
    if (result)
        switch (tok->eqn_type) {
            case SYMBOL_EQN: tok->enode_kov = tok->leaf_symbol->var_kov; break;
            case CONSTANT_EQN: tok->enode_kov = spec_get_kind_of_value(tok->leaf_constant);
break;
            case OPERATION_EQN:
                if (tok->eqn_operation == EQUALS_OPERATION)
                    ⟨Typecheck the set-equals node at the top level 41⟩
                else if (tok->eqn_operation == POWER_OPERATION)
                    ⟨Typecheck a raise-to-integer-power node 42⟩
                else
                    ⟨Typecheck a general operation node 44⟩;
                break;
            default: internal_error("forbidden enode found in parsed equation");
        }
    return result;
}
```

§**41.** We are in the realm of quasinumerical KOVs here, so there's no possible fuss about implicit casting. (But if we ever add a floating-point number KOV to Inform, we'll have to worry about writing an integer to a float here.)

⟨Typecheck the set-equals node at the top level 41⟩ ≡

```
if (kov_compare(tok->enode_operands[0]->enode_kov,
    tok->enode_operands[1]->enode_kov) == FALSE) {
    result = FALSE;
    equation_problem(_P_(C10EquationIncomparable), eqn, "",
        "this equation tries to set two values equal which have "
        "different kinds from each other.");
}
tok->enode_kov = tok->enode_operands[0]->enode_kov;
```

This code is used in §40.

§**42.** The restriction on powers is needed to make it possible to know the dimensions of the result. If h is a length, h^2 is an area but h^3 is a volume; so if all we have is h^n, and we don't know the value of n, we're unable to see what equations h^n can appear in.

⟨Typecheck a raise-to-integer-power node 42⟩ ≡

```
equation_node *base = tok->enode_operands[0];
equation_node *power = tok->enode_operands[1];
if (kov_is_dimensionless(base->enode_kov)) {
    if ((is_kova(power->enode_kov, NUMBER_TY) == FALSE) ||
        (power->eqn_type != CONSTANT_EQN)) {
        result = FALSE;
        equation_problem(_P_(C10EquationNonIntPower), eqn, "",
            "the '^' raise-to-power symbol can only be used to raise to a "
            "constant positive integer. So 'mv^2' is fine, but not 'k^F' "
            "for a force 'F'.");
    }
    tok->enode_kov = base->enode_kov;
} else if ((is_kova(power->enode_kov, NUMBER_TY) == FALSE) ||
    (power->eqn_type != CONSTANT_EQN)) {
    result = FALSE;
    equation_problem(_P_(C10EquationDimensionPower), eqn, "",
        "the '^' raise-to-power symbol can only be used to raise a value "
        "with dimensions to a specific number. So 'mv^2' is fine, but not "
        "'mv^n' or 'mv^(1+n)'. (This is because I would need to work out what "
        "kind of value 'v^n' would be, and the answer would depend on 'n', "
        "but I wouldn't know what 'n' is.)");
} else ⟨Take the dimensional power of the KOV of the base 43⟩;
```

This code is used in §40.

§**43.** To work out the KOV of b^n, we use repeated multiplication or division of dimensions; if $n = 0$ then we have a dimensionless value, and choose `NUMBER_TY` as the simplest possibility.

⟨Take the dimensional power of the KOV of the base 43⟩ ≡
```
kind_of_value *kov = kova(NUMBER_TY);
int op = TIMES_OPERATION;
int n = value_of_NUMBER_type(power->leaf_constant);
if (n < 0) { n = -n; op = DIVIDE_OPERATION; }
while (n > 0) {
    kov = arithmetic_on_kovs(kov, base->enode_kov, op);
    n--;
}
tok->enode_kov = kov;
```

This code is used in §42.

§**44.** The following is easy because it was the content of the whole "Dimensions.w" section:

⟨Typecheck a general operation node 44⟩ ≡
```
if (arithmetic_op_is_unary(tok->eqn_operation))
    tok->enode_kov = arithmetic_on_kovs(
        tok->enode_operands[0]->enode_kov, NULL, tok->eqn_operation);
else
    tok->enode_kov = arithmetic_on_kovs(
        tok->enode_operands[0]->enode_kov, tok->enode_operands[1]->enode_kov,
        tok->eqn_operation);
if (tok->enode_kov == NULL) {
    result = FALSE;
    LOG("Failed at operation:\n"); log_equation_node(tok);
    if (arithmetic_op_is_unary(tok->eqn_operation))
        ⟨Issue unary equation typechecking problem message 45⟩
    else
        ⟨Issue binary equation typechecking problem message 46⟩;
}
```

This code is used in §40.

§**45.** These unary operations all typecheck, at present, because they come into being only by rearranging other operations which have already typechecked; that would stop being true if we added notations for root and cube root.

⟨Issue unary equation typechecking problem message 45⟩ ≡
```
quote_source(1, current_sentence);
specification *spec1 = new_generic_CONSTANT_type(tok->enode_operands[0]->enode_kov);
quote_spec(4, spec1);
switch(i) {
    case UNARY_MINUS_OPERATION:
        quote_text(6, "negating");
        break;
    case ROOT_OPERATION:
        quote_text(6, "taking the square root of");
        break;
    case CUBEROOT_OPERATION:
        quote_text(6, "taking the cube root of");
        break;
```

```
    }
    handmade_problem(_P_(BelievedImpossible));
    issue_problem_segment(
        "You wrote %1, but that equation seems to involve %6 %4, which is not "
        "good arithmetic.");
    issue_problem_end();
```

This code is used in §44.

§**46.**

⟨Issue binary equation typechecking problem message 46⟩ ≡

```
    quote_source(1, current_sentence);
    specification *spec1 = new_generic_CONSTANT_type(tok->enode_operands[0]->enode_kov);
    specification *spec2 = new_generic_CONSTANT_type(tok->enode_operands[1]->enode_kov);
    quote_spec(4, spec1);
    quote_spec(5, spec2);
    switch(i) {
        case PLUS_OPERATION:
            quote_text(6, "adding"); quote_text(7, "to");
            break;
        case MINUS_OPERATION:
            quote_text(6, "subtracting"); quote_text(7, "from");
            break;
        case TIMES_OPERATION:
            quote_text(6, "multiplying"); quote_text(7, "by");
            break;
        case DIVIDE_OPERATION:
        case REMAINDER_OPERATION:
            quote_text(6, "dividing"); quote_text(7, "by");
            break;
        case POWER_OPERATION:
            quote_text(6, "raising"); quote_text(7, "the the power of");
            break;
    }
    handmade_problem(_P_(C10EquationBadArithmetic));
    issue_problem_segment(
        "You wrote %1, but that equation seems to involve "
        "%6 %4 %7 %5, which is not good arithmetic.");
    issue_problem_end();
```

This code is used in §44.

§**47. Compiling.** We want each equation to have a distinct value at runtime. The following routines do nothing (for now, anyway) except to give the I6 identifiers for equations a value; they are never called.

```
void compile_equations(OUTPUT_STREAM) {
    equation *eqn;
    LOOP_OVER(eqn, equation) {
        WRITE("[ %s;\n", eqn->eqn_I6_identifier); INDENT;
        WRITE("return 0;\n");
        OUTDENT; WRITE("];\n");
    }
}
```

The function compile_equations is invoked by a command in a `.i6t` template file.

§**48.** These identifiers are used to compile equation names as values, so:

```
char *eqn_get_I6_representation(equation *eqn) {
    return eqn->eqn_I6_identifier;
}
```

The function eqn_get_I6_representation is called from 7/vasp.

§**49. Solving.** So we can finally turn to putting equations to use. Firstly, when a named equation is used, a "where..." clause is sometimes given to make temporary assignments (see above); what happens is that the S-parser temporarily sets the usage words of the equation to the relevant text...

```
void eqn_set_usage_notes(equation *eqn, int u1, int u2) {
    eqn->usage_w1 = u1; eqn->usage_w2 = u2;
}
```

The function eqn_set_usage_notes is called from 5/mlc.

§**50.** ...so that, when we come to solve the equation (i.e., later on in the invocation compiler), we know where to find these temporary assignments. They are wiped out once this compilation is over.

```
void compile_solution(OUTPUT_STREAM, int w1, int w2, equation *eqn) {
    if (eqn->usage_w1 >= 0)
        eqn_declare_variables_inner(eqn, eqn->usage_w1, eqn->usage_w2, TRUE);
    compile_solution_inner(OUT, w1, w2, eqn);
    eqn_remove_temp_variables(eqn);
    eqn->usage_w1 = -1; eqn->usage_w2 = -1;
}
```

The function compile_solution is called from 12/cinv.

§**51.** With that dance out of the way, we can concentrate on the actual task. We have to compile code which assigns the correct value to the symbol specified by (w_1, w_2), according to the equation `eqn`.

```
void compile_solution_inner(OUTPUT_STREAM, int w1, int w2, equation *eqn) {
    equation_symbol *to_solve = NULL;
    ⟨Identify which symbol in the equation we are solving for 52⟩;
    ⟨Rearrange the equation so that this symbol is the entire LHS 53⟩;
    ⟨Identify the symbols in the equation with local variables 54⟩;
    WRITE("! Solving %s for '%s'\n", eqn->eqn_I6_identifier, to_solve->variable_name);
    enode_compile(OUT, eqn->parsed_equation);
}
```

§**52.** Note the case sensitivity here.

⟨Identify which symbol in the equation we are solving for 52⟩ ≡

```
    if (w1 == w2) {
        char *p = lw_array[w1].lw_rawtext;
        equation_symbol *ev;
        for (ev = eqn->symbol_list; ev; ev = ev->next)
            if (strcmp(p, ev->variable_name) == 0)
                to_solve = ev;
    }
    if (to_solve == NULL) {
        quote_source(1, current_sentence);
        quote_words(2, w1, w2);
        quote_words(3, eqn->word_ref1, eqn->word_ref2);
        handmade_problem(_P_(C10EquationBadTarget));
        issue_problem_segment(
            "In %1, you asked to let %2 be given by the equation '%3', "
            "but '%2' isn't a symbol in that equation.");
        issue_problem_end();
        return;
    }
```

This code is used in §51.

§**53.** This step is not easy, but is delegated to `eqn_rearrange` below, so it looks easy here. The surprising thing is the fresh round of typechecking: why do we do that? The answer is not that we doubt whether the equation is still valid – the rearranged equation should pass if and only if the original did, if we've implemented all of this correctly – but because the alterations made to the tree mean that the assignments of KOVs at each node are now potentially incorrect. Re-typechecking will recalculate these.

⟨Rearrange the equation so that this symbol is the entire LHS 53⟩ ≡

```
if (eqn_rearrange(eqn, to_solve) == FALSE) {
    quote_source(1, current_sentence);
    quote_words(2, w1, w2);
    quote_words(3, eqn->word_ref1, eqn->word_ref2);
    handmade_problem(_P_(C10EquationInsoluble));
    issue_problem_segment(
        "In %1, you asked to let %2 be given by the equation '%3', "
        "but I am unable to rearrange the equation in any simple way "
        "so that it sets '%2' equal to something else. Maybe you could "
        "write a more explicit equation? (You're certainly better at "
        "maths than I am; I can only make easy deductions.)");
    issue_problem_end();
    return;
}
if (eqn_typecheck(eqn) == FALSE) return;
LOG("Rearranged equation reads:\n"); log_equation_node(eqn->parsed_equation);
```

This code is used in §51.

§**54.** Suppose we read a phrase such as

let PE be given by PE = mgh, where g = 9.801 m/ss;

We can only compile code to do this if we can identify values for the symbols. "g" is not a problem because a temporary assignment supplies this. For each symbol `ev` which isn't a constant, we must set `ev->local_map` to the number of a local variable, or else issue a problem message.

⟨Identify the symbols in the equation with local variables 54⟩ ≡

```
equation_symbol *ev;
for (ev = eqn->symbol_list; ev; ev = ev->next)
    if (ev->var_const == NULL) {
        ev->local_map = parse_name_local_to_current_stack_frame(
            ev->variable_wn, ev->variable_wn);
        if (ev->local_map < 0)
            ⟨Can't find an unset symbol among the local variables 55⟩
        else
            ⟨Check that the KOV of the local variable matches that of the symbol 56⟩;
    }
```

This code is used in §51.

§**55.** In the above example, finding "PE" should not be a problem: this is the `to_solve` symbol, and it must be a current local variable name since the "let" will have created it as such if it didn't already exist. But things can certainly go wrong with "m" and "h", which need to exist as local variables in the current stack frame.

⟨Can't find an unset symbol among the local variables 55⟩ ≡

```
if (ev == to_solve) internal_error("can't find 'let' variable to assign");
quote_source(1, current_sentence);
quote_words(2, w1, w2);
quote_words(3, eqn->word_ref1, eqn->word_ref2);
quote_words(4, ev->variable_wn, ev->variable_wn);
handmade_problem(_P_(C10EquationSymbolMissing));
issue_problem_segment(
    "In %1, you asked to let %2 be given by the equation '%3', "
    "but I can't see what to use for '%4'. The usual idea is "
    "to set the other variables in the equation using 'let': "
    "so adding 'let %4 be ...' before trying to find '%2' "
    "should work.");
issue_problem_end();
return;
```

This code is used in §54.

§**56.** In the case of the symbol we are setting, the local variable might be one which has only just been created and thus has no value yet – not having set it, Inform hasn't given it a kind of value more explicit than `ANY_VALUE_TY`. We can improve that by giving it the KOV of the symbol it is to match.

In all other cases, the local variable already exists and has a fixed KOV. This must exactly match that of the symbol. (Again, if we ever need implicit casting between quasinumerical KOVs, we'll have to return to this.)

⟨Check that the KOV of the local variable matches that of the symbol 56⟩ ≡

```
kind_of_value *kov = kov_of_local(ev->local_map);
if (is_kova(kov, ANY_VALUE_TY)) {
    kov = ev->var_kov;
    set_KOV_of_local_variable(ev->local_map, kov);
}
if (kov_compare(kov, ev->var_kov) == FALSE) {
    quote_source(1, current_sentence);
    quote_words(2, w1, w2);
    quote_words(3, eqn->word_ref1, eqn->word_ref2);
    quote_words(4, ev->variable_wn, ev->variable_wn);
    quote_kov(5, kov);
    quote_kov(6, ev->var_kov);
    handmade_problem(_P_(C10EquationSymbolWrongKOV));
    issue_problem_segment(
        "In %1, you asked to let %2 be given by the equation '%3', "
        "but in that equation '%4' is supposedly %6 - whereas right "
        "here, it seems to be %5. Perhaps two different quantities have "
        "ended up with the same symbol in the source text?");
    issue_problem_end();
    return;
}
```

This code is used in §54.

§**57.** Actual compilation is simple, since the tree is set up for it.

```
void enode_compile(OUTPUT_STREAM, equation_node *tok) {
    switch (tok->eqn_type) {
        case SYMBOL_EQN:
            if (tok->leaf_symbol->var_const)
                spec_compile(OUT, tok->leaf_symbol->var_const);
            else
                WRITE("t_%d", tok->leaf_symbol->local_map); break;
        case CONSTANT_EQN: spec_compile(OUT, tok->leaf_constant); break;
        case OPERATION_EQN: WRITE("("); ⟨Compile a single operation 58⟩; WRITE(")"); break;
        default: internal_error("forbidden enode found in parsed equation");
    }
}
```

§**58.** And here we handle operation nodes:

⟨Compile a single operation 58⟩ ≡

```
    equation_node *left = tok->enode_operands[0];
    equation_node *right = tok->enode_operands[1];
    switch (tok->eqn_operation) {
        case POWER_OPERATION:
            ⟨Compile a power of the left operand 59⟩; break;
        case EQUALS_OPERATION:
            enode_compile(OUT, left); WRITE(" = "); enode_compile(OUT, right); break;
        case PLUS_OPERATION:
            enode_compile(OUT, left); WRITE(" + "); enode_compile(OUT, right); break;
        case MINUS_OPERATION:
            enode_compile(OUT, left); WRITE(" - "); enode_compile(OUT, right); break;
        case TIMES_OPERATION:
            enode_compile(OUT, left); WRITE(" * "); enode_compile(OUT, right);
            kov_rescale_multiplication(OUT, left->enode_kov, right->enode_kov);
            break;
        case DIVIDE_OPERATION:
            WRITE("IntegerDivide("); enode_compile(OUT, left);
            kov_rescale_division(OUT, left->enode_kov, right->enode_kov);
            WRITE(", "); enode_compile(OUT, right); WRITE(")");
            break;
        case UNARY_MINUS_OPERATION:
            WRITE("-("); enode_compile(OUT, left); WRITE(")");
            break;
        case ROOT_OPERATION:
            WRITE("SquareRoot("); enode_compile(OUT, left);
            kov_rescale_root(OUT, left->enode_kov, 2); WRITE(")");
            break;
        case CUBEROOT_OPERATION:
            WRITE("CubeRoot("); enode_compile(OUT, left);
            kov_rescale_root(OUT, left->enode_kov, 3); WRITE(")");
            break;
        default: internal_error("unimplemented operation");
    }
```

This code is used in §57.

§**59.** We accomplish powers by repeated multiplication. This is partly because I6 has no "to the power of" function, partly because the powers involved will always be small, partly because of the need for scaling to come out right.

⟨Compile a power of the left operand 59⟩ ≡

```
int p = value_of_NUMBER_type(right->leaf_constant);
if (p < 0) internal_error("can't compile negative powers yet");
if (p > 0) enode_compile(OUT, left);
while (p > 1) {
    WRITE(" * "); enode_compile(OUT, left);
    kov_rescale_multiplication(OUT, left->enode_kov, left->enode_kov);
    p--;
}
```

This code is used in §58.

§**60. Rearrangement.** We carry out only the simplest of operations, but it's surprising how often that's good enough: if it isn't, we simply return `FALSE`.

Everything we do will be reversible, which is important since we are changing the `parsed_equation` tree, and we don't want to be changing our view of what the equation means in the process. One thing that never changes is that the top node of the equation is always the unique "equal to" node in the tree.

Suppose we are solving for `v`, which occurs in just one place in the equation. Either it's at the top level under the `=`, in which case we now have an explicit formula for `v`, or it's stuck underneath some operation node. We rearrange the tree to move this operation over to the other side, which allows `v` to make progress – see below for a proof that this terminates.

```
int eqn_rearrange(equation *eqn, equation_symbol *to_solve) {
    while (TRUE) {
        ⟨Swap the two sides if necessary so that v occurs only once and on the left 61⟩;
        equation_node *old_LHS = eqn->parsed_equation->enode_operands[0];
        equation_node *old_RHS = eqn->parsed_equation->enode_operands[1];
        if (old_LHS->eqn_type != OPERATION_EQN) break;
        if (old_LHS->enode_arity == 2)
            ⟨Rearrange to move v upwards through this binary operator 63⟩
        else
            ⟨Rearrange to move v upwards through this unary operator 65⟩;
    }
    return TRUE;
}
```

§**61.** We have no ability to gather terms, so the variable `v` we are solving for can only occur once in the formula. In Inform's idea of equations, `A = B` and `B = A` have the same meaning, so we'll place `v` on the left.

⟨Swap the two sides if necessary so that v occurs only once and on the left 61⟩ ≡

```
int lc = enode_count_var(eqn->parsed_equation->enode_operands[0], to_solve);
int rc = enode_count_var(eqn->parsed_equation->enode_operands[1], to_solve);
if (lc + rc != 1) return FALSE;
if (lc == 0) {
    equation_node *swap = eqn->parsed_equation->enode_operands[0];
    eqn->parsed_equation->enode_operands[0] = eqn->parsed_equation->enode_operands[1];
    eqn->parsed_equation->enode_operands[1] = swap;
}
```

This code is used in §60.

§**62.** The main loop above terminates because on each iteration, either

(i) the tree depth of `v` below `=` decreases by 1, or
(ii) the tree depth of `v` remains the same but the number of `MINUS_OPERATION` or `DIVIDE_OPERATION` nodes in the tree decreases by 1.

Since at any given time there are a finite number of `MINUS_OPERATION` or `DIVIDE_OPERATION` nodes, case (ii) cannot repeat indefinitely, and we must therefore eventually fall into case (i); and then subsequently do so again, and so on; and so the tree depth of `v` will ultimately fall to 1, at which point it is at the top level as required and we break out of the loop.

§**63.** So the rearrangement moves have to make sure the "(i) or (ii)" property always holds. The simplest case to understand is `+`. Suppose we have:

```
=
    +
        V
        E
    R
```

representing $(V + E) = R$, where V is the sub-equation containing v. (E is an arbitrary sub-equation, and R is the right hand side.) One of the two operands of `+` will be "promoted", moving upwards in the tree, and since we can choose to promote either V or E, we'll choose V, thus obtaining:

```
=
    V
    -
        R
        E
```

that is, $V = (R - E)$. Since V has moved upwards, so has the unique instance of v, and therefore the tree depth of v has decreased by 1 – property (i). Multiplication is similar, but turns into division on the right hand side.

But now consider `-`. When we rearrange:

```
=
    -
        E
        V
    R
```

representing $(E - V) = R$ we no longer have a choice of which operand of `-` to promote: we have to promote the right operand, and that produces $E = (R + V)$. The tree depth of v is not improved,

and it's now over on the right hand side. The next iteration of the main loop will swap sides again so that we have $(R + V) = E$. But a tricky node (subtraction or division) has been exchanged out of the tree for an easy one (addition or multiplication), so we fail property (i) but achieve property (ii).

```
⟨Rearrange to move v upwards through this binary operator 63⟩ ≡
    rearrange to move this operator
    int op = old_LHS->eqn_operation;
    if (op == POWER_OPERATION)
        ⟨Rearrange to remove a power 64⟩
    else {
        int promote = 0, new_op = PLUS_OPERATION;
        if (enode_count_var(old_LHS->enode_operands[1], to_solve) > 0) promote = 1;
        switch (op) {
            case PLUS_OPERATION: new_op = MINUS_OPERATION; break;
            case MINUS_OPERATION: new_op = PLUS_OPERATION; promote = 0; break;
            case TIMES_OPERATION: new_op = DIVIDE_OPERATION; break;
            case DIVIDE_OPERATION: new_op = TIMES_OPERATION; promote = 0; break;
            default: LOG("%d\n", op); internal_error("strange operator in rearrangement");
        }
        equation_node *E = old_LHS->enode_operands[1 - promote];
        the new LHS is the promoted operand:
        eqn->parsed_equation->enode_operands[0] = old_LHS->enode_operands[promote];
        the new RHS is the operator which used to be the LHS...
        eqn->parsed_equation->enode_operands[1] = old_LHS;
        ...the former RHS being the operand replacing the promoted one...
        old_LHS->enode_operands[0] = old_RHS;
        old_LHS->enode_operands[1] = E;
        ...except that the operator reverses in "sense"
        old_LHS->eqn_operation = new_op;
    }
```

This code is used in §60.

§64. Solving $x^v = y$ for v requires logs, which are not in our scheme; and solving $v^n = y$ for non-constant n is no better. So in either case we surrender by returning `FALSE`.

In fact, the only cases we can solve at present are $V^2 = y$ and $V^3 = y$. It would be easy to add solutions for $V^4 = y$, $V^6 = y$ and in general for $V^k = y$ where the only prime factors of k are 2 and 3, but this is not something people are likely to need very much. The taking of 4th or higher roots hardly ever occurs in physical equations, and anyone wanting this will have to write more explicit source text.

Anyway, rearrangement for our easy cases is indeed easy:

```
=
    ^
        V
        2
    R
```

becomes

```
=
    V
    square-root
        R
```

and V is always promoted, so we achieve property (i); and similarly for cube roots.

⟨Rearrange to remove a power 64⟩ ≡

```
if (old_LHS->enode_operands[1]->eqn_type != CONSTANT_EQN) return FALSE;
int p = value_of_NUMBER_type(old_LHS->enode_operands[1]->leaf_constant);
int new_op = -1;
if (p == 2) new_op = ROOT_OPERATION;
if (p == 3) new_op = CUBEROOT_OPERATION;
if (new_op == -1) return FALSE;
eqn->parsed_equation->enode_operands[0] = old_LHS->enode_operands[0];
eqn->parsed_equation->enode_operands[1] = old_LHS;
old_LHS->eqn_operation = new_op;
old_LHS->enode_arity = 1;
old_LHS->enode_operands[0] = old_RHS;
```

This code is used in §63.

§**65.** The unary operations are easy in a similar way – they only have one operand, so we always promote V and achieve property (i). A square root is rearranged as a square, and a cube root as a cube. (It's important that everything we do is reversible – we generate exactly those powers which we are able to undo again if necessary.) Unary minus is easier still – we need only move it to the other side; thus $-V = R$ becomes $V = -R$, and `v` again rises.

⟨Rearrange to move v upwards through this unary operator 65⟩ ≡

```
int op = old_LHS->eqn_operation;
switch (op) {
    case UNARY_MINUS_OPERATION:
        eqn->parsed_equation->enode_operands[0] = old_LHS->enode_operands[0];
        eqn->parsed_equation->enode_operands[1] = old_LHS;
        old_LHS->enode_operands[0] = old_RHS;
        break;
    case ROOT_OPERATION:
        eqn->parsed_equation->enode_operands[0] = old_LHS->enode_operands[0];
        eqn->parsed_equation->enode_operands[1] = old_LHS;
        old_LHS->eqn_operation = TIMES_OPERATION;
        old_LHS->enode_arity = 2;
        old_LHS->enode_operands[0] = old_RHS;
        old_LHS->enode_operands[1] = old_RHS;
        break;
    case CUBEROOT_OPERATION:
        eqn->parsed_equation->enode_operands[0] = old_LHS->enode_operands[0];
        eqn->parsed_equation->enode_operands[1] = old_LHS;
        old_LHS->eqn_operation = TIMES_OPERATION;
        old_LHS->enode_arity = 2;
        old_LHS->enode_operands[0] = old_RHS;
        old_LHS->enode_operands[1] = enode_new_op(TIMES_OPERATION);
        old_LHS->enode_operands[1]->enode_operands[0] = old_RHS;
        old_LHS->enode_operands[1]->enode_operands[1] = old_RHS;
        break;
    default: internal_error("unanticipated operator in rearrangement");
}
```

This code is used in §60.

§**66.** And that's the whole rearranger, except for the utility routine which counts instances of the magic variable `v` at or below a given point in the equation tree.

```
int enode_count_var(equation_node *tok, equation_symbol *to_solve) {
    int c = 0, i;
    if (tok == NULL) return c;
    if ((tok->eqn_type == SYMBOL_EQN) && (tok->leaf_symbol == to_solve))
        return 1;
    for (i=0; i<tok->enode_arity; i++)
        c += enode_count_var(tok->enode_operands[i], to_solve);
    return c;
}
```

§**67. Internal test case.** This is a little like those "advise all parties" law exam questions: we parse the equation, then rearrange to solve it for each variable in turn.

```
void eqn_internal_test(int e1, int e2) {
    int i, wh1 = -1, wh2 = -1;
    [[e1, e2 == ... where ... : i --> e1, e2 ... wh1, wh2]];
    equation *eqn = new_equation(e1, e2, TRUE);
    eqn_set_wherewithal(eqn, wh1, wh2);
    examine_equation(eqn);
    begin_reporting_internal_test();
    log_equation_parsed(eqn);
    equation_symbol *ev;
    for (ev = eqn->symbol_list; ev; ev = ev->next) {
        if (eqn_rearrange(eqn, ev) == FALSE)
            LOG("Too hard to rearrange to solve for %s\n", ev->variable_name);
        else {
            LOG("Rearranged to solve for %s:\n", ev->variable_name);
            log_equation_parsed(eqn);
        }
    }
    end_reporting_internal_test();
}
```

The function eqn_internal_test is called from 13/test.

§**68. Indexing and logging.** And finally:

```
void log_equation(equation *eqn) {
    LOG("{$W}", eqn->word_ref1, eqn->word_ref2);
}

void log_equation_parsed(equation *eqn) {
    if (eqn == NULL) LOG("<null>\n");
    else log_equation_node(eqn->parsed_equation);
}

void index_equations(void) {
    int ec = 0; equation *eqn;
    LOOP_OVER(eqn, equation) { ec++; }
    if (ec == 0) return;
    INDEX("<p><b>List of Equations</b> (<i>About equations</i>");
    index_doc_link("EQUATIONS"); INDEX(")<p>\n");
    LOOP_OVER(eqn, equation) {
        int mw = eqn->equation_no_w2;
        if (eqn->equation_name_w2 > mw) mw = eqn->equation_name_w2;
        print_raw_text_to_file(eqn->equation_created_at->word_ref1, mw, ifl);
        index_link(lw_array[eqn->equation_created_at->word_ref1].lw_source);
        INDEX(" (");
        print_raw_text_to_file(eqn->word_ref1, eqn->word_ref2, ifl);
        INDEX(")<br>\n");
    }
}
```

The function log_equation is called from 2/dl.

The function index_equations is invoked by a command in a `.i6t` template file.

Listed-In Relations 10/libp

Purpose

To define the binary predicates corresponding to table columns, and which determine whether a given value is listed in that column.

Definitions

¶1. ...

§**1. Initial stock.** There is none, since no tables or columns exist when Inform starts up.

```
void LISTED_IN_KBP_create_initial_stock(void) {
}
```

The function LISTED_IN_KBP_create_initial_stock is called from 5/bp.

§**2. Subsequent creations.** When a column called, say, "pledged delegate count" appears in one or more tables in the source text, Inform creates a `table_column` structure to represent the common identity of all table columns with this name. (They are required all to share the same kind of value in their entries.) For each different table column, a BP is created to represent the meaning of "X is a pledged delegate count listed in T". Arguably there should just be one super-powerful predicate `listed-in(X, C, T)`, but that would need to be a ternary predicate, not binary, and Inform doesn't at present support those. So we make a one-parameter family of `listed-in-C(X, T)` binary predicates instead.

```
binary_predicate *make_listed_in_predicate(table_column *tc) {
    binary_predicate *bp = make_pair_of_BPs(LISTED_IN_KBP,
        bptd_blank(),
        bptd_new(NULL, kova(TABLE_TY)),
        "listed_in", "lists-in", -1, NULL, NULL,
        sch_new_1d("(ct_1=ExistsTableRowCorr(ct_0=*2,%d,*1))",
            tc_get_column_id(tc)),
        0);
    bp->a_listed_in_predicate = TRUE;
    return bp;
}
```

The function make_listed_in_predicate is called from 10/tab.

§**3.** Once again there is a timing constraint. Tables are created quite early on in Inform's run, but the entries in the columns aren't parsed until much later. Since the kind of value stored in a column is often determined only by looking at those values, it follows that we can't specify what goes into the left-hand term at the time when the column is created. So we fill this in later, instead:

```
void supply_KOV_for_listed_in_tc(binary_predicate *bp, kind_of_value *kov) {
    bp->term_details[0].implies_kov = kov;
    bp->reversal->term_details[1].implies_kov = kov;
}
```

The function supply_KOV_for_listed_in_tc is called from 10/tab.

§4. **Second stock.** By this time, they all exist.

```
void LISTED_IN_KBP_create_second_stock(void) {
}
```

The function LISTED_IN_KBP_create_second_stock is called from 5/bp.

§5. **Typechecking.**

```
int LISTED_IN_KBP_typecheck(binary_predicate *bp,
        kind_of_value **kovs_of_terms, kind_of_value **kovs_required, tc_problem_kit *tck) {
    return DECLINE_TO_MATCH;
}
```

The function LISTED_IN_KBP_typecheck is called from 5/bp.

§6. **Assertion.** These relations cannot be asserted.

```
int LISTED_IN_KBP_assert(binary_predicate *bp,
        world_object *wo0, specification *spec0,
        world_object *wo1, specification *spec1) {
    return FALSE;
}
```

The function LISTED_IN_KBP_assert is called from 5/bp.

§7. **Compilation.** Note the side-effect here: we ensure that the `ct` local variables will be present in the current stack frame, since we're going to need them to hold the table reference for any successful lookup.

```
int LISTED_IN_KBP_compile(int task, binary_predicate *bp, annotated_i6_schema *asch) {
    if (task == TEST_ATOM_TASK) we_need_ct_locals();
    return FALSE;
}
```

The function LISTED_IN_KBP_compile is called from 5/bp.

§8. **Problem message text.**

```
int LISTED_IN_KBP_describe_for_problems(binary_predicate *bp, char *problem_buffer) {
    strcat(problem_buffer, "the listed in relation");
    return TRUE;
}
```

The function LISTED_IN_KBP_describe_for_problems is called from 5/bp.

Interactive Fiction ID 10/ifid

Purpose

Computes and makes available the IFID (Interactive Fiction ID) number for an Inform-generated work of IF, in compliance with the Treaty of Babel.

Template interpreter commands

```
2   {-array:UUID_ARRAY}
```

Definitions

---

§**1.** The Interactive Fiction ID number for an Inform 7-compiled work is the same as the UUID unique ID generated by the Inform 7 application.

UUIDs are not generated within NI, but by the host application: NI expects to read them in the form of the `uuid.txt` file placed in the project bundle by that application. After some agonising, I decided that the Treaty did not actually oblige me to crash NI out if this file did not exist: but in such cases the UUID is empty.

```
define MAX_UUID_LENGTH 128                          the UUID is truncated to this if necessary

char uuid_text[MAX_UUID_LENGTH];
int uuid_read = -1;

int read_uuid(void) {
    FILE *xf; char *fn;
    int i=0; int c;
    if (uuid_read >= 0) return uuid_read;
    uuid_read = 0;
    fn = top_level_filename(UUID_LEAFNAME);
    xf = iso_fopen(fn, "r");
    if (xf == NULL) {
        uuid_text[0] = 0;                           the UUID is the empty string if the file is missing
        return 0;
    }
    while (((c = fgetc(xf)) != EOF)                 the UUID file is plain text, not Unicode
        && (i < MAX_UUID_LENGTH-1))
        uuid_text[i++] = toupper(c);
    fclose(xf);
    uuid_text[i] = 0;
    uuid_read = i;
    return i;
}
```

The function read_uuid is called from 10/bib.

§**2.** The IFID is written into the compiled story file, too, both in order that it can be printed by the VERSION command and to brand the file so that it can still be identified even if it loses touch with its iFiction record. We store the IFID in plain text, with a "magic string" identifier around it, in byte-accessible memory.

```
void compile_UUID_ARRAY_array(OUTPUT_STREAM) {
    int i;
    read_uuid();
    WRITE("Array UUID_ARRAY string \"UUID://");
    for (i=0; uuid_text[i]; i++) WRITE("%c", toupper(uuid_text[i]));
    WRITE("//\";\n");
}
```

The function compile_UUID_ARRAY_array is invoked by a command in a `.i6t` template file.

Bibliographic Data 10/bib

Purpose

To manage the special quantities providing bibliographic data on the work of IF being generated (title, author's name and so forth); to write the iFiction record for the work of IF compiled, its release instructions and its picture manifest, if any.

Template interpreter commands

```
4   {-callv:index_library_card}
5   {-callv:write_ifiction_and_blurb}
21   {-callv:compile_bibliographic_i6_constants}
```

Definitions

¶1. Much of this section is best understood by reference to the Treaty of Babel, a cross-IF-system standard for bibliographic data and packaging agreed between the major IF design systems in 2006. Inform aims to comply fully with the Treaty and the code below should be maintained as such.

The bibliographic data for the story is kept in the following quantities, which are used to build the iFiction record and the releasing blurb at the end of a successful compilation.

```
quantity *story_title = NULL;
quantity *story_author = NULL;
quantity *story_headline = NULL;
quantity *story_genre = NULL;
quantity *story_description = NULL;
quantity *story_release_number = NULL;
quantity *story_creation_year = NULL;
int episode_number = -1;                    for a work which is part of a numbered series
char *series_name = NULL;
```

¶2. Auxiliary files are not really files to NI at all: simply names passed along. They are the auxiliary files included in the iFiction record generated for a released project, if the source asks to do so: they might for instance be maps or booklets which the author intends to accompany the final story file. (Because they are treated only as names and are never opened, the following structure contains no file handles.)

```
typedef struct auxiliary_file {
    char *leafname_in_materials_folder;                    e.g., "Collegio.pdf"
    char *brief_description;                               e.g., "Collegio Magazine"
    MEMORY_MANAGEMENT
} auxiliary_file;
```

The structure auxiliary_file is private to this section.

---

§**1.** Here we parse the special titling line at the top of the text source, setting two initial values for the bibliographic variables: `story_title` and `story_author`.

```
sentence_handler BIBLIOGRAPHIC_SH_handler =
    { BIBLIOGRAPHIC_NT, -1, 2, bibliographic_data };

void bibliographic_data(parse_node *PN) {
    int w1, w2, aw1 = -1, aw2 = -1, i;
    [[w1, w2 <-- PN]];
    [[w1, w2 == ... by ... : i --> w1, w2 ... aw1, aw2]];
    if ((is_kova(is_a_literal(w1, w2), TEXT_TY) == FALSE) &&
        (is_kova(is_a_literal(w1, w2), TEXT_ROUTINE_TY) == FALSE)) {
        sentence_problem(_P_(C10BadTitleSentence),
            "the initial bibliographic sentence can only be a title in double-quotes",
            "possibly followed with 'by' and the name of the author.");
    }
    if (story_title && story_author) {
        specification *the_title = parse_expression(w1, w1, VALUE_EXPCON);
        initialise_global_variable(story_title, the_title, NULL);
        if (aw1 >= 0) {
            char author_buffer[1024];
            if (is_kova(is_a_literal(aw1, aw2), TEXT_TY) == FALSE) {
                sprintf(author_buffer, "\"");
                print_raw_text_to_string(aw1, aw2,
                    author_buffer+strlen(author_buffer));
                for (i=1; author_buffer[i]; i++)
                    if (author_buffer[i] == '\"') author_buffer[i] = '\'';
                sprintf(author_buffer+strlen(author_buffer), "\" ");
                feed_into_lexer(author_buffer, FALSE, FALSE);
                aw1 = lexer_feed_w1; aw2 = lexer_feed_w2;
            }
            specification *the_author = parse_expression(aw1, aw1, VALUE_EXPCON);
            initialise_global_variable(story_author, the_author, NULL);
        }
    } else internal_error("story title and author quantities not created");
}

int story_author_is(char *p) {
    if ((story_author) && (qty_has_initial_value_set(story_author))) {
```

```
        specification *spec = qty_get_initial_value(story_author);
        spec_set_kind_of_value(spec, kova(TEXT_TY));
        int result = FALSE;
        int b = divert_constant_text_bibliographically;
        TEMPORARY_STREAM;
        divert_constant_text_bibliographically = TRUE;
        spec_compile(TEMP, spec);
        divert_constant_text_bibliographically = b;
        if (strcmp(p, STREAM_TEXT(TEMP)) == 0) result = TRUE;
        CLOSE_TEMPORARY_STREAM;
        return result;
    }
    return FALSE;
}
```

The function story_author_is is called from 4/ext.

§**2. Episodes in a series.** This sets the two global variables for episode number and series title, which are handled less grandly than the quantities above.

```
sentence_handler EPISODE_SH_handler =
    { SENTENCE_NT, EPISODE_VB, 1, set_episode_data };

void set_episode_data(parse_node *pn) {
    int w1, w2, of1, of2, x1, x2, i;
    [[w1, w2 <-- pn->down->next->next]];
    if ([[w1, w2 == ... of ... : i --> x1, x2 ... of1, of2]] &&
        (is_kova(is_a_literal(x1, x2), NUMBER_TY)) &&
        (is_kova(is_a_literal(of1, of2), TEXT_TY))) {
        is_a_literal(x1, x1);
        episode_number = last_literal_evaluated;
        dequote_word(of2);
        series_name = lw_array[of2].lw_text;
        return;
    }
    LOG("Found data <$W>\n", w1, w2);
    sentence_problem(_P_(C10BadEpisode),
        "this is not the right way to specify how the story "
        "fits into a larger narrative",
        "and should take the form 'The story is episode 2 of "
        "\"Belt of Orion\", where the episode number has to be a "
        "whole number 0, 1, 2, ... and the series name has to be "
        "plain text without [substitutions].");
}
```

§**3. Releasing instructions.** The following somewhat miscellaneous variables hold the instructions given in the source text for how to release the story file – the content of the "Release along with..." sentence, in fact. We then give the routine which parses this sentence, setting the relevant variables.

```
int release_website = FALSE;                                    Release along with a website?
char *website_template_leafname = "Standard";                   If so, the template name for it
int release_booklet = FALSE;                                    Release along with introductory booklet?
int release_cover = FALSE;                                      Release along with cover art?
parse_node *cover_filename_sentence = NULL;                     Where this was requested
int release_solution = FALSE;                                   Release along with a solution?
int release_source = FALSE;                                     Release along with the source text?
int release_card = FALSE;                                       Release along with the iFiction card?
int solution_public = FALSE;                                    If released, will this be linked on a website?
int source_public = TRUE;                                       If released, will this be linked on a website?
int card_public = FALSE;                                        If released, will this be linked on a website?

sentence_handler RELEASE_SH_handler =
    { SENTENCE_NT, RELEASE_VB, 1, handle_release_declaration };

void handle_release_declaration(parse_node *p) {
    new_release_declaration(
        new_nounphrase_articled(p->down->next->next->word_ref1,
            p->down->next->next->word_ref2));
}

void new_release_declaration(parse_node *p) {
    int w1 = p->word_ref1, w2 = p->word_ref2, i, x1, x2, cw1, cw2;
    if (is_list_divided(w1, w2, LOOK_FOR_OR+LOOK_FOR_AND)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        new_release_declaration(new_nounphrase_articled(lw1, lw2));
        new_release_declaration(new_nounphrase_articled(rw1, rw2));
        return;
    }
    int privacy = NOT_APPLICABLE;
    if [[w1, w2 == private ... --> w1, w2]] privacy = FALSE;
    else if [[w1, w2 == public ... --> w1, w2]] privacy = TRUE;
    if [[w1, w2 == solution]] {
        release_solution = TRUE;
        if (privacy != NOT_APPLICABLE) solution_public = privacy;
        return;
    }
    if [[w1, w2 == source text]] {
        release_source = TRUE;
        if (privacy != NOT_APPLICABLE) source_public = privacy;
        return;
    }
    if [[w1, w2 == library card]] {
        release_card = TRUE;
        if (privacy != NOT_APPLICABLE) card_public = privacy;
        return;
    }
    if (privacy != NOT_APPLICABLE) {
        quote_source(1, p);
        handmade_problem(_P_(C10NoSuchPublicRelease));
```

```
        issue_problem_segment(
            "I don't know how to release along with %1: the only features of "
            "a release which can be marked as public or private are the 'source "
            "text', 'solution' and 'library card'.");
        issue_problem_end();
        return;
    }
    if [[w1, w2 == cover art]] {
        release_cover = TRUE;
        cover_filename_sentence = current_sentence;
        return;
    }
    if [[w1, w2 == existing story file]] { existing_story_file = TRUE; return; }
    if ([[w1, w2 == file of ... called ... : i --> x1, x2 ... cw1, cw2]]
        && (is_kova(is_a_literal(x1, x2), TEXT_TY))
        && (is_kova(is_a_literal(cw1, cw2), TEXT_TY))) {
        auxiliary_file *af = CREATE(auxiliary_file);
        dequote_word(x1); dequote_word(cw1);
        af->leafname_in_materials_folder = lw_array[cw1].lw_text;
        af->brief_description = lw_array[x1].lw_text;
        return;
    }
    if [[w1, w2 == introductory booklet]] { release_booklet = TRUE; return; }
    if [[w1, w2 == website]] { release_website = TRUE; return; }
    if ([[w1, w2 == ... website --> x1, x2]]
        && (is_kova(is_a_literal(x1, x2), TEXT_TY))) {
        dequote_word(x1);
        website_template_leafname = lw_array[x1].lw_text;
        release_website = TRUE;
        return;
    }
    quote_source(1, p);
    handmade_problem(_P_(C10ReleaseAlong));
    issue_problem_segment(
        "I don't know how to release along with %1: the only forms I can "
        "accept are - 'Release along with cover art', '...a website', "
        "'the solution', 'the library card', 'the introductory booklet', 'the source text',
"
        "'an existing story file' or '...a file of \"Something Useful\" "
        "called \"Something.pdf\"'.");
    issue_problem_end();
}
```

§4. **The Library Card.** The Library Card is part of the Contents index, and is intended as a natural way to present bibliographic data to the user. In effect, it's a simplified form of the iFiction record, without the XML overhead and without the technicalities handled automatically by NI.

```
int write_bibliographic_datum(OUTPUT_STREAM, quantity *q, int desc_mode) {
    if (q == NULL) return FALSE;
    if (qty_has_initial_value_set(q) == FALSE) return FALSE;
    TEMPORARY_STREAM;
    compile_literal_text(TEMP, qty_treat_as_plain_text_word(q));
    if (STREAM_EXTENT(TEMP) > 0) {
        if (desc_mode) WRITE("<description>");
        STREAM_COPY(OUT, TEMP);
        if (desc_mode) WRITE("</description>\n");
    }
    CLOSE_TEMPORARY_STREAM;
    return TRUE;
}

void library_card_entry(char *field, quantity *q, char *t) {
    char *font = "<font face=\"courier\" size=2 color=\"#000080\">";
    first_html_column(ifl, 0);
    INDEX("%s%s</font>", font, field);
    next_html_column(ifl, 0);
    INDEX("%s<b>", font);
    divert_constant_text_bibliographically = TRUE;
    BEGIN_COMPILATION_MODE;
    COMPILATION_MODE_ENTER(COMPILE_TEXT_TO_XML_CMODE);
    if (write_bibliographic_datum(ifl, q, (q==story_description)?TRUE:FALSE)
        == FALSE) INDEX("%s", t);
    divert_constant_text_bibliographically = FALSE;
    END_COMPILATION_MODE;
    INDEX("</b></font>");
    end_html_row(ifl);
}

void index_library_card(void) {
    auxiliary_file *af;
    INDEX("<p><hr><p>");
    index_anchor("LCARD");
    INDEX("<b>Library Card</b> (<i>About library cards:</i>");
    index_doc_link("LCARDS"); INDEX(")<p>\n");
    begin_html_table(ifl, "cornsilk", FALSE, 1, 3, 3, 0, 0);
    library_card_entry("Story title", story_title, "Untitled");
    library_card_entry("Story author", story_author, "Anonymous");
    library_card_entry("Story headline", story_headline, "An Interactive Fiction");
    library_card_entry("Story genre", story_genre, "Fiction");
    if (episode_number >= 0) {
        char episode_text[MAX_STRING_LENGTH+20];
        sprintf(episode_text, "%d of %s", episode_number, series_name);
        library_card_entry("Episode", NULL, episode_text);
    }
    library_card_entry("Release number", story_release_number, "1");
    library_card_entry("Story creation year", story_creation_year, "(This year)");
    library_card_entry("IFID number", NULL, uuid_text);
    library_card_entry("Story description", story_description, "None");
```

```
    LOOP_OVER(af, auxiliary_file) {
        library_card_entry(af->leafname_in_materials_folder, NULL,
            af->brief_description);
    }
    end_html_table(ifl);
    INDEX("<p>\n");
    write_current_VM(ifl);
    index_VM_usage();
}
```

The function index_library_card is invoked by a command in a `.i6t` template file.

§**5. The main metadata routine.** Here we combine writing the iFiction record and the release instructions. (In a single routine, since they share the need to know about certain external files.)

```
void write_ifiction_and_blurb(void) {
    STREAM xf_struct; STREAM *xf = &xf_struct;
    char *fn;
    int i, cover_picture_number = 0;
    char materials_folder[MAX_FILENAME_LENGTH];
    char cover_filename[MAX_FILENAME_LENGTH];
    char *cover_art_format = NULL;
    char header[64];
    FILE *COVER_FILE;
    unsigned int width, height;
    char *story_format = "zcode";
    char *manifest_filename;
    if (release_cover) cover_picture_number = 1;
    if ((story_filename_extension) &&
        (strcmp(story_filename_extension, "ulx") == 0))
        story_format = "glulx";
    ⟨WIB - Ensure Materials folder if needed 6⟩;
    ⟨WIB - Check cover art image if any 7⟩;
    ⟨WIB - Read header of existing story file if present 8⟩
    read_uuid();
    divert_constant_text_bibliographically = TRUE;
    ⟨WIB - Write iFiction record 9⟩;
    ⟨WIB - Write release blurb 10⟩;
    ⟨WIB - Write manifest file 11⟩;
    divert_constant_text_bibliographically = FALSE;
}
```

The function write_ifiction_and_blurb is invoked by a command in a `.i6t` template file.

§6. Some forms of release need to take files from, or put files into, a subfolder called Release of the project's associated Materials folder. This is where we read (and if necessary create) that folder.

⟨WIB - Ensure Materials folder if needed 6⟩ ≡

```
if (bundle_name) strcpy(materials_folder, bundle_name);
else materials_folder[0] = 0;
i = strlen(materials_folder)-1;
while ((i>0) && (materials_folder[i] != '.')) i--;
if (i>0) strcpy(materials_folder+i, " Materials");
if ((release_website) || (release_booklet) || (release_cover) ||
    (release_solution) || (existing_story_file) ||
    (NUMBER_CREATED(blorb_figure) > 1)) {
    if ((materials_folder[0] == 0) ||
        (platform_specific_mkdir(materials_folder) == FALSE)) {
        release_problem(_P_(Untestable),
            "In order to release the story file along with other "
            "resources, I tried to create a folder alongside this "
            "Inform project, but was unable to do so. The folder "
            "was to have been called",
            materials_folder);
        return;
    }
    if (materials_folder[0]) {
        char release_folder[MAX_FILENAME_LENGTH];
        sprintf(release_folder, "%s%cRelease", materials_folder, FOLDER_SEPARATOR);
        if (platform_specific_mkdir(release_folder) == FALSE) {
            release_problem(_P_(Untestable),
                "In order to release the story file along with other "
                "resources, I tried to create a folder alongside this "
                "Inform project, but was unable to do so. The folder "
                "was to have been called",
                release_folder);
            return;
        }
    }
}
```

This code is used in §5.

§**7.** Using the utility routines above, we find out the format of the cover art and see that its dimensions conform to Treaty of Babel requirements.

⟨WIB - Check cover art image if any 7⟩ ≡

```
if (release_cover) {
    unsigned int rv = 0;
    cover_art_format = "jpg";
    sprintf(cover_filename, "%s%cCover.jpg", materials_folder, FOLDER_SEPARATOR);
    COVER_FILE = iso_fopen(cover_filename, "rb" );
    if (COVER_FILE) {
        rv = get_jpg_dimensions(COVER_FILE, &width, &height);
        fclose(COVER_FILE);
        if (rv == 0) {
            release_problem(_P_(Untestable),
                "The cover image seems not to be a JPEG despite the name",
                cover_filename);
            return;
        }
    }
    if (rv == 0) {
        sprintf(cover_filename, "%s%cCover.png", materials_folder, FOLDER_SEPARATOR);
        COVER_FILE = iso_fopen(cover_filename, "rb" );
        if (COVER_FILE == NULL) {
            current_sentence = cover_filename_sentence;
            release_problem_at_sentence(_P_(Untestable),
                "The release instructions said that there is a cover image "
                "to attach to the story file, but I was unable to find it, "
                "having looked for both 'Cover.png' and 'Cover.jpg' in the "
                "'Materials' folder for this project", cover_filename);
            return;
        }
        cover_art_format = "png";
        rv = get_png_dimensions(COVER_FILE, &width, &height);
        fclose(COVER_FILE);
        if (rv == 0) {
            release_problem(_P_(Untestable),
                "The cover image seems not to be a PNG despite the name",
                cover_filename);
            return;
        }
    }
    if ((width < 120) || (width > 1200) || (height < 120) || (height > 1200)) {
        release_problem(_P_(Untestable),
            "The height and width of the cover image, in pixels, must be "
            "between 120 and 1024 inclusive",
            cover_filename);
        return;
    }
    if ((width > 2*height) || (height > 2*width)) {
        release_problem(_P_(Untestable),
            "We recommend a square cover image, but at any rate it is "
            "required to be no more rectangular than twice as wide as it "
            "is high (or vice versa)",
            cover_filename);
```

```
            return;
        }
    }
```

This code is used in §5.

§8.

⟨WIB - Read header of existing story file if present 8⟩ ≡

```
    if (existing_story_file) {
        FILE *STORYF;
        int i;
        char story_filename[2*MAX_FILENAME_LENGTH];
        if (story_filename_extension == NULL)
            sprintf(story_filename, "%s%cstory.z5",
                materials_folder, FOLDER_SEPARATOR);
        else
            sprintf(story_filename, "%s%cstory.%s",
                materials_folder, FOLDER_SEPARATOR, story_filename_extension);
        STORYF = iso_fopen(story_filename, "rb");
        if (STORYF == NULL) {
            fatal_error2("Unable to find the promised existing story file",
                story_filename);
        }
        for (i=0; i<0x40; i++) header[i] = fgetc(STORYF);
        fclose(STORYF);
    }
```

This code is used in §5.

§9. For the format of this file, see the Treaty of Babel.

⟨WIB - Write iFiction record 9⟩ ≡

```
    fn = top_level_filename(METADATA_LEAFNAME);
    if (STREAM_OPEN_TO_FILE(xf, fn, UTF8_ENC) == FALSE) fatal_error2("Can't open metadata file",
fn);
    BEGIN_COMPILATION_MODE;
    COMPILATION_MODE_ENTER(COMPILE_TEXT_TO_XML_CMODE);
    write_ifiction_record(xf, story_format, header,
        cover_picture_number, cover_art_format, height, width);
    END_COMPILATION_MODE;
    STREAM_CLOSE(xf);
```

This code is used in §5.

§10. The release file is a set of instructions for how to turn the compiled story file, together with any picture resources and iFiction record, into a "blorb": one of the components of Inform 7 is its "releasing agent", presently `cblorb`, whose task is to act on the instructions we are about to write. This cannot be done directly by NI since the story file itself does not exist at any point during NI's run (NI merely produces I6 source code, which has to be run through I6 to obtain the story file). Besides, it makes good sense to separate these functions.

The format for the release file is an extension of the "blurb" format documented in the *Inform Designer's Manual*, fourth edition (the "DM4"). At some point we intend to publish a description of blurb 2.0, this extended version, but at present the format is in flux, and we are contemplating replacing the releasing agent with a better one; which may well change the blurb specification.

Note that the code below does not generate an `author` blurb instruction, which would lead to an AUTH chunk in the final blorb. This is partly because the AUTH chunk is now obsolete in the wake of the Treaty of Babel, but also because it avoids problems with Unicode: an AUTH can only contain plainest ASCII, whereas the author's name known to NI may very well use characters not representable in ASCII. There is no good way round this: so, farewell AUTH.

Similarly, we do not supply a release number. The release number of a blorb has a different meaning from that of the story file embedded in it: the number refers to the release of the picture and sound resources found in the blorb, and NI knows nothing about this, so it makes no comment.

⟨WIB - Write release blurb 10⟩ ≡

```
fn = top_level_filename(BLURB_LEAFNAME);
if (STREAM_OPEN_TO_FILE(xf, fn, ISO_ENC) == FALSE)
    fatal_error2("Can't open blurb file", fn);
write_release_blurb(xf, materials_folder, cover_picture_number, cover_art_format);
STREAM_CLOSE(xf);
```

This code is used in §5.

§11.

⟨WIB - Write manifest file 11⟩ ≡

```
manifest_filename = top_level_filename(MANIFEST_LEAFNAME);
if (STREAM_OPEN_TO_FILE(xf, manifest_filename, UTF8_ENC) == FALSE)
    fatal_error2("Can't open manifest file", fn);
fig_write_picture_manifest(xf, release_cover);
STREAM_CLOSE(xf);
```

This code is used in §5.

§12.

```
void write_ifiction_record(OUTPUT_STREAM, char *story_format, char *header,
    int cover_picture_number, char *cover_art_format, int height, int width) {
    WRITE("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
    WRITE("<ifindex version=\"1.0\" "
        "xmlns=\"http://babel.ifarchive.org/protocol/iFiction/\">\n"); INDENT;
    WRITE("<story>\n"); INDENT;
    ⟨Write the identification tag of the iFiction record 13⟩;
    ⟨Write the bibliographic tag of the iFiction record 14⟩;
    if (NUMBER_CREATED(auxiliary_file) > 0)
        ⟨Write the resources tag of the iFiction record 15⟩;
    if (release_cover)
        ⟨Write the cover tag of the iFiction record 16⟩;
    ⟨Write the releases tag of the iFiction record 17⟩;
```

```
    ⟨Write the colophon tag of the iFiction record 18⟩;
    ⟨Write the format-specific tag of the iFiction record 19⟩;
    OUTDENT; WRITE("</story>\n");
    OUTDENT; WRITE("</ifindex>\n");
}
```

§13.

⟨Write the identification tag of the iFiction record 13⟩ ≡

```
    WRITE("<identification>\n"); INDENT;
    WRITE("<ifid>%s</ifid>\n", uuid_text);
    if (existing_story_file) {
        WRITE("<ifid>ZCODE-%d-%c%c%c%c%c%c",
            header[2]*256+header[3],
            header[0x12], header[0x13], header[0x14],
            header[0x15], header[0x16], header[0x17]);
        if ((header[0x12] != '8') || (isdigit(header[0x12])))
            WRITE("-%04x", header[0x1c]*256 + header[0x1d]);
        WRITE("</ifid>\n");
    }
    WRITE("<format>%s</format>\n", story_format);
    OUTDENT; WRITE("</identification>\n");
```

This code is used in §12.

§14.

⟨Write the bibliographic tag of the iFiction record 14⟩ ≡

```
    WRITE("<bibliographic>\n"); INDENT;
    WRITE("<title>");
    if (write_bibliographic_datum(OUT, story_title, FALSE) == FALSE) WRITE("Untitled");
    WRITE("</title>\n");
    WRITE("<author>");
    if (write_bibliographic_datum(OUT, story_author, FALSE) == FALSE) WRITE("Anonymous");
    WRITE("</author>\n");
    WRITE("<headline>");
    if (write_bibliographic_datum(OUT, story_headline, FALSE) == FALSE)
        WRITE("An Interactive Fiction");
    WRITE("</headline>\n");
    WRITE("<genre>");
    if (write_bibliographic_datum(OUT, story_genre, FALSE) == FALSE) WRITE("Fiction");
    WRITE("</genre>\n");
    WRITE("<firstpublished>");
    if (write_bibliographic_datum(OUT, story_creation_year, FALSE) == FALSE)
        WRITE("%d", (the_present->tm_year)+1900);
    WRITE("</firstpublished>\n");
    write_bibliographic_datum(OUT, story_description, TRUE);
    WRITE("<language>en</language>\n");
    WRITE("<group>Inform</group>\n");
    if (episode_number >= 0) {
        WRITE("<seriesnumber>%d</seriesnumber>\n", episode_number);
        WRITE("<series>%s</series>\n", series_name);
    }
    OUTDENT; WRITE("</bibliographic>\n");
```

This code is used in §12.

§15.

⟨Write the resources tag of the iFiction record 15⟩ ≡

```
auxiliary_file *af;
WRITE("<resources>\n"); INDENT;
LOOP_OVER(af, auxiliary_file) {
    WRITE("<auxiliary>\n"); INDENT;
    WRITE("<leafname>");
    write_xml_safe_text(OUT, af->leafname_in_materials_folder);
    WRITE("</leafname>\n");
    WRITE("<description>");
    write_xml_safe_text(OUT, af->brief_description);
    WRITE("</description>\n");
    OUTDENT; WRITE("</auxiliary>\n");
}
OUTDENT; WRITE("</resources>\n");
```

This code is used in §12.

§16.

⟨Write the cover tag of the iFiction record 16⟩ ≡

```
WRITE("<cover>\n"); INDENT;
WRITE("<format>%s</format>\n", cover_art_format);
WRITE("<height>%d</height>\n", height);
WRITE("<width>%d</width>\n", width);
OUTDENT; WRITE("</cover>\n");
```

This code is used in §12.

§17.

⟨Write the releases tag of the iFiction record 17⟩ ≡

```
WRITE("<releases>\n"); INDENT;
WRITE("<attached>\n"); INDENT;
WRITE("<release>\n"); INDENT;
if (existing_story_file) {
    WRITE("<releasedate>%s%c%c-%c%c-%c%c</releasedate>\n",
        ((isdigit(header[0x12])) &&
            (header[0x12] != '8') && (header[0x12] != '9'))?"20":"19",
        header[0x12], header[0x13], header[0x14],
        header[0x15], header[0x16], header[0x17]);
    WRITE("<version>%d</version>\n", header[2]*256+header[3]);
    if ((isdigit(header[0x3c])) &&
        ((isdigit(header[0x3d])) || (header[0x3d] == '.')) &&
        (isdigit(header[0x3e])) && (isdigit(header[0x3f]))) {
        if (header[0x3d] == '.') {
            WRITE("<compiler>Inform 6</compiler>\n");
            WRITE("<compilerversion>%c%c%c%c</compilerversion>\n",
                header[0x3c], header[0x3d], header[0x3e], header[0x3f]);
        }
        else {
            WRITE("<compiler>Inform 1-5</compiler>\n");
```

```
                WRITE("<compilerversion>%c%c%c%c</compilerversion>\n",
                    header[0x3c], header[0x3d], header[0x3e], header[0x3f]);
            }
        } else {
            WRITE("<compiler>ZILCH</compiler>\n");
            WRITE("<compilerversion>%d</compilerversion>\n", header[0x00]);
        }
    } else {
        WRITE("<releasedate>%04d-%02d-%02d</releasedate>\n",
            (the_present->tm_year)+1900, (the_present->tm_mon)+1, the_present->tm_mday);
        if ((story_release_number != NULL) &&
            (qty_has_initial_value_set(story_release_number))) {
            WRITE("<version>");
            compile_quantity_initial(OUT, story_release_number);
            WRITE("</version>\n");
        } else WRITE("<version>1</version>\n");
        WRITE("<compiler>Inform 7</compiler>\n");
        WRITE("<compilerversion>%s</compilerversion>\n", NI_BUILD);
    }
    OUTDENT; WRITE("</release>\n");
    OUTDENT; WRITE("</attached>\n");
    OUTDENT; WRITE("</releases>\n");
```

This code is used in §12.

§18.

⟨Write the colophon tag of the iFiction record 18⟩ ≡

```
    WRITE("<colophon>\n"); INDENT;
    WRITE("<generator>Inform 7</generator>\n");
    WRITE("<generatorversion>%s</generatorversion>\n", NI_BUILD);
    WRITE("<originated>20%02d-%02d-%02d</originated>\n",
        (the_present->tm_year)-100, (the_present->tm_mon)+1, the_present->tm_mday);
    OUTDENT; WRITE("</colophon>\n");
```

This code is used in §12.

§19.

⟨Write the format-specific tag of the iFiction record 19⟩ ≡

```
    WRITE("<%s>\n", story_format); INDENT;
    if (existing_story_file) {
        WRITE("<serial>%c%c%c%c%c%c</serial>\n",
            header[0x12], header[0x13], header[0x14],
            header[0x15], header[0x16], header[0x17]);
        WRITE("<release>%d</release>\n", header[2]*256+header[3]);
        WRITE("<checksum>%04x</checksum>\n", header[0x1c]*256 + header[0x1d]);
        if ((isdigit(header[0x3c])) &&
            ((isdigit(header[0x3d])) || (header[0x3d] == '.')) &&
            (isdigit(header[0x3e])) && (isdigit(header[0x3f]))) {
            WRITE("<compiler>Inform v%c%c%c%c</compiler>\n",
                header[0x3c], header[0x3d], header[0x3e], header[0x3f]);
        } else {
            WRITE("<compiler>Infocom ZIL</compiler>\n");
        }
```

```
    } else {
        WRITE("<serial>%02d%02d%02d</serial>\n",
            (the_present->tm_year)-100, (the_present->tm_mon)+1, the_present->tm_mday);
        if ((story_release_number != NULL) &&
            (qty_has_initial_value_set(story_release_number))) {
            WRITE("<release>");
            compile_quantity_initial(OUT, story_release_number);
            WRITE("</release>\n");
        } else WRITE("<release>1</release>\n");
        WRITE("<compiler>Inform 7 build %s</compiler>\n", NI_BUILD);
    }
    if (release_cover)
        WRITE("<coverpicture>%d</coverpicture>\n", cover_picture_number);
    OUTDENT; WRITE("</%s>\n", story_format);
```

This code is used in §12.

§20. .

define BIBLIOGRAPHIC_TEXT_TRUNCATION 31

```
void write_release_blurb(OUTPUT_STREAM, char *materials_folder,
    int cover_picture_number, char *cover_art_format) {
    char story_file_leafname[128];
    WRITE("! Blurb file created by Inform build %s\n", NI_BUILD);
    WRITE("\n! Identification\n\n", NI_BUILD);
    WRITE("project folder \"%s\"\n", bundle_name);
    if ((release_website) || (release_booklet) || (release_cover) ||
        (release_solution) || (existing_story_file))
        WRITE("release to \"%s%cRelease\"\n", materials_folder, FOLDER_SEPARATOR);
    WRITE("\n! Blorb instructions\n\n", NI_BUILD);
    WRITE("storyfile leafname \"");
    if ((story_title != NULL) &&
        (qty_has_initial_value_set(story_title))) {
        BEGIN_COMPILATION_MODE;
        COMPILATION_MODE_ENTER(TRUNCATE_TEXT_CMODE);
        compile_quantity_initial(OUT, story_title);
        END_COMPILATION_MODE;
    } else WRITE("story");
    WRITE(".%s", get_VM_blorbed_extension());
    WRITE("\"\n");
    if (story_filename_extension == NULL)
        sprintf(story_file_leafname, "output.z5");
    else sprintf(story_file_leafname, "output.%s", story_filename_extension);
    if (existing_story_file) {
        if (story_filename_extension == NULL)
            WRITE("storyfile \"%s%cstory.z5\" include\n",
                materials_folder, FOLDER_SEPARATOR);
        else
            WRITE("storyfile \"%s%cstory.%s\" include\n",
                materials_folder, FOLDER_SEPARATOR, story_filename_extension);
    } else {
        WRITE("storyfile \"%s\" include\n",
```

```
        build_filename(story_file_leafname));
}
WRITE("ifiction \"%s\" include\n",
    top_level_filename(METADATA_LEAFNAME));
if (release_cover) {
    WRITE("cover \"%s%cCover.%s\"\n",
        materials_folder, FOLDER_SEPARATOR, cover_art_format);
    WRITE("picture %d \"%s%cCover.%s\"\n", cover_picture_number,
        materials_folder, FOLDER_SEPARATOR, cover_art_format);
} else {
    WRITE("cover \"%s%cReserved%cDefaultCover.jpg\"\n",
        pathname_of_built_in_extensions, FOLDER_SEPARATOR, FOLDER_SEPARATOR);
    WRITE("picture %d \"%s%cReserved%cDefaultCover.jpg\"\n", 1,
        pathname_of_built_in_extensions, FOLDER_SEPARATOR, FOLDER_SEPARATOR);
}
fig_write_blurb_commands(OUT, materials_folder);
sfx_write_blurb_commands(OUT, materials_folder);
WRITE("\n! Placeholder variables\n\n", NI_BUILD);
WRITE("placeholder [IFID] = \"%s\"\n", uuid_text);
if ((story_release_number != NULL) &&
    (qty_has_initial_value_set(story_release_number))) {
    WRITE("placeholder [RELEASE] = \"");
    compile_quantity_initial(OUT, story_release_number);
    WRITE("\"\n");
} else WRITE("placeholder [RELEASE] = \"1\"\n");
BEGIN_COMPILATION_MODE;
COMPILATION_MODE_ENTER(COMPILE_TEXT_TO_XML_CMODE);
if ((story_creation_year) && (qty_has_initial_value_set(story_creation_year))) {
    WRITE("placeholder [YEAR] = \"");
    compile_quantity_initial(OUT, story_creation_year);
    WRITE("\"\n");
} else WRITE("placeholder [YEAR] = \"%d\"\n", (the_present->tm_year)+1900);
if ((story_title) && (qty_has_initial_value_set(story_title))) {
    qty_treat_as_plain_text_word(story_title);
    WRITE("placeholder [TITLE] = \"");
    compile_quantity_initial(OUT, story_title);
    WRITE("\"\n");
} else WRITE("placeholder [TITLE] = \"Untitled\"\n");
if ((story_author) && (qty_has_initial_value_set(story_author))) {
    qty_treat_as_plain_text_word(story_author);
    WRITE("placeholder [AUTHOR] = \"");
    compile_quantity_initial(OUT, story_author);
    WRITE("\"\n");
} else WRITE("placeholder [AUTHOR] = \"Anonymous\"\n");
if ((story_description) && (qty_has_initial_value_set(story_description))) {
    qty_treat_as_plain_text_word(story_description);
    WRITE("placeholder [BLURB] = \"");
    compile_quantity_initial(OUT, story_description);
    WRITE("\"\n");
} else WRITE("placeholder [BLURB] = \"A work of interactive fiction.\"\n");
END_COMPILATION_MODE;
```

```
    WRITE("\n! Other material to release\n\n", NI_BUILD);
    if (release_source) {
        if (source_public) WRITE("source public\n");
        else WRITE("source\n");
    }
    if (release_solution) {
        if (solution_public) WRITE("solution public\n");
        else WRITE("solution\n");
    }
    if (release_card) {
        if (card_public) WRITE("ifiction public\n");
        else WRITE("ifiction\n");
    }
    auxiliary_file *af;
    LOOP_OVER(af, auxiliary_file)
        WRITE("auxiliary \"%s%c%s\" \"%s\"\n",
            materials_folder, FOLDER_SEPARATOR,
            af->leafname_in_materials_folder, af->brief_description);
    if (release_booklet)
        WRITE("auxiliary \"%s%cReserved%cIntroductionToIF.pdf\" \"Introduction to IF\"\n",
            pathname_of_built_in_extensions, FOLDER_SEPARATOR, FOLDER_SEPARATOR);
    if (release_website) {
        WRITE("\n! Website instructions\n\n", NI_BUILD);
        WRITE("template path \"%s%cTemplates\"\n", materials_folder, FOLDER_SEPARATOR);
        WRITE("template path \"%s\"\n", pathname_of_templates);
        WRITE("template path \"%s%cReserved%cTemplates\"\n",
            pathname_of_built_in_extensions, FOLDER_SEPARATOR, FOLDER_SEPARATOR);
        if (strcmp(website_template_leafname, "Classic") != 0) WRITE("css\n");
        WRITE("website \"%s\"\n", website_template_leafname);
    }
}
```

§**21. Bibliographic constants in I6.** Some of this bibliographic information is also accessible at run-time, and is used by the I6 library, for instance to produce the banner.

```
void compile_bibliographic_i6_constants(OUTPUT_STREAM) {
    encode_constant_text_bibliographically = TRUE;
    if ((story_title) && (qty_has_initial_value_set(story_title))) {
        qty_treat_as_plain_text_word(story_title);
        WRITE("Constant Story ");
        compile_quantity_initial(OUT, story_title);
        WRITE(";\n");
    } else {
        WRITE("Constant Story \"Welcome\";\n");
    }
    if ((story_headline) && (qty_has_initial_value_set(story_headline))) {
        qty_treat_as_plain_text_word(story_headline);
        WRITE("Constant Headline ");
        compile_quantity_initial(OUT, story_headline);
        WRITE(";\n");
    } else {
```

```
        WRITE("Constant Headline \"An Interactive Fiction\";\n");
    }
    if ((story_author) && (qty_has_initial_value_set(story_author))) {
        qty_treat_as_plain_text_word(story_author);
        WRITE("Constant Story_Author ");
        compile_quantity_initial(OUT, story_author);
        WRITE(";\n");
    }
    if ((story_release_number) && (qty_has_initial_value_set(story_release_number))) {
        WRITE("Release ");
        compile_quantity_initial(OUT, story_release_number);
        WRITE(";\n");
    }
    WRITE("Serial \"%02d%02d%02d\";\n",
        (the_present->tm_year)-100, (the_present->tm_mon)+1, the_present->tm_mday);
    encode_constant_text_bibliographically = FALSE;
}
```

The function compile_bibliographic_i6_constants is invoked by a command in a `.i6t` template file.

Figures 10/fig

Purpose

To register the names associated with picture resource numbers, which are defined to allow the final story file to display pictures, and to produce the thumbnail index of figures.

10/fig.§3-5 Blurb and manifest; §6 Thumbnail Index

Template interpreter commands

```
5   {-array:tableoffigures}
6   {-callv:index_figures}
```

Definitions

¶1. To be viable, figures have to be of an image format which Blorb recognises, and in any case we only allow two formats: JPEG and PNG.

```
typedef struct blorb_figure {
    int word_ref1, word_ref2;                    text of figure name
    int leafname_word;                           word number of text like "plant.jpg"
    int figure_number;                           resource number of this picture inside Blorb
    MEMORY_MANAGEMENT
} blorb_figure;
```

The structure blorb_figure is private to this section.

---

§**1.** Resources in a Blorb file have unique ID numbers which are positive integers, but these are not required to start from 1, nor to be contiguous. For Inform, ID number 1 is reserved for the cover image (whether or not any cover image is provided: it is legal for there to be figures but no cover, and vice versa). Other figures, and sound effects, then mix freely as needed from ID number 3 on upwards. We skip 2 so that it can be guaranteed that no sound resource has ID 1 or 2: this is to help people trying to play sounds in the Z-machine, where operand 1 or 2 in the `@sound` opcode signifies not a sound resource number but a long or short beep. If a genuine sound effect had resource ID 1 or 2, therefore, it would be unplayable on the Z-machine.

```
int next_free_resource_ID = 3;

int get_next_free_blorb_resource_ID(void) {
    return next_free_resource_ID++;
}
```

The function get_next_free_blorb_resource_ID is called from 10/sfx.

§**2.** Figure allocation now follows.

```
sentence_handler FIGURE_SH_handler =
    { SENTENCE_NT, FIGURE_VB, 1, handle_figure_definition };
void handle_figure_definition(parse_node *p) {
    register_figure(p->down->next->word_ref1,
        p->down->next->word_ref2,
        p->down->next->next->word_ref1,
        p->down->next->next->word_ref2);
}
void register_figure(int f1, int f2, int fn1, int fn2) {
    blorb_figure *bf; meaning_list *ml; int wn;
    if [[fn1, fn2 == of cover art]] {
        wn = -1;
    } else {
        if (is_kova(is_a_literal(fn1, fn2), TEXT_TY) == FALSE) {
            sentence_problem(_P_(C10PictureNotTextual),
                "a figure can only be declared as a quoted file name",
                "which should be the name of a JPEG or PNG image inside the "
                "project's Materials folder. For instance, 'Figure 2 is the "
                "file \"Crossed Swords.png\".'");
            return;
        }
        dequote_word(fn1);
        wn = fn1;
    }
    bf = CREATE(blorb_figure);
    bf->leafname_word = wn;
    bf->word_ref1 = f1; bf->word_ref2 = f2;
    if (wn >= 0)
        bf->figure_number = get_next_free_blorb_resource_ID();
    else
        bf->figure_number = 1;
    LOGIF(FIGURE_CREATIONS, "Created figure <$W> = filename '%s' = resource ID %d\n",
        f1, f2, (wn>=0)?lw_array[wn].lw_text:"<cover art>", bf->figure_number);
    ml = SP_excerpt(MISCELLANEOUS_MC, f1, f2);
    if ((ml != NULL) && (em_get_secondary_code(ml_meaning(ml)) == FIGURE_SMC)) {
        sentence_problem(_P_(C10PictureDuplicate),
            "this is already the name of a Figure",
            "so there must be some duplication somewhere.");
        return;
    }
    register_excerpt_meaning(
        MISCELLANEOUS_MC, FIGURE_SMC, f1, f2, STORE_POINTER_blorb_figure(bf));
}
int fig_get_resource_ID(blorb_figure *bf) {
    return bf->figure_number;
}
```

The function fig_get_resource_ID is called from 5/lit.

§**3. Blurb and manifest.** The picture manifest is used by the implementation of Glulx within the Inform application to connect picture ID numbers with filenames relative to the Materials folder for its project.

```
void fig_write_picture_manifest(OUTPUT_STREAM, int releasing_cover) {
    blorb_figure *bf;
    WRITE("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
    WRITE("<!DOCTYPE plist PUBLIC \"-//Apple Computer//DTD PLIST 1.0//EN\" "
        "\"http://www.apple.com/DTDs/PropertyList-1.0.dtd\">\n");
    WRITE("<plist version=\"1.0\">\n"); INDENT;
    WRITE("<dict>\n"); INDENT;
    WRITE("<key>Graphics</key>\n");
    WRITE("<dict>\n"); INDENT;
    if (releasing_cover) {
        WRITE("<key>1</key>\n");
        WRITE("<string>Cover.jpg</string>\n");
    }
    LOOP_OVER(bf, blorb_figure) {
        if (bf->leafname_word == -1) continue;
        WRITE("<key>%d</key>\n", bf->figure_number);
        WRITE("<string>Figures%c%s</string>\n", FOLDER_SEPARATOR, lw_array[bf->leafname_word].lw_text);
    }
    OUTDENT; WRITE("</dict>\n");
    sfx_write_sounds_manifest(OUT);
    OUTDENT; WRITE("</dict>\n");
    OUTDENT; WRITE("</plist>\n");
}
```

The function fig_write_picture_manifest is called from 10/bib.

§**4.** The following writes Blurb commands for all of the figures, but not for the cover art, which is handled by Bibliographic Data.

```
void fig_write_blurb_commands(OUTPUT_STREAM, char *resources) {
    blorb_figure *bf;
    LOOP_OVER(bf, blorb_figure) {
        if (bf->leafname_word == -1) continue;
        WRITE("picture %d \"%s%cFigures%c%s\"\n",
            bf->figure_number, resources,
            FOLDER_SEPARATOR, FOLDER_SEPARATOR, lw_array[bf->leafname_word].lw_text);
    }
}
```

The function fig_write_blurb_commands is called from 10/bib.

§5.

```
void compile_tableoffigures_array(OUTPUT_STREAM) {
    WRITE("Array TableOfFigures --> ");
    blorb_figure *bf;
    LOOP_OVER(bf, blorb_figure) WRITE("%d ", bf->figure_number);
    WRITE(" 0 0;\n");
    WRITE("[ PrintFigureName f;\n"); INDENT;
    WRITE("switch(f) {\n"); INDENT;
    LOOP_OVER(bf, blorb_figure) {
        WRITE("%d: print \"", bf->figure_number);
        print_raw_text_to_file(bf->word_ref1, bf->word_ref2, OUT);
        WRITE("\";\n");
    }
    WRITE("default: print \"<no-such-figure>\";\n");
    OUTDENT; WRITE("}\n");
    OUTDENT; WRITE("];\n");
}
```

The function compile_tableoffigures_array is invoked by a command in a `.i6t` template file.

§6. **Thumbnail Index.** The index is presented with thumbnails of a given pixel width, which the HTML renderer automatically scales to fit. Height is adjusted so as to match this width, preserving the aspect ratio.

define THUMBNAIL_WIDTH 80

```
void index_figures(void) {
    blorb_figure *bf; int i; FILE *FIGURE_FILE;
    char resources[MAX_FILENAME_LENGTH];
    char image_filename[MAX_FILENAME_LENGTH];
    char line2[MAX_FILENAME_LENGTH];
    int rv;
    if (NUMBER_CREATED(blorb_figure) < 2) return;                    cover art always creates 1
    strcpy(resources, bundle_name);
    i = strlen(resources)-1;
    while ((i>0) && (resources[i] != '.')) i--;
    if (i>0) sprintf(resources+i, " Materials%cFigures", FOLDER_SEPARATOR);
    INDEX("<p><b>List of Figures</b> (<i>About figures</i>");
    index_doc_link("FIGURES"); INDEX(")<p>\n");
    begin_html_table(ifl, "#ffffff", TRUE, 0, 0, 0, 0, 0);
    LOOP_OVER(bf, blorb_figure) {
        unsigned int width, height;
        if (bf->leafname_word == -1) continue;
        sprintf(image_filename, "%s%c%s", resources, FOLDER_SEPARATOR,
            lw_array[bf->leafname_word].lw_text);
        rv = 0;
        FIGURE_FILE = iso_fopen(image_filename, "rb");
        if (FIGURE_FILE) {
            char *real_format = "JPEG";
            rv = get_jpg_dimensions(FIGURE_FILE, &width, &height);
            fclose(FIGURE_FILE);
            if (rv == 0) {
                FIGURE_FILE = iso_fopen(image_filename, "rb");
```

```
            if (FIGURE_FILE) {
                real_format = "PNG";
                rv = get_png_dimensions(FIGURE_FILE, &width, &height);
                fclose(FIGURE_FILE);
            }
        }
        if (rv == 0) {
            sprintf(line2, "<i>Unknown image format</i><br>\n");
        } else {
            sprintf(line2, "%s format: %d (width) by %d (height) pixels<br>\n",
                real_format, width, height);
        }
    } else {
        sprintf(line2, "<i>Missing from the Materials%cFigures folder</i><br>\n",
            FOLDER_SEPARATOR);
    }
    first_html_column(ifl, THUMBNAIL_WIDTH+10);
    if (rv == 0)
        INDEX("<img border=\"0\" src=\"inform:/doc_images/image_problem.png\"> ");
    else
        INDEX("<img border=\"1\" src=\"file://%s\" width=\"%d\" height=\"%d\"> ",
        image_filename, THUMBNAIL_WIDTH, THUMBNAIL_WIDTH*height/width);
    next_html_column(ifl, 0);
    print_raw_text_to_file(bf->word_ref1, bf->word_ref2, ifl);
    index_link(lw_array[bf->word_ref1].lw_source);
    INDEX("<br>\n%sFilename: \"%s\" - resource number %d",
        line2, lw_array[bf->leafname_word].lw_text, bf->figure_number);
    end_html_row(ifl);
}
end_html_table(ifl);
INDEX("<p>");
}
```

The function index_figures is invoked by a command in a `.i6t` template file.

Sound Effects 10/sfx

Purpose

To register the names associated with sound resource numbers, which are defined to allow the final story file to play sound effects, and to produce the index of sound effects.

10/sfx.§1-3 Blurb and manifest; §4 Sounds Index

Template interpreter commands

```
3   {-array:tableofsounds}
4   {-callv:index_sounds}
```

Definitions

¶1. To be viable, sound files have to be of a format which Blorb recognises, and in any case we only allow two formats: AIFF (uncompressed) and OGG (compressed).

Sound effects and images have unique ID numbers (i.e., we cannot have both sound number 45 and also image 45): these are positive integers, but are not required to start from 1, nor to be contiguous. For Inform, ID number 1 is reserved for the cover image (whether or not any cover image is provided: it is legal for there to be figures but no cover, and vice versa). Resource numbers are then numbered contiguously upwards from 2.

```
typedef struct blorb_sound {
    int word_ref1, word_ref2;                    text of sound name
    int sfx_leafname_word;                       word number of text like "eerie.ogg"
    int sound_number;                            resource number of this picture inside Blorb
    MEMORY_MANAGEMENT
} blorb_sound;
```

The structure blorb_sound is private to this section.

```
sentence_handler SOUND_SH_handler =
    { SENTENCE_NT, SOUND_VB, 1, handle_sound_definition };
void handle_sound_definition(parse_node *p) {
    register_sound(p->down->next->word_ref1,
        p->down->next->word_ref2,
        p->down->next->next->word_ref1,
        p->down->next->next->word_ref2);
}
void register_sound(int f1, int f2, int fn1, int fn2) {
    blorb_sound *bs; meaning_list *ml; int wn;
    if (is_kova(is_a_literal(fn1, fn2), TEXT_TY) == FALSE) {
        sentence_problem(_P_(C10SoundNotTextual),
            "a sound effect can only be declared as a quoted file name",
            "which should be the name of an AIFF or OGG file inside the "
            "Sounds subfolder of the project's Materials folder. For "
            "instance, 'Sound of Swordplay is the file "
            "\"Crossed Swords.aiff\".'");
        return;
```

```
    }
    dequote_word(fn1);
    wn = fn1;
    bs = CREATE(blorb_sound);
    bs->sfx_leafname_word = wn;
    bs->word_ref1 = f1; bs->word_ref2 = f2;
    bs->sound_number = get_next_free_blorb_resource_ID();
    LOGIF(FIGURE_CREATIONS, "Created sound effect <$W> = filename '%s' = resource ID %d\n",
        f1, f2, lw_array[wn].lw_text, bs->sound_number);
    ml = SP_excerpt(MISCELLANEOUS_MC, f1, f2);
    if ((ml != NULL) && (em_get_secondary_code(ml_meaning(ml)) ==
        SOUND_EFFECT_SMC)) {
        sentence_problem(_P_(C10SoundDuplicate),
            "this is already the name of a sound effect",
            "so there must be some duplication somewhere.");
        return;
    }
    register_excerpt_meaning(
        MISCELLANEOUS_MC, SOUND_EFFECT_SMC, f1, f2,
        STORE_POINTER_blorb_sound(bs));
}

int sfx_get_resource_ID(blorb_sound *bs) {
    return bs->sound_number;
}
```

The function sfx_get_resource_ID is called from 5/lit.

§**1. Blurb and manifest.** The sounds manifest is used by the implementation of Glulx within the Inform application to connect picture ID numbers with filenames relative to the `Materials/Sounds` folder for its project. (It's part of the XML manifest file created from `Figures.w`.)

```
void sfx_write_sounds_manifest(OUTPUT_STREAM) {
    blorb_sound *bs;
    if (NUMBER_CREATED(blorb_sound) == 0) return;
    WRITE("<key>Sounds</key>\n");
    WRITE("<dict>\n"); INDENT;
    LOOP_OVER(bs, blorb_sound) {
        WRITE("<key>%d</key>\n", bs->sound_number);
        WRITE("<string>Sounds%c%s</string>\n", FOLDER_SEPARATOR, lw_array[bs->sfx_leafname_word].lw_text);
    }
    OUTDENT; WRITE("</dict>\n");
}
```

The function sfx_write_sounds_manifest is called from 10/fig.

§**2.** The following writes Blurb commands for all of the sounds.

```
void sfx_write_blurb_commands(OUTPUT_STREAM, char *resources) {
    blorb_sound *bs;
    LOOP_OVER(bs, blorb_sound) {
        WRITE("sound %d \"%s%cSounds%c%s\"\n",
            bs->sound_number, resources,
            FOLDER_SEPARATOR, FOLDER_SEPARATOR, lw_array[bs->sfx_leafname_word].lw_text);
    }
}
```

The function sfx_write_blurb_commands is called from 10/bib.

§**3.**

```
void compile_tableofsounds_array(OUTPUT_STREAM) {
    WRITE("Array TableOfSounds --> ");
    blorb_sound *bs;
    LOOP_OVER(bs, blorb_sound) WRITE("%d ", bs->sound_number);
    WRITE(" 0 0;\n");
    WRITE("[ PrintSoundName f;\n"); INDENT;
    WRITE("switch(f) {\n"); INDENT;
    WRITE("0: print \"<silence>\";\n");
    LOOP_OVER(bs, blorb_sound) {
        WRITE("%d: print \"", bs->sound_number);
        print_raw_text_to_file(bs->word_ref1, bs->word_ref2, OUT);
        WRITE("\";\n");
    }
    WRITE("default: print \"<no-such-sound>\";\n");
    OUTDENT; WRITE("}\n");
    OUTDENT; WRITE("];\n");
}
```

The function compile_tableofsounds_array is invoked by a command in a `.i6t` template file.

§**4. Sounds Index.** The index is only a little helpful for sounds.

```
void index_sounds(void) {
    blorb_sound *bs; int i; FILE *SOUND_FILE;
    char resources[MAX_FILENAME_LENGTH];
    char image_filename[MAX_FILENAME_LENGTH];
    char line2[MAX_FILENAME_LENGTH];
    int rv;
    if (NUMBER_CREATED(blorb_sound) == 0) return;
    strcpy(resources, bundle_name);
    i = strlen(resources)-1;
    while ((i>0) && (resources[i] != '.')) i--;
    if (i>0) sprintf(resources+i, " Materials%cSounds", FOLDER_SEPARATOR);
    INDEX("<p><b>List of Sounds</b> (<i>About sounds</i>");
    index_doc_link("SOUNDS"); INDEX(")<p>\n");
    begin_html_table(ifl, "#ffffff", TRUE, 0, 0, 0, 0, 0);
    LOOP_OVER(bs, blorb_sound) {
        unsigned int duration, pBitsPerSecond, pChannels, pSampleRate, fsize,
            midi_version, no_tracks;
```

```
int preview = TRUE, waveform_style = TRUE;
sprintf(image_filename, "%s%c%s", resources, FOLDER_SEPARATOR,
    lw_array[bs->sfx_leafname_word].lw_text);
rv = 0;
SOUND_FILE = iso_fopen(image_filename, "rb");
if (SOUND_FILE) {
    char *real_format = "AIFF";
    rv = get_aiff_duration(SOUND_FILE, &duration, &pBitsPerSecond,
        &pChannels, &pSampleRate);
    fseek(SOUND_FILE, 0, SEEK_END);
    fsize = (int) (ftell(SOUND_FILE));
    fclose(SOUND_FILE);
    if (rv == 0) {
        SOUND_FILE = iso_fopen(image_filename, "rb");
        if (SOUND_FILE) {
            real_format = "Ogg Vorbis";
            preview = FALSE;
            rv = get_ogg_duration(SOUND_FILE, &duration, &pBitsPerSecond,
                &pChannels, &pSampleRate);
            fclose(SOUND_FILE);
        }
    }
    if (rv == 0) {
        SOUND_FILE = iso_fopen(image_filename, "rb");
        if (SOUND_FILE) {
            waveform_style = FALSE;
            real_format = "MIDI";
            preview = TRUE;
            rv = get_midi_file_information(SOUND_FILE,
                &midi_version, &no_tracks);
            fclose(SOUND_FILE);
        }
    }
    if (rv == 0) {
        sprintf(line2, "<i>Unknown sound format</i><br>\n");
    } else {
        if (waveform_style == FALSE)
            sprintf(line2, "Type %d %s file with %d track%s<br>\n"
                "<i>Warning: not officially supported in glulx yet</i><br>\n",
                midi_version, real_format, no_tracks,
                (no_tracks == 1)?"":"s");
        else {
            int min = (duration/6000), sec = (duration%6000)/100,
                centisec = (duration%100);
            sprintf(line2, "%d.%01dKB %s file: duration ",
                fsize/1024, (fsize%1024)/102, real_format);
            if (min > 0)
                sprintf(line2+strlen(line2), "%d minutes ", min);
            if ((sec > 0) || (centisec > 0)) {
                if (centisec == 0)
                    sprintf(line2+strlen(line2), "%d seconds<br>", sec);
                else
                    sprintf(line2+strlen(line2), "%d.%02d seconds<br>",
```

```
                            sec, centisec);
                    } else sprintf(line2+strlen(line2), "exactly<br>");
                    sprintf(line2+strlen(line2),
                        "Sampled as %d.%01dkHz %s (%d.%01d kilobits/sec)<br>",
                        pSampleRate/1000, (pSampleRate%1000)/100,
                        (pChannels==1)?"Mono":"Stereo",
                        pBitsPerSecond/1000, (pSampleRate%1000)/100);
                }
            }
        } else {
            sprintf(line2, "<i>Missing from the Materials%cSounds folder</i><br>\n",
                FOLDER_SEPARATOR);
        }
        first_html_column(ifl, THUMBNAIL_WIDTH+10);
        if (rv == 0)
            INDEX("<img border=\"0\" src=\"inform:/doc_images/image_problem.png\"> ");
        else {
            if (preview)
            INDEX("<embed src=\"file://%s\" width=\"%d\" height=\"64\" "
            "autostart=\"false\" volume=\"50%%\" mastersound></embed> ",
                image_filename, THUMBNAIL_WIDTH);
            else
            INDEX("<img border=\"0\" src=\"inform:/doc_images/sound_okay.png\"> ");
        }
        next_html_column(ifl, 0);
        print_raw_text_to_file(bs->word_ref1, bs->word_ref2, ifl);
        index_link(lw_array[bs->word_ref1].lw_source);
        INDEX("<br>\n%sFilename: \"%s\" - resource number %d",
            line2, lw_array[bs->sfx_leafname_word].lw_text, bs->sound_number);
        end_html_row(ifl);
    }
    end_html_table(ifl);
    INDEX("<p>");
}
```

The function index_sounds is invoked by a command in a `.i6t` template file.

External Files 10/exf

Purpose

To register the names associated with external files, and build the small I6 arrays associated with each.

10/exf.§2-3 I6 arrays of file structures; §4 External Files Index

Template interpreter commands

```
3   {-callv:external_file_arrays}
4   {-callv:index_external_files}
```

Definitions

¶1. Each file can be text or binary, has a name, and can be owned by a this project, by an unspecified other project, or by a project named by IFID.

```
define OWNED_BY_THIS_PROJECT 1
define OWNED_BY_ANOTHER_PROJECT 2
define OWNED_BY_SPECIFIC_PROJECT 3

typedef struct external_file {
    int word_ref1, word_ref2;                    text of name
    int unextended_filename;                     word number of text like "bones"
    char exf_I6_identifier[32];                  an I6 identifier
    int file_is_binary;                          true or false
    int file_ownership;                          one of the above
    char IFID_of_owner[48];                      an I6 identifier
    MEMORY_MANAGEMENT
} external_file;
```

The structure external_file is private to this section.

§**1.** Very little needs to be done with files in I7 itself.

```
sentence_handler FILE_SH_handler =
    { SENTENCE_NT, FILE_VB, 1, handle_file_definition };

void handle_file_definition(parse_node *p) {
    register_file(p->down->next->word_ref1,
        p->down->next->word_ref2,
        p->down->next->next->word_ref1,
        p->down->next->next->word_ref2);
}

void register_file(int f1, int f2, int fn1, int fn2) {
    external_file *exf; meaning_list *ml; char *p; int i;
    int bad_filename = FALSE;
    if ((fn1 != fn2) || (vocab_test_flags(fn1, TEXT_MC) == FALSE)) {
        sentence_problem(_P_(C10FilenameNotTextual),
            "a file can only be called with a single quoted piece of text",
            "as in: 'The File of Wisdom is called \"wisdom\".'");
        return;
```

```
}
p = lw_array[fn1].lw_text;
if (strlen(p) < 5) bad_filename = TRUE;
if (isalpha(p[1]) == FALSE) bad_filename = TRUE;
for (i=0; p[i]; i++) {
    if (p[i] == '"') {
        if ((i==0) || (p[i+1] == 0)) continue;
    }
    if (i>24) bad_filename = TRUE;
    if ((isalpha(p[i])) || (isdigit(p[i]))) continue;
    LOG("Objected to character %c\n", p[i]);
    bad_filename = TRUE;
}
if (bad_filename) {
    LOG("Filename: %s\n", p);
    sentence_problem(_P_(C10FilenameUnsafe),
        "filenames must be very conservatively chosen",
        "in order to be viable on a wide range of computers. They must "
        "consist of 3 to 23 English letters or digits, with the first being "
        "a letter. Spaces are not allowed, and nor are periods. (A file "
        "extension, such as '.glkdata', may be added on some platforms "
        "automatically: this is invisible to Inform.)");
    return;
}
exf = CREATE(external_file);
exf->unextended_filename = fn1;
exf->file_is_binary = FALSE;
exf->file_ownership = OWNED_BY_THIS_PROJECT;
if [[f1, f2 == ... CLOSEBRACKET]] {
    for (i=f2-4; i>f1; i--) {
        if [[i, f2 == OPENBRACKET owned by ... CLOSEBRACKET]] {
            int own1 = i+3, own2 = f2-1;
            f2 = i-1;
            LOGIF(FIGURE_CREATIONS, "External file owner: $W\n", own1, own2);
            if [[own1, own2 == another project]] {
                exf->file_ownership = OWNED_BY_ANOTHER_PROJECT;
                break;
            }
            if [[own1, own2 == project ###]] {
                if (vocab_test_flags(own2, TEXT_MC)) {
                    int j, invalid = FALSE;
                    p = lw_array[own2].lw_text;
                    for (j=1; (j<47) && (p[j]); j++) {
                        if ((p[j] == '"') && (p[j+1] == 0)) break;
                        exf->IFID_of_owner[j-1] = p[j];
                        if ((isalpha(p[j])) || (isdigit(p[j]))) continue;
                        if (p[j] == '-') continue;
                        invalid = TRUE;
                        LOG("Objected to character %c\n", p[j]);
                    }
                    exf->IFID_of_owner[j-1] = 0;
                    if ((invalid) || (j==47)) {
                        sentence_problem(_P_(C10BadFileIFID),
```

```
                                "the owner of the file should be specified "
                                "using a valid double-quoted IFID",
                                "as in: 'The File of Wisdom (owned by project "
                                "\"4122DDA8-A153-46BC-8F57-42220F9D8795\") "
                                "is called \"wisdom\".'");
                            } else
                                exf->file_ownership = OWNED_BY_SPECIFIC_PROJECT;
                            break;
                        }
                    }
                    sentence_problem(_P_(C10BadFileOwner),
                        "the owner of this file is wrongly specified",
                        "since it is not one of the three possibilities - "
                        "(1) Specify nothing: making the file belong to this "
                        "project. (2) Specify only that it belongs to someone, "
                        "without saying whom: 'The File of Wisdom (owned by "
                        "another project) is called \"wisdom\".' (3) Specify "
                        "that it belongs to a project with a given double-quoted "
                        "IFID: 'The File of Wisdom (owned by project "
                        "\"4122DDA8-A153-46BC-8F57-42220F9D8795\") "
                        "is called \"wisdom\".'");
                }
            }
        }
        [[f1, f2 == the ... --> f1, f2]];
        if [[f1, f2 == text ... --> f1, f2]] exf->file_is_binary = FALSE;
        else if [[f1, f2 == binary ... --> f1, f2]] exf->file_is_binary = TRUE;
        exf->word_ref1 = f1; exf->word_ref2 = f2;
        LOGIF(FIGURE_CREATIONS, "Created external file <$W> = filename '%s'\n",
            f1, f2, lw_array[exf->unextended_filename].lw_text);
        ml = SP_excerpt(MISCELLANEOUS_MC, f1, f2);
        if ((ml != NULL) &&
            (em_get_secondary_code(ml_meaning(ml)) == EXTERNAL_FILE_SMC)) {
            sentence_problem(_P_(C10FilenameDuplicate),
                "this is already the name of a file",
                "so there must be some duplication somewhere.");
            return;
        }
        register_excerpt_meaning(
            MISCELLANEOUS_MC, EXTERNAL_FILE_SMC,
                f1, f2, STORE_POINTER_external_file(exf));
        isn_compose_identifier(exf->exf_I6_identifier, 'X', exf->allocation_id,
            exf->word_ref1, exf->word_ref2);
}
```

§2. **I6 arrays of file structures.** External files are written in I6 as their array names:

```
void compile_exf_constant(OUTPUT_STREAM, int id) {
    external_file *exf;
    LOOP_OVER(exf, external_file) {
        if (exf->allocation_id == id) {
            WRITE("%s", exf->exf_I6_identifier);
            return;
        }
    }
    LOG("ID: %d\n", id);
    internal_error("no exf structure with this ID");
}
```

The function compile_exf_constant is called from 7/vasp.

§3. And the following declares the arrays:

```
void external_file_arrays(OUTPUT_STREAM) {
    external_file *exf;
    WRITE("Array TableOfExternalFiles --> ");
    LOOP_OVER(exf, external_file) WRITE("%s ", exf->exf_I6_identifier);
    WRITE("0 0;\n");
    LOOP_OVER(exf, external_file) {
        WRITE("Array %s --> AUXF_MAGIC_VALUE AUXF_STATUS_IS_UNUSED %s 0 0\n",
            exf->exf_I6_identifier,
            (exf->file_is_binary)?"true":"false");
        INDENT;
        WRITE("%s ", lw_array[exf->unextended_filename].lw_rawtext);
        switch (exf->file_ownership) {
            case OWNED_BY_THIS_PROJECT:
                WRITE("UUID_ARRAY;\n");
                break;
            case OWNED_BY_ANOTHER_PROJECT:
                WRITE("NULL;\n");
                break;
            case OWNED_BY_SPECIFIC_PROJECT:
                WRITE("IFID_ARRAY_%d;\n", exf->allocation_id);
                WRITE("Array IFID_ARRAY_%d string \"//%s//\";\n",
                    exf->allocation_id, exf->IFID_of_owner);
                break;
        }
        OUTDENT;
    }
    WRITE("[ PrintExternalFileName f;\n"); INDENT;
    WRITE("switch(f) {\n"); INDENT;
    LOOP_OVER(exf, external_file) {
        WRITE("%s: print \"", exf->exf_I6_identifier);
        print_raw_text_to_file(exf->word_ref1, exf->word_ref2, OUT);
        WRITE("\";\n");
    }
    WRITE("default: print \"<no-such-file>\";\n");
    OUTDENT; WRITE("}\n");
    OUTDENT; WRITE("];\n");
```

```
}
```

The function external_file_arrays is invoked by a command in a .i6t template file.

§4. **External Files Index.** More or less perfunctory, but still of some use, if only as a list.

```
void index_external_files(void) {
    external_file *exf;
    if (NUMBER_CREATED(external_file) == 0) return;
    INDEX("<p><b>List of External Files</b> (<i>About files</i>");
    index_doc_link("EFILES"); INDEX(")<p>\n");
    begin_html_table(ifl, "#ffffff", TRUE, 0, 0, 0, 0, 0);
    LOOP_OVER(exf, external_file) {
        first_html_column(ifl, THUMBNAIL_WIDTH+10);
        if (exf->file_is_binary)
            INDEX("<img border=\"0\" src=\"inform:/doc_images/exf_binary.png\"> ");
        else {
            INDEX("<img border=\"0\" src=\"inform:/doc_images/exf_text.png\"> ");
        }
        next_html_column(ifl, 0);
        print_raw_text_to_file(exf->word_ref1, exf->word_ref2, ifl);
        index_link(lw_array[exf->word_ref1].lw_source);
        INDEX("<br>\nFilename: %s %s- owned by ",
            (exf->file_is_binary)?"- binary ":"",
            lw_array[exf->unextended_filename].lw_text);
        switch (exf->file_ownership) {
            case OWNED_BY_THIS_PROJECT: INDEX("this project"); break;
            case OWNED_BY_ANOTHER_PROJECT: INDEX("another project"); break;
            case OWNED_BY_SPECIFIC_PROJECT:
                INDEX("project with IFID number <b>%s</b>",
                    exf->IFID_of_owner);
                break;
        }
        end_html_row(ifl);
    }
    end_html_table(ifl);
    INDEX("<p>");
}
```

The function index_external_files is invoked by a command in a .i6t template file.

Inform 6 Inclusions 10/isin

Purpose

To include Inform 6 code almost verbatim in the output of NI, as instructed by low-level Inform 7 sentences.

Template interpreter commands

```
2   {-callv:compile_use_options}
2   {-routine:TestUseOption}
2   {-callv:compile_icl_commands}
```

Definitions

¶1. The preferred way to pass "do it this way, not that way" instructions to NI, in the spirit of natural-language input, is not to use command-line arguments in the shell but to write suitable sentences in the source text. For instance:

> Use American dialect and the serial comma.

Use options like "American dialect" take the place of what would be switches like `--american-dialect` in a conventional Unix tool. They are themselves defined by NI source (almost always the Standard Rules). They generally control run-time behaviour, and as a result NI deals with them simply by defining suitable I6 constants in the I6 code being output: conditional compilation of the standard I6 library then delivers the desired result. Anyway, the use options known to NI are stored in the following small structure.

Two use options are special: "memory economy" and "dynamic memory allocation". These can affect some of the run-time data generated by NI, and have effects which are not confined to conditional compilation at the I6 stage.

```
typedef struct use_option {
    int word_ref1, word_ref2;                    word range where name is stored
    struct parse_node *option_expansion;           definition as given in source
    int option_used;                             set if this option has been taken
    int source_file_scoped;                    scope is the current source file only?
    int minimum_setting_value;                     for those which are numeric
    MEMORY_MANAGEMENT
} use_option;

int memory_economy_in_force = FALSE;
int dynamic_memory_allocation = 0;
```

The structure use_option is private to this section.

¶2. We can also meddle with the I6 memory settings which will be used to finish compiling the story file, when NI has done its work. We need this because we have no practical way for NI to predict when it is going to generate code which will break I6's limits: the only reasonable way it can work is for the user to hit the limit occasionally, and then raise that limit by hand with a sentence in the I7 source text.

```
typedef struct i6_memory_setting {
    char ICL_identifier[64];                    see the DM4 for the I6 memory setting names
    int number;                                 e.g., 50000 means "at least 50,000"
    MEMORY_MANAGEMENT
} i6_memory_setting;
```

The structure i6_memory_setting is private to this section.

¶3. Inclusions are managed with the following modest structure.

```
typedef struct i6_inclusion_matter {
    struct parse_node *material_to_include;     normally an I6 escape (- ... -)
    struct world_object *wo_to_include_with;    or into an object or class definition
    MEMORY_MANAGEMENT
} i6_inclusion_matter;
```

The structure i6_inclusion_matter is private to this section.

---

§**1.** Inclusions are primitive things, but still a little more is involved than cut and paste.

```
sentence_handler INFORM6CODE_SH_handler =
    { INFORM6CODE_NT, -1, 2, inform_6_inclusion };
void inform_6_inclusion(parse_node *PN) {
    int iw1 = PN->word_ref1+3, iw2 = PN->word_ref2, sw1 = -1, stage = -2;
    world_object *wo;
    i6_inclusion_matter *inclm;
    if (iw1 > iw2) {
        I6T_intervention_new(1, "Output.i6t", "I6 Inclusions",
            lw_array[PN->word_ref1 + 2].lw_rawtext, NULL);
        return;
    }
    if [[iw1, iw2 == before ... --> sw1, iw2]] stage = -1;
    if [[iw1, iw2 == instead of ... --> sw1, iw2]] stage = 0;
    if [[iw1, iw2 == after ... --> sw1, iw2]] stage = 1;
    if ((stage != -2) && (vocab_test_flags(sw1, TEXT_MC))) {
        if [[sw1, iw2 == ###]] {
            dequote_word(sw1);
            I6T_intervention_new(stage, lw_array[sw1].lw_text, NULL, lw_array[PN->word_ref1
+ 2].lw_rawtext, NULL);
            return;
        }
        if ([[sw1, iw2 == ### in ###]] && (vocab_test_flags(iw2, TEXT_MC)))  {
            dequote_word(sw1); dequote_word(iw2);
            I6T_intervention_new(stage, lw_array[iw2].lw_text, lw_array[sw1].lw_text, lw_array[PN->word_ref1
+ 2].lw_rawtext, NULL);
            return;
```

```
        }
    }
    if [[iw1, iw2 == when defining ... --> iw1, iw2]] {
        wo = NULL;
        specification *spec = parse_expression(iw1, iw2, TYPE_EXPCON);
        if (spec_is_UNKNOWN(spec)) {
            sentence_problem(_P_(C10WhenDefiningUnknown),
                "I do not understand what definition you're referring to",
                "so I can't make an Inform 6 inclusion there.");
            return;
        }
        if ((species_is(spec, DESCRIPTION_SPC)) && (spec_get_described_kind(spec)))
            wo = spec_get_described_kind(spec);
        else if ((species_is(spec, DESCRIPTION_SPC)) && (spec_get_described_object(spec)))
            wo = spec_get_described_object(spec);
        else if (spec_is_actual_CONSTANT_of_kova(spec, OBJECT_TY))
            wo = OBJECT_spec_to_world_object(spec);
        if (wo) {
            inclm = CREATE(i6_inclusion_matter);
            inclm->material_to_include = PN;
            inclm->wo_to_include_with = wo;
            return;
        }
    }
    current_sentence = PN;
    if [[iw1, iw2 == before the library]] {
        sentence_problem(_P_(C10BeforeTheLibrary),
            "this syntax was withdrawn in January 2008",
            "in favour of a more finely controlled I6 inclusion command. "
            "The effect you want can probably be achieved by writing "
            "'after \"Definitions.i6t\".' instead of 'before the library.'");
        return;
    }
    sentence_problem(_P_(C10BadI6Inclusion),
        "this is not a form of I6 code inclusion I recognise",
        "because the clause at the end telling me where to put the code "
        "excerpt is not one of the possibilities I know. The clause can "
        "either be blank (in which case I'll find somewhere sensible to "
        "put it), or 'when defining' plus the name of an object or kind "
        "of object, or 'before', 'instead of' or 'after' a double-quoted "
        "name of a template layer segment, or of a part of one. For "
        "instance, 'before \"Parser.i6t\".' or 'after \"Pronouns\" in "
        "\"Language.i6t\".'");
}
void compile_inclusions_for_wo(OUTPUT_STREAM, world_object *wo) {
    i6_inclusion_matter *inclm;
    LOOP_OVER (inclm, i6_inclusion_matter)
        if (inclm->wo_to_include_with == wo) {
            interpret_I6T_file(OUT,
                lw_array[inclm->material_to_include->word_ref1 + 2].lw_rawtext, NULL, -1);
            WRITE("\n");
        }
```

```
}
```

The function compile_inclusions_for_wo is called from 9/cot.

§**2.** With that general machinery for I6 inclusions now built, we can move on to use options. (Each one actually used creates a before-the-library I6 inclusion of its own.)

```
sentence_handler USEMEANS_SH_handler =
    { SENTENCE_NT, USEMEANS_VB, 1, new_use_option };
void new_use_option(parse_node *p) {
    int w1, w2, min_setting = -1;
    use_option *uo = CREATE(use_option);
    w1 = p->down->next->word_ref1+1;
    w2 = p->down->next->word_ref2;
    if ([[w1, w2 == ... of at least ###]] &&
        (is_kova(is_a_literal(w2,w2), NUMBER_TY))) {
        min_setting = atoi(lw_array[w2].lw_rawtext);
        w2 = w2 - 4;
    }
    uo->word_ref1 = w1;
    uo->word_ref2 = w2;
    uo->option_expansion = p->down->next->next;
    uo->option_used = FALSE;
    uo->source_file_scoped = FALSE;
    uo->minimum_setting_value = min_setting;
    if [[uo == authorial modesty]] uo->source_file_scoped = TRUE;
    if (lw_array[uo->word_ref1].lw_identity == the_V) uo->word_ref1++;
    register_excerpt_meaning(
        MISCELLANEOUS_MC, USE_OPTION_SMC, uo->word_ref1, uo->word_ref2,
        STORE_POINTER_use_option(uo));
}
sentence_handler USE_SH_handler =
    { SENTENCE_NT, USE_VB, 2, handle_set_use_option };
void handle_set_use_option(parse_node *p) {
    set_use_options(p->down->next->word_ref1, p->down->next->word_ref2);
}
void set_use_options(int w1, int w2) {
    use_option *uo;
    meaning_list *ml;
    int min_setting = -1;
    if (is_list_divided(w1, w2, LOOK_FOR_AND)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        set_use_options(lw1, lw2);
        set_use_options(rw1, rw2);
        return;
    }
    if ([[w1, w2 == ### of ###]] &&
        (vocab_test_flags(w2, NUMBER_MC))) {
        i6_memory_setting *ms;
        int n = vocab_get_literal_number_value(lw_array[w2].lw_identity);
        LOOP_OVER(ms, i6_memory_setting)
            if (compare_raw_word_by_strcmp(w1, ms->ICL_identifier)) {
                if (ms->number < n) ms->number = n;
```

```
                return;
            }
        ms = CREATE(i6_memory_setting);
        if (strlen(lw_array[w1].lw_rawtext) > 63) {
            sentence_problem(_P_(C10BadICLIdentifier),
                "that is too long to be an ICL identifier",
                "so can't be the name of any I6 memory setting.");
        } else strcpy(ms->ICL_identifier, lw_array[w1].lw_rawtext);
        ms->number = n;
        return;
    }
    if ([[w1, w2 == ... of at least ###]] &&
        (vocab_test_flags(w2, NUMBER_MC))) {
        min_setting = vocab_get_literal_number_value(lw_array[w2].lw_identity);
        w2 = w2 - 4;
    }
    if (lw_array[w1].lw_identity == the_V) w1++;
    ml = SP_excerpt(MISCELLANEOUS_MC, w1, w2);
    if ((ml != NULL) && (em_get_secondary_code(ml_meaning(ml)) == USE_OPTION_SMC)) {
        extension_file *ef = NULL;
        uo = RETRIEVE_POINTER_use_option(em_data(ml_meaning(ml)));
        if (uo->minimum_setting_value == -1) {
            if (min_setting != -1)
                sentence_problem(_P_(C10UONotNumerical),
                    "that 'Use' option does not have a numerical setting",
                    "but is either used or not used.");
        } else {
            if (min_setting >= uo->minimum_setting_value)
                uo->minimum_setting_value = min_setting;
        }
        if (uo->source_file_scoped) {
            ef = sf_get_extension_corresponding(lw_array[w1].lw_source.file_of_origin);
            if (ef == NULL) {
                uo->option_used = TRUE;
                if [[w1, w2 == authorial modesty]]
                    set_general_authorial_modesty();
            } else {
                if [[w1, w2 == authorial modesty]]
                    ef_set_authorial_modesty(ef);
            }
        } else uo->option_used = TRUE;
        set_immediate_option_flags(w1, w2, uo);
        return;
    }
    log_word_range(w1, w2);
    sentence_problem(_P_(C10UnknownUseOption),
        "that isn't a 'Use' option known to me",
        "and needs to be one of the ones listed in the documentation.");
}
void set_immediate_option_flags(int w1, int w2, use_option *uo) {
    if (is_list_divided(w1, w2, LOOK_FOR_AND)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        set_immediate_option_flags(lw1, lw2, uo);
```

```
        set_immediate_option_flags(rw1, rw2, uo);
        return;
    }
    if [[w1, w2 == telemetry recordings]] telemetry_recording = TRUE;
    if [[w1, w2 == memory economy]] memory_economy_in_force = TRUE;
    if [[w1, w2 == numbered rules]] br_set_numbered_rules();
    if ((uo) && [[w1, w2 == dynamic memory allocation]])
        dynamic_memory_allocation = uo->minimum_setting_value;
}

int get_dynamic_memory_allocation(void) {
    return dynamic_memory_allocation;
}

void compile_use_options(OUTPUT_STREAM) {
    use_option *uo;
    LOOP_OVER(uo, use_option)
        if ((uo->option_used) || (uo->minimum_setting_value >= 0)) {
            WRITE("! Use option:\n");
            interpret_I6T_file(OUT, lw_array[uo->option_expansion->word_ref1 + 1].lw_rawtext,
                NULL, uo->minimum_setting_value);
            WRITE("\n");
        }
}

void compile_TestUseOption_routine(OUTPUT_STREAM) {
    use_option *uo;
    WRITE("[ TestUseOption UO;\n"); INDENT;
    LOOP_OVER(uo, use_option)
        if (uo->option_used)
            WRITE("if (UO == %d) rtrue;\n", uo->allocation_id);
    WRITE("rfalse;\n");
    OUTDENT; WRITE("];\n");
}

void compile_icl_commands(OUTPUT_STREAM) {
    i6_memory_setting *ms;
    LOOP_OVER(ms, i6_memory_setting)
        WRITE("!%% $%s=%d\n", ms->ICL_identifier, ms->number);
}
```

The function set_immediate_option_flags is called from 2/dl.

The function get_dynamic_memory_allocation is called from 9/rsdt.

The function compile_use_options is invoked by a command in a `.i6t` template file.

The function compile_TestUseOption_routine is invoked by a command in a `.i6t` template file.

The function compile_icl_commands is invoked by a command in a `.i6t` template file.

11 Actions and Activities

`11/chron`: *Chronology.w* To parse representations of periods of time or of historical repetition, and to keep track of the state of things so that it will be possible in future to ask questions concerning the past.

`11/act`: *Actions.w* To define, map to I6 and index individual actions.

`11/anl`: *Action Name Lists.w* Action name lists provide a disjunction in the choice of action made by an action pattern. For instance, "taking or dropping the disc" results in a two-entry ANL. An empty ANL is also legal, and means "doing something" – the generic I7 text for "any action at all".

`11/ap`: *Action Patterns.w* An action pattern is a description which may match many actions or none. The text "doing something" matches every action, while "throwing something at a door in a dark room" is seldom matched. Here we parse such text into a data structure called an `action_pattern`.

`11/los`: *Looping Over Scope.w* To compile routines capable of being passed as arguments to the I6 library routine for looping over parser scope at run-time, and so to provide an implementation for conditions such as "in the presence of Mrs Dalloway".

`11/nap`: *Named Action Patterns.w* A named action pattern is a named categorisation of actions, such as "acting suspiciously", which is a disjunction of action patterns (stored in a linked list).

`11/ina`: *Actions Index.w* To construct the Actions index page.

`11/av`: *Activities.w* To create and manage activities, which are action-like bundles of rules controlling how the I6 runtime code carries out tasks such as "printing the name of something". Each has its own page in the I7 documentation. An activity list is a disjunction of actitivies.

Chronology 11/chron

Purpose

To parse representations of periods of time or of historical repetition, and to keep track of the state of things so that it will be possible in future to ask questions concerning the past.

11/chron.§2 Compiling chronology; §3-6 The periodic monitoring

Template interpreter commands

```
{-callv:allow_no_further_past_tenses}
{-callv:past_actions_i6_routines}
{-callv:past_tenses_i6_escape}
{-callv:chronology_extents_i6_escape}
```

Definitions

¶1. In two different contexts (action patterns and specifications) we will need a represention for periods of time. We measure time in two units: "times", as in something having been true four times in the past; and "turns", as in something having been true for five turns now. We will also need to keep track of four tenses when discussing time periods, and also whether we mean "three or more" (say) or "less than seven" turns or times. We represent this using the relevant I6 comparison operator: for instance a length of 5 and an operator `>=` denotes "five or more". Lastly, we can denote a closed interval of time as well, by setting `until` to the upper end.

```
define TIMES_UNIT 1                    used for "for the third time"
define TURNS_UNIT 2                    used for "for three turns"

define IS_TENSE 0                      Present
define WAS_TENSE 1                     Past
define HASBEEN_TENSE 2                 Present perfect
define HADBEEN_TENSE 3                 Past perfect

typedef struct time_period {
    int valid_tp;                      flag distinguishing validly and invalidly parsed periods
    int units;                         one of the two above
    int length;                        the duration or else the lower limit of an interval
    int until;                         -1 or else the upper limit of an interval
    char *inform6_operator;            <=, ==, >, and so forth
    int tense;                         one of the four above
} time_period;
```

The structure time_period is private to this section.

¶2. Some conditions stored in SPs, and also some actions stored in APs, are evaluated in the past tense, and to do this we need to lodge them into storage: the following simple structures are used for this.

```
typedef struct past_tense_condition_record {
    struct specification *condition;                    condition to be evaluated
    struct parse_node *where_ptc_tested;                sentence in which condition is found
    MEMORY_MANAGEMENT
} past_tense_condition_record;

typedef struct past_tense_action_record {
    struct action_pattern historic_action;              action pattern to be matched
    struct parse_node *where_pta_tested;                sentence in which AP is found
    MEMORY_MANAGEMENT
} past_tense_action_record;
```

The structure past_tense_condition_record is private to this section.

The structure past_tense_action_record is private to this section.

---

§**1.** This simple procedure looks at the text between `w1` and `w2` to look for an indication of a period of time, such as "twice" or "for more than the third time".

```
time_period new_time_period(void) {
    time_period tp;
    tp.valid_tp = 0;
    tp.length = 0;
    tp.until = -1;
    tp.units = TIMES_UNIT;
    tp.inform6_operator = NULL;
    tp.tense = IS_TENSE;
    return tp;
}

time_period *store_tp(time_period tp) {
    time_period *ntp = CREATE(time_period);
    *ntp = tp;
    return ntp;
}

void log_tense_number(int t) {
    switch (t) {
        case IS_TENSE: LOG("IS_TENSE"); break;
        case WAS_TENSE: LOG("WAS_TENSE"); break;
        case HASBEEN_TENSE: LOG("HASBEEN_TENSE"); break;
        case HADBEEN_TENSE: LOG("HADBEEN_TENSE"); break;
        default: LOG("<invalid-tense>"); break;
    }
}

void log_time_period(time_period *tp) {
    if (tp->valid_tp == FALSE) { LOG("---"); return; }
    LOG("<");
    if (tp->inform6_operator) LOG("%s", tp->inform6_operator);
    if (tp->until >= 0) LOG("%d", tp->until); else LOG("%d", tp->length);
    switch(tp->units) {
        case TURNS_UNIT: LOG(" turns"); break;
```

```
        case TIMES_UNIT: LOG(" times"); break;
        default: LOG(": <invalid-units>"); break;
    }
    if (tp->tense != IS_TENSE) {
        LOG(": "); log_tense_number(tp->tense);
    }
    LOG(">");
}

int tp_is_valid(time_period *tp) {
    return tp->valid_tp;
}
void tp_make_invalid(time_period *tp) {
    tp->valid_tp = FALSE;
}

int tp_get_tense(time_period *tp) {
    return tp->tense;
}

void tp_set_tense(time_period *tp, int t) {
    tp->tense = t;
}

int tp_duration_count(time_period *tp) {
    if (tp->until >= 0) {
        return tp->until - tp->length + 1;
    } else {
        if (tp->inform6_operator == NULL) return 1;
        if (strcmp(tp->inform6_operator, "==") == 0) return 1;
        if (strcmp(tp->inform6_operator, "<=") == 0) return tp->length;
        if (strcmp(tp->inform6_operator, "<") == 0) return tp->length - 1;
        if (strcmp(tp->inform6_operator, ">=") == 0) return 1000000 - tp->length;
        if (strcmp(tp->inform6_operator, ">") == 0) return 1000000 - (tp->length - 1);
    }
    return 0;
}

int compare_specificity_of_tp(time_period *tp1, time_period *tp2) {
    int dc1, dc2;

    if ((tp1) && (tp1->valid_tp == FALSE)) tp1 = NULL;
    if ((tp2) && (tp2->valid_tp == FALSE)) tp2 = NULL;

    if ((tp1 == NULL) && (tp2 == NULL)) return 0;

    if ((tp1) && (tp2 == NULL)) return 1;
    if ((tp2) && (tp1 == NULL)) return -1;

    dc1 = tp_duration_count(tp1); dc2 = tp_duration_count(tp2);

    if (dc1 > dc2) return -1;
    if (dc1 < dc2) return 1;

    return 0;
}

time_period parse_time_period(int w1, int w2) {
    int i1 = -1, i2 = -1, j = -1, l = 0, m = -1, x2 = w2;
    time_period tp = new_time_period();
    if [[w1, w2 == *** once]] { i1 = 1; j = 1; l = TIMES_UNIT; }
    if [[w1, w2 == *** twice]] { i1 = 2; j = 1; l = TIMES_UNIT; }
```

```
    if [[w1, w2 == *** thrice]] { i1 = 3; j = 1; l = TIMES_UNIT; }
    if [[w1, w2 == *** time/times]] m = TIMES_UNIT;
    if [[w1, w2 == *** turn/turns]] m = TURNS_UNIT;
    if ((m != -1) && (j == -1) && (w2-1 >= w1) &&
        (is_an_ordinal_or_cardinal(w2-1, w2-1))) {
        j = 2; i1 = last_literal_evaluated; l = m;
        if ((w2-3 >= w1) && (compare_word(w2-2, to_V)) &&
            (is_an_ordinal_or_cardinal(w2-3, w2-3))) {
            i2 = i1; j = 4; i1 = last_literal_evaluated;
        }
    }
    if (j >= 0) {
        char *op = NULL;
        int w3 = w2-j;
        [[w1, w3 == ... the --> w1, w3]];
            if [[w1, w3 == *** only/exactly]] { w3--; op = "=="; }
        else if [[w1, w3 == *** at most]]      { w3-=2; op = "<="; }
        else if [[w1, w3 == *** less than]]    { w3-=2; op = "<"; }
        else if [[w1, w3 == *** at least]]     { w3-=2; op = ">="; }
        else if [[w1, w3 == *** more than]]    { w3-=2; op = ">"; }
        else if [[w1, w3 == *** under]]        { w3--; op = "<"; }
        else if [[w1, w3 == *** over]]         { w3--; op = ">"; }
        [[w1, w3 == ... the --> w1, w3]];
        [[w1, w3 == ... for --> w1, w3]];
        tp.inform6_operator = op;
        tp.length = i1;
        tp.until = i2;
        tp.units = l;
        tp.valid_tp = w3;
    }
    if (tp.valid_tp) LOGIF(TIME_PERIODS, "Parsed time period: <$W> = $t\n",
        tp.valid_tp, x2, &tp);
    return tp;
}
```

The function new_time_period is called from 7/cosp and 11/ap.
The function store_tp is called from 5/ml and 7/cosp.
The function log_tense_number is called from 5/conj.
The function log_time_period is called from 2/dl and 7/cosp.
The function tp_is_valid is called from 5/candp, 11/anl and 11/ap.
The function tp_make_invalid is called from 11/ap.
The function tp_get_tense is called from 5/mlc.
The function tp_set_tense is called from 5/mlc and 7/cosp.
The function compare_specificity_of_tp is called from 11/ap.
The function parse_time_period is called from 5/candp, 11/anl and 11/ap.

§2. **Compiling chronology.** First, the here and now.

```
int no_past_tenses = 0, no_past_actions = 0;
int too_late_for_past_tenses = FALSE;

void ap_compile_forced_to_present(OUTPUT_STREAM, action_pattern ap) {
    ap_convert_to_present_tense(&ap);                                    prevent recursion
    TEMPORARY_STREAM;
    compile_action_pattern_match(TEMP, ap, FALSE);
    STREAM_COPY(OUT, TEMP);
    CLOSE_TEMPORARY_STREAM;
}

void compile_past_action_pattern(OUTPUT_STREAM, time_period duration,
    action_pattern ap) {
    char *op = duration.inform6_operator;
    past_tense_action_record *pta;
    LOGIF(TIME_PERIODS,
        "compile_past_action_pattern on: $A\nat: $t\n", &ap, &duration);
    if (ap_makes_callings(&ap)) {
        sentence_problem(_P_(C11PTAPMakesCallings),
            "a description of an action cannot both refer to past history "
            "and also use '(called ...)'",
            "because that would require Inform in general to remember "
            "too much information about past events.");
    }
    if (too_late_for_past_tenses) internal_error("too late for a PAP");
    if (op == NULL) op = "==";
    WRITE("(PAPR_%d() && ", no_past_actions);
    if (duration.until >= 0) {
        if (duration.units == TIMES_UNIT)
            WRITE("((%d <= (TimesActionHasHappened-->%d)) && "
                "(%d >= (TimesActionHasHappened-->%d)) && "
                "(ActionCurrentlyHappeningFlag->%d))",
                duration.length, no_past_actions,
                duration.until, no_past_actions, no_past_actions);
        else
            WRITE("((%d <= (TurnsActionHasBeenHappening-->%d)) && "
                "(%d >= (TurnsActionHasBeenHappening-->%d))) ",
                duration.length, no_past_actions,
                duration.until, no_past_actions);
    } else {
        if (duration.units == TIMES_UNIT)
            WRITE("(((TimesActionHasHappened-->%d) %s %d) && "
                "(ActionCurrentlyHappeningFlag->%d))",
                no_past_actions, op, duration.length, no_past_actions);
        else
            WRITE("((TurnsActionHasBeenHappening-->%d) %s %d)",
                no_past_actions, op, duration.length);
    }
    WRITE(")");

    pta = CREATE(past_tense_action_record);
    pta->where_pta_tested = current_sentence;
    pta->historic_action = ap;
    no_past_actions++;
```

```
}
void compile_past_tense_condition(OUTPUT_STREAM, time_period duration,
    specification *spec) {
    int turns_flag = 0,
        perfect_flag = ((duration.tense)/2)%2,
        past_flag = (duration.tense)%2;
    char *op = duration.inform6_operator;
    past_tense_condition_record *ptc;
    LOGIF(TIME_PERIODS,
        "compile_past_tense_condition on: $S\nat: $t\n", spec, &duration);
    if (too_late_for_past_tenses) internal_error("too late for a PTC");
    if (duration.units == TURNS_UNIT) turns_flag = 1;
    if ((past_flag == 0) && (perfect_flag == 0) && (op == NULL)) op = "==";
    else if (op == NULL) op = ">=";
    if ((species_is(spec, TEST_PROPOSITION_SPC)) && (spec_get_proposition(spec))) {
        pcalc_prop *unnegated_prop = prop_unnegate(spec_get_proposition(spec));
        if (unnegated_prop) {
            if (spec_test_flag(spec, CONDITION_NEGATED_SPFLAG))
                spec_clear_flag(spec, CONDITION_NEGATED_SPFLAG);
            else
                spec_set_flag(spec, CONDITION_NEGATED_SPFLAG);
            spec_set_proposition(spec, unnegated_prop);
        }
        if (prop_detect_locals(spec_get_proposition(spec)) > 0) {
            sentence_problem(_P_(BelievedImpossible),      the S-parser catches these earlier now
                "conditions written in the past tense or referring to the past "
                "with clauses like 'for the second time' cannot refer to "
                "temporary values",
                "because these have no past. For instance, the name given in a "
                "'repeat...' or a 'let' can't be talked about as having "
                "existed before.");
            return;
        }
    }
    if (spec_test_flag(spec, CONDITION_NEGATED_SPFLAG)) WRITE("(~~");
    WRITE("(TestSinglePastState(%d, %d, false, %d)",
        past_flag, no_past_tenses, 1 + turns_flag + 4*perfect_flag);
    if (duration.length != 0) WRITE(" %s %d ", op, duration.length);
    WRITE(")");
    if (spec_test_flag(spec, CONDITION_NEGATED_SPFLAG)) WRITE(")");
    if (no_past_tenses >= 1024) {                  limit imposed by the Z-machine implementation
        limit_problem(_P_(Untestable),                                  well, not conveniently
            "conditions written in the past tense", 1024);
        return;
    }
    ptc = CREATE(past_tense_condition_record);
    ptc->where_ptc_tested = current_sentence;
    ptc->condition = spec_copy(spec);
    spec_clear_flag(ptc->condition, CONDITION_NEGATED_SPFLAG);
    spec_set_condition_tense(ptc->condition, NULL);
    no_past_tenses++;
}
```

The function compile_past_action_pattern is called from 11/ap.
The function compile_past_tense_condition is called from 7/cosp.

§**3. The periodic monitoring.** As can be seen from the above routines, we depend on the array `TrackedActions`, on a routine called `TestSinglePastState` and on magical things happening at run-time at the start of every action. Once these are defined, it's too late to create any further past tense references, so:

```
void allow_no_further_past_tenses(void) {
    too_late_for_past_tenses = TRUE;
}
```

The function allow_no_further_past_tenses is invoked by a command in a `.i6t` template file.

§**4.** The `{-A}` ("past actions") escape. A series of `if` statements checking whether each past-tense action is now true, and updating the relevant array entries accordingly.

This looks straightforward, but a tricky point arises from the fact that a turn is not the same thing as a slot for an action. Usually they correspond, but not always. So if the action for a turn is taking, and that causes a further action of opening a box, say, what do we say about the number of turns for which the taking and opening actions have been happening?

In `adjust` mode, we are at the start or end of a "try" action which arises mid-turn. As can be seen, adjust mode is allowed only to change the record of how many turns an action has been happening – from 0 to 1, and otherwise making no change, if the action is the one being tried; or else back to 0 if it isn't. In particular, an action implicitly tried does not affect the "times" count.

This is a delicate business. The better way to solve this would be to stack a local copy of these arrays each time an action starts, and restore it (with updates) each time it finishes. But there is just no storage available in the Z-machine to contemplate this with equanimity, so we compromise. Historically, it's been a fertile source of tricky bugs. The test case `ActionInterrupted` should be checked if this code is ever tampered with.

```
void past_actions_i6_routines(OUTPUT_STREAM) {
    int once_only = TRUE;
    past_tense_action_record *pta;
    LOOP_OVER(pta, past_tense_action_record) {
        phsf_create_nonphrase_stack_frame();
        int pt = pta->allocation_id;
        current_sentence = pta->where_pta_tested;          ensure problems reported correctly
        WRITE("[ PAPR_%d;\n", pt); INDENT;
        WRITE("if (");
        ap_compile_forced_to_present(OUT, pta->historic_action);
        WRITE(") rtrue;\n");
        WRITE("rfalse;\n");
        OUTDENT; WRITE("];\n");
        if ((do_we_need_ct_locals()) && (once_only)) {
            once_only = FALSE;
            sentence_problem(_P_(C11PastTableLookup),
                "it's not safe to look up table entries in a way referring "
                "to past history",
                "because it leads to dangerous ambiguities. For instance, "
                "does 'taking an item listed in the Table of Treasure "
                "for the first time' mean that this is the first time taking "
                "any of the things in the table, or only the first time "
                "this one? And so on.");
```

```
        }
        phsf_remove_nonphrase_stack_frame();
    }
    WRITE("Array PastActionsI6Routines --> ");
    LOOP_OVER(pta, past_tense_action_record) {
        WRITE("PAPR_%d ", pta->allocation_id);
    }
    WRITE("0 0;\n");
}
```

The function past_actions_i6_routines is invoked by a command in a `.i6t` template file.

§**5.** The `{-C}` ("chronology") escape. The body of a switch statement in which the ith case results in the I6 variable `new` being set equal to the I6 condition stored up as past tense condition number i.

```
void past_tenses_i6_escape(OUTPUT_STREAM) {
    int once_only = TRUE;
    past_tense_condition_record *ptc;
    LOGIF(TIME_PERIODS,
        "Creating %d past tense conditions in TestSinglePastState\n",
            NUMBER_CREATED(past_tense_condition_record));
    phsf_create_nonphrase_stack_frame();
    phsf_determines_the_past();
    LOOP_OVER(ptc, past_tense_condition_record) {
        specification *spec = ptc->condition;
        LOGIF(TIME_PERIODS, "Number %d: proposition $D\n",
            ptc->allocation_id, spec_get_proposition(spec));
        WRITE("%d: new = ", ptc->allocation_id);
        current_sentence = ptc->where_ptc_tested;          ensure problems reported correctly
        if (prop_contains_callings(spec_get_proposition(spec))) {
            sentence_problem(_P_(C11PastCallings),
                "it's not safe to use '(called ...)' in a way referring "
                "to past history",
                "because this would make a temporary value to hold the "
                "quantity in question, but at a different time from when "
                "it would be needed.");
        }
        TEMPORARY_STREAM;
        spec_compile(TEMP, spec);
        STREAM_COPY(OUT, TEMP);
        CLOSE_TEMPORARY_STREAM;
        WRITE(";\n");
        if ((do_we_need_ct_locals()) && (once_only)) {
            once_only = FALSE;
            sentence_problem(_P_(C11PastTableEntries),
                "it's not safe to look up table entries in a way referring "
                "to past history",
                "because it leads to dangerous ambiguities. For instance, "
                "does 'taking an item listed in the Table of Treasure "
                "for the first time' mean that this is the first time taking "
                "any of the things in the table, or only the first time "
                "this one? And so on.");
        }
```

```
    }
    phsf_remove_nonphrase_stack_frame();
    LOGIF(TIME_PERIODS,
        "Creation of past tense conditions complete\n");
}
```

The function past_tenses_i6_escape is invoked by a command in a `.i6t` template file.

§**6.** The `{-E}` ("extents") escape. Sizes for the arrays. We need to do these after compiling the code in the escapes above, because they might continue to spawn each other as they compile: e.g. a past tense action pattern might itself refer to a condition in the still more remote past, as in the case of "After switching on the lunar clock when the lunar clock has been switched on for more than five times".

```
void chronology_extents_i6_escape(OUTPUT_STREAM) {
    WRITE("Constant NO_PAST_TENSE_CONDS %d;\n", no_past_tenses);
    WRITE("Constant NO_PAST_TENSE_ACTIONS %d;\n", no_past_actions);
}
```

The function chronology_extents_i6_escape is invoked by a command in a `.i6t` template file.

Actions 11/act

Purpose

To define, map to I6 and index individual actions.

Template interpreter commands

```
2   {-callv:Inform_name_the_actions}
6   {-array:ActionHappened}
10   {-callv:compile_action_routines}
11   {-array:ActionData}
11   {-array:ActionCoding}
```

Definitions

¶1. An action is an impulse to do something within the model world, and which may not be possible. Much of the work of designing an interactive fiction consists in responding to the actions of the player, sometimes in ways which the player expects, sometimes not. Design systems for interactive fiction therefore need to provide flexible and convenient ways to discuss actions.

An activity is by contrast something done by the run-time code during play: for instance, printing the name of an object, or asking a disambiguation question. These tasks must similarly be customisable by the designer, and a system of rulebooks is used which parallels the treatment of actions.

¶2. Some fields of this structure reflect the history of NI, and of I6 for that matter: in particular, very few actions if any now use an I6-library defined verb routine, but the ability is kept against future need; and the idea of a flexible number of parameters – which I6 allowed, thus parsing "listen" and "listen to the frog" as the same action, with 0 and 1 parameters respectively – has been dropped in I7. (We use the activities for selecting missing parameters instead.) So for now the minimum and maximum below are always equal.

```
typedef struct action_name {
    int word_ref1, word_ref2;                          such as "drop" or "take"
    int past_word_ref1, past_word_ref2;                such as "dropped" or "taken"
    int it_optional;                                   noun optional when describing the second noun?
    char an_I6_identifier[32];                         compile to this action name in I6
    int out_of_world;                                  action is declared as out of world?
    int censored;                                      from the index, due to bad language?
    int use_verb_routine_in_I6_library;                rather than compiling our own?
    struct rulebook *check_rules;                      rulebooks private to this action
    struct rulebook *carry_out_rules;
    struct rulebook *report_rules;
    struct stacked_variable_owner *owned_by_an;        action variables owned here
    struct parse_node *designers_specification;        where created
    int requires_light;                                does this action require light to be carried out?
    int min_parameters, max_parameters;                in the range 0 to 2
    int noun_access;                                   one of the possibilities below
```

```
    int second_access;
    struct kind_of_value *noun_kov;                    if there is at least 1 parameter
    struct kind_of_value *second_kov;                  if there are 2 parameters
    struct grammar_line *list_with_action;             list of grammar producing this
    int an_specification_text_word;                    description used in index
    int an_index_group;                                paragraph number it belongs to (1, 2, 3, ...)
    MEMORY_MANAGEMENT
} action_name;
stacked_variable_owner_list *all_nonempty_stacked_action_vars = NULL;
```

The structure action_name is shared with 11/ina.

¶3. The access possibilities for the noun and second noun are as follows.

```
define UNRESTRICTED_ACCESS 0                       question not meaningful, e.g. for a number
define IMPOSSIBLE_ACCESS 1                         action doesn't take a noun, so no question of access
define DOESNT_REQUIRE_ACCESS 2                     actor need not be able to touch this object
define REQUIRES_ACCESS 3                           actor must be able to touch this object
define REQUIRES_POSSESSION 4                       actor must be carrying this object
```

---

§**1.** The constructor function for action names divides them into two according to their implementation. Some actions are fully implemented in I6, and do not have the standard check/carry out/report rulebooks: others are full I7 actions. (As I7 has matured, more and more actions have moved from the first category to the second.)

```
action_name *act_new(int w1, int w2, int implemented_by_I7) {
    action_name *an;
    an = CREATE(action_name);
    an->word_ref1 = w1; an->word_ref2 = w2;
    make_past_of_participle(w1, w2);
    an->past_word_ref1 = plw1; an->past_word_ref2 = plw2;
    an->it_optional = TRUE;
    an->an_I6_identifier[0] = 0;
    an->use_verb_routine_in_I6_library = TRUE;
    an->check_rules = NULL;
    an->carry_out_rules = NULL;
    an->report_rules = NULL;
    an->requires_light = FALSE;
    an->min_parameters = 0;
    an->max_parameters = 2;
    an->noun_access = UNRESTRICTED_ACCESS;
    an->second_access = UNRESTRICTED_ACCESS;
    an->noun_kov = kova(OBJECT_TY);
    an->second_kov = kova(OBJECT_TY);
    an->designers_specification = NULL;
    an->list_with_action = NULL;
    an->censored = FALSE;
    an->out_of_world = FALSE;
    an->an_specification_text_word = -1;
    register_reworded_meaning(MISCELLANEOUS_MC, ACTION_NAME_SMC, 0, 0, w1, w2, action_V,
        STORE_POINTER_action_name(an));
```

```
    LOGIF(ACTION_CREATIONS, "Created action: $W\n", w1, w2);
    if (implemented_by_I7) {
        int wn;
        an->use_verb_routine_in_I6_library = FALSE;
        wn = lexer_wordcount;
        feed_into_lexer("check", TRUE, FALSE);
        splice_words(an->word_ref1, an->word_ref2);
        an->check_rules =
            rb_new_automatic(wn, lexer_wordcount-1, ACTION_FOCUS,
                NO_OUTCOME, TRUE, FALSE, FALSE);
        rb_fragment_by_actions(an->check_rules, 1);
        wn = lexer_wordcount;
        feed_into_lexer("carry out", TRUE, FALSE);
        splice_words(an->word_ref1, an->word_ref2);
        an->carry_out_rules =
            rb_new_automatic(wn, lexer_wordcount-1, ACTION_FOCUS,
                NO_OUTCOME, TRUE, FALSE, FALSE);
        rb_fragment_by_actions(an->carry_out_rules, 2);
        wn = lexer_wordcount;
        feed_into_lexer("report", TRUE, FALSE);
        splice_words(an->word_ref1, an->word_ref2);
        an->report_rules =
            rb_new_automatic(wn, lexer_wordcount-1, ACTION_FOCUS,
                NO_OUTCOME, TRUE, FALSE, FALSE);
        rb_fragment_by_actions(an->report_rules, 1);
        an->owned_by_an = stvo_new(20000+an->allocation_id);
    } else {
        an->owned_by_an = NULL;
    }
    action_name *an2;
    LOOP_OVER(an2, action_name)
        if (an != an2)
            if (action_names_overlap(an, an2)) {
                an->it_optional = FALSE;
                an2->it_optional = FALSE;
            }
    return an;
}
```

pointing at, pointing it at

```
int action_names_overlap(action_name *an1, action_name *an2) {
    int w1 = an1->word_ref1, w2 = an1->word_ref2, x1 = an2->word_ref1, x2 = an2->word_ref2;
    int i, j;
    for (i = w1, j = x1; (i <= w2) && (j <= x2); i++, j++) {
        if ([[word i == it]] && (compare_words(i+1, j))) return TRUE;
        if ([[word j == it]] && (compare_words(j+1, i))) return TRUE;
        if (compare_words(i, j) == FALSE) return FALSE;
    }
    return FALSE;
}
void log_action_name(action_name *an) {
    if (an == NULL) LOG("<null-action-name>");
    else LOG("$W", an->word_ref1, an->word_ref2);
```

```
}
action_name *act_by_name(int w1, int w2) {
    action_name *an;
    LOOP_OVER(an, action_name)
        if (compare_word_range(w1, w2, an->word_ref1, an->word_ref2))
            return an;
    LOOP_OVER(an, action_name)
        if ([[an == ... to]]
            && (compare_word_range(w1, w2, an->word_ref1, an->word_ref2-1)))
            return an;
    return NULL;
}
action_name *act_longest_null(int w1, int w2, int tense, int *excess) {
    action_name *an;
    LOOP_OVER(an, action_name)
        if (an->max_parameters == 0) {
            int aw1, aw2;
            if (tense == IS_TENSE) { aw1 = an->word_ref1; aw2 = an->word_ref2; }
            else { aw1 = an->past_word_ref1; aw2 = an->past_word_ref1; }
            if ((w2 - w1 >= aw2 - aw1) &&
                (compare_word_range(w1, w1 + aw2 - aw1, aw1, aw2))) {
                *excess = w1 + aw2 - aw1 + 1;
                return an;
            }
        }
    return NULL;
}
int act_it_optional(action_name *an) {
    return an->it_optional;
}
char *an_get_I6_representation(action_name *an) {
    return an->an_I6_identifier;
}
rulebook *an_get_fragmented_rulebook(action_name *an, rulebook *rb) {
    if (rb == built_in_rulebooks[CHECK_RB]) return an->check_rules;
    if (rb == built_in_rulebooks[CARRY_OUT_RB]) return an->carry_out_rules;
    if (rb == built_in_rulebooks[REPORT_RB]) return an->report_rules;
    internal_error("asked for peculiar fragmented rulebook"); return NULL;
}
rulebook *an_switch_fragmented_rulebook(action_name *new_an, rulebook *orig) {
    action_name *an;
    if (new_an == NULL) return orig;
    LOOP_OVER(an, action_name) {
        if (orig == an->check_rules) return new_an->check_rules;
        if (orig == an->carry_out_rules) return new_an->carry_out_rules;
        if (orig == an->report_rules) return new_an->report_rules;
    }
    return orig;
}
void an_set_specification_text(action_name *an, int wn) {
    an->an_specification_text_word = wn;
}
```

```
int an_get_specification_text(action_name *an) {
    return an->an_specification_text_word;
}
```

The function log_action_name is called from 2/dl.
The function act_by_name is called from 13/tfg.
The function act_longest_null is called from 11/ap and 12/phud.
The function act_it_optional is called from 11/anl.
The function an_get_I6_representation is called from 7/vasp, 7/cmsp, 11/anl, 11/ap, 12/br and 13/gl.
The function an_get_fragmented_rulebook is called from 12/phud.
The function an_switch_fragmented_rulebook is called from 12/phud.
The function an_set_specification_text is called from 8/mass.

§**2.** Most actions are given automatically generated Inform 6 names in the compiled code: `Q4_green`, for instance. A few must however correspond to names of significance in the I6 library.

```
void Inform_name_the_actions(void) {
    action_name *an;
    LOOP_OVER(an, action_name)
        if (an->an_I6_identifier[0] == 0)
            isn_compose_identifier(an->an_I6_identifier,
                'A', an->allocation_id, an->word_ref1, an->word_ref2);
}
void an_translates(parse_node *pn) {
    parse_node *p1 = pn->down->next;
    parse_node *p2 = pn->down->next->next;
    action_name *an = act_by_name(p1->word_ref1, p1->word_ref2);
    if (an == NULL) {
        LOG("Tried action name $W\n", p1->word_ref1, p1->word_ref2);
        sentence_problem(_P_(C11TranslatesNonAction),
            "this does not appear to be the name of an action",
            "so cannot be translated into I6 at all.");
        return;
    }
    if (an->an_I6_identifier[0]) {
        LOG("Tried action name $W = %s\n", p1->word_ref1, p1->word_ref2, an->an_I6_identifier);
        sentence_problem(_P_(C11TranslatesActionAlready),
            "this action has already been translated",
            "so there must be some duplication somewhere.");
        return;
    }
    strcpy(an->an_I6_identifier, lw_array[p2->word_ref1].lw_text);
    LOGIF(ACTION_CREATIONS, "Translated action: $l as %s\n", an, lw_array[p2->word_ref1].lw_text);
}
int an_get_stem_length(action_name *an) {
    int w1 = an->word_ref1, w2 = an->word_ref2, k = w1, s = 0;
    if (w1<0) return 0;                                    should never happen
    while (k<=w2) {
        if [[word k == it]] return k-w1;
        if (![[word k == it]]) s++;
        k++;
    }
    return w2-w1+1;
```

```
    return s;
}
```

The function Inform_name_the_actions is invoked by a command in a .i6t template file.
The function an_translates is called from 2/isn.
The function an_get_stem_length is called from 11/anl.

§3. Action variables.

```
void an_add_variable(action_name *an, parse_node *cnode) {
    specification *spec;
    int nw1 = -1, nw2 = -1, tw1 = -1, tw2 = -1, mw1 = -1, mw2 = -1, i;
    kind_of_value *kov;
    stacked_variable *stv = NULL;
    if (pn_get_node_type(cnode) != PROPERTYCALLED_NT)
        internal_error("ac_add_variable on a node of unknown type");
    if (an->owned_by_an == NULL) {
        quote_source(1, current_sentence);
        quote_words(2, nw1, nw2);
        handmade_problem(_P_(Untestable));                    since we no longer define such actions
        issue_problem_segment(
            "You wrote %1, which I am reading as a request to make "
            "a new named variable for an action - a value associated "
            "with a action and which has a name. But this is a low-level "
            "action implemented internally by the Inform 6 library, and "
            "I am unable to give it any variables. Sorry.");
        issue_problem_end();
        return;
    }
    [[tw1, tw2 <-- cnode->down]];
    [[nw1, nw2 <-- cnode->down->next]];
    if [[nw1, nw2 == ... OPENBRACKET ... CLOSEBRACKET : i --> nw1, nw2 ... mw1, mw2]] {
        int x1, x2;
        if ((!([[mw1, mw2 == matched as ... --> mw1, mw2]])) ||
            (mw1 != mw2) || (is_kova(is_a_literal(mw1, mw2), TEXT_TY) == FALSE)) {
            quote_source(1, current_sentence);
            quote_words(2, mw1, mw2);
            handmade_problem(_P_(C11BadMatchingSyntax));
            issue_problem_segment(
                "You wrote %1, which I am reading as a request to make "
                "a new named variable for an action - a value associated "
                "with a action and which has a name. The request seems to "
                "say in parentheses that the name in question is '%2', but "
                "I only recognise the form '(matched as \"some text\")' here.");
            issue_problem_end();
            return;
        }
        dequote_word(mw1);
        x1 = lexer_wordcount;
        feed_into_lexer(lw_array[mw1].lw_text, FALSE, FALSE);
        x2 = lexer_wordcount-1;
        mw1 = x1; mw2 = x2;
        if (mw2 > mw1) {
```

```
            quote_source(1, current_sentence);
            quote_words(2, mw1, mw2);
            handmade_problem(_P_(C11MatchedAsTooLong));
            issue_problem_segment(
                "You wrote %1, which I am reading as a request to make "
                "a new named variable for an action - a value associated "
                "with a action and which has a name. You say that it should "
                "be '(matched as \"%2\")', but I can only recognise such "
                "matches when a single keyword is used to introduce the "
                "clause, and this is more than one word.");
            issue_problem_end();
            return;
        }
    }
    if [[tw1, tw2 == object]] spec = new_generic_CONSTANT_type(kova(OBJECT_TY));
    else spec = parse_expression(tw1, tw2, TYPE_EXPCON);
    if [[nw1, nw2 == ... and ... : i]] {
        quote_source(1, current_sentence);
        quote_words(2, nw1, nw2);
        handmade_problem(_P_(C11ActionVarAnd));
        issue_problem_segment(
            "You wrote %1, which I am reading as a request to make "
            "a new named variable for an action - a value associated "
            "with a action and which has a name. The request seems to "
            "say that the name in question is '%2', but I'd prefer to "
            "avoid 'and' in such names, please.");
        issue_problem_end();
        return;
    }
    if (species_is(spec, DESCRIPTION_SPC)) {
        if ((spec_get_described_kind(spec)) && (number_of_adjectives_applied_to(spec) == 0))
{
            spec = new_generic_CONSTANT_type(kovko(spec_get_described_kind(spec)));
        } else {
            quote_source(1, current_sentence);
            quote_words(2, tw1, tw2);
            handmade_problem(_P_(C11ActionVarOverspecific));
            issue_problem_segment(
                "You wrote %1, which I am reading as a request to make "
                "a new named variable for an action - a value associated "
                "with a action and which has a name. The request seems to "
                "say that the value in question is '%2', but this is too "
                "specific a description. (Instead, a kind of value "
                "(such as 'number') or a kind of object (such as 'room' "
                "or 'thing') should be given. To get a property whose "
                "contents can be any kind of object, use 'object'.)");
            issue_problem_end();
            return;
        }
    }
    if (spec_is_generic_CONSTANT(spec) == FALSE) {
        LOG("Offending SP: $X", spec);
        quote_source(1, current_sentence);
```

```
        quote_words(2, tw1, tw2);
        handmade_problem(_P_(C11ActionVarUnknownKOV));
        issue_problem_segment(
            "You wrote %1, but '%2' is not the name of a kind of "
            "value which I know (such as 'number' or 'text').");
        issue_problem_end();
        return;
    }
    if (is_kova(spec_get_kind_of_value(spec), ANY_VALUE_TY)) {
        quote_source(1, current_sentence);
        quote_words(2, tw1, tw2);
        handmade_problem(_P_(C11ActionVarValue));
        issue_problem_segment(
            "You wrote %1, but saying that a variable is a 'value' "
            "does not give me a clear enough idea what it will hold. "
            "You need to say what kind of value: for instance, 'A door "
            "has a number called street address.' is allowed because "
            "'number' is specific about the kind of value.");
        issue_problem_end();
        return;
    }
    if (stvo_empty(an->owned_by_an)) {
        all_nonempty_stacked_action_vars =
            stvol_add_owner(all_nonempty_stacked_action_vars, an->owned_by_an);
    }
    kov = spec_get_kind_of_value(spec);
    stv = stvo_add(an->owned_by_an, nw1, nw2, kov, spec_get_described_kind(spec));
    LOGIF(ACTION_CREATIONS, "Created action variable for $1: $W ($u/$O)\n",
        an, nw1, nw2, kov, spec_get_described_kind(spec));
    if (mw1 >= 0) {
        stv_set_matching_text(stv, mw1, mw2);
        LOGIF(ACTION_CREATIONS, "Match with text: $W + SP\n", mw1, mw2);
    }
}
stacked_variable *an_parse_match_clause(action_name *an, int w1, int w2) {
    return stvo_parse_match_clause(an->owned_by_an, w1, w2);
}
void compile_action_name_var_creators(OUTPUT_STREAM) {
    action_name *an;
    LOOP_OVER(an, action_name) {
        if ((an->owned_by_an) &&
                (stvo_empty(an->owned_by_an) == FALSE))
            stvl_compile_frame_creator(OUT, an->owned_by_an,
                "ANSTVC_%d", an->allocation_id);
    }
}
```

The function an_add_variable is called from 8/mass.

The function an_parse_match_clause is called from 11/ap.

§**4.** I7-implemented actions are created explicitly by I7 sentences.

```
sentence_handler NEW_ACTION_SH_handler =
    { SENTENCE_NT, NEW_ACTION_VB, 1, act_parse_definition };

void act_parse_definition(parse_node *p) {
    int said_what_acts_on = FALSE;
    int aw1 = p->down->next->word_ref1, aw2 = p->down->next->word_ref2;
    action_name *an = act_by_name(aw1, aw2);
    if (an) {
        sentence_problem(_P_(C11ActionAlreadyExists),
            "that seems to be an action already existing",
            "so it cannot be redefined now. If you would like to reconfigure "
            "an action in the standard set - for instance if you prefer "
            "'unlocking' to apply to only one thing, not two - create a new "
            "action for what you need ('keyless unlocking', perhaps) and then "
            "change the grammar to use the new action rather than the old "
            "('Understand \"unlock [something]\" as keyless unlocking.').");
        return;
    }
    an = act_new(aw1, aw2, TRUE);
    if (p->down->next->next) {
        int x1 = p->down->next->next->word_ref1;
        int x2 = p->down->next->next->word_ref2;
        an->designers_specification = p->down->next->next;
        [[x1, x2 == COMMA ... --> x1, x2]];
        [[x1, x2 == and ... --> x1, x2]];
        if [[x1, x2 == out of world ***]]  { an->out_of_world = TRUE; x1+=3; }
        [[x1, x2 == COMMA ... --> x1, x2]];
        [[x1, x2 == and ... --> x1, x2]];
        if [[x1, x2 == censored ***]] { an->censored = TRUE; x1++; }
        [[x1, x2 == COMMA ... --> x1, x2]];
        [[x1, x2 == and ... --> x1, x2]];
        [[x1, x2 == with ... --> x1, x2]];
        if [[x1, x2 == past participle ...]] {
            x1+=2;
            set_past_participle(
                &(an->past_word_ref1), &(an->past_word_ref2), x1++);
            if ((x1<=x2) && ([[x1, x1 == and/COMMA/applying/requiring]] == FALSE)) {
                sentence_problem(_P_(C11MultiwordPastParticiple),
                    "a past participle must be given as a single word",
                    "even if the action name itself is longer than that. "
                    "(For instance, the action name 'hanging around until' "
                    "should have past participle given just as 'hung'; I "
                    "can already deduce the rest.)");
                return;
            }
        }
        [[x1, x2 == COMMA ... --> x1, x2]];
        [[x1, x2 == and ... --> x1, x2]];
        if [[x1, x2 == applying to ...]] {
            said_what_acts_on = TRUE;
            x1 = act_requirement_clause(an, x1+2, x2);
```

```
        }
        [[x1, x2 == COMMA ... --> x1, x2]];
        [[x1, x2 == and ... --> x1, x2]];
        if [[x1, x2 == requiring light ***]] {
            an->requires_light = TRUE;
            x1+=2;
        }
        if (x1 <= x2) {
            sentence_problem(_P_(C11ActionClauseUnknown),
                "the action definition contained text I couldn't follow",
                "and may be too complicated.");
            return;
        }
    }
    if (said_what_acts_on == FALSE) {
        an->noun_access = IMPOSSIBLE_ACCESS;
        an->second_access = IMPOSSIBLE_ACCESS;
        an->min_parameters = 0;
        an->max_parameters = 0;
    }
    if (an->max_parameters >= 2) {
        if ((is_kova(an->noun_kov, OBJECT_TY) == FALSE) &&
            (is_kova(an->second_kov, OBJECT_TY) == FALSE)) {
            sentence_problem(_P_(C11ActionBothValues),
                "this action definition asks to have a single action apply "
                "to two different things which are not objects",
                "and unfortunately a fundamental restriction is that an "
                "action can apply to two objects, or one object and one "
                "value, but not to two values. Sorry about that.");
            return;
        }
    }
}
```

§5. **Action censorship.** This is for bowdlerisation of the index: it avoids putting Rude Words into the index, where anybody might read them.

```
int act_is_censored(action_name *an) {
    if (an->censored) return TRUE;
    return FALSE;
}
int act_is_out_of_world(action_name *an) {
    if (an->out_of_world) return TRUE;
    return FALSE;
}
kind_of_value *act_get_data_type_of_noun(action_name *an) {
    return an->noun_kov;
}
kind_of_value *act_get_data_type_of_second_noun(action_name *an) {
    return an->second_kov;
}
void act_set_text_to_name_tensed(action_name *an, int *x1, int *x2, int tense) {
```

```
    if (tense == HASBEEN_TENSE) {
        *x1 = an->past_word_ref1;
        *x2 = an->past_word_ref2;
    } else {
        *x1 = an->word_ref1;
        *x2 = an->word_ref2;
    }
}

int act_can_have_parameters(action_name *an) {
    if (an->max_parameters > 0) return TRUE;
    return FALSE;
}

int act_get_max_parameters(action_name *an) {
    return an->max_parameters;
}
```

The function act_is_censored is called from 13/gl.
The function act_is_out_of_world is called from 13/gl.
The function act_get_data_type_of_noun is called from 7/cmsp and 11/ap.
The function act_get_data_type_of_second_noun is called from 7/cmsp and 11/ap.
The function act_set_text_to_name_tensed is called from 11/anl.
The function act_can_have_parameters is called from 11/anl.
The function act_get_max_parameters is called from 11/ap.

§6. **Past tense.** Simpler actions – those with no parameter, or a single parameter which is a thing – can be tested in the past tense. The run-time support for this is a general bitmap revealing which actions have ever happened, plus a bitmap for each object revealing which have ever been applied to the object in question. This is where we compile the bitmaps in their fresh, empty form.

```
int act_can_be_compiled_in_past_tense(action_name *an) {
    if (an->min_parameters > 1) return FALSE;
    if (an->max_parameters > 1) return FALSE;
    if ((an->max_parameters == 1) &&
        (is_kova(an->noun_kov, OBJECT_TY) == FALSE))
            return FALSE;
    return TRUE;
}

void act_compile_action_bitmap_property(OUTPUT_STREAM) {
    int i;
    WRITE(" with action_bitmap");
    for (i=0; i<=((NUMBER_CREATED(action_name))/16); i++) WRITE(" 0");
    WRITE("\n");
}

void compile_ActionHappened_array(OUTPUT_STREAM) {
    int i = 0;
    WRITE("Array ActionHappened -->");
    for (i=0; i<=((NUMBER_CREATED(action_name))/16); i++) {
        WRITE(" 0");
    }
    WRITE(";\n\n");
}
```

The function act_can_be_compiled_in_past_tense is called from 11/ap.
The function act_compile_action_bitmap_property is called from 9/cot.
The function compile_ActionHappened_array is invoked by a command in a `.i6t` template file.

§**7.** Clauses in action definitions to do with what arguments an action requires.

```
int act_requirement_clause(action_name *an, int x1, int x2) {
    int objc = 0, duplicate = FALSE;
    an->noun_access = REQUIRES_ACCESS;
    an->second_access = REQUIRES_ACCESS;
    an->noun_kov = kova(OBJECT_TY);
    an->second_kov = kova(OBJECT_TY);
    if [[x1, x2 == nothing ***]] { x1++; goto AppSpecDone; }
    objc = 1;
    if [[x1, x2 == one ... --> x1, x2]] ;
    else if [[x1, x2 == two ... --> x1, x2]] duplicate = TRUE;
    x1 = act_individual_requirement_clause(an, x1, x2, objc);
    if (x1 >= x2) goto AppSpecDone;
    if [[x1, x2 == and one ... --> x1, x2]] objc = 2;
    else goto AppSpecDone;
    x1 = act_individual_requirement_clause(an, x1, x2, objc);
    AppSpecDone:
    if (objc == 0) an->noun_access = IMPOSSIBLE_ACCESS;
    if (objc < 2) an->second_access = IMPOSSIBLE_ACCESS;
    if (duplicate) {
        an->second_access = an->noun_access;
        an->second_kov = an->noun_kov;
    }
    an->min_parameters = 0;
    if (an->noun_access != IMPOSSIBLE_ACCESS)
        an->min_parameters = 1;
    if (an->second_access != IMPOSSIBLE_ACCESS)
        an->min_parameters = 2;
    an->max_parameters = an->min_parameters;
    return x1;
}
int act_individual_requirement_clause(action_name *an, int x1, int x2, int objc) {
    int y2, access = REQUIRES_ACCESS;
    kind_of_value *kov = kova(OBJECT_TY);
    specification *spec;
    if [[x1, x2 == visible ***]] {
        x1++; access = DOESNT_REQUIRE_ACCESS;
    } else {
        if [[x1, x2 == touchable ***]] {
            x1++; access = REQUIRES_ACCESS;
        } else {
            if [[x1, x2 == carried ***]] {
                x1++; access = REQUIRES_POSSESSION;
            }
        }
    }
```

```
    if [[x1, x2 == thing/things ***]] {
        x1++;
    } else {
        y2 = x1+1;
        while ((y2<x2) && ([[y2, x2 == and/COMMA ...]] == FALSE)) y2++;
        if (y2 != x2) y2--;
        spec = parse_expression(x1, y2, TYPE_EXPCON);
        if (spec_is_generic_CONSTANT(spec)) {
            kov = spec_get_kind_of_value(spec);
            access = UNRESTRICTED_ACCESS;
        } else {
            LOG("Error at $W: offending SP: $X", x1, y2, spec);
            sentence_problem(_P_(C11ActionMisapplied),
                "an action can only apply to things or to kinds of value",
                "for instance: 'photographing is an action applying to "
                "one visible thing'.");
        }
        x1 = y2+1;
    }
    if (objc == 1) {
        an->noun_access = access;
        an->noun_kov = kov;
    } else {
        an->second_access = access;
        an->second_kov = kov;
    }
    return x1;
}
```

§8. The grammar list.

```
void act_add_gl(action_name *an, grammar_line *gl) {
    if (an->list_with_action == NULL) an->list_with_action = gl;
    else gl_list_with_action_add(an->list_with_action, gl);
}
```

The function act_add_gl is called from 13/gl.

§9. Typechecking grammar for an action.

```
void act_check_types_for_grammar(action_name *an, int tok_values,
    kind_of_value **tok_value_kovs) {
    int required = 0; char *failed_on = "<internal error>";
    if (an->noun_access != IMPOSSIBLE_ACCESS)
        required++;
    if (an->second_access != IMPOSSIBLE_ACCESS)
        required++;
    if (required < tok_values) {
        switch(required) {
            case 0:
                failed_on =
                    "this action applies to nothing, but you have provided "
                    "material in square brackets which expands to something";
                break;
            case 1:
                failed_on =
                    "this action applies to just one thing, but you have "
                    "put more than one thing in square brackets";
                break;
            default:
                failed_on =
                    "this action applies to two things, the maximum possible, "
                    "but you have put more than two in square brackets";
                break;
        }
        goto Unmatched;
    }
    if (tok_values >= 1) {
        switch(an->noun_access) {
            case UNRESTRICTED_ACCESS: {
                kind_of_value *supplied_data_type = tok_value_kovs[0];
                kind_of_value *desired_data_type = an->noun_kov;
                if (can_we_cast_kovs(supplied_data_type, desired_data_type)
                    != ALWAYS_MATCH) {
                    failed_on =
                        "the thing you suggest this action should act on "
                        "has the wrong kind of value";
                    goto Unmatched;
                }
                break;
            }
            case REQUIRES_ACCESS:
            case REQUIRES_POSSESSION:
            case DOESNT_REQUIRE_ACCESS:
                if (is_kova(tok_value_kovs[0], OBJECT_TY) == FALSE) {
                    failed_on =
                        "the thing you suggest this action should act on "
                        "is not an object at all";
                    goto Unmatched;
                }
                break;
```

```
    }
}
if (tok_values == 2) {
    switch(an->second_access) {
        case UNRESTRICTED_ACCESS: {
            kind_of_value *supplied_data_type = tok_value_kovs[1];
            kind_of_value *desired_data_type = an->second_kov;
            if (can_we_cast_kovs(supplied_data_type, desired_data_type)
                != ALWAYS_MATCH) {
                failed_on =
                    "the second thing you suggest this action should act on "
                    "has the wrong kind of value";
                goto Unmatched;
            }
            break;
        }
        case REQUIRES_ACCESS:
        case REQUIRES_POSSESSION:
        case DOESNT_REQUIRE_ACCESS:
            if (is_kova(tok_value_kovs[1], OBJECT_TY) == FALSE) {
                failed_on =
                    "the second thing you suggest this action should act on "
                    "is not an object at all";
                goto Unmatched;
            }
            break;
    }
}
return;
Unmatched:
    LOG("%d token values supplied\n", tok_values);
    { int i;
        for (i=0; i<tok_values; i++) {
            LOG("Token value %d: $u = %s\n", i,
                tok_value_kovs[i], kov_get_NI_source_name(tok_value_kovs[i]));
        }
        LOG("Expected noun kov: $u = %s\n",
            an->noun_kov, kov_get_NI_source_name(an->noun_kov));
        LOG("Expected second kov: $u = %s\n",
            an->second_kov, kov_get_NI_source_name(an->second_kov));
        LOG("Noun access level: %d\n", an->noun_access);
        LOG("Second access level: %d\n", an->second_access);
    }
    quote_source(1, current_sentence);
    if (an->designers_specification == NULL) {
        quote_text(2, "<none given>");
    } else {
        quote_words(2,
        an->designers_specification->word_ref1,
        an->designers_specification->word_ref2);
    }
    quote_words(3, an->word_ref1, an->word_ref2);
    quote_text(4, failed_on);
```

```
        handmade_problem(_P_(C11GrammarMismatchesAction));
        issue_problem_segment("The grammar you give in %1 is not compatible "
            "with the %3 action (defined as '%2') - %4.");
        issue_problem_end();
}
```

The function act_check_types_for_grammar is called from 13/gl.

§**10. Compiling data about actions.** In I6, there was no common infrastructure for the implementation of actions: each defined its own `-Sub` routine. Here, we do have a common infrastructure, and we access it with a single call.

```
void compile_action_routines(OUTPUT_STREAM) {
    action_name *an;
    LOOP_OVER(an, action_name) {
        if (an->use_verb_routine_in_I6_library) continue;
        WRITE("[ %sSub; return GenericVerbSub(%d,%d,%d); ];\n",
            an->an_I6_identifier,
            an->check_rules->allocation_id,
            an->carry_out_rules->allocation_id,
            an->report_rules->allocation_id);
    }
}
```

The function compile_action_routines is invoked by a command in a `.i6t` template file.

§**11. Compiling data about actions.** There are also collective tables of data about actions.

```
void compile_ActionData_array(OUTPUT_STREAM) {
    action_name *an;
    int slots, mn, ms, ml, mnp, msp, hn, hs, j, j0, somethings;
    WRITE("Array ActionData table\n");
    LOOP_OVER(an, action_name) {
        if (an->use_verb_routine_in_I6_library) continue;
        mn = 0; ms = 0; ml = 0; mnp = 1; msp = 1; hn = 0; hs = 0;
        if (an->requires_light) ml = 1;
        if (an->noun_access == REQUIRES_ACCESS) mn = 1;
        if (an->second_access == REQUIRES_ACCESS) ms = 1;
        if (an->noun_access == REQUIRES_POSSESSION) { mn = 1; hn = 1; }
        if (an->second_access == REQUIRES_POSSESSION) { ms = 1; hs = 1; }
        if (an->noun_access == IMPOSSIBLE_ACCESS) mnp = 0;
        if (an->second_access == IMPOSSIBLE_ACCESS) msp = 0;
        WRITE("  ##%-32s  $$%d%d%d%d%d%d%d%d  ",
            an->an_I6_identifier,
            hs, hn,
            (an->out_of_world)?1:0, msp, mnp, ml, ms, mn);
        WRITE("%d ", kov_I6_ID(an->noun_kov));
        WRITE("%d ", kov_I6_ID(an->second_kov));
        slots = 0;
        j = an->word_ref1; j0 = -1; somethings = 0;
        while (j <= an->word_ref2) {
            if [[word j == it]] {
                if (j0 >= 0) {
                    WRITE("\"");
```

```
                    print_action_text_to(j0, j-1, an->word_ref1, OUT);
                    WRITE("\" ");
                    j0 = -1;
                    slots++;
                }
                cat_something(OUT, an, somethings++);
                slots++;
            } else {
                if (j0<0) j0 = j;
            }
            j++;
        }
        if (j0 >= 0) {
            WRITE("\"");
            print_action_text_to(j0, j-1, an->word_ref1, OUT);
            WRITE("\" ");
            slots++;
        }
        if (somethings < an->max_parameters) {
            cat_something(OUT, an, somethings++);
            slots++;
        }
        for (; slots<5; slots++) WRITE("NULL ");
        if ((an->owned_by_an) &&
                (stvo_empty(an->owned_by_an) == FALSE))
            WRITE(" ANSTVC_%d", an->allocation_id);
        else WRITE(" 0");
        WRITE(" %d", 20000+an->allocation_id);
        WRITE("\n");
    }
    WRITE(";\n");
    compile_action_name_var_creators(OUT);
    note_VM_usage("action", -1, -1, NULL, 12, 0, TRUE);
}

void cat_something(OUTPUT_STREAM, action_name *an, int n) {
    kind_of_value *kov = an->noun_kov;
    if (n>0) kov = an->second_kov;
    WRITE("%s ", kov_get_name_of_printing_rule_ACTIONS(kov));
}

void print_action_text_to(int w1, int w2, int start, OUTPUT_STREAM) {
    if (w1 == start) {
        print_text_to_file(w1, w1, OUT);
        if (w1 == w2) return;
        WRITE(" ");
        w1++;
    }
    print_raw_text_to_file(w1, w2, OUT);
}

void compile_ActionCoding_array(OUTPUT_STREAM) {
    int i = 0;
    action_name *an;
    WRITE("Array ActionCoding -->");
    LOOP_OVER(an, action_name) {
```

```
        if ((i++)%8 == 0) WRITE("\n   ");
        if ((an->an_I6_identifier)[0] == '_')
            WRITE(" 0");
        else WRITE(" ##%s", an->an_I6_identifier);
    }
    WRITE(";\n\n");
}
```

The function compile_ActionData_array is invoked by a command in a `.i6t` template file.
The function compile_ActionCoding_array is invoked by a command in a `.i6t` template file.

§12. Indexing.

```
int act_index(action_name *an, int pass,
    extension_file **ext, heading **current_area, int f, int *new_par, int bold,
    int on_details_page) {
    if (an->use_verb_routine_in_I6_library) return f;
    heading *definition_area = heading_of(lw_array[an->word_ref1].lw_source);
    *new_par = FALSE;
    if (pass == 1) {
        extension_file *this_extension =
            hd_get_extension_containing(definition_area);
        if (*ext != this_extension) {
            *ext = this_extension;
            if (*ext == NULL) {
                INDEX("<p><b>New actions defined in the source</b><br>");
            } else
            if (*ext != standard_rules_extension) {
                INDEX("<p><b>Actions defined by the extension ");
                ef_write_name_to_file(*ext, ifl);
                INDEX(" by ");
                ef_write_author_to_file(*ext, ifl);
                INDEX("</b><br>");
            }
            f = FALSE;
            *new_par = TRUE;
        }
        if ((definition_area != *current_area) && (*ext == standard_rules_extension)) {
            int x1, x2, j;
            if (f) INDEX("<p>");
            hd_get_text_of_heading(definition_area, &x1, &x2);
            if (x1 >= 0) {
                j = is_word_intermediate(HYPHEN_V, x1, x2);
                if (j >= 0) x1 = j+1;
                INDEX("<b>");
                print_raw_text_to_file(x1, x2, ifl);
                INDEX("</b><br>");
                f = FALSE;
                *new_par = TRUE;
            } else {
                if (*ext == NULL) {
                    INDEX("<b>");
                    INDEX("New actions");
                    INDEX("</b><br>");
```

```
                f = FALSE;
                *new_par = TRUE;
            }
        }
    }
}
if ((pass == 1) && (f)) INDEX(", ");
f = TRUE;
*current_area = definition_area;
if (pass == 2) INDEX("<b>");
if (an->out_of_world) INDEX("<font color=\"#800000\">");
if (pass == 1) {
    if (bold) INDEX("<b>");
    print_raw_text_to_file(an->word_ref1, an->word_ref2, ifl);
    if (bold) INDEX("</b>");
} else {
    int j = an->word_ref1;
    int somethings = 0;
    while (j <= an->word_ref2) {
        if [[word j == it]] {
            act_index_something(an, somethings++);
        } else {
            print_raw_text_to_file(j, j, ifl);
            INDEX(" ");
        }
        j++;
    }
    if (somethings < an->max_parameters)
        act_index_something(an, somethings++);
}
if (an->out_of_world) INDEX("</font>");
if (pass == 2) {
    int swn = an_get_specification_text(an);
    INDEX("</b>");
    index_link(lw_array[an->designers_specification->word_ref1].lw_source);
    index_anchor(an->an_I6_identifier);
    if (an->requires_light) INDEX(" (requires light)");
    INDEX(" (<i>past tense</i> ");
    print_raw_text_to_file(an->past_word_ref1, an->past_word_ref2, ifl);
    INDEX(")<br>\n");
    INDEX("<p>");
    if (swn >= 0) {
        print_text_to_file(swn, swn, ifl);
        INDEX("<p>");
    }
    INDEX("<hr>");
    INDEX("<p><b>Typed commands leading to this action</b><p>\n");
    if (an->censored == FALSE) {
        if (gl_index_list_with_action(an->list_with_action) == FALSE)
            INDEX("<i>None</i>");
    } else INDEX("<i>None that I wish to tell you about</i>");
    if (stvo_empty(an->owned_by_an) == FALSE) {
        INDEX("<p><b>Named values belonging to this action</b><p>\n");
```

```
            stvo_index(an->owned_by_an);
        }
        INDEX("<p><b>Rules controlling this action</b><p>\n");
        rb_index_action_rules(an, NULL, SETTING_ACTION_VARIABLES_RB, "set action variables for");
        rb_index_action_rules(an, NULL, BEFORE_RB, "before");
        rb_index_action_rules(an, NULL, INSTEAD_RB, "instead of");
        rb_index_action_rules(an, an->check_rules, CHECK_RB, "check");
        rb_index_action_rules(an, an->carry_out_rules, CARRY_OUT_RB, "carry out");
        rb_index_action_rules(an, NULL, AFTER_RB, "after");
        rb_index_action_rules(an, an->report_rules, REPORT_RB, "report");
        INDEX("<p>");
    } else {
        index_link(lw_array[an->designers_specification->word_ref1].lw_source);
        index_detail_link("A", an->allocation_id, (on_details_page)?FALSE:TRUE);
    }
    return f;
}
void act_index_something(action_name *an, int argc) {
    kind_of_value *kov = NULL;                          redundant assignment to appease gcc -O2
    INDEX("<font color=\"#000080\">");
    if (argc == 0) kov = an->noun_kov;
    if (argc == 1) kov = an->second_kov;
    if (is_kova(kov, OBJECT_TY)) INDEX("something");
    else if (is_kova(kov, UNDERSTANDING_TY)) INDEX("some text");
    else if (is_kovcon(kov) == FALSE) index_data_type_name(kov_I6_ID(kov), TRUE);
    else {
        int w1, w2;
        kov_get_name(kov, &w1, &w2, FALSE);
        if (w1 >= 0) {
            print_raw_text_to_file(w1, w2, ifl);
        } else {
            INDEX("%s", vocab_get_exemplar(kov_get_parsing_name(kov), FALSE));
        }
    }
    INDEX("</font> ");
}
```

The function act_index is called from 11/ina.

Action Name Lists 11/anl

Purpose

Action name lists provide a disjunction in the choice of action made by an action pattern. For instance, "taking or dropping the disc" results in a two-entry ANL. An empty ANL is also legal, and means "doing something" – the generic I7 text for "any action at all".

11/anl.§2 Specificity of ANLs

Definitions

```
typedef struct action_name_list {
    struct action_name *action_listed;                    the action in this ANL list entry
    struct action_name_list *next;                        next in this ANL list
    int word_position;                                    and some values used temporarily during parsing
    int parity;
    int parc;
    int parameter_w1[2], parameter_w2[2];
    int in_w1, in_w2;
    int abbreviation_level;                               number of words missing
    int anyone_specified;
} action_name_list;
```

The structure action_name_list is shared with 11/ap.

---

§**1.** The action name list is the part of an action pattern identifying which actions are allowed: "taking, dropping or examining" for example. The following routine extracts this from a potentially longer text (e.g. "taking, dropping or examining a door").

```
action_name_list *anl_new(void) {
    action_name_list *new_anl = CREATE(action_name_list);
    new_anl->action_listed = NULL;
    new_anl->parc = 0;
    new_anl->word_position = -1;
    new_anl->parity = 1;
    new_anl->in_w1 = -1;
    new_anl->in_w2 = -1;
    new_anl->abbreviation_level = 0;
    new_anl->anyone_specified = FALSE;
    return new_anl;
}

void log_action_name_list(action_name_list *anl) {
    int i, c;
    for (c=0; anl; anl = anl->next, c++) {
        LOG("ANL Link (%d@%d): %s ", c, anl->word_position,
            (anl->parity==1)?"+":"-");
        if (anl->action_listed)
            LOG("$W", anl->action_listed->word_ref1, anl->action_listed->word_ref2);
        else LOG("NULL");
        for (i=0; i<anl->parc; i++)
```

```
            LOG(" [%d: $W]", i, anl->parameter_w1[i], anl->parameter_w2[i]);
        LOG(" [in: $W]\n", anl->in_w1, anl->in_w2);
    }
}

void log_action_name_list_briefly(action_name_list *anl) {
    for (; anl; anl = anl->next) {
        if (anl->action_listed == NULL) LOG("ANY / ");
        else LOG("$W / ", anl->action_listed->word_ref1, anl->action_listed->word_ref2);
    }
}

action_name *anl_get_singleton_action(action_name_list *anl) {
    action_name *an;
    if (anl == NULL) internal_error("Supposed singleton ANL is empty");
    an = anl->action_listed;
    if (an == NULL) internal_error("Singleton ANL points to null AN");
    return an;
}

action_name_list *make_action_name_list(int w1, int w2, int tense,
    int depth, int parity) {
    action_name_list *anl_list = NULL, *new_anl = NULL;
    action_name *an;
    int restorative_w2;

    if ((depth == 0) && [[w1, w2 == doing something/anything ***]]) {
        int j, x1 = w1+2;
        if [[x1, w2 == other than ...]] {
            w1 += 4; goto NegatedList;
        }
        if [[x1, w2 == except ...]] {
            w1 += 3; goto NegatedList;
        }
        new_anl = anl_new();
        new_anl->word_position = w1;
        w1 += 2;
        if ((w2 > w1) && ((j = is_the_word_in_intermediate(w1, w2))>=0)) {
            new_anl->in_w1 = j+1;
            new_anl->in_w2 = w2;
            w2 = j-1;
        }
        if (w2 >= w1) {
            if [[w1, w2 == to/with ...]] {
                new_anl->parc = 1;
                new_anl->parameter_w1[0] = w1+1;
                new_anl->parameter_w2[0] = w2;
            } else return NULL;
        }
        return new_anl;
        NegatedList: parity = -1;
    }

    if (is_list_divided(w1, w2, LOOK_FOR_OR)) {
        int lw1 = left_w1, lw2 = left_w2,
            rw1 = right_w1, rw2 = right_w2;
        action_name_list *left_atom =
```

```
        make_action_name_list(lw1, lw2, tense, depth+1, parity);
    action_name_list *right_tail =
        make_action_name_list(rw1, rw2, tense, depth+1, parity);
    if (left_atom == NULL) return NULL;
    if (right_tail == NULL) return NULL;
    new_anl = right_tail;
    while (new_anl->next != NULL) new_anl = new_anl->next;
    new_anl->next = left_atom;
    return right_tail;
}
LOGIF(AP_PARSING, "Parsing ANL from $W (tense %d)\n", w1, w2, tense);
new_anl = anl_new();
anl_list = NULL;
restorative_w2 = w2;
LOOP_OVER(an, action_name) {
    int j, w_m, x_m, x_ended = FALSE;
    int x1, x2;
    int fc = 0;
    int it_optional = act_it_optional(an);
    w2 = restorative_w2;
    act_set_text_to_name_tensed(an, &x1, &x2, tense);
    new_anl->action_listed = an;
    new_anl->parc = 0;
    new_anl->word_position = w1;
    new_anl->parity = parity;
    new_anl->in_w1 = -1;
    new_anl->in_w2 = -1;
    w_m = w1; x_m = x1;
    while ((w_m <= w2) && (x_m <= x2)) {
        if (lw_array[x_m++].lw_identity != lw_array[w_m++].lw_identity) {
            fc=1; goto DontInclude;
        }
        if (x_m > x2) { x_ended = TRUE; break; }
        if [[word x_m == it]] {
            if (w_m > w2) x_ended = TRUE; else {
                int j = is_word_intermediate_inc(
                    lw_array[x_m+1].lw_identity, (it_optional)?(w_m):(w_m+1), w2);
                if (j<0) { fc=2; goto DontInclude; }
                if (j-1 >= w_m) {
                    new_anl->parameter_w1[new_anl->parc] = w_m;
                    new_anl->parameter_w2[new_anl->parc] = j-1;
                    new_anl->parc++;
                } else {
                    new_anl->parameter_w1[new_anl->parc] = -1;
                    new_anl->parameter_w2[new_anl->parc] = -1;
                    new_anl->parc++;
                }
                w_m = j; x_m++;
            }
        }
        if (x_ended) break;
    }
    if ((w_m > w2) && (x_ended == FALSE)) {
```

```
            if ([[word w1 == listening]] && [[x_m, x2 == to]]) x_ended = TRUE;
            else { fc=3; goto DontInclude; }
        }
        if (x_m <= x2) new_anl->abbreviation_level = x2-x_m+1;
        j = is_the_word_in_intermediate(w_m, w2);
        if ((j>=0) && (j<w2)) {
            new_anl->in_w1 = j+1;
            new_anl->in_w2 = w2;
            w2 = j-1;
        }
        if (w_m <= w2) {
            if (act_can_have_parameters(an) == FALSE) { goto DontInclude; }
            if (parity == -1)
                [[w_m, w2 == to/with ... --> w_m, w2]];
            new_anl->parameter_w1[new_anl->parc] = w_m;
            new_anl->parameter_w2[new_anl->parc] = w2;
            new_anl->parc++;
        }
        if (strcmp(an_get_I6_representation(an), "_Pattern_") == 0)
            new_anl->action_listed = NULL;
        new_anl->next = NULL;
        if (anl_list == NULL) anl_list = new_anl;
        else {
            action_name_list *pos = anl_list, *prev = NULL;
            while ((pos) && (pos->abbreviation_level < new_anl->abbreviation_level))
                prev = pos, pos = pos->next;
            if (prev) prev->next = new_anl; else anl_list = new_anl;
            new_anl->next = pos;
        }
        new_anl = anl_new();
        DontInclude: ;
    }
    LOGIF(AP_PARSING, "Parsing ANL from $W resulted in:\n$L\n", w1, w2, anl_list);
    return anl_list;
}
action_name_list *anl_extract_actions_only(int w1, int w2) {
    int aw1, aw2, d1, d2, i, f = FALSE;
    action_name_list *anl;
    time_period duration;
    if ([[w1, w2 == asking ... to ... : i --> aw1, aw2 ... d1, d2]] &&
        [[d1, d2 == try ... --> w1, w2]]) {
        ;
    } else {
        if [[w1, w2 == ... trying ... : i --> aw1, aw2 ... w1, w2]] {
            ;
        } else {
            if [[w1, w2 == an actor ... --> w1, w2]]
                f = TRUE;
            if [[w1, w2 == trying ... --> w1, w2]] ;
            else {
                int bl = 0;
                if (vocab_test_flags(w1, ING_MC) == FALSE)
                for (i=w1+1; i<=w2; i++) {
```

```
                    if [[word i == OPENBRACKET]] bl++;
                    if [[word i == CLOSEBRACKET]] bl--;
                    if ((bl == 0) &&
                        (vocab_test_flags(i, ING_MC)) &&
                        (![[word i == thing/something]])) {
                        w1 = i;
                        break;
                    }
                }
            }
        }
    }
    duration = parse_time_period(w1, w2);
    if (tp_is_valid(&duration)) w2 = tp_is_valid(&duration);
    i = is_word_intermediate_inc(presence_V, w1, w2);
    if (i>=w1+3) {
        if ((compare_word(i-2, in_V)) &&
            (compare_word(i-1, the_V)) &&
            (compare_word(i+1, of_V))) {
            w2 = i-3;
        }
    }
    anl = make_action_name_list(w1, w2, IS_TENSE, 0, TRUE);
    if (anl) anl->anyone_specified = f;
    return anl;
}
action_name *anl_get_single_action(action_name_list *anl) {
    int posn = -1, matchl = -1;
    action_name *anf = NULL;
    LOGIF(RULE_ATTACHMENTS, "Getting single action from:\n$L\n", anl);
    while (anl) {
        if (anl->parity == -1) return NULL;
        if (anl->action_listed) {
            int k = an_get_stem_length(anl->action_listed) - anl->abbreviation_level;
            if (anl->word_position != posn) {
                if (posn >= 0) return NULL;
                posn = anl->word_position;
                anf = anl->action_listed;
                matchl = k;
            } else {
                if (k > matchl) {
                    matchl = k;
                    anf = anl->action_listed;
                }
            }
        }
        anl = anl->next;
    }
    LOGIF(RULE_ATTACHMENTS, "Posn %d AN $l\n", posn, anf);
    return anf;
}
int anl_get_explicit_anyone_flag(action_name_list *anl) {
    if (anl == NULL) return FALSE;
```

```
    return anl->anyone_specified;
}

int count_actions_covered(action_name_list *anl) {
    int k, parity = TRUE;
    if (anl == NULL) return 0;
    for (k=0; anl; anl = anl->next) {
        if (anl->parity == 1) parity = TRUE;
        else parity = FALSE;
        k++;
    }
    if (parity == FALSE) k = 1000000-k;                    1,000,000 being a poor man's infinity
    return k;
}

void compile_action_name_list(OUTPUT_STREAM, action_name_list *anl) {
    if (anl == NULL) return;
    LOGIF(COMPILE_ACTION_PATTERNS, "CANL: $L", anl);
    WRITE("(action %s", (anl->parity==1)?"==":"~=");
    while (anl != NULL) {
        WRITE("##%s", an_get_I6_representation(anl->action_listed));
        if (anl->next != NULL) WRITE(" or ");
        anl = anl->next;
    }
    WRITE(")");
}
```

The function log_action_name_list is called from 2/dl.
The function log_action_name_list_briefly is called from 11/ap.
The function anl_get_singleton_action is called from 7/cmsp and 11/ap.
The function make_action_name_list is called from 11/ap.
The function anl_extract_actions_only is called from 12/phud.
The function anl_get_single_action is called from 11/ap and 12/phud.
The function anl_get_explicit_anyone_flag is called from 12/phud.
The function compile_action_name_list is called from 11/ap.

§**2. Specificity of ANLs.** The following is one of NI's standardised comparison routines, which takes a pair of objects A, B and returns 1 if A makes a more specific description than B, 0 if they seem equally specific, or −1 if B makes a more specific description than A. This is transitive, and intended to be used in sorting algorithms.

```
int compare_specificity_of_anl(action_name_list *anl1, action_name_list *anl2) {
    int count1, count2;
    count1 = count_actions_covered(anl1);
    count2 = count_actions_covered(anl2);
    if (count1 < count2) return 1;
    if (count1 > count2) return -1;
    return 0;
}
```

The function compare_specificity_of_anl is called from 11/ap.

Action Patterns 11/ap

Purpose

An action pattern is a description which may match many actions or none. The text "doing something" matches every action, while "throwing something at a door in a dark room" is seldom matched. Here we parse such text into a data structure called an `action_pattern`.

11/ap.§19-20 Action pattern specificity; §21-22 Compiling action patterns

Definitions

¶1. Action patterns are essentially a conjunction of specifications – the action must be this, *and* the noun must be that, *and*... While they allow disjunction in the choice of action, all of that code is a matter for the action name list to handle. The AP structure is a list of conditions all of which must apply at once.

One surprising point is that the AP is used not only for action patterns, but also in a slightly generalised role, as the condition for a rule to be applied. Most rules are indeed predicated on actions – "instead of eating the cake" – but some are instead in "parametrised" rulebooks, which means they apply to a parameter object instead of an action – "reaching inside the cabinet". These not-really-action APs are used in no other context, and employ the `parameter_spec` field below, ignoring the rest.

```
typedef struct action_pattern {
    int word_ref1, word_ref2;                          text giving rise to this AP
    struct named_action_pattern *named;                give a named action pattern...
    struct action_name_list *action;                   ...or give an ANL list (not both)
    int test_anl;                                      actually test the action when compiled
    int applies_to_any_actor;                          treat player and other people equally
    int request;                                       a request from the player for someone to do this?
    struct specification *actor_spec;
    struct specification *noun_spec;
    struct specification *second_spec;
    struct specification *presence_spec;               in the presence of...
    struct specification *room_spec;                   in...
    struct specification *when;                        when... (any condition here)
    struct specification *from_spec;                   for the "going" action only
    struct specification *to_spec;                     ditto
    struct specification *by_spec;                     ditto
    struct specification *through_spec;                ditto
    struct specification *pushing_spec;                ditto
    int nowhere_flag;                                  ditto: a flag for "going nowhere"
    struct ap_optional_clause *optional_clauses;
    int chief_action_owner_id;                         stacked variable ID number of main action
    struct time_period duration;                       to hold "for the third time", etc.
    struct specification *parameter_spec;              alternatively, just this
    int AP_parity;                                     +1 if meant positively, -1 if negatively
    int valid;                                         recording success or failure in parsing to an AP
    struct action_pattern *next;                       for forming APs into linked lists
} action_pattern;

typedef struct ap_optional_clause {
```

```
    struct stacked_variable *pertains_to;
    struct specification *clause_spec;
    struct ap_optional_clause *next;
    MEMORY_MANAGEMENT
} ap_optional_clause;
```

The structure action_pattern is private to this section.
The structure ap_optional_clause is private to this section.

¶2. When we parse action patterns, we record why they fail, in order to make it easier to produce helpful error messages. (We can't simply fire off errors at the time they occur, because text is often parsed in several contexts at once, so just because it fails this one does not mean it is wrong.) PAPF stands for "parse action pattern failure".

```
define MISC_PAPF 1
define NOPARTICIPLE_PAPF 2
define MIXEDNOUNS_PAPF 3
define WHEN_PAPF 4
define WHENOKAY_PAPF 5
define IMMISCIBLE_PAPF 6

int pap_failure_reason;                    one of the above
int permit_trying_omission = FALSE;        allow the keyword 'trying' to be omitted
```

```
action_pattern new_action_pattern(void) {
    action_pattern ap;
    ap.word_ref1 = -1; ap.word_ref2 = -1;
    ap.action = NULL;
    ap.test_anl = TRUE;
    ap.actor_spec = NULL;
    ap.noun_spec = NULL; ap.second_spec = NULL; ap.room_spec = NULL;
    ap.parameter_spec = NULL;
    ap.valid = FALSE;
    ap.next = NULL;
    ap.named = NULL;
    ap.when = NULL;
    ap.presence_spec = NULL;
    ap.AP_parity = 1;
    ap.from_spec = NULL;
    ap.to_spec = NULL;
    ap.by_spec = NULL;
    ap.through_spec = NULL;
    ap.pushing_spec = NULL;
    ap.nowhere_flag = FALSE;
    ap.request = FALSE;
    ap.applies_to_any_actor = FALSE;
    ap.duration = new_time_period();
    ap.optional_clauses = NULL;
    ap.chief_action_owner_id = 0;
    return ap;
}
```

```
ap_optional_clause *apoc_new(stacked_variable *stv, specification *spec) {
    ap_optional_clause *apoc = CREATE(ap_optional_clause);
    apoc->stv_to_match = stv;
    apoc->clause_spec = spec;
    apoc->next = NULL;
    return apoc;
}
void ap_add_optional_clause(action_pattern *ap, ap_optional_clause *apoc) {
    int oid = stv_get_owner_id(apoc->stv_to_match);
    int off = stv_get_offset(apoc->stv_to_match);
    if (ap->optional_clauses == NULL) {
        ap->optional_clauses = apoc;
        apoc->next = NULL;
    } else {
        ap_optional_clause *oapoc = ap->optional_clauses, *papoc = NULL;
        while (oapoc) {
            int ooff = stv_get_offset(oapoc->stv_to_match);
            if (off < ooff) {
                if (oapoc == ap->optional_clauses) {
                    apoc->next = ap->optional_clauses;
                    ap->optional_clauses = apoc;
                    papoc = NULL;
                } else {
                    apoc->next = papoc->next;
                    papoc->next = apoc;
                    papoc = NULL;
                }
                break;
            }
            papoc = oapoc;
            oapoc = oapoc->next;
        }
        if (papoc) {
            apoc->next = NULL;
            papoc->next = apoc;
        }
    }
    if (oid == 20007 /* i.e., going */ ) {
        switch (off) {
            case 0: ap->from_spec = apoc->clause_spec; break;
            case 1: ap->to_spec = apoc->clause_spec; break;
            case 2: ap->through_spec = apoc->clause_spec; break;
            case 3: ap->by_spec = apoc->clause_spec; break;
            case 4: ap->pushing_spec = apoc->clause_spec; break;
        }
    }
    ap->chief_action_owner_id = oid;
}
int ap_count_optional_clauses(action_pattern *ap) {
    int n = 0;
    ap_optional_clause *apoc;
    for (apoc = ap->optional_clauses; apoc; apoc = apoc->next) {
        if ((ap->chief_action_owner_id != 20007) ||
```

```
            (stv_get_offset(apoc->stv_to_match) >= 5))
            n++;
    }
    return n;
}

int compare_specificity_of_apoc_list(action_pattern *ap1, action_pattern *ap2) {
    int rct1 = ap_count_optional_clauses(ap1);
    int rct2 = ap_count_optional_clauses(ap2);
    int off1, off2;
    ap_optional_clause *apoc1, *apoc2;

    if (rct1 > rct2) return 1;
    if (rct1 < rct2) return -1;
    if (rct1 == 0) return 0;
    if (ap1->chief_action_owner_id != ap2->chief_action_owner_id) return 0;

    apoc1 = ap1->optional_clauses; apoc2 = ap2->optional_clauses;
    while ((apoc1) && (apoc2)) {
        off1 = stv_get_offset(apoc1->stv_to_match);
        off2 = stv_get_offset(apoc2->stv_to_match);
        if (off1 == off2) {
            int rv = compare_specificity_of_spec(apoc1->clause_spec, apoc2->clause_spec);
            if (rv != 0) return rv;
            apoc1 = apoc1->next;
            apoc2 = apoc2->next;
        }
        if (off1 < off2) apoc1 = apoc1->next;
        if (off1 > off2) apoc2 = apoc2->next;
    }
    return 0;
}

void log_action_pattern(action_pattern *ap) {
    if (ap->valid != TRUE) LOG("  [Invalid]");
    else LOG("  [Valid]");
    if (ap->named) LOG("  Named:'$W'", ap->named->word_ref1, ap->named->word_ref2);
    LOG("  Action: ");
    if (ap->action == NULL) LOG("unspecified");
    else log_action_name_list_briefly(ap->action);
    if (ap->noun_spec) LOG("  Noun: $S", ap->noun_spec);
    if (ap->second_spec) LOG("  Second: $S", ap->second_spec);
    if (ap->from_spec) LOG("  From: $S", ap->from_spec);
    if (ap->to_spec) LOG("  To: $S", ap->to_spec);
    if (ap->by_spec) LOG("  By: $S", ap->by_spec);
    if (ap->through_spec) LOG("  Through: $S", ap->through_spec);
    if (ap->pushing_spec) LOG("  Pushing: $S", ap->pushing_spec);
    if (ap->room_spec) LOG("  Room: $S", ap->room_spec);
    if (ap->parameter_spec) LOG("  Parameter: $S", ap->parameter_spec);
    if (ap->presence_spec) LOG("  Presence: $S", ap->presence_spec);
    if (ap->nowhere_flag) LOG("  Nowhere  ");
    LOG("\n");
}

action_pattern *store_ap(action_pattern ap) {
    action_pattern *sap = CREATE(action_pattern);
    *sap = ap;
```

```
    return sap;
}

int ap_is_named(action_pattern *ap) {
    if (ap->named == NULL) return FALSE;
    return TRUE;
}

int ap_is_valid(action_pattern *ap) {
    return ap->valid;
}

int ap_is_request(action_pattern *ap) {
    return ap->request;
}

int ap_within_action_context(action_pattern *ap, action_name *an) {
    action_name_list *anl;
    if (ap == NULL) return TRUE;
    if (ap->named) return nap_within_action_context(ap->named, an);
    if (ap->action == NULL) return TRUE;
    for (anl = ap->action; anl; anl = anl->next)
        if (anl->action_listed == an)
            return TRUE;
    return FALSE;
}

action_name *ap_required_action(action_pattern *ap) {
    if ((ap->action) && (ap->action->next == NULL) && (ap->action->parity == 1))
        return ap->action->action_listed;
    return NULL;
}

int ap_is_unspecific(action_pattern *ap) {
    if (ap_required_action(ap) == NULL) return TRUE;
    if (ap_clause_is_unspecific(ap->noun_spec)) return TRUE;
    if (ap_clause_is_unspecific(ap->second_spec)) return TRUE;
    if (ap_clause_is_unspecific(ap->actor_spec)) return TRUE;
    return FALSE;
}

int ap_clause_is_unspecific(specification *spec) {
    if (spec == NULL) return FALSE;
    if (species_is(spec, DESCRIPTION_SPC) == FALSE) return FALSE;
    return TRUE;
}

int ap_is_overspecific(action_pattern *ap) {
    if (ap->when != NULL) return TRUE;
    if (ap->room_spec != NULL) return TRUE;
    if (ap->presence_spec != NULL) return TRUE;
    if (ap->optional_clauses != NULL) return TRUE;
    if (ap->nowhere_flag) return TRUE;
    if (ap->applies_to_any_actor) return TRUE;
    return FALSE;
}

void coerce_TEST_ACTION_to_TRY_ACTION_by_ap(specification *spec, action_pattern *ap) {
    coerce_TEST_ACTION_to_TRY_ACTION(spec,
        ap->noun_spec, ap->second_spec, ap->actor_spec,
```

```
        ap->request, ap->action);
}

void ap_suppress_action_testing(action_pattern *ap) {
    if (tp_is_valid(&(ap->duration)) == FALSE) ap->test_anl = FALSE;
}
```

The function new_action_pattern is called from 12/phud and 12/phrcd.
The function log_action_pattern is called from 2/dl and 8/mass.
The function store_ap is called from 5/ml, 5/tandv and 12/phtd.
The function ap_is_named is called from 8/mass.
The function ap_is_valid is called from 5/tandv, 5/candp, 8/refpt, 8/mass, 12/phud, 12/phrcd and 12/phtd.
The function ap_is_request is called from 12/phud.
The function ap_within_action_context is called from 11/nap and 12/phrcd.
The function ap_required_action is called from 12/phrcd.
The function ap_is_unspecific is called from 7/tc.
The function ap_is_overspecific is called from 7/tc.
The function coerce_TEST_ACTION_to_TRY_ACTION_by_ap is called from 7/tc.
The function ap_suppress_action_testing is called from 12/phrcd.

§**1.** We are allowed to give names to certain kinds of behaviour by "categorising" an action.

```
void categorise_action_as(action_pattern ap, int w1, int w2) {
    LOGIF(AP_PARSING, "Categorising the action:\n$A...as $W\n",
        &ap, w1, w2);
    if (ap.actor_spec) {
        sentence_problem(_P_(C11NamedAPWithActor),
            "behaviour characterised by named action patterns can only specify the action",
            "not the actor: as a result, it cannot include requests to other people to "
            "do things.");
        return;
    }
    nap_add_ap(store_ap(ap), w1, w2);
}
```

The function categorise_action_as is called from 8/mass.

§**2.** Before the rather complicated code to parse a general text into an action pattern, we put down some utility routines for type-checking that what we get will make sense.

The exceptional treatment of the `PROPERTY_TY` kind of value below is to allow "examining scenery" to be an action pattern, where an either/or property has a name which is really a noun rather than an adjective, luring people into treating it as such.

```
specification *last_spec_failing_to_validate = NULL;
kind_of_value *last_kov_failing_to_validate = NULL;
kind_of_value *last_kov_found_failing_to_validate = NULL;

void clear_validation_case(void) {
    last_spec_failing_to_validate = NULL;
    last_kov_failing_to_validate = NULL;
    last_kov_found_failing_to_validate = NULL;
}

int get_validation_case(specification **spec, kind_of_value **kov,
    kind_of_value **kov2) {
    *spec = last_spec_failing_to_validate;
    *kov = last_kov_failing_to_validate;
```

```
    *kov2 = last_kov_found_failing_to_validate;
    if ((*spec == NULL) || (*kov == NULL)) return FALSE;
    return TRUE;
}
int validate_parameter(specification *spec, kind_of_value *kov) {
    specification *vts;
    kind_of_value *kov_found = NULL;
    if (spec == NULL) return TRUE;
    LOGIF(AP_PARSING, "Validating parameter in action pattern: $S ($u)\n",
        spec, kov);
    if (spec_is_UNKNOWN(spec)) {
        if [[spec == something/anything]] return TRUE;
        if [[spec == something/anything else]] return TRUE;
        goto DontValidate;
    }
    if (spec_is_generic(spec)) return FALSE;
    kov_found = spec_evaluates_to(spec);
    if ((is_kova(kov_found, PROPERTY_TY)) && (is_kova(kov, OBJECT_TY)))
        return TRUE;
    if ((is_kova(kov_found, SNIPPET_TY)) && (is_kova(kov, UNDERSTANDING_TY)))
        return TRUE;
    if ((is_kova(kov, UNDERSTANDING_TY)) && (species_is(spec, CONSTANT_SPC) == FALSE) &&
        (is_kova(kov_found, TEXT_TY)))
        goto DontValidate;
    vts = new_generic_CONSTANT_type(kov);
    if (can_we_match_value_descriptions(spec, vts) == NEVER_MATCH) {
        if ((is_kova(kov, UNDERSTANDING_TY)) && (species_is(spec, CONSTANT_SPC))) {
            vts = new_actual_CONSTANT_spec(kova(SNIPPET_TY));
            if (can_we_match_value_descriptions(spec, vts) != NEVER_MATCH) return TRUE;
        }
        if (is_kova(kov_found, ANY_VALUE_TY)) return TRUE;        pick up later in type-checking
        goto DontValidate;
    }
    return TRUE;
    DontValidate:
        LOGIF(AP_PARSING,
            "Fails to validate for type-checking reasons: wanted $u, found $u\n",
            kov, kov_found);
        last_spec_failing_to_validate = spec_copy(spec);
        last_kov_failing_to_validate = kov;
        last_kov_found_failing_to_validate = kov_found;
        return FALSE;
}
int validate_when(specification *spec) {
    specification *cond_t = new_generic_CONDITION_type();
    LOGIF(AP_PARSING, "Validating 'when' clause in action pattern: $S\n", spec);
    if (spec == NULL) return TRUE;
    if (spec_is_UNKNOWN(spec)) return FALSE;
    if (typecheck(spec, cond_t) == NEVER_MATCH) return FALSE;
    return TRUE;
}
specification *nullify_nonspecific_references(specification *spec) {
```

```
    if (spec == NULL) return spec;
    if (spec_is_UNKNOWN(spec)) return NULL;
    if ((species_is(spec, DESCRIPTION_SPC)) &&
        [[spec == something/anything]]) {
        spec_set_described_kind(spec, kind_thing);
    }
    return spec;
}

int check_going(specification *spec, char *keyword,
    world_object *ka, world_object *kb) {
    if (spec == NULL) return TRUE;
    if (spec_describes_an_object_vaguely_or_exactly(spec)) {
        world_object *oref = spec_object_exactly_described_if_any(spec);
        if ((oref == NULL) || (ka == NULL) || (wo_of_kind(oref, ka)) ||
            ((kb) && (wo_of_kind(oref, kb)))) return TRUE;
        quote_source(1, current_sentence);
        quote_object(2, oref);
        quote_text(3, keyword);
        quote_object(4, ka);
        quote_object(5, oref->kind);
        if (kb) quote_object(6, kb);
        handmade_problem(_P_(C11GoingWrongKind));
        if (kb)
        issue_problem_segment(
            "In the sentence %1, %2 seems to be intended as something the "
            "player might be going %3, but this has the wrong kind: %5 "
            "rather than %4 or %6.");
        else
        issue_problem_segment(
            "In the sentence %1, %2 seems to be intended as something the player "
            "might be going %3, but this has the wrong kind: %5 rather than %4.");
        issue_problem_end();
        return TRUE;
    }
    quote_source(1, current_sentence);
    quote_words(2, spec->word_ref1, spec->word_ref2);
    quote_text(3, keyword);
    handmade_problem(_P_(C11GoingWithoutObject));
    issue_problem_segment(
        "In the sentence %1, '%2' seems to be intended as something the player "
        "might be going %3, but it doesn't make sense in that context.");
    issue_problem_end();
    return FALSE;
}
```

The function clear_validation_case is called from 7/tc.

The function get_validation_case is called from 7/tc.

The function validate_parameter is called from 11/av.

The function validate_when is called from 9/pp, 11/av, 12/phsf, 12/def and 13/gl.

§**3.** First a much easier, parametric form of parsing, used for the APs which form the usage conditions for rules in object-based rulebooks.

```
action_pattern parse_parametric_action_pattern(int aw1, int aw2, int tense) {
    action_pattern ap = new_action_pattern();
    specification *value_t = new_generic_CONSTANT_type(kova(OBJECT_TY));
    ap.parameter_spec = parse_action_parameter(aw1, aw2, TRUE);
    if ((spec_is_UNKNOWN(ap.parameter_spec)) ||
        (can_we_match_value_descriptions(ap.parameter_spec, value_t) == NEVER_MATCH))
        ap.valid = FALSE;
    else ap.valid = TRUE;
    return ap;
}
```

The function parse_parametric_action_pattern is called from 12/phud.

§**4.** A useful utility: parsing a parameter in an action pattern.

```
specification *parse_action_parameter(int w1, int w2, int allow_adjlist) {
    specification *spec;
    if (allow_adjlist) spec = parse_expression(w1, w2, DESCRIPTIVE_TYPE_EXPCON);
    else spec = parse_expression(w1, w2, TYPE_EXPCON);
    if (spec_is_UNKNOWN(spec)) {
        meaning_list *ml = SP_excerpt(QUANTITY_MC, w1, w2);
        if (ml) {
            quantity *q = RETRIEVE_POINTER_quantity(em_data(ml_meaning(ml)));
            if ((q) && (qty_is_a_variable(q))) spec = new_QUANTITY_spec(q);
        }
    }
    return spec;
}
```

§**5.** The main action pattern parser is called only by the following shell routine, which exists in order to change some parsing rules.

```
int suppress_ap_parsing = FALSE;
action_pattern parse_action_pattern(int w1, int w2, int tense) {
    action_pattern ap;
    int d, wh_flag = FALSE;
    if ((w1<0) || (w2<w1)) internal_error("PAP on illegal word range");
    if [[w1, w2 == when/while ***]] wh_flag = TRUE;
    d = vocab_disjunction_of_flags(w1, w2);
    if ((suppress_ap_parsing) ||
        (((tense == IS_TENSE) && ((d & (ING_MC+NAMED_AP_MC)) == 0)
            && (wh_flag==FALSE)))) {
        ap = new_action_pattern(); ap.valid = FALSE;
        ap.word_ref1 = w1; ap.word_ref2 = w2;
        pap_failure_reason = NOPARTICIPLE_PAPF;
        return ap;
    }
    LOGIF(AP_PARSING, "Parse action pattern (tense %d): $W\n", tense, w1, w2);
    if ((w1 == w2) && (wh_flag)) {
        ap = new_action_pattern(); ap.valid = FALSE;
```

```
        ap.word_ref1 = w1; ap.word_ref2 = w2;
        pap_failure_reason = MISC_PAPF;
    } else {
        ap = parse_action_pattern_dash(w1, w2, tense);
    }
    LOGIF(AP_PARSING, "PAP result (pfr %d): $A\n", pap_failure_reason, &ap);
    return ap;
}
```

The function parse_action_pattern is called from 5/tandv, 5/candp, 7/tc, 8/refpt, 8/mass, 12/phud and 12/phtd.

§**6.** We can't put it off any longer. Here goes.

```
int last_successful_w1 = -1, last_successful_w2 = -1;
int ap_trying_omission_position = -1;
action_pattern parse_action_pattern_dash(int w1, int w2, int tense) {
    int non_temporal_w1, non_temporal_w2;
    int failure_this_call = MISC_PAPF;
    int i, j, k = 0, aw1, aw2, d1, d2;
    action_pattern ap;
    int ap_parity = 1;
    action_name_list *anl = NULL;
    specification *wts = NULL, *itp = NULL;
    time_period duration;
    ap = new_action_pattern(); ap.valid = FALSE;
    ap.word_ref1 = w1; ap.word_ref2 = w2;
    ap_trying_omission_position = -1;
    if ([[w1, w2 == asking ... to ... : i --> aw1, aw2 ... d1, d2]] &&
        (aw2 >= aw1) &&
        [[d1, d2 == try ... --> w1, w2]]) {
        ap.actor_spec = parse_expression(aw1, aw2, TYPE_OR_VALUE_EXPCON);
        ap.request = TRUE;
    } else {
        if [[w1, w2 == ... trying ... : i --> aw1, aw2 ... w1, w2]] {
            ap.actor_spec = parse_expression(aw1, aw2, TYPE_OR_VALUE_EXPCON);
            ap.request = FALSE;
        } else {
            if [[w1, w2 == an actor ... --> w1, w2]] {
                ap.applies_to_any_actor = TRUE;
                [[w1, w2 == trying ... --> w1, w2]] ;
            } else {
                if [[w1, w2 == trying ... --> w1, w2]] ;
                else if (permit_trying_omission) {
                    int bl = 0;
                    for (i=w1+1; i<=w2; i++) {
                        if [[word i == OPENBRACKET]] bl++;
                        if [[word i == CLOSEBRACKET]] bl--;
                        if ((bl == 0) &&
                            (vocab_test_flags(i, ING_MC)) &&
                            (![[word i == thing/something]])) {
                            specification *try_stem;
                            world_object *wo;
                            suppress_ap_parsing = TRUE;
```

```
                            try_stem = parse_expression(w1, i-1, TYPE_OR_VALUE_EXPCON);
                            suppress_ap_parsing = FALSE;
                            for (j=w1, k=0; j<=i-1; j++)
                                if (vocab_test_flags(j, ING_MC)) k++;
                            if (k>0) continue;
                            wo = wo_of_CONSTANT_OBJECT_if_any(try_stem);
                            if (wo_has_ING_name(wo)) continue;
                            if ((spec_get_storage_form(try_stem) == LOCAL_VARIABLE_SPC) ||
                                (spec_get_storage_form(try_stem) == NONLOCAL_VARIABLE_SPC)
||
                                (species_is(try_stem, CONSTANT_SPC)) ||
                                (species_is(try_stem, DESCRIPTION_SPC))) {
                                ap_trying_omission_position = i;
                                ap.actor_spec = try_stem;
                                ap.request = FALSE;
                                w1 = i;
                                break;
                            }
                        }
                    }
                }
            }
        }
    }
    if ((ap.actor_spec) && (spec_is_UNKNOWN(ap.actor_spec))) goto Failed;
    LOGIF(AP_PARSING, "PAR after preamble: $W\n", w1, w2);
    ⟨PAR - (a) Parse Terminal Duration 7⟩;
    ⟨PAR - (b) Parse Doing It 8⟩;
    non_temporal_w1 = w1; non_temporal_w2 = w2;
    ⟨PAR - (c) Parse Initial Not 9⟩;
    ⟨PAR - (d) Parse Terminal When Clause 10⟩;
    ⟨PAR - (e) Parse Terminal Presence Clause 11⟩;
    ⟨PAR - (f) Parse Special Going Clauses 12⟩;
    ⟨PAR - (g) Parse Initial In Clause 13⟩;
    ⟨PAR - (h) Parse Named Action Pattern 14⟩;
    ⟨PAR - (i) Parse Initial Action Name List 15⟩;
    ⟨PAR - (j) Parse Parameters 16⟩;
    ⟨PAR - (k) Verify Mixed Action 17⟩;
    Failed:
        LOGIF(AP_PARSING, "Parse action failed: $W\n", w1, w2);
        ap.valid = FALSE;
        pap_failure_reason = failure_this_call;
        ap.optional_clauses = NULL;
        ap.from_spec = NULL;
        ap.to_spec = NULL;
        ap.by_spec = NULL;
        ap.through_spec = NULL;
        ap.pushing_spec = NULL;
        ap.nowhere_flag = FALSE;
        return ap;
    Succeeded:
```

```
        pap_failure_reason = 0;
        last_successful_w1 = non_temporal_w1;
        last_successful_w2 = non_temporal_w2;
        LOGIF(AP_PARSING, "Last successful w1,w2 set to: $W\n",
            last_successful_w1, last_successful_w2);
        ap.word_ref1 = w1; ap.word_ref2 = w2;
        ap.action = anl;
        if ((anl != NULL) && (anl->action_listed == NULL)) ap.action = NULL;
        ap.valid = TRUE;
        ap.when = wts;
        ap.presence_spec = itp;
        ap.duration = duration;
        ap.AP_parity = ap_parity;
        ap.actor_spec = nullify_nonspecific_references(ap.actor_spec);
        ap.noun_spec = nullify_nonspecific_references(ap.noun_spec);
        ap.second_spec = nullify_nonspecific_references(ap.second_spec);
        ap.room_spec = nullify_nonspecific_references(ap.room_spec);
        ap.presence_spec = nullify_nonspecific_references(ap.presence_spec);
        if (check_going(ap.from_spec, "from", kind_room, kind_region) == FALSE)
            ap.valid = FALSE;
        if (check_going(ap.to_spec, "to", kind_room, kind_region) == FALSE)
            ap.valid = FALSE;
        if (check_going(ap.by_spec, "by", kind_thing, NULL) == FALSE)
            ap.valid = FALSE;
        if (check_going(ap.through_spec, "through", kind_door, NULL) == FALSE)
            ap.valid = FALSE;
        if (check_going(ap.pushing_spec, "with", kind_thing, NULL) == FALSE)
            ap.valid = FALSE;
        if (ap.valid == FALSE) goto Failed;
        LOGIF(AP_PARSING, "Matched action pattern: $A\n", &ap);
        return ap;
}
```

§**7.** An indication of duration at the end of an action pattern. Trim this off.

⟨PAR - (a) Parse Terminal Duration 7⟩ ≡

```
    duration = parse_time_period(w1, w2);
    if (tp_is_valid(&duration)) w2 = tp_is_valid(&duration);
```

This code is used in §6.

§**8.** Match "doing it" as a repetition of the previous successfully matched action pattern.

⟨PAR - (b) Parse Doing It 8⟩ ≡

```
    if [[w1, w2 == doing it]] {
        if (last_successful_w1 >= 0) {
            w1 = last_successful_w1; w2 = last_successful_w2;
            LOGIF(AP_PARSING, "Doing it refers to $W\n", w1, w2);
        } else {
            LOGIF(AP_PARSING, "Doing it fails, no last successful w1\n");
        }
    }
```

This code is used in §6.

§**9.** Record a change in parity and trim off an initial "not".

⟨PAR - (c) Parse Initial Not 9⟩ ≡

```
ap_parity = 1;
if [[w1, w2 == not ... --> w1, w2]] ap_parity = -1;
```

This code is used in §6.

§**10.** Parse, and trim away, a "when..." condition.

⟨PAR - (d) Parse Terminal When Clause 10⟩ ≡

```
j = is_word_intermediate_inc(when_V, w1, w2);
if (j<0) j = is_word_intermediate_inc(while_V, w1, w2);
if ((rule_circumstances_mode) && [[w1, w2 == when/while ***]]) j=w1;
if ((j>=0) && (j<w2)) {
    int vr;
    ph_stack_frame *phsf = NULL;
    if (current_stack_frame() == NULL)
        phsf = phsf_create_nonphrase_stack_frame();
    failure_this_call = WHENOKAY_PAPF;
    LOGIF(AP_PARSING, "A when clause <$W> is suspected.\n", j+1, w2);
    wts = parse_expression(j+1, w2, CONDITION_EXPCON);
    if (phsf) phsf_remove_nonphrase_stack_frame();
    vr = validate_when(wts);
    LOGIF(AP_PARSING, "When validation result: %d.\n", vr);
    if (vr) w2 = j-1; else {
        wts = NULL;
        failure_this_call = WHEN_PAPF;
    }
    if (w2 < w1) { ap.valid = TRUE; goto Succeeded; }
}
```

This code is used in §6.

§**11.** Parse, and trim away, an "in the presence of..." clause.

⟨PAR - (e) Parse Terminal Presence Clause 11⟩ ≡

```
j = is_word_intermediate_inc(presence_V, w1, w2);
if (j>=0) {
    if ((compare_word(j-2, in_V)) &&
        (compare_word(j-1, the_V)) &&
        (compare_word(j+1, of_V))) {
        itp = parse_action_parameter(j+2, w2, FALSE);
        LOGIF(AP_PARSING, "ITPO $W = $S\n", j+2, w2, itp);
        if (spec_is_UNKNOWN(itp) == FALSE) w2 = j-3; else itp = NULL;
        if (w2 < w1) { ap.valid = TRUE; goto Succeeded; }
    }
}
```

This code is used in §6.

§**12.** Special clauses are allowed after "going..."; trim them away as they are recorded.

⟨PAR - (f) Parse Special Going Clauses 12⟩ ≡ {

```
    action_name_list *preliminary_anl =
        make_action_name_list(w1, w2, tense, 0, 1);
    action_name *chief_an =
        anl_get_single_action(preliminary_anl);
    if (chief_an == NULL) {
        int x;
        chief_an = act_longest_null(w1, w2, tense, &x);
    }
    if (chief_an) {
        int I_am_going = FALSE;
        if (((tense == IS_TENSE) && [[w1, w2 == going ***]]) ||
            ((tense == HASBEEN_TENSE) && [[w1, w2 == gone ***]]))
            I_am_going = TRUE;
        stacked_variable *last_stv_specified = NULL;
        i = w1+1; j = -1;
        LOGIF(AP_PARSING, "Trying special clauses at <$W> = %d, %d\n", i, w2, i, w2);
        while (i < w2) {
            stacked_variable *stv = NULL;
            if (unexpectedly_upper_case(i) == FALSE)
                stv = an_parse_match_clause(chief_an, i, w2);
            if (stv != NULL) {
                LOGIF(AP_PARSING,
                    "Special clauses found on <$W> with %d, %d, %d\n",
                    i, w2, i, j, k);
                if (last_stv_specified == NULL) j = i-1;
                else ap_add_optional_clause(&ap,
                    apoc_new(last_stv_specified, parse_verified_action_parameter(k, i-1, FALSE)));
                k = i+1;
                last_stv_specified = stv;
            }
            i++;
        }
        if (last_stv_specified != NULL)
            ap_add_optional_clause(&ap,
                apoc_new(last_stv_specified, parse_verified_action_parameter(k, w2, FALSE)));
        if (j >= 0) w2 = j;
        if (I_am_going) {
            if [[w1, w2 == ... nowhere --> w1, w2]] ap.nowhere_flag = TRUE;
            else if [[w1, w2 == ... somewhere --> w1, w2]] ap.nowhere_flag = 2;
        }
    }
}
```

This code is used in §6.

§**13.** The initial "in" clause, e.g., "in the Pantry", requires special handling to prevent it from clashing with other interpretations of "in" elsewhere in the grammar.

⟨PAR - (g) Parse Initial In Clause 13⟩ ≡

```
    if ((rule_circumstances_mode) && [[w1, w2 == in ...]]) {
        ap.room_spec = parse_action_parameter(w1+1, w2, FALSE);
        if (validate_parameter(ap.room_spec, kova(OBJECT_TY))) {
            ap.valid = TRUE; goto Succeeded; }
    }
```

This code is used in §6.

§**14.** We now have only the actual action left. First we see if it is a named pattern ("behaving suspiciously", say), rather than a match against an action's ordinary syntax ("taking something"):

⟨PAR - (h) Parse Named Action Pattern 14⟩ ≡

```
{ named_action_pattern *nap;
    int c1 = w1;
    int c2 = w2;
    j = is_the_word_in_intermediate(w1, w2);
    if (j>=0) { c2 = j-1; }
    if ((c1<=c2) && (nap = nap_by_name(c1, c2))) {
        ap.named = nap;
        ap.valid = TRUE;
        ap.when = wts;
        if (j>=0) {
            ap.room_spec = parse_action_parameter(j+1, w2, FALSE);
            if (validate_parameter(ap.room_spec, kova(OBJECT_TY)) == FALSE)
                ap.valid = FALSE;
        }
        if (ap.valid) goto Succeeded;
        goto Failed;
    }
}
```

This code is used in §6.

§**15.** Extract the information as to which actions are intended: e.g., from "taking or dropping something", that it will be taking or dropping.

⟨PAR - (i) Parse Initial Action Name List 15⟩ ≡

```
    anl = make_action_name_list(w1, w2, tense, 0, 1);
    if (anl == NULL) goto Failed;
    LOGIF(AP_PARSING, "ANL from PAR(i):\n$L\n", anl);
```

This code is used in §6.

§**16.** Now to fill in the gaps. At this point we have the action name list as a linked list of all possible lexical matches, but want to whittle it down to remove those which do not semantically make sense. For instance, "taking inventory" has two possible lexical matches: "taking inventory" with 0 parameters, or "taking" with 1 parameter "inventory", and we cannot judge without parsing the expression "inventory". The list passes muster if at least one match succeeds at the first word position represented in the list, which is to say the last one lexically, since the list is reverse-ordered. (This is so that "taking or dropping something" requires only "dropping" to have its objects specified; "taking", of course, does not.) We delete all entries in the list at this crucial word position except for the one matched.

⟨PAR - (j) Parse Parameters 16⟩ ≡

```
{ action_name_list *anl_link = anl;
    int first_word_represented = anl->word_position;
    int no_positions = 0, position_at[100], position_min_parc[100],
        positions_with_min_parc[3];
    while (anl_link != NULL) {
        int i, j=-1;
        for (i=0; i<no_positions; i++)
            if (anl_link->word_position == position_at[i])
                j = i;
        if (j == -1) {
            if (no_positions == 100) goto Failed;
            position_at[no_positions] = anl_link->word_position;
            position_min_parc[no_positions] = 1000000;
            j = no_positions++;
        }
        if (anl_link->parc < position_min_parc[j])
            position_min_parc[j] = anl_link->parc;
        anl_link = anl_link->next;
    }
    for (i=0; i<3; i++) positions_with_min_parc[i] = 0;
    for (i=0; i<no_positions; i++) {
        LOGIF(AP_PARSING, "ANL position %d (word %d): min parc %d\n",
            i, position_at[i], position_min_parc[i]);
        if ((position_min_parc[i] >= 0) && (position_min_parc[i] < 3))
            positions_with_min_parc[position_min_parc[i]]++;
    }
    if ((positions_with_min_parc[1] > 1) ||
        (positions_with_min_parc[2] > 1)) {
        failure_this_call = MIXEDNOUNS_PAPF; goto Failed;
    }
    anl_link = anl;
    while (anl_link != NULL) {
        kind_of_value *check_n, *check_s;
        int pw1, pw2;
        if (anl_link->word_position != first_word_represented)
            goto Failed;
        ap.noun_spec = NULL; ap.second_spec = NULL;
        if (anl_link->parc >= 1) {
            if (anl_link->parameter_w1[0] >= 0)
                put_action_object_into_ap(&ap,
                    1, anl_link->parameter_w1[0], anl_link->parameter_w2[0]);
```

```
        }
        pw1 = anl_link->parameter_w1[1];
        pw2 = anl_link->parameter_w2[1];
        if (anl_link->parc >= 2) {
            if (pw1 >= 0) {
                if ((anl_link->action_listed != NULL)
                    && (is_kova(act_get_data_type_of_second_noun(anl_link->action_listed),
                        UNDERSTANDING_TY))
                    && [[pw1, pw2 == something/anything/it]]) {
                    ap.second_spec = new_actual_CONSTANT_spec(kova(UNDERSTANDING_TY));
                    ap.second_spec->word_ref1 = anl_link->parameter_w1[1];
                    ap.second_spec->word_ref2 = anl_link->parameter_w1[1];
                } else {
                    put_action_object_into_ap(&ap, 2, pw1, pw2);
                }
            }
        }
        if (anl_link->in_w1 >= 0)
            put_action_object_into_ap(&ap, 3, anl_link->in_w1, anl_link->in_w2);
        check_n = kova(OBJECT_TY); check_s = kova(OBJECT_TY);
        if (anl_link->action_listed != NULL) {
            check_n = act_get_data_type_of_noun(anl_link->action_listed);
            check_s = act_get_data_type_of_second_noun(anl_link->action_listed);
        }
        ap.valid = TRUE;
        if (validate_parameter(ap.noun_spec, check_n) == FALSE)
            ap.valid = FALSE;
        if (validate_parameter(ap.second_spec, check_s) == FALSE)
            ap.valid = FALSE;
        if (validate_parameter(ap.room_spec, kova(OBJECT_TY)) == FALSE)
            ap.valid = FALSE;
        if (ap.valid) {
            anl = anl_link;
            while (anl_link) {
                while ((anl_link->next) &&
                    (anl_link->next->word_position == anl_link->word_position))
                    anl_link->next = anl_link->next->next;
                anl_link = anl_link->next;
            }
            goto ParameterListMade;
        }
        anl_link = anl_link->next;
    }
    goto Failed;
    ParameterListMade: ;
}
```

This code is used in §6.

§**17.** Not all actions can cohabit. We require that as far as the user has specified the parameters, the actions in the list must all agree (i) to be allowed to have such a parameter, and (ii) to be allowed to have a parameter of the same type. Thus "waiting or taking something" fails (waiting takes 0 parameters, but we specified one), and so would "painting or taking something" if painting had to be followed by a colour, say. Note that the "doing anything" action is always allowed a parameter (this is the case when the first action name in the list is `NULL`).

⟨PAR - (k) Verify Mixed Action 17⟩ ≡

```
{ action_name_list *anl_link = anl;
    int nnz = 0;
    if (anl->action_listed == NULL) {
        if (anl->parc >= 2) goto Failed;
        goto Succeeded;
    }
    while (anl_link != NULL) {
        if ((anl_link->action_listed != NULL) &&
            (act_get_max_parameters(anl_link->action_listed) < anl->parc)) {
            failure_this_call = IMMISCIBLE_PAPF;
            goto Failed;
        }
        if (anl_link->parc != 0) {
            nnz++;
            if (nnz > 1) {
                failure_this_call = IMMISCIBLE_PAPF;
                goto Failed;
            }
        }
        if (anl->parc >= 1) {
            if (anl_link->action_listed == NULL) {
                failure_this_call = IMMISCIBLE_PAPF;
                goto Failed;
            }
            if (kov_compare(act_get_data_type_of_noun(anl->action_listed),
                act_get_data_type_of_noun(anl_link->action_listed)) == FALSE) {
                failure_this_call = IMMISCIBLE_PAPF;
                goto Failed;
            }
        }
        if (anl->parc >= 2) {
            if (kov_compare(act_get_data_type_of_second_noun(anl->action_listed),
                act_get_data_type_of_second_noun(anl_link->action_listed)) == FALSE) {
                failure_this_call = IMMISCIBLE_PAPF;
                goto Failed;
            }
        }
        anl_link = anl_link->next;
    }
    goto Succeeded;
}
```

This code is used in §6.

§**18.**

```
specification *parse_verified_action_parameter(int w1, int w2, int allow_adjlist) {
    specification *spec = parse_action_parameter(w1, w2, allow_adjlist);
    if (spec_is_UNKNOWN(spec)) {
        quote_source(1, current_sentence);
        quote_words(2, w1, w2);
        handmade_problem(_P_(C11BadOptionalAPClause));
        issue_problem_segment(
            "In %1, I tried to read a description of an action - a complicated "
            "one involving optional clauses; but '%2' wasn't something I "
            "recognised.");
        issue_problem_end();
    }
    return spec;
}
```

§**19. Action pattern specificity.** The following is one of NI's standardised comparison routines, which takes a pair of objects A, B and returns 1 if A makes a more specific description than B, 0 if they seem equally specific, or −1 if B makes a more specific description than A. This is transitive, and intended to be used in sorting algorithms.

```
int ap_count_rooms(action_pattern *ap) {
    int c = 0;
    if (ap->room_spec) c += 2;
    if (ap->from_spec) c += 2;
    if (ap->to_spec) c += 2;
if (ap-¿nowhere_flag)c+ = 1;
    return c;
}

int ap_count_going(action_pattern *ap) {
    int c = 0;
    if (ap->pushing_spec) c += 2;
    if (ap->by_spec) c += 2;
    if (ap->through_spec) c += 2;
    return c;
}

int ap_count_aspects(action_pattern *ap) {
    int c = 0;
    if (ap == NULL) return 0;
    if ((ap->pushing_spec) ||
        (ap->by_spec) ||
        (ap->through_spec))
        c++;
    if ((ap->room_spec) ||
        (ap->from_spec) ||
        (ap->to_spec))
        c++;
    if ((ap->nowhere_flag) ||
        (ap->noun_spec) ||
```

```
        (ap->second_spec) ||
        (ap->actor_spec))
        c++;
    if (ap->presence_spec) c++;
    if ((tp_is_valid(&(ap->duration))) || (ap->when))
        c++;
    if (ap->parameter_spec) c++;
    return c;
}

int compare_specificity_of_ap(action_pattern *ap1, action_pattern *ap2) {
    int rv, suspend_usual_from_and_room = FALSE, rct1, rct2;

    if ((ap1 == NULL) && (ap2)) return -1;
    if ((ap1) && (ap2 == NULL)) return 1;
    if ((ap1 == NULL) && (ap2 == NULL)) return 0;

    LOGIF(SPECIFICITIES,
        "Comparing specificity of action patterns:\n(1) $A(2) $A\n", ap1, ap2);

    if ((ap1->valid == FALSE) && (ap2->valid != FALSE)) return -1;
    if ((ap1->valid != FALSE) && (ap2->valid == FALSE)) return 1;

    c_s_stage_law = "III.1 - Object To Which Rule Applies";

    rv = compare_specificity_of_spec(ap1->parameter_spec, ap2->parameter_spec);
    if (rv != 0) return rv;

    c_s_stage_law = "III.2.1 - Action/Where/Going In Exotic Ways";

    rct1 = ap_count_going(ap1); rct2 = ap_count_going(ap2);
    if (rct1 > rct2) return 1;
    if (rct1 < rct2) return -1;

    rv = compare_specificity_of_spec(ap1->pushing_spec, ap2->pushing_spec);
    if (rv != 0) return rv;

    rv = compare_specificity_of_spec(ap1->by_spec, ap2->by_spec);
    if (rv != 0) return rv;

    rv = compare_specificity_of_spec(ap1->through_spec, ap2->through_spec);
    if (rv != 0) return rv;

    c_s_stage_law = "III.2.2 - Action/Where/Room Where Action Takes Place";

    rct1 = ap_count_rooms(ap1); rct2 = ap_count_rooms(ap2);
    if (rct1 > rct2) return 1;
    if (rct1 < rct2) return -1;

    if ((ap1->from_spec) && (ap1->room_spec == NULL)
        && (ap2->room_spec) && (ap2->from_spec == NULL)) {
        rv = compare_specificity_of_spec(ap1->from_spec, ap2->room_spec);
        if (rv != 0) return rv;
        suspend_usual_from_and_room = TRUE;
    }

    if ((ap2->from_spec) && (ap2->room_spec == NULL)
        && (ap1->room_spec) && (ap1->from_spec == NULL)) {
        rv = compare_specificity_of_spec(ap1->room_spec, ap2->from_spec);
        if (rv != 0) return rv;
        suspend_usual_from_and_room = TRUE;
```

```
    }
    if (suspend_usual_from_and_room == FALSE) {
        rv = compare_specificity_of_spec(ap1->from_spec, ap2->from_spec);
        if (rv != 0) return rv;
    }
    if (suspend_usual_from_and_room == FALSE) {
        rv = compare_specificity_of_spec(ap1->room_spec, ap2->room_spec);
        if (rv != 0) return rv;
    }
    rv = compare_specificity_of_spec(ap1->to_spec, ap2->to_spec);
    if (rv != 0) return rv;
    c_s_stage_law = "III.2.3 - Action/Where/In The Presence Of";
    rv = compare_specificity_of_spec(ap1->presence_spec, ap2->presence_spec);
    if (rv != 0) return rv;
    c_s_stage_law = "III.2.4 - Action/Where/Other Optional Clauses";
    rv = compare_specificity_of_apoc_list(ap1, ap2);
    if (rv != 0) return rv;
    c_s_stage_law = "III.3.1 - Action/What/Second Thing Acted On";
    rv = compare_specificity_of_spec(ap1->second_spec, ap2->second_spec);
    if (rv != 0) return rv;
    c_s_stage_law = "III.3.2 - Action/What/Thing Acted On";
    rv = compare_specificity_of_spec(ap1->noun_spec, ap2->noun_spec);
    if (rv != 0) return rv;
    if ((ap1->nowhere_flag) && (ap2->nowhere_flag == FALSE)) return -1;
    if ((ap1->nowhere_flag == FALSE) && (ap2->nowhere_flag)) return 1;
    c_s_stage_law = "III.3.3 - Action/What/Actor Performing Action";
    rv = compare_specificity_of_spec(ap1->actor_spec, ap2->actor_spec);
    if (rv != 0) return rv;
    c_s_stage_law = "III.4.1 - Action/How/What Happens";
    rv = compare_specificity_of_anl(ap1->action, ap2->action);
    if (rv != 0) return rv;
    c_s_stage_law = "III.5.1 - Action/When/Duration";
    rv = compare_specificity_of_tp(&(ap1->duration), &(ap2->duration));
    if (rv != 0) return rv;
    c_s_stage_law = "III.5.2 - Action/When/Circumstances";
    rv = compare_specificity_of_CONDITIONs(ap1->when, ap2->when);
    if (rv != 0) return rv;
    c_s_stage_law = "III.6.1 - Action/Name/Is This Named";
    if ((ap1->named != NULL) && (ap2->named == NULL))
        return 1;
    if ((ap1->named == NULL) && (ap2->named != NULL))
        return -1;
    return 0;
}
```

The function ap_count_aspects is called from 12/phrcd.
The function compare_specificity_of_ap is called from 12/phrcd.

§**20.** And an anticlimactic little routine for putting objects into action patterns in the noun or second noun position.

```
void put_action_object_into_ap(action_pattern *ap, int pos,
    int w1, int w2) {
    specification *spec = parse_expression(w1, w2, TYPE_OR_VALUE_EXPCON);
    if ((spec_is_CONSTANT_of_kova(spec, TEXT_TY)) ||
        (spec_is_CONSTANT_of_kova(spec, TEXT_ROUTINE_TY)))
        spec_set_kind_of_value(spec, kova(UNDERSTANDING_TY));
    spec->word_ref1 = w1; spec->word_ref2 = w2;
    LOGIF(AP_PARSING, "PAOIA (position %d) $W = $S\n", pos, w1, w2, spec);
    switch(pos) {
        case 1: ap->noun_spec = spec; break;
        case 2: ap->second_spec = spec; break;
        case 3: ap->room_spec = spec; break;
    }
}
```

§**21. Compiling action patterns.** In the following routines, we compile a single clause in what may be a complex condition which determines whether a rule should fire. The flag `f` indicates whether any condition has already been printed, and is updated as the return value of the routine. (Thus, it's permissible for the routines to compile nothing and return `f` unchanged.) The simple case first:

```
int CAP_insert_clause(int f, OUTPUT_STREAM, char *i6_condition) {
    if (f) WRITE(" && ");
    WRITE("(%s)", i6_condition);
    return TRUE;
}
```

§**22.** The more complex clauses mostly act on a single I6 global variable. In almost all cases, this falls through to the standard method for testing a condition: we force it to propositional form, substituting the global in for the value of free variable 0. However, rule clauses are allowed a few syntaxes not permitted to ordinary conditions, and these are handled as exceptional cases first:

(a) A table reference such as "a Queen listed in the Table of Monarchs" expands.

(b) Writing "from R", where R is a region, tests if the room being gone from is in R, not if it is equal to R. Similarly for other room-related clauses such as "through" and "in".

(c) Given a piece of run-time parser grammar, we compile a test against the standard I6 topic variables: there are two of these, so this is the exceptional case where the clause doesn't act on a single I6 global, and in this case we therefore ignore `I6_global_name`.

```
int CAP_match_clause(int f, OUTPUT_STREAM, quantity *I6_global_quantity,
    specification *spec, int verify_as_description_of_object) {
    specification *I6_var_TS;
    int force_proposition = FALSE;
    if (spec == NULL) return f;
    LOGIF(COMPILE_ACTION_PATTERNS, "[MPE on $Q: $S]\n", I6_global_quantity, spec);
    I6_var_TS = new_QUANTITY_spec(I6_global_quantity);
```

```
if (f) WRITE(" && ");
int c1, c2;
spec_get_calling(spec, &c1, &c2);
if (c1 >= 0) {
    int ln = ensure_called_local(c1, c2, spec_get_described_kov(spec));
    WRITE("(t_%d = %s, (", ln, qty_get_I6_representation(I6_global_quantity));
}
force_proposition = TRUE;
if (spec_is_UNKNOWN(spec)) {
    if (problem_count == 0) internal_error("MPE found unknown SP");
    force_proposition = FALSE;
}
else if (family_is(spec, STORAGE_FMY)) {
    force_proposition = TRUE;
    if (species_is(spec, TABLE_ENTRY_SPC)) {
        if (spec_get_argc(spec) != 2) internal_error("MPE with bad no of args");
        we_need_ct_locals();
        WRITE("(ct_1=ExistsTableRowCorr(ct_0=");
        spec_compile(OUT, spec_get_argument(spec, 1));
        WRITE(",");
        spec_compile(OUT, spec_get_argument(spec, 0));
        WRITE(",");
        spec_compile(OUT, I6_var_TS);
        WRITE("))");
        force_proposition = FALSE;
    }
}
else if (family_is(spec, VALUE_FMY)) {
    if (spec_is_generic_CONSTANT(spec)) {
        WRITE("(true)");
        force_proposition = FALSE;
    }
    if (spec_is_CONSTANT_of_kova(spec, UNDERSTANDING_TY)) {
        if [[spec == something/anything]]
            WRITE("(true)");
        else {
            WRITE("(");
            spec_compile(OUT, spec);
            WRITE("(consult_from, consult_words)~=GPR_FAIL)");
        }
        force_proposition = FALSE;
    }
    if ((I6_global_quantity != parameter_object_quantity) &&
        (spec_is_CONSTANT_of_kova(spec, OBJECT_TY))) {
        world_object *wo = spec_object_exactly_described_if_any(spec);
        if ((wo) && (wo_of_kind(wo, kind_region))) {
            WRITE("(TestRegionalContainment(");
            spec_compile(OUT, I6_var_TS);
            WRITE(",");
            spec_compile(OUT, spec);
            WRITE("))");
            force_proposition = FALSE;
        }
```

```
        }
    }
    else if (species_is(spec, DESCRIPTION_SPC)) {
        if ((I6_global_quantity != parameter_object_quantity) &&
            ((spec_get_described_object(spec)) &&
            (wo_of_kind(spec_get_described_object(spec), kind_region)))) {
            WRITE("(TestRegionalContainment(");
            spec_compile(OUT, I6_var_TS);
            WRITE(",");
            spec_compile(OUT, spec);
            WRITE("))");
            force_proposition = FALSE;
        } else {
            spec_convert_docket_to_proposition(spec);
            force_proposition = FALSE;
        }
    }
    pcalc_prop *prop = spec_get_proposition(spec);
    if (family_is(spec, STORAGE_FMY))
        LOGIF(COMPILE_ACTION_PATTERNS, "Storage has $D\n", prop);
    if ((force_proposition) && (prop == NULL)) {
        prop = prop_from_spec(spec, FALSE);
        LOGIF(COMPILE_ACTION_PATTERNS, "[MPE forced proposition: $D]\n", prop);
        if (prop == NULL) internal_error("MPE unable to force proposition");
        if (verify_as_description_of_object)
            prop_verify_descriptive(prop,
                "an action or activity to apply to things matching a given "
                "description", spec);
    }
    if (prop) {
        LOGIF(COMPILE_ACTION_PATTERNS, "[MPE faces proposition: $D]\n", prop);
        prop_type_check(prop, tc_no_problem_reporting());
        compile_test_of_proposition(OUT, I6_var_TS, prop);
    }
    if (c1 >= 0) WRITE("))");
    return TRUE;
}
void compile_action_pattern_to_block(OUTPUT_STREAM, action_pattern *ap) {
    action_name *an = anl_get_singleton_action(ap->action);
    WRITE("STORED_ACTION_TY_New(");
    WRITE("##%s, ", an_get_I6_representation(an));
    if (ap->noun_spec)
        spec_compile(OUT, ap->noun_spec);
    else
        WRITE("0");
    WRITE(", ");
    if (ap->second_spec)
        spec_compile(OUT, ap->second_spec);
    else
        WRITE("0");
    WRITE(", ");
```

```
    if (ap->actor_spec)
        spec_compile(OUT, ap->actor_spec);
    else
        WRITE("player");
    WRITE(", %d", ap->request);
    WRITE(")");
}
void compile_action_pattern_match(OUTPUT_STREAM, action_pattern ap, int naming_mode) {
    int f = FALSE;
    kind_of_value *kov_of_noun = kova(OBJECT_TY);
    ap_optional_clause *apoc;
    LOGIF(COMPILE_ACTION_PATTERNS, "Compiling action pattern:\n  $A", &ap);
    if (tp_is_valid(&(ap.duration))) {
        compile_past_action_pattern(OUT, ap.duration, ap);
        goto AP_Compiled;
    }
    if (ap.AP_parity == -1) WRITE("(~~(");
    if (ap.named != NULL) {
        if (f) WRITE(" && ");
        WRITE("(%s())", nap_get_I6_representation(ap.named));
        f = TRUE;
    }
    if ((ap.action != NULL) && (ap.test_anl)) {
        if (f) WRITE(" && ");
        compile_action_name_list(OUT, ap.action);
        f = TRUE;
    }
    if (naming_mode == FALSE) {
        if (ap.applies_to_any_actor == FALSE) {
            if (ap.actor_spec != NULL) {
                if (f) WRITE(" && ");
                WRITE(" (actor~=player)");
                if (ap.request) WRITE(" && (act_requester)");
                else WRITE(" && (act_requester==nothing)");
                f = TRUE;
                f = CAP_match_clause(f, OUT, I6_actor_quantity, ap.actor_spec, TRUE);
            } else {
                if (f) WRITE(" && ");
                WRITE(" (actor==player)");
                f = TRUE;
            }
        } else {
            if (f) WRITE(" && ");
            if (ap.request) WRITE("(act_requester)");
            else WRITE("(act_requester==nothing)");
            f = TRUE;
        }
    }
    if ((ap.action == NULL) && (ap.noun_spec)) {
        if (f) WRITE(" && ");
        WRITE(" (noun) && (noun == inp1)");
        f = TRUE;
```

```
}
if ((ap.action == NULL) && (ap.second_spec)) {
    if (f) WRITE(" && ");
    WRITE(" (second) && (second == inp2)");
    f = TRUE;
}
if (ap.action) {
    kov_of_noun = act_get_data_type_of_noun(ap.action->action_listed);
    if (kov_of_noun == NULL) kov_of_noun = kova(OBJECT_TY);
}
f = CAP_match_clause(f, OUT, I6_noun_quantity, ap.noun_spec,
    (is_kova(kov_of_noun, OBJECT_TY))?TRUE:FALSE);
if (ap.action) {
    kov_of_noun = act_get_data_type_of_second_noun(ap.action->action_listed);
    if (kov_of_noun == NULL) kov_of_noun = kova(OBJECT_TY);
}
f = CAP_match_clause(f, OUT, I6_second_quantity, ap.second_spec,
    (is_kova(kov_of_noun, OBJECT_TY))?TRUE:FALSE);
if (ap.room_spec) {
    if ((ap.applies_to_any_actor == FALSE) && (naming_mode == FALSE) &&
        (ap.actor_spec == NULL))
        f = CAP_match_clause(f, OUT, real_location_quantity, ap.room_spec, TRUE);
    else {
        if (f) WRITE(" && ");
        WRITE(" (actor_location = LocationOf(actor))");
        f = TRUE;
        f = CAP_match_clause(f, OUT, actor_location_quantity, ap.room_spec, TRUE);
    }
}
f = CAP_match_clause(f, OUT, parameter_object_quantity,
    ap.parameter_spec, TRUE);

apoc = ap.optional_clauses;
while (apoc) {
    char lvalue_text[64];
    stv_compile_rvalue_to_text(lvalue_text, apoc->stv_to_match);
    f = CAP_match_clause(f, OUT, qty_temporary_quantity(lvalue_text),
        apoc->clause_spec, TRUE);
    apoc = apoc->next;
}

if (ap.nowhere_flag) {
    if (ap.nowhere_flag == 1)
        f = CAP_insert_clause(f, OUT, "(MStack-->MstVON(20007,1)) == nothing");
    else {
        specification *somewhere = new_DESCRIPTION_spec();
        spec_set_described_kind(somewhere, kind_room);
        f = CAP_match_clause(f, OUT,
            qty_temporary_quantity("(MStack-->MstVON(20007,1))"), somewhere, TRUE);
    }
} else {
    if ((ap.to_spec == NULL) &&
        ((ap.from_spec != NULL)||(ap.by_spec != NULL)||
        (ap.through_spec != NULL)||(ap.pushing_spec != NULL)))
        f = CAP_insert_clause(f, OUT, "(MStack-->MstVON(20007,1)) ~= nothing");
```

```
    }
    if (ap.presence_spec != NULL) {
        world_object *to_be_present =
            spec_object_exactly_described_if_any(ap.presence_spec);
        if (f) WRITE(" && ");
        if (to_be_present) {
            WRITE("(");
            CAP_match_clause(FALSE, OUT,
                qty_temporary_quantity(wo_get_I6_representation(to_be_present)),
                ap.presence_spec, TRUE);
            WRITE("&& (TestScope(%s, actor)))",
                wo_get_I6_representation(to_be_present));
        } else {
            int ln, pc1, pc2;
            loop_over_scope *los = los_new(ap.presence_spec);
            spec_get_calling(ap.presence_spec, &pc1, &pc2);
            if (pc1 >= 0) {
                ln = ensure_called_local(pc1, pc2, spec_get_described_kov(ap.presence_spec));
                WRITE("(los_rv=false, LoopOverScope(LOS_%d, actor), t_%d=los_rv)",
                los->allocation_id, ln);
            } else {
                WRITE("(los_rv=false, LoopOverScope(LOS_%d, actor), los_rv)",
                los->allocation_id);
            }
        }
        f = TRUE;
    }
    if (ap.when != NULL) {
        if (f) WRITE(" && ");
        spec_compile(OUT, ap.when);
        f = TRUE;
    }
    if (f == FALSE) WRITE("TRUE");
    if (ap.AP_parity == -1) WRITE("))");
    AP_Compiled: ;
}

void ap_convert_to_present_tense(action_pattern *ap) {
    tp_make_invalid(&(ap->duration));
}

int pta_acceptable(specification *spec) {
    world_object *wo;
    if (spec == NULL) return TRUE;
    if (species_is(spec, DESCRIPTION_SPC) == FALSE) return TRUE;
    wo = spec_object_exactly_described_if_any(spec);
    if ((wo) && (wo->kind_flag == FALSE)) return TRUE;
    return FALSE;
}

int ap_makes_callings(action_pattern *ap) {
    if (spec_makes_callings(ap->noun_spec)) return TRUE;
    if (spec_makes_callings(ap->second_spec)) return TRUE;
    if (spec_makes_callings(ap->actor_spec)) return TRUE;
    if (spec_makes_callings(ap->room_spec)) return TRUE;
```

```
    if (spec_makes_callings(ap->parameter_spec)) return TRUE;
    if (spec_makes_callings(ap->presence_spec)) return TRUE;
    return FALSE;
}
void compile_past_tense_action(OUTPUT_STREAM, action_pattern *ap) {
    int bad_form = FALSE;
    WRITE("(");
    if (ap->AP_parity == -1) WRITE("~~(");
    WRITE("TestActionBitmap(");
    if (ap->noun_spec == NULL)
        WRITE("nothing");
    else
        spec_compile(OUT, ap->noun_spec);
    WRITE(",");
    if (ap->action == NULL)
        WRITE("-1");
    else {
        if (ap->action->next) bad_form = TRUE;
        if (act_can_be_compiled_in_past_tense(ap->action->action_listed) == FALSE)
            bad_form = TRUE;
        WRITE("##%s", an_get_I6_representation(ap->action->action_listed));
    }
    if (pta_acceptable(ap->noun_spec) == FALSE) bad_form = TRUE;
    if (pta_acceptable(ap->second_spec) == FALSE) bad_form = TRUE;
    if (pta_acceptable(ap->actor_spec) == FALSE) bad_form = TRUE;
    if (ap->room_spec) bad_form = TRUE;
    if (ap->parameter_spec) bad_form = TRUE;
    if (ap->presence_spec) bad_form = TRUE;
    if (ap->when) bad_form = TRUE;
    if (ap->optional_clauses) bad_form = TRUE;
    WRITE(")");
    if (ap->AP_parity == -1) WRITE(")");
    WRITE(")");
    LOG("Past tense action:\n$L\n", ap->action);
    if (bad_form)
        sentence_problem(_P_(C11PTAPTooComplex),
            "that is too complex a past tense action",
            "at least for this version of Inform to handle: we may improve "
            "matters in later releases. The restriction is that the "
            "actions used in the past tense may take at most one "
            "object, and that this must be a physical thing (not a "
            "value, in other words). And no details of where or what "
            "else was then happening can be specified.");
}
```

The function CAP_match_clause is called from 11/los.

The function compile_action_pattern_to_block is called from 7/vasp.

The function compile_action_pattern_match is called from 7/vasp, 7/cosp, 11/chron, 11/nap and 12/phrcd.

The function ap_convert_to_present_tense is called from 11/chron.

The function ap_makes_callings is called from 5/candp and 11/chron.

The function compile_past_tense_action is called from 7/cosp.

Looping Over Scope 11/los

Purpose

To compile routines capable of being passed as arguments to the I6 library routine for looping over parser scope at run-time, and so to provide an implementation for conditions such as "in the presence of Mrs Dalloway".

Template interpreter commands

```
0   {-callv:compile_los_routines}
```

Definitions

```
typedef struct loop_over_scope {
    struct specification *what_to_find;
    MEMORY_MANAGEMENT
} loop_over_scope;
```

The structure loop_over_scope is private to this section.

```
loop_over_scope *los_new(specification *what) {
    loop_over_scope *los = CREATE(loop_over_scope);
    los->what_to_find = spec_copy_and_copy_docket(what);
    spec_clear_calling(los->what_to_find);
    return los;
}
void compile_los_routines(OUTPUT_STREAM) {
    loop_over_scope *los;
    LOOP_OVER(los, loop_over_scope) {
        ph_stack_frame *phsf = phsf_create_nonphrase_stack_frame();
        phsf_add_it_variable(phsf, -1, -1, kova(OBJECT_TY));
        OUT = begin_compiling_phrase(OUT);
        begin_code_blocks();
        INDENT;
        WRITE("if (");
        CAP_match_clause(FALSE, OUT, qty_temporary_quantity("t_0"),
            los->what_to_find, TRUE);
        WRITE(") los_rv = t_0;\n");
        OUTDENT; WRITE("];\n");
        OUT = write_routine_header();
        WRITE("[ LOS_%d ", los->allocation_id);
        copy_compiled_phrase();
        end_code_blocks();
        phsf_remove_nonphrase_stack_frame();
    }
}
```

The function los_new is called from 11/ap.

The function compile_los_routines is invoked by a command in a `.i6t` template file.

Named Action Patterns 11/nap

Purpose

A named action pattern is a named categorisation of actions, such as "acting suspiciously", which is a disjunction of action patterns (stored in a linked list).

Template interpreter commands

```
0   {-callv:compile_named_action_patterns}
```

Definitions

```
typedef struct named_action_pattern {
    struct action_pattern *first;                    list of APs defining this NAP
    int word_ref1, word_ref2;                        text of declaration
    char nap_I6_identifier[32];                      for an I6 routine to test this NAP
    MEMORY_MANAGEMENT
} named_action_pattern;
```

The structure named_action_pattern is private to this section.

```
named_action_pattern *nap_new(int w1, int w2) {
    named_action_pattern *nap = CREATE(named_action_pattern);
    nap->first = NULL;
    nap->word_ref1 = w1;
    nap->word_ref2 = w2;
    register_excerpt_meaning(NAMED_AP_MC, 0, w1, w2,
        STORE_POINTER_named_action_pattern(nap));
    sprintf(nap->nap_I6_identifier, "NAP_%d", nap->allocation_id);
    return nap;
}

named_action_pattern *nap_by_name(int w1, int w2) {
    meaning_list *ml = SP_excerpt(NAMED_AP_MC, w1, w2);
    if (ml) return RETRIEVE_POINTER_named_action_pattern(em_data(ml_meaning(ml)));
    return NULL;
}

char *nap_get_I6_representation(named_action_pattern *nap) {
    return nap->nap_I6_identifier;
}

void nap_add_ap(action_pattern *app, int w1, int w2) {
    named_action_pattern *nap;

    nap = nap_by_name(w1, w2);
    if (nap) {
        action_pattern *list;
        list = nap->first; while (list->next) list = list->next;
        list->next = app;
        return;
    }

    nap = nap_new(w1, w2);
    nap->first = app;
```

```
}
int nap_within_action_context(named_action_pattern *nap, action_name *an) {
    action_pattern *ap;
    for (ap = nap->first; ap; ap = ap->next)
        if (ap_within_action_context(ap, an)) return TRUE;
    return FALSE;
}
void nap_index_all(void) {
    named_action_pattern *nap;
    action_pattern *ap;
    int num_naps = NUMBER_CREATED(named_action_pattern);
    index_anchor("NAP");
    INDEX("<p><b>Named kinds of action</b><p>");
    if (num_naps == 0) {
        INDEX("<i>No names have yet been defined. (Here's how: ");
        index_doc_link("NAP");
        INDEX(")</i>");
    }
    LOOP_OVER(nap, named_action_pattern) {
        INDEX("<b>");
        print_raw_text_to_file(nap->word_ref1, nap->word_ref2, ifl);
        INDEX("</b>");
        index_link(lw_array[nap->word_ref1].lw_source);
        ap = nap->first;
        INDEX("<br>  <i>defined as any of the following acts:</i>\n");
        while (ap != NULL) {
            INDEX("<br>\n");
            INDEX("    ");
            print_raw_text_to_file(ap->word_ref1, ap->word_ref2, ifl);
            index_link(lw_array[ap->word_ref1].lw_source);
            ap = ap->next;
        }
        INDEX("<p>\n");
    }
    INDEX("<hr><p>\n");
}
void compile_named_action_patterns(OUTPUT_STREAM) {
    named_action_pattern *nap;
    action_pattern *ap;
    LOOP_OVER(nap, named_action_pattern) {
        WRITE("[ NAP_%d;\n", nap->allocation_id); INDENT;
        ap = nap->first;
        while (ap != NULL) {
            WRITE("if (");
            compile_action_pattern_match(OUT, *ap, TRUE);
            WRITE(") rtrue;\n");
            ap = ap->next;
        }
        WRITE("rfalse;\n");
        OUTDENT; WRITE("];\n");
    }
}
```

The function nap_by_name is called from 11/ap and 12/phud.
The function nap_get_I6_representation is called from 11/ap.
The function nap_add_ap is called from 11/ap.
The function nap_within_action_context is called from 11/ap.
The function nap_index_all is called from 11/ina.
The function compile_named_action_patterns is invoked by a command in a `.i6t` template file.

Actions Index 11/ina

Purpose

To construct the Actions index page.

Template interpreter commands

```
4   {-callv:index_page_Actions}
```

Definitions

¶1. The following modest structure is used for the indexing of command verbs, and is too deeply boring to comment upon. These are the headwords of commands which can be typed at run-time, like QUIT or INVENTORY. For indexing purposes, we divide these headwords into five "natures":

```
define NORMAL_COMMAND 1
define ALIAS_COMMAND 2
define OUT_OF_WORLD_COMMAND 3
define TESTING_COMMAND 4
define BARE_DIRECTION_COMMAND 5

typedef struct command_index_entry {
    int nature;                                              one of the above values
    char *command_headword;                  text of command headword, such as "REMOVE"
    struct grammar_verb *gv_indexed;                                    ...leading to...
    struct command_index_entry *next_alphabetically;                    next in linked list
    MEMORY_MANAGEMENT
} command_index_entry;

command_index_entry *sorted_command_index = NULL;                  in alphabetical order of text
```

The structure command_index_entry is private to this section.

```
void index_meta_verb(char *t) {
    command_index_entry *vie;
    vie = CREATE(command_index_entry);
    vie->command_headword = t;
    vie->nature = OUT_OF_WORLD_COMMAND;
    vie->gv_indexed = NULL;
    vie->next_alphabetically = NULL;
}

void index_test_verb(char *t) {
    command_index_entry *vie;
    vie = CREATE(command_index_entry);
    vie->command_headword = t;
    vie->nature = TESTING_COMMAND;
    vie->gv_indexed = NULL;
    vie->next_alphabetically = NULL;
}

void index_verb_definition(char *p, char *trueverb) {
    int i = 1;
    if ((p[0] == 0) || (p[1] == 0)) return;
    if (trueverb) {
```

```
        if (strcmp(trueverb, "0") != 0) {
            INDEX("%s", trueverb);
            for (i=1; p[i+1]; i++) if (p[i] == ' ') break;
            for (; p[i+1]; i++) if (p[i] != ' ') break;
            if (p[i+1]) INDEX(" ");
        }
    }
    for (; p[i+1]; i++) {
        char c = p[i];
        switch(c) {
            case '"': INDEX("&quot;"); break;
            default: INDEX("%c", c); break;
        }
    }
}
command_index_entry *vie_new_from(char *headword, grammar_verb *gv, int nature) {
    command_index_entry *vie;
    vie = CREATE(command_index_entry);
    vie->command_headword = headword;
    vie->nature = nature;
    vie->gv_indexed = gv;
    vie->next_alphabetically = NULL;
    return vie;
}
void index_alphabetical_grammar(void) {
    command_index_entry *vie, *vie2, *last_vie2, *list_start = NULL;
    grammar_verb *gv;
    char head_letter;
    LOOP_OVER(gv, grammar_verb)
        gv_make_command_index_entries(gv);
    vie = CREATE(command_index_entry);
    vie->command_headword = "0";
    vie->nature = BARE_DIRECTION_COMMAND;
    vie->gv_indexed = NULL;
    vie->next_alphabetically = NULL;
    LOOP_OVER(vie, command_index_entry) {
        if (list_start == NULL) { list_start = vie; continue; }
        vie2 = list_start;
        last_vie2 = NULL;
        while (vie2 && (strcmp(vie->command_headword, vie2->command_headword) > 0)) {
            last_vie2 = vie2;
            vie2 = vie2->next_alphabetically;
        }
        if (last_vie2 == NULL) {
            vie->next_alphabetically = list_start; list_start = vie;
        } else {
            last_vie2->next_alphabetically = vie; vie->next_alphabetically = vie2;
        }
    }
    for (vie = list_start, head_letter = 0; vie; vie = vie->next_alphabetically) {
        grammar_verb *gv;
        if (vie->command_headword[0] != head_letter) {
```

```
            if (head_letter) INDEX("<br>");
            head_letter = vie->command_headword[0];
        }
        gv = vie->gv_indexed;
        switch (vie->nature) {
            case NORMAL_COMMAND:
                gv_index_normal(gv, vie->command_headword);
                break;
            case ALIAS_COMMAND:
                gv_index_alias(gv, vie->command_headword);
                break;
            case OUT_OF_WORLD_COMMAND:
                INDEX("<font color=\"#800000\">");
                INDEX("&quot;%s&quot;, <i>a command for controlling play</i></font><br>",
                    vie->command_headword);
                break;
            case TESTING_COMMAND:
                INDEX("<font color=\"#800000\">");
                INDEX("&quot;%s&quot;, <i>a testing command not available "
                    "in the final game</i></font><br>",
                    vie->command_headword);
                break;
            case BARE_DIRECTION_COMMAND:
                INDEX("&quot;[direction]&quot; - <i>Going</i><br>");
                break;
        }
    }
}
void index_alphabetical_actions(void) {
    int nr = NUMBER_CREATED(action_name);
    action_name **sorted = I7_calloc(nr, sizeof(action_name *), INDEX_SORTING_MREASON);
    if (sorted) {
        ⟨Sort the action names 2⟩;
        ⟨Tabulate the action names 1⟩;
        I7_free(sorted, INDEX_SORTING_MREASON);
    }
}
```

The function index_test_verb is called from 13/gv.
The function index_verb_definition is called from 13/gl.
The function vie_new_from is called from 13/gv.

§**1.**

⟨Tabulate the action names 1⟩ ≡

```
    begin_html_table(ifl, NULL, TRUE, 0, 0, 0, 0, 0);
    first_html_column(ifl, 0);
    int i;
    action_name *an;
    for (i=0; i<nr; i++) {
        if (i == nr/2) next_html_column(ifl, 0);
        an = sorted[i];
        open_html_paragraph(ifl, 1, "tight");
        print_text_to_file(an->word_ref1, an->word_ref2, ifl);
        index_detail_link("A", an->allocation_id, TRUE);
```

```
        INDEX("</p>");
    }
    end_html_row(ifl);
    end_html_table(ifl);
```

This code is used in §0.

§**2.** As usual, we sort with the C library's `qsort`.

⟨Sort the action names 2⟩ ≡

```
    int i = 0;
    action_name *an;
    LOOP_OVER(an, action_name) sorted[i++] = an;
    qsort(sorted, nr, sizeof(action_name *), compare_action_names);
```

This code is used in §0.

§**3.** The following means the table is sorted in alphabetical order of action name.

```
int compare_action_names(const void *ent1, const void *ent2) {
    const action_name *an1 = *((const action_name **) ent1);
    const action_name *an2 = *((const action_name **) ent2);
    return rangecmp(an1->word_ref1, an1->word_ref2, an2->word_ref1, an2->word_ref2);
}
```

§**4.**

```
void index_page_Actions(void) {
    int f = FALSE, par_count = 0;
    action_name *an;
    heading *current_area = NULL;
    extension_file *ext = NULL;
    LOOP_OVER(an, action_name) {
        int new_par = FALSE;
        f = act_index(an, 1, &ext, &current_area, f, &new_par, FALSE, FALSE);
        if (new_par) par_count++;
        an->an_index_group = par_count;
    }
    INDEX("<p><hr>");
    nap_index_all();
    index_anchor("COMMANDS");
    INDEX("<p><b>Commands available to the player</b><p>");
    index_alphabetical_grammar();
    INDEX("<p><hr>");
    index_anchor("LEXICON");
    INDEX("<p><b>Actions in alphabetical order</b><p>\n");
    index_alphabetical_actions();
    index_anchor("ARULES");
    index_action_rulebooks();
    LOOP_OVER(an, action_name) {
        open_index_file("A.html", "<Actions", an->allocation_id,
            "A single action in detail.|About the action rulebooks<ARSUMMARY>");
        f = FALSE;
```

```
        int new_par = FALSE;
        action_name *an2;
        current_area = NULL;
        ext = NULL;
        LOOP_OVER(an2, action_name) {
            if (an2->an_index_group == an->an_index_group)
                f = act_index(an2, 1, &ext, &current_area, f, &new_par, (an2 == an)?TRUE:FALSE,
TRUE);
        }
        INDEX("<p><hr><p>");
        act_index(an, 2, &ext, &current_area, FALSE, &new_par, FALSE, FALSE);
    }
}
```

The function index_page_Actions is invoked by a command in a `.i6t` template file.

Activities 11/av

Purpose

To create and manage activities, which are action-like bundles of rules controlling how the I6 runtime code carries out tasks such as "printing the name of something". Each has its own page in the I7 documentation. An activity list is a disjunction of actitivies.

11/av.§1 Activity variables; §2-3 Activity indexing

Template interpreter commands

```
1   {-array:activity_var_creators}
2   {-callv:compile_activity_constants}
2   {-array:Activity_before_rulebooks}
2   {-array:Activity_for_rulebooks}
2   {-array:Activity_after_rulebooks}
2   {-array:Activity_atb_rulebooks}
```

Definitions

```
typedef struct activity {
    int word_ref1, word_ref2;                          text of the name of the activity
    struct rulebook *before_rules;                     rulebooks for when this is followed
    struct rulebook *for_rules;
    struct rulebook *after_rules;
    struct stacked_variable_owner *owned_by_av;        activity variables owned here
    char av_I6_identifier[32];                         an I6 identifier for a constant identifying this
    int av_documentation_symbol_wn;                    cross-reference to HTML documentation, if any
    int activity_indexed;                              has this been indexed yet?
    struct activity_crossref *cross_references;
    MEMORY_MANAGEMENT
} activity;

typedef struct activity_list {
    struct activity *activity;                         what activity
    struct specification *acting_on;                   the parameter
    struct specification *only_when;                   condition for when this applies
    int ACL_parity;                                    +1 if meant positively, -1 if negatively
    struct activity_list *next;                        next in activity list
} activity_list;

typedef struct activity_crossref {
    struct phrase *rule_dependent;
    struct activity_crossref *next;
} activity_crossref;
```

The structure activity is private to this section.
The structure activity_list is private to this section.
The structure activity_crossref is private to this section.

¶1.

```
define PRINTING_THE_NAME_ACT 0
define PRINTING_THE_PLURAL_NAME_ACT 1
define PRINTING_A_NUMBER_OF_ACT 2
define PRINTING_ROOM_DESC_DETAILS_ACT 3
define LISTING_CONTENTS_ACT 4
define GROUPING_TOGETHER_ACT 5
define WRITING_A_PARAGRAPH_ABOUT_ACT 6
define LISTING_NONDESCRIPT_ITEMS_ACT 7

define PRINTING_NAME_OF_DARK_ROOM_ACT 8
define PRINTING_DESC_OF_DARK_ROOM_ACT 9
define PRINTING_NEWS_OF_DARKNESS_ACT 10
define PRINTING_NEWS_OF_LIGHT_ACT 11
define REFUSAL_TO_ACT_IN_DARK_ACT 12

define CONSTRUCTING_STATUS_LINE_ACT 13
define PRINTING_BANNER_TEXT_ACT 14

define READING_A_COMMAND_ACT 15
define DECIDING_SCOPE_ACT 16
define DECIDING_CONCEALED_POSSESS_ACT 17
define DECIDING_WHETHER_ALL_INC_ACT 18
define CLARIFYING_PARSERS_CHOICE_ACT 19
define ASKING_WHICH_DO_YOU_MEAN_ACT 20
define PRINTING_A_PARSER_ERROR_ACT 21
define SUPPLYING_A_MISSING_NOUN_ACT 22
define SUPPLYING_A_MISSING_SECOND_ACT 23
define IMPLICITLY_TAKING_ACT 24
define STARTING_VIRTUAL_MACHINE_ACT 25

define AMUSING_A_VICTORIOUS_PLAYER_ACT 26
define PRINTING_PLAYERS_OBITUARY_ACT 27
define DEALING_WITH_FINAL_QUESTION_ACT 28

define PRINTING_LOCALE_DESCRIPTION_ACT 29
define CHOOSING_NOTABLE_LOCALE_OBJ_ACT 30
define PRINTING_LOCALE_PARAGRAPH_ACT 31
```

```
sentence_handler NEW_ACTIVITY_SH_handler =
    { SENTENCE_NT, NEW_ACTIVITY_VB, 1, create_activity };

void create_activity(parse_node *pn) {
    int w1 = pn->down->next->word_ref1;
    int w2 = pn->down->next->word_ref2;
    activity *av = CREATE(activity);
    int wn, i, arity = ACTION_FOCUS;
    int future_action_flag = FALSE;
    specification *spec;

    av->av_documentation_symbol_wn = dref_position_of_symbol(&w1, &w2, FALSE);
    i = is_word_intermediate(OPENBRACKET_V, w1, w2);
    if ((i>=0) && (w2>w1+2) && [[word w2 == CLOSEBRACKET]]) {
```

```
    int b1 = i+1, b2 = w2-1;
    while (b1<=b2) {
        if [[b1, b2 == future action ***]] {
            future_action_flag = TRUE;
            b1 += 2;
            continue;
        }
        if [[b1, b2 == COMMA ... --> b1, b2]] continue;
        sentence_problem(_P_(C11ActivityNoteUnknown),
            "one of the notes about this activity makes no sense",
            "and should be either 'documented at SYMBOL' or 'future action'.");
        break;
    }
    w2 = i-1;
}
av->word_ref1 = w1; av->word_ref2 = w2;
isn_compose_identifier(av->av_I6_identifier, 'V', av->allocation_id, w1, w2);
LOGIF(ACTION_CREATIONS, "Created activity: $W\n", w1, w2);
if [[w1, w2 == ... something/anything --> w1, w2]] {
    arity = PARAMETER_FOCUS;
    [[w1, w2 == ... of --> w1, w2]];
    av->word_ref2 = w2;
}
spec = parse_expression(w1, w2, VALUE_EXPCON);
if (!(spec_is_UNKNOWN(spec)) && (!(species_is(spec, PROPERTY_VALUE_SPC)))) {
    LOG("$W means $S\n", w1, w2, spec);
    sentence_problem(_P_(C11BadActivityName),
        "this already has a meaning",
        "and so cannot be the name of a newly created activity.");
    return;
}
register_excerpt_meaning(ACTIVITY_MC, 0, w1, w2, STORE_POINTER_activity(av));
register_reworded_meaning(ACTIVITY_MC, 0, 0, 0, w1, w2, activity_V,
    STORE_POINTER_activity(av));
wn = lexer_wordcount;
feed_into_lexer("before", TRUE, FALSE);
splice_words(av->word_ref1, av->word_ref2);
av->before_rules =
    rb_new_automatic(wn, lexer_wordcount-1, arity,
        NO_OUTCOME, FALSE, future_action_flag, TRUE);
wn = lexer_wordcount;
feed_into_lexer("for", TRUE, FALSE);
splice_words(av->word_ref1, av->word_ref2);
av->for_rules =
    rb_new_automatic(wn, lexer_wordcount-1, arity,
        SUCCESS_OUTCOME, FALSE, future_action_flag, TRUE);
wn = lexer_wordcount;
feed_into_lexer("after", TRUE, FALSE);
splice_words(av->word_ref1, av->word_ref2);
av->after_rules =
    rb_new_automatic(wn, lexer_wordcount-1, arity,
        NO_OUTCOME, FALSE, future_action_flag, TRUE);
```

```
    av->owned_by_av = stvo_new(10000+av->allocation_id);
    rb_make_stvs_accessible(av->before_rules, av->owned_by_av);
    rb_make_stvs_accessible(av->for_rules, av->owned_by_av);
    rb_make_stvs_accessible(av->after_rules, av->owned_by_av);
    av->activity_indexed = FALSE;
    av->cross_references = NULL;
}
```

§1. Activity variables.

```
void av_add_variable(activity *av, parse_node *cnode) {
    specification *spec;
    int nw1 = -1, nw2 = -1, tw1 = -1, tw2 = -1, i;
    if ((pn_get_node_type(cnode) != PROPERTYCALLED_NT) &&
        (pn_get_node_type(cnode) != NOUNPHRASE_NT)) {
        LOG("Tree: $T\n", cnode);
        internal_error("ac_add_variable on a node of unknown type");
    }
    if (pn_get_node_type(cnode) == NOUNPHRASE_NT) {
        quote_source(1, current_sentence);
        quote_words(2, nw1, nw2);
        handmade_problem(_P_(C11ActivityVariableNameless));
        issue_problem_segment(
            "You wrote %1, which I am reading as a request to make "
            "a new named variable for an activity - a value associated "
            "with a activity and which has a name. Here, though, there "
            "seems to be no name for the variable as such, only an indication "
            "of its kind. Try something like 'The printing the banner text "
            "activity has a number called the accumulated vanity'.");
        issue_problem_end();
        return;
    }
    [[tw1, tw2 <-- cnode->down]];
    [[nw1, nw2 <-- cnode->down->next]];
    spec = parse_expression(tw1, tw2, TYPE_EXPCON);
    if [[nw1, nw2 == ... and ... : i]] {
        quote_source(1, current_sentence);
        quote_words(2, nw1, nw2);
        handmade_problem(_P_(C11ActivityVarAnd));
        issue_problem_segment(
            "You wrote %1, which I am reading as a request to make "
            "a new named variable for an activity - a value associated "
            "with a activity and which has a name. The request seems to "
            "say that the name in question is '%2', but I'd prefer to "
            "avoid 'and' in such names, please.");
        issue_problem_end();
        return;
    }
    if (species_is(spec, DESCRIPTION_SPC)) {
        if ((spec_get_described_kind(spec)) && (number_of_adjectives_applied_to(spec) == 0))
{
            spec = new_generic_CONSTANT_type(kovko(spec_get_described_kind(spec)));
```

```
        } else {
            quote_source(1, current_sentence);
            quote_words(2, tw1, tw2);
            handmade_problem(_P_(C11ActivityVarOverspecific));
            issue_problem_segment(
                "You wrote %1, which I am reading as a request to make "
                "a new named variable for an activity - a value associated "
                "with a activity and which has a name. The request seems to "
                "say that the value in question is '%2', but this is too "
                "specific a description. (Instead, a kind of value "
                "(such as 'number') or a kind of object (such as 'room' "
                "or 'thing') should be given. To get a property whose "
                "contents can be any kind of object, use 'object'.)");
            issue_problem_end();
            return;
        }
    }
    if (!(spec_is_generic_CONSTANT(spec))) {
        LOG("Offending SP: $X", spec);
        quote_source(1, current_sentence);
        quote_words(2, tw1, tw2);
        handmade_problem(_P_(C11ActivityVarUnknownKOV));
        issue_problem_segment(
            "You wrote %1, but '%2' is not the name of a kind of "
            "value which I know (such as 'number' or 'text').");
        issue_problem_end();
        return;
    }
    if (is_kova(spec_get_kind_of_value(spec), ANY_VALUE_TY)) {
        quote_source(1, current_sentence);
        quote_words(2, tw1, tw2);
        handmade_problem(_P_(C11ActivityVarValue));
        issue_problem_segment(
            "You wrote %1, but saying that a variable is a 'value' "
            "does not give me a clear enough idea what it will hold. "
            "You need to say what kind of value: for instance, 'A door "
            "has a number called street address.' is allowed because "
            "'number' is specific about the kind of value.");
        issue_problem_end();
        return;
    }
    stvo_add(av->owned_by_av, nw1, nw2, spec_get_kind_of_value(spec), NULL);
}

void compile_activity_var_creators_array(OUTPUT_STREAM) {
    activity *av;
    WRITE("Array activity_var_creators -->");
    LOOP_OVER(av, activity) {
        if (stvo_empty(av->owned_by_av)) WRITE(" 0");
        else WRITE(" AVSTVC_%d", av->allocation_id);
    }
    WRITE(" 0;\n");
    LOOP_OVER(av, activity) {
        if (stvo_empty(av->owned_by_av) == FALSE)
```

```
            stvl_compile_frame_creator(OUT, av->owned_by_av,
                " AVSTVC_%d", av->allocation_id);
    }
}
```

The function av_add_variable is called from 8/mass.
The function compile_activity_var_creators_array is invoked by a command in a `.i6t` template file.

§2. Activity indexing.

```
void av_index_by_number(int id, int indent) {
    activity *av;
    LOOP_OVER(av, activity)
        if (av->allocation_id == id) av_index(av, indent);
}

void av_index(activity *av, int indent) {
    int empty = TRUE;
    char *doc_link = NULL, *text = NULL;
    if (av->activity_indexed) return;
    av->activity_indexed = TRUE;
    if (rb_is_empty(av->before_rules, NULL) == FALSE) empty = FALSE;
    if (rb_is_empty(av->for_rules, NULL) == FALSE) empty = FALSE;
    if (rb_is_empty(av->after_rules, NULL) == FALSE) empty = FALSE;
    if (av->cross_references) empty = FALSE;
    if (av->av_documentation_symbol_wn >= 0)
        doc_link = lw_array[av->av_documentation_symbol_wn].lw_rawtext;
    if (empty) text = "There are no rules before, for or after this activity.";
    index_rules_box(NULL, av->word_ref1, av->word_ref2, doc_link,
        NULL, av, text, indent);
}

void av_index_details(activity *av) {
    rb_index(av->before_rules, "before", NULL, NULL);
    rb_index(av->for_rules, "for", NULL, NULL);
    rb_index(av->after_rules, "after", NULL, NULL);
    av_index_cross_references(av);
}

char *av_get_I6_representation(activity *av) {
    return av->av_I6_identifier;
}

activity_list *parse_activity_list(int w1, int w2, int p) {
    meaning_list *ml = NULL;
    activity_list *al;
    int split;
    if [[w1, w2 == not ... --> w1, w2]]
        return parse_activity_list(w1, w2, FALSE);
    if (is_list_divided(w1, w2, LOOK_FOR_OR)) {
        int lw1 = left_w1, lw2 = left_w2,
            rw1 = right_w1, rw2 = right_w2;
        activity_list *lal, *ral;
        lal = parse_activity_list(lw1, lw2, p);
        ral = parse_activity_list(rw1, rw2, p);
        if ((lal == NULL) || (ral == NULL)) return NULL;
        lal->next = ral;
```

```
        return lal;
    }
    al = CREATE(activity_list);
    for (split = w1+1; split<=w2+1; split++) {
        ml = SP_excerpt(ACTIVITY_MC, w1, split-1);
        if (ml) {
            if (split<=w2) {
                specification *possible_operand;
                int w3 = split;
                [[w3, w2 == of ... --> w3, w2]];
                possible_operand = parse_expression(w3, w2, TYPE_EXPCON);
                if (validate_parameter(possible_operand, kova(OBJECT_TY))) {
                    al->acting_on = possible_operand;
                    break;
                }
            } else break;
        }
    }
    if (ml != NULL) al->activity = RETRIEVE_POINTER_activity(em_data(ml_meaning(ml)));
    else {
        al->acting_on = NULL;
        al->only_when = parse_expression(w1, w2, CONDITION_EXPCON);
        if (validate_when(al->only_when) == FALSE) return NULL;
    }
    al->next = NULL;
    al->ACL_parity = p;
    return al;
}

void compile_activity_list(OUTPUT_STREAM, activity_list *al) {
    if (al->ACL_parity == FALSE) WRITE("(~~(");
    else WRITE("((");
    while (al != NULL) {
        if (al->activity != NULL) {
            if (al->acting_on == NULL)
            WRITE("(TestActivity(%s))", al->activity->av_I6_identifier);
            else WRITE("(TestActivity(%s, Prop_%d))",
                al->activity->av_I6_identifier,
                compile_deferred_description_test(al->acting_on));
        }
        else {
            WRITE("(");
            spec_compile(OUT, al->only_when);
            WRITE(")");
        }
        al = al->next;
        if (al != NULL) WRITE(" || ");
    }
    WRITE("))");
}

void compile_activity_constants(OUTPUT_STREAM) {
    activity *av;
    LOOP_OVER(av, activity) {
        WRITE("Constant %s = %d;\n", av->av_I6_identifier, av->allocation_id);
```

```
    }
}

void compile_Activity_before_rulebooks_array(OUTPUT_STREAM) {
    activity *av; int i = 0;
    WRITE("Array Activity_before_rulebooks --> ");
    LOOP_OVER(av, activity) {
        WRITE("%d ", av->before_rules->allocation_id); i++;
    }
    if (i==0) WRITE("NULL ");
    WRITE("NULL;\n");
}

void compile_Activity_for_rulebooks_array(OUTPUT_STREAM) {
    activity *av; int i = 0;
    WRITE("Array Activity_for_rulebooks --> ");
    LOOP_OVER(av, activity) {
        WRITE("%d ", av->for_rules->allocation_id); i++;
    }
    if (i==0) WRITE("NULL ");
    WRITE("NULL;\n");
}

void compile_Activity_after_rulebooks_array(OUTPUT_STREAM) {
    activity *av; int i = 0;
    WRITE("Array Activity_after_rulebooks --> ");
    LOOP_OVER(av, activity) {
        WRITE("%d ", av->after_rules->allocation_id); i++;
    }
    if (i==0) WRITE("NULL ");
    WRITE("NULL;\n");
}

void compile_Activity_atb_rulebooks_array(OUTPUT_STREAM) {
    activity *av; int i = 0;
    WRITE("Array Activity_atb_rulebooks -> ");
    LOOP_OVER(av, activity) {
        WRITE("%d ", rb_used_by_future_actions(av->before_rules));
        i++;
    }
    if (i==0) WRITE("$ff ");
    WRITE("$ff;\n");
}

void avl_annotate_for_cross_references(activity_list *avl, phrase *ph) {
    for (; avl; avl = avl->next)
        if (avl->activity) {
            activity *av = avl->activity;
            activity_crossref *acr = CREATE(activity_crossref);
            acr->next = av->cross_references;
            av->cross_references = acr;
            acr->rule_dependent = ph;
        }
}

void av_index_cross_references(activity *av) {
    activity_crossref *acr;
    for (acr = av->cross_references; acr; acr = acr->next) {
```

```
        phrase *ph = acr->rule_dependent;
        if ((ph->declaration_node) && (ph->declaration_node->word_ref1 >= 0)) {
            open_html_paragraph(ifl, 2, "tight");
            INDEX("NB: ");
            print_text_to_file(ph->declaration_node->word_ref1,
                ph->declaration_node->word_ref2, ifl);
            index_link(lw_array[ph->declaration_node->word_ref1].lw_source);
            INDEX("</p>");
        }
    }
}
```

The function av_index_by_number is called from 12/rb.
The function av_index is called from 12/rb.
The function av_index_details is called from 12/rb.
The function av_get_I6_representation is called from 7/vasp.
The function parse_activity_list is called from 12/phrcd.
The function compile_activity_list is called from 12/phrcd.
The function compile_activity_constants is invoked by a command in a `.i6t` template file.
The function compile_Activity_before_rulebooks_array is invoked by a command in a `.i6t` template file.
The function compile_Activity_for_rulebooks_array is invoked by a command in a `.i6t` template file.
The function compile_Activity_after_rulebooks_array is invoked by a command in a `.i6t` template file.
The function compile_Activity_atb_rulebooks_array is invoked by a command in a `.i6t` template file.
The function avl_annotate_for_cross_references is called from 12/phrcd.

12 Phrases and Rules

`12/iphr`: *Introduction to Phrases.w* An exposition of the data structures used inside Inform to hold phrases, rules and rulebooks.

`12/cs`: *Construction Sequence.w* To deal with all the `.i6t` interpreted commands which bring about the compilation of phrases, and to ensure that they are used in the correct order.

`12/ph`: *Phrases.w* To create one `phrase` object for each phrase declaration in the source text.

`12/phud`: *Phrase Usage.w* To parse the preamble of a phrase declaration to a `ph_usage_data` (PHUD) structure containing a still mostly textual representation of the conditions for its usage; to issue problem messages diagnosing bad syntax in these preambles; later, to parse further in order to convert the PHUD to a no longer textual PHRCD structure; finally, to use a PHUD to create a suitable primary booked rule (BR) for any phrase destined to be a rule, employing any predeclared BR for the phrase which may exist.

`12/phrcd`: *Phrase Runtime Context Data.w* To create, logically compare the specificity of, and compile I6 tests which check the current validity of, the circumstances in which a rule phrase should be used. All interesting PHRCD structures are manufactured from PHUDs, however, and the code for that is found in the Phrase Usage section.

`12/phtd`: *Phrase Type Data.w* To create, manage, compare the logical specificity of, and assist excerpt parsing concerning, the type of a To phrase. This involves whether it is void, determines a condition or returns a value (and if so, what kind of value); and also what parameters it takes (possibly values, possible other types used by inline definitions) and their types in turn.

`12/phod`: *Phrase Options.w* To create and subsequently parse against the list of phrase options with which the user can choose to invoke a To phrase.

`12/phsf`: *Stack Frames.w* To create, and parse local value names within the context of, stack frames. A stack frame is a context in which sequences of void-phrase expression evaluations can be performed, that is, in which procedural code can be compiled. Each phrase compiled to an I6 routine needs such a context, but a stack frame can also arise when other routines not derived from phrases are being compiled. In this section we also manage the beginning and ending of code blocks (as in "If ... begin; ...; end if").

`12/stv`: *Stacked Variables.w* To permit quantities to belong to rulebooks, and to have limited name-scope.

`12/cinv`: *Compile Invocations.w* Here we generate Inform 6 code to execute the phrase(s) called for by an invocation list.

`12/ambig`: *Runtime Ambiguities.w* To keep track of runtime ambiguities likely to arise, classifying them into equivalent questions, in order to minimise the amount of run-time checking code which will be needed.

`12/cph`: *Compile Phrases.w* For each phrase whose declaration involves a list of expressions to be evaluated in void context – that is, whose declaration is not an inline piece of I6 – we compile a suitable I6 routine. We also provide some utilities for other sections of NI needing to compile I6 routines not derived directly from phrases.

`12/toph`: *To Phrases.w* The first of four ways phrases are invoked: in the definitions of other phrases, and in I7 expressions and conditions. Here a phrase is used much as a function would be used in a C-like language.

`12/phin`: *Phrasebook Index.w* To compile the HTML page for the Phrasebook index.

`12/def`: *Adjectival Definitions.w* The second of four ways phrases are invoked: as definitions of adjectives which can be used as unary predicates in the calculus. (And we also look after adjectives arising from I6 or I7 conditions, and from I6 routines.)

`12/tiph`: *Timed Phrases.w* The third of four ways phrases are invoked: as timed events, which need no special NI data structure and are simply compiled into a pair of timetable I6 arrays to be processed at run-time.

`12/br`: *Rules.w* The fourth of four ways phrases are invoked: as rules capable of being placed in rulebooks. Each such phrase is pointed to by one or more `booked_rule` structures, which are analogous to looseleaf pages for use in ring-binders: a page can either be left looseleaf, or can be bound into a single rulebook. NI requires that phrases be able to belong to multiple rulebooks, and this is achieved by creating a different BR for each such membership. Here we create and manage both individual BRs and linked lists of BRs in logical specificity order.

`12/rb`: *Rulebooks.w* To create, manage, compile and index rulebooks, the content of which is a linked list of booked rules together with some general conventions as to how they are to be used.

`12/rps`: *Rule Placement Sentences.w* To parse and act upon explicit sentences like "The fire alarm rule is listed after the burglar alarm rule in the House Security rules."

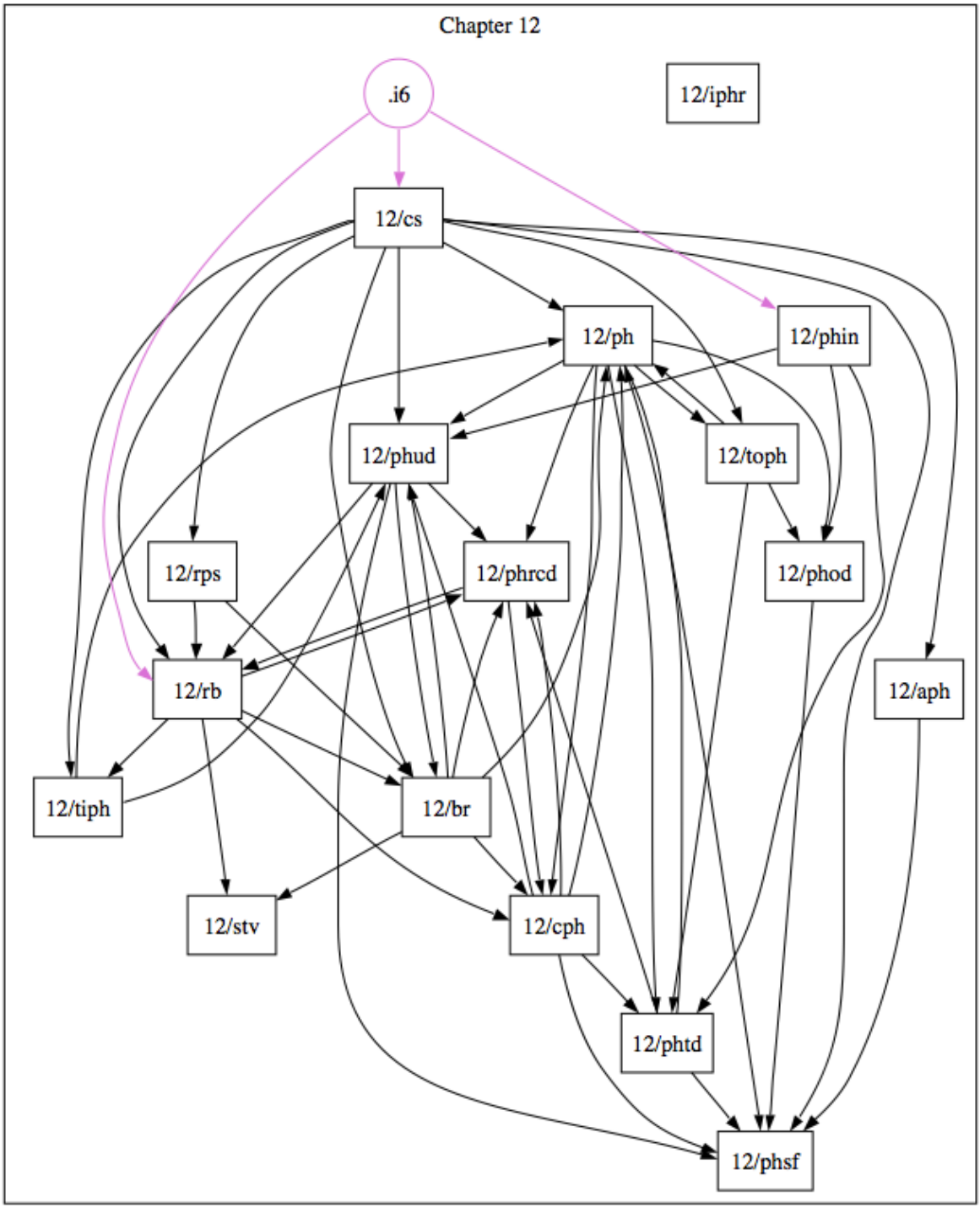
Introduction to Phrases 12/iphr

Purpose

An exposition of the data structures used inside Inform to hold phrases, rules and rulebooks.

§**1.** Much of the typical source text for an Inform work of IF consists of phrase declarations, consisting of a preamble, then a colon (except in a few cases where commas are permitted) and then a list of phrase invocations divided by semicolons. Some declarations ("To..." phrases) declare what most Inform writers think of as phrases; some define adjectives ("Definition: ..."); others declare "rules" ("Instead of jumping..."). Grammatically, every phrase is a single sentence, even when quite long and written out in a tabulated, computer language sort of form: rather as Philip Larkin's poem *MCMIV* is a single sentence through all four stanzas.

Internally, these forms are handled by largely common code and data structures with as much commonality between them as possible.


Chapter 12
.i6
12/iphr
12/cs
12/ph
12/phin
12/phud
12/toph
12/rps
12/phrcd
12/phod
12/rb
12/aph
12/tiph
12/br
12/stv
12/cph
12/phtd
12/phsf

§**2. Phrases are not invocations.** In common with Inform 7 documentation generally, NI uses the word "phrase" in a more specialised way than the usual English sense of "a small group of words standing together as a conceptual unit, typically forming a component of a clause".

NI uses the term "excerpt" for this, reserving "phrase" for a more specific meaning, or rather for three related ones:

(1a) One of the possible forms of instruction to do something or to decide something, generally defined by text such as "To award (N - a number) points: ...".

(1b) One of the rules, defined by text such as "Before eating the cake, ...".

(2) An actual instruction to do something, such as "award 50 points". These are properly speaking "invocations" of phrases rather than phrases themselves, and are – as we have seen – stored in the references list of `PHRASE` specifications during type-checking and code generation.

The `phrase` structure holds details of each possible form (1a) or (1b). Each of these is defined *either* as a list of invocations (2) of phrases (1a) to do something, in which case the definition will be compiled as an I6 routine, *or* as an I6 schema. Here are examples of these two forms of definition, both for phrases (1a) to do something:

```
To spring the trap: say "Sproing!"; move the player to the Cage.
```

```
To award (some - number) points: (- score=score+some; -).
```

When we compile an invocation (2) of these, "spring the trap" is compiled to an I6 function call to the definition routine, which might typically look like `R_231();`, whereas "award 50 points" is compiled inline to `score=score+50;`.

A phrase (1b) might be defined using I7 source like so:

```
A reaching inside rule for the flask: allow access.
```

These phrases have no invocations, and often no names: they simply take effect when needed. Their definitions are always as I6 routines.

§**3. Data structures for phrase declarations.**

Each declaration corresponds to a `phrase` structure. This is an anthology of five sub-structures for different purposes: PHUD, PHTD, PHSF, PHRCD and PHOD. The data in these structures is used to represent the information in the preamble, and also information needed during compilation of the body of the declaration (local variable names which come and go, for instance).

Further structures represent individual local variables (`local_variable`) within PHSFs, and individual phrase options (`phrase_option`) within PHODs.

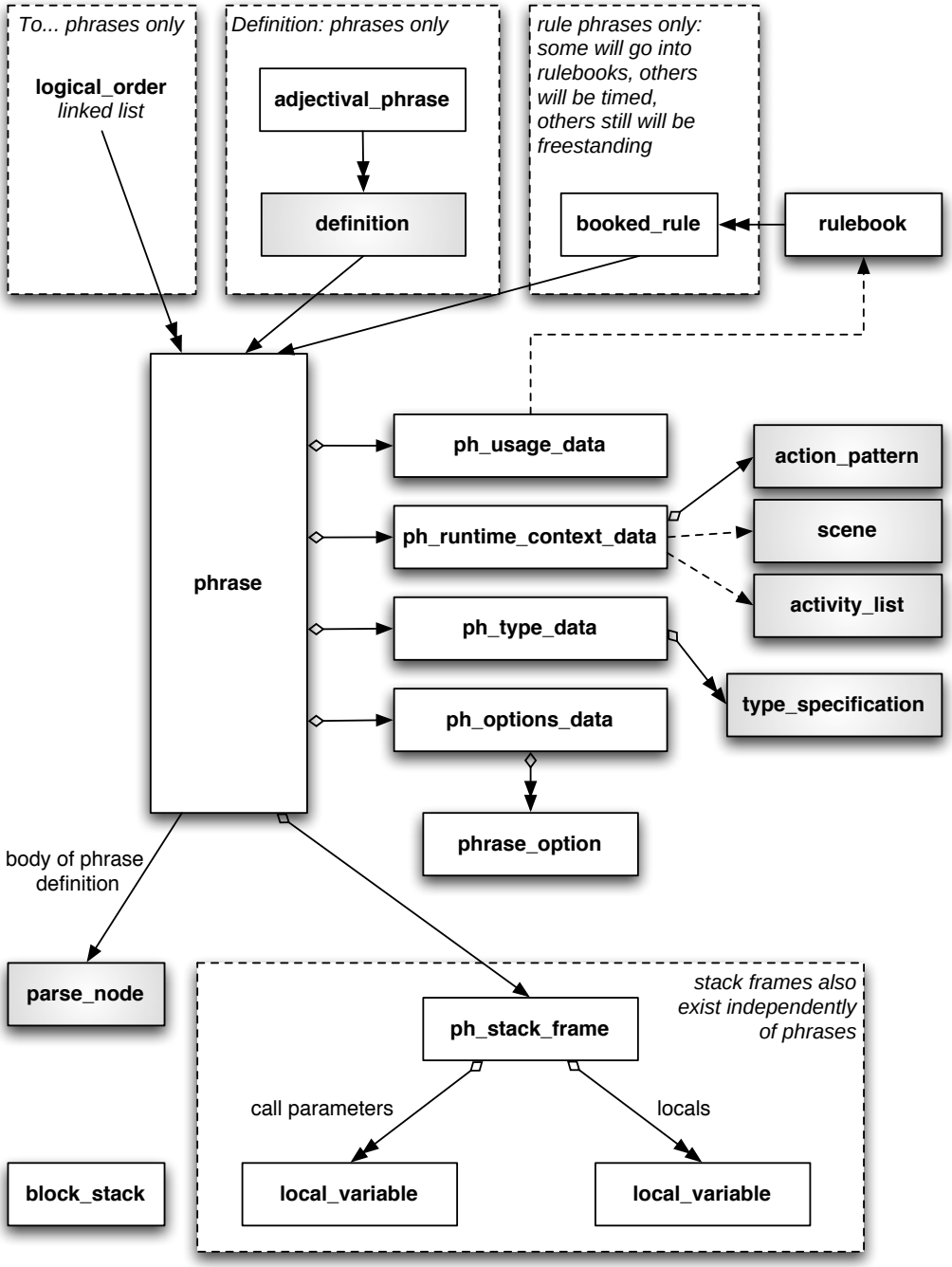
Whereas "To..." phrases compile to I6 routines which are called as necessary from I6 expressions explicitly compiled by I7, rules are placed into I6 arrays and are invoked at the whim of the I6 run-time code. There are two main sets of arrays: the timetable arrays, used for rules whose usage is quoted in terms of time ("At 10:23 AM: ..."), and rulebooks, used for rules quoted as applying to actions, activities or similar. In both cases, a small structure called a `booked_rule` is used to represent an individual booking within the timetable or a rulebook: this is analogous to a page on which a rule is written. BRs need to exist because the same rule can be written into more than one rulebook; there is only one phrase structure for the rule, but a different BR for each place where it is used.

NI does not use a structure to represent the timetable, but simply generates it when needed. Rulebooks, however, are stored as `rulebook` structures.

Finally, we also use booked rules to provide definitions of adjectives. Each individually named adjective has an `adjectival_phrase` structure.

None of the routines in this chapter is especially complex, but the situation is complicated by (a) the large number of varied structures being dealt with and (b) some awkward timing constraints which cause these structures to be built and used in unobvious sequences. To cope with this, the code for the substructures is

encapsulated so far as is possible, and the first section of the chapter centralises all of the timing contraints into one place, documented the sequence which is needed.



§**4. The five phrase substructures.** To recap, a `phrase` is internally divided into five substructures, whose names are abbreviated as PHUD, PHTD, PHSF, PHRCD and PHOD.

(i) The phrase usage data (PHUD) contains the results of parsing the preamble of the declaration to see what kind of phrase is to be created. For "To..." phrases, little is recorded. For rules, the PHUD contains all the information needed to place the rule within its rulebook.

(ii) Conversely, the phrase type data (PHTD) is primarily useful for "to..." phrases. It records the pattern of text to be registered as an excerpt with the excerpt parser, and the types required for the tokens in the definition.

Because they are registered directly with the excerpt parser, "To..." phrases are not placed in any rulebook or stored in any run-time array. But the order in which they are registered is important, because the excerpt parser returns the first match it can, so that it is important that more specific phrases are stored before less specific ones. The judgement on specificity of a phrase is made on its PHTD alone.

(iii) The phrase options data (PHOD) contains the names of phrase options, if any apply, and whether or not they are mutually exclusive. It is used only for "To..." phrases, and is separated from the PHTD because it is parsed separately from the excerpt parser mechanism and because it is not used in type-checking.

(iv) The phrase run-time context data (PHRCD) contains data structures which store the fully-parsed preamble. This is of use only with rules, and contains (for instance) the parameter to be used when placing the rule into its rulebook – an action pattern structure. When such phrases are created, they are bound up into "booked rules" (see below) and these BRs are placed into rulebooks. Rules in rulebooks are sorted into order of specificity, and this judgement is made on the contents of the PHRCD alone.

(v) The phrase stack frame (PHSF) contains the local named values valid in a given compilation context. (Names of phrase options can also be used locally and for this the PHOD is used, but they are used as conditions not values.) These named values include local variables, which are added during compilation of the phrase, but also the call parameters of the phrase, which are generated from the tokens recorded in the PHTD. The PHSF is not necessarily empty for rules, because rule preambles sometimes contain "callings": "Instead of taking a container (called the bag): ..." requires a PHSF which contains "bag" as a locally named value, for instance.

Uniquely of the five substructures of `phrase`, the PHSF can exist independently of any `phrase` structure in more than a temporary way. Free-standing PHSF stack frames are used when compiling text which contains substitutions, and for other code-compilation purposes where code does not arise explicitly from phrase declarations in the source.

To sum up, a "to..." phrase makes use of a PHUD (minimally), a PHTD (for sorting, registration and type-checking), a PHOD, and a PHSF. A rule phrase makes use of a PHUD (more heavily), a PHRCD (for sorting and to compile checks on applicability), and a PHSF.

The obvious simplification in this scheme would be to include the PHRCD within the PHUD, and the PHOD within the PHTD, thus reducing the picture to just three structures. We do not do this because there are significant timing difficulties.

Construction Sequence 12/cs

Purpose

To deal with all the `.i6t` interpreted commands which bring about the compilation of phrases, and to ensure that they are used in the correct order.

Template interpreter commands

```
4   {-callv:traverse_for_phrase_names}
6   {-callv:traverse_for_phrases}
7   {-callv:register_phrase_meanings}
8   {-callv:parse_rule_parameters}
8   {-callv:add_rules_to_rulebooks}
8   {-callv:parse_rule_placements}
9   {-callv:compile_phrases}
10   {-callv:resolve_predeclared_booked_rules}
10   {-callv:compile_adjectival_phrases}
10   {-array:TimedEventsTable}
10   {-array:TimedEventTimesTable}
10   {-array:rulebooks_array}
10   {-callv:compile_rulebooks}
10   {-array:RulebookNames}
10   {-callv:compile_rule_printing_switch}
```

---

§**1. Timing issues.** Suppose we compare the run of NI to a single day. At dawn the program starts up, initialises the memory manager, opens the debugging and problem logs, etc. In mid-morning it runs through the assertion sentences twice, acting to create the world objects and their properties, the rulebooks, the named global variables and so forth. It then completes work with the initial state of things, constructing its world model. By noon, a decisive change is reached: it is now possible to parse non-constant expressions, which require the use of phrases to evaluate them. In the afternoon, the phrase declarations are compiled into I6 routines, and in the late afternoon the grammar is put together, text with substitution routines are compiled, and so forth. At nightfall, NI completes its run.

However, work on phrases is not left entirely until the afternoon. In the early morning, we run through the phrase preambles to look for named rules. (For instance, a phrase like "Instead of pushing the red button (this is the fire alarm rule): ...") This looking-for-names is done by parsing the preamble text to a PHUD in what is called "coarse mode", which can only get an approximate idea of the contents; it is so early in the morning that more sophisticated parsing is not possible, because so many named values do not exist yet. The PHUD thus produced is immediately thrown away once we have our answer; there is in any case no `phrase` structure yet to which it could be attached.

For each rule name which is found (here "the fire alarm rule"), an empty "booked rule" structure is created. Booked rules are analogous to pages in a rulebook; just as the same idea might be written onto pages in different books, the same phrase can appear in more than one booked rule. No rulebooks will exist until mid-morning, so at this stage any booked rules created will just be looseleaf pages, and since no phrases will exist until they afternoon, they will also be blank; but we can imagine them containing a Post-It reading "put phrase X here once it is created". The name is then registered as an excerpt whose data points to this booked rule. This needs to be done because the rule name is a value which might be used as a property of

an object, or the initial state of a global variable, or an entry in a Table; and therefore the name has to be registered before the mid-morning run through the assertions. Such a booked rule is said to be "predeclared".

Rulebooks, unlike rules, are created with standard assertion sentences, just like objects. ("When play begins is a rulebook.") They therefore appear during the mid-morning run, and their names are registered immediately. Rulebooks are like ring-binders, and they are created empty of pages.

In the late morning, all objects, kinds, rulebooks, variables, rules, etc., have their names. Because rulebook names in particular exist, we can get further in parsing the preambles to phrase declarations. So NI now runs through the preambles again and parses them for a second time to PHUDs, but this time in "fine mode" rather than "coarse mode", and this time the result is not thrown away. If the phrase is a "To..." phrase declaration, then the PHUD was pretty sketchy and we parse more substantial PHTD and PHOD structures to accompany it. If it is a rule, the PHUD contains a great deal of useful information, and we accompany it with essentially blank PHTD and PHOD structures. Either way, we have now assembled a PHUD, PHTD and PHOD: these are combined into a new `phrase` structure. The PHSF structure is initially created as a function of the PHTD, converting the token names (if there are any) to call parameter local variables. A blank PHRCD structure is created to fill out the set of five substructures.

As they are created, the "To..." phrases are insertion-sorted into a list of phrases in logical precedence order. Phrases which are destined to be adjective definitions are similarly insertion-sorted into the list of meanings for the relevant adjective, again in logical precedence order. (This can be done now because it relies only on names of kinds and types, all of which have existed since mid-morning.)

As phrases destined to be rules are created, they are each given a "primary booking" by being copied onto a new blank rulebook page. If a predeclared BR for the phrase already exist, then that one is used (we remove the metaphorical Post-It note and write in the metaphorical text of the phrase); if not, we create a new booked rule structure for the phrase. Either way, at this point every rule-like phrase now has one and only one booked rule structure in existence with the phrase written on it.

It is now nearly noon, and things appear to be a little untidy. Why are the "To..." phrases not yet registered with the excerpt parser? Why are the rules all over the floor on single pages, while the ring-binder rulebooks still lie empty? The answer in both cases is that we needed to wait until all of the "To..." phrases had been created as structures before we could safely proceed. The first phrase cannot be registered until we know the complete logical order of the phrases. With rules, the problem is worse: these cannot even be sorted yet, since they are sorted by their PHRCDs, and unlike the PHTDs a PHRCD cannot even be parsed until all the phrases have been registered – a rule might have been specified to happen "when in darkness", say, requiring that NI recognise the text "in darkness" as the name of a phrase.

So, with the clock approaching noon, we finally have the "To..." phrases sorted into logical order. We at last register these in sequence, and just as this is done, the hour chimes. From this point onwards, arbitrary run-time expressions can be parsed by NI, and there are no longer any names-which-do-not-exist-yet problems.

In the afternoon, we begin by binding up the rulebooks. First, we go through the phrases destined to be rules, and for each we translate the PHUD (which contains mainly textual representations of the usage information, e.g. "taking something (called the thingummy) which is in a lighted room during Scene Two when the marble slab is open") to a PHRCD (which contains fully parsed NI data structures, e.g., an action pattern and a pointer to a `scene` structure). This PHUD-to-PHRCD parsing is at last possible, since it is afternoon. During the PHUD-to-PHRCD parsing process, we make sure that the relevant phrase's PHSF is the current stack frame, because it's here that the names of any callings (e.g. "thingummy") are created as local variables to be valid throughout the phrase. The PHUD will never be used again.

Second, we go through the rules once again. Every rule-phrase has a BR (booked rule) structure, which is like a looseleaf rulebook page with the phrase written on it. In some cases (such as "Instead of taking something: ...") the preamble indicates a rulebook for which it is destined (here "instead"), but other rule preambles indicate no specific rulebook (as in "The fire alarm rule: ..."). Where we do have a rulebook in mind, we add the BR to the rulebook, insertion sorting it into logical precedence order as deduced from the PHRCD.

It might seem as if the rulebooks are now complete, but this is not true, because we still have to deal with any assertion sentences requiring particular named rules to be placed in particular positions in rulebooks. We do that now, creating new BRs as required.

It is now mid-afternoon, and the rulebooks are complete. It is time to compile the I6 routines which will provide the run-time definitions of all these phrases. First we compile the "To..." phrases, except that we omit phrases defined inline with an explicit I6 escape; these do not compile to I6 function calls, so they need no I6 functions as their definitions. A "To..." phrase is compiled using only the PHSF, the current stack frame: the other substructures are no longer needed.

Secondly we compile the rules. We do this in rulebook order simply to make the I6 code easier to read: it makes no difference otherwise. But we need to remember that there are still rule phrases which are looseleaf, that is, whose booked rules are not found in any rulebook. So we need to finish up by compiling rule routines for rules not in any rulebook. A rule phrase is compiled using both the PHSF, as the current stack frame, and also the PHRCD, which is used to compile the applicability test for the rule. (For instance, for "Instead of jumping during Scene V", the test would be that the action is jumping and that Scene V is currently under way.) The other substructures are no longer needed.

This is a long process, and stretches on until evening. We have four final tasks: to compile the I6 arrays which describe the rulebook contents; to compile a timetable array for those looseleaf rules which specified their usage as times ("At 3:15 PM: ...") or as events ("When the alarm clock goes off: ..."); to reconcile the names of predeclared BRs; and to compile I6 routines for the unary predicates representing adjectives at run-time, which will mean using the BRs assigned as definitions to adjectives.

The reconciliation needs some explanation. Suppose back in early morning we predeclared a BR under the name of the "fire alarm rule". An I6 constant may have been needed during the mid-morning which evaluated to the I6 routine for this phrase. But of course that phrase's I6 routine name could not have been known then, because the phrase did not exist until late morning. We get around this by creating a constant in the form `BR_1`, `BR_2`, etc., in early morning, and then compiling I6 constant declarations in the evening which define these constants in terms of the I6 routines they are now known to have represented: thus `Constant BR_1 = R_321;`, for instance.

And at last the sun slips below the horizon. As it does so, note that in spite of all our efforts in the evening, there may still be a few looseleaf rules in existence: rules which were never written into any rulebook, or into the timetable, or used to define anything. This is not necessarily a mistake – sometimes rules are defined which the user intends to dynamically assign to rulebooks at run-time, and which have no rulebook assignment at compile time.

§2. Summary of time sequence.

Dawn. NI starts up.

Early morning. Looseleaf, blank booked rules are made for any named rules, and names registered for them.

Mid-morning. NI works through the assertion sentences, creating and naming the world, and in particular creating and naming rulebooks. These are initially empty.

Late morning. The phrase structures are created. "To..." phrases are sorted into logical precedence order. Rule phrases have booked rules made for them, unless they already have one from early morning: however, their conditions of use remain as largely unparsed data in PHUDs.

Just before noon. The "To..." phrases are registered, in logical precedence order.

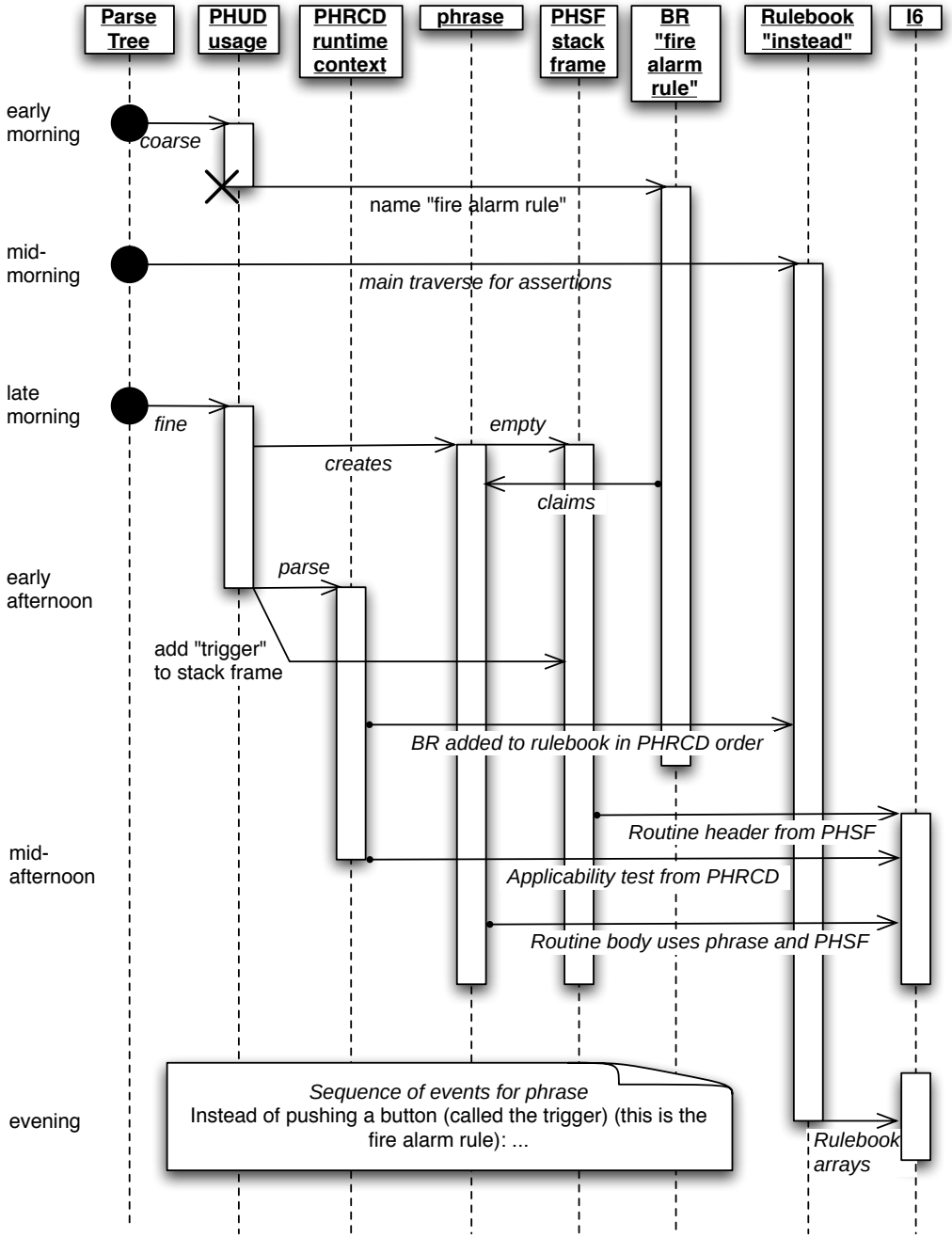
Noon. NI passes a significant milestone: before this point, only constant expressions could be parsed, but from now on, any expression can be parsed, including one which makes use of the "To..." phrases.

Early afternoon. The rule phrases have their PHUDs parsed into PHRCDs. Any rule phrase whose preamble specifies a rulebook is logically sorted into that rulebook. The explicit sentences in the source which describe how rules should occur in rulebooks are then obeyed, and the rulebooks are finally completed.

Mid-afternoon. The bodies of phrase declarations are compiled into suitable I6 routines.

Evening. The arrays giving the run-time form of rulebooks and of the event timetable are compiled, and constants are created which enable names of routines not compiled until afternoon to have been used as constants in the morning. Definitions of adjectives are compiled.

Nightfall. NI shuts down.


Parse Tree
PHUD usage
PHRCD runtime context
phrase
PHSF stack frame
BR "fire alarm rule"
Rulebook "instead"
I6
early morning
coarse
name "fire alarm rule"
mid-morning
main traverse for assertions
late morning
fine
empty
creates
claims
early afternoon
parse
add "trigger" to stack frame
BR added to rulebook in PHRCD order
Routine header from PHSF
mid-afternoon
Applicability test from PHRCD
Routine body uses phrase and PHSF
evening
Sequence of events for phrase
Instead of pushing a button (called the trigger) (this is the fire alarm rule): ...
Rulebook arrays

§**3.** To make sure that none of this happens out of sequence, we take the following precautions. (It isn't as unlikely that this might occur as it may seem – the timing is actually controlled by the sequence of calls in the interpreted `Main.i6t` file, which hackers might try to alter.)

```
define DAWN_PHT 0
define EARLY_MORNING_PHT 1
define MID_MORNING_PHT 2
define LATE_MORNING_PHT 3
define PRE_NOON_PHT 4
define NOON_PHT 5
define EARLY_AFTERNOON_PHT 6
define MID_AFTERNOON_PHT 7
define EVENING_PHT 8
define NIGHTFALL_PHT 9

int phrase_time_now = DAWN_PHT;

void advance_phrase_time_to(int advance_to) {
    if (advance_to < phrase_time_now) {
        LOG("Advance to: %d   Time now: %d\n", advance_to, phrase_time_now);
        internal_error(
            "The necessary phrase construction events are out of sequence");
    }
    phrase_time_now = advance_to;
}
```

§4. Early morning.

```
void traverse_for_phrase_names(void) {
    parse_node *p;
    advance_phrase_time_to(EARLY_MORNING_PHT);
    for (TREE_START(p); p; TREE_NEXT(p))
        if (pn_get_node_type(p) == ROUTINE_NT)
            phud_predeclare_name_in(p);
}
```

The function traverse_for_phrase_names is invoked by a command in a `.i6t` template file.

§**5. Mid-morning.** This is when NI is making its main traverses through assertions. We will ignore all commands (i.e., lines in the body of phrase declarations) and phrase preambles, so we give them null handlers. On the other hand, we do indeed want to act on declarations of I6 library rules: that is, sentences like "The can't act in the dark rule translates into I6 as `"CANT_ACT_IN_THE_DARK_R"`.", which cause a booked rule to be created which corresponds not to a `phrase` structure but to an already-compiled `I6` routine.

```
sentence_handler COMMAND_SH_handler = { COMMAND_NT, -1, 0, NULL };
sentence_handler ROUTINE_SH_handler = { ROUTINE_NT, -1, 0, NULL };

void handle_library_rule(parse_node *p) {
    new_br_from_i6(p->down->next->word_ref1, p->down->next->word_ref2,
        p->down->next->next->word_ref1, FALSE);
}
```

The function handle_library_rule is called from 2/isn.

§6. Late morning.

```
void traverse_for_phrases(void) {
    parse_node *p;
    int progress_target = 0, progress_made = 0;
    advance_phrase_time_to(LATE_MORNING_PHT);
    for (TREE_START(p); p; TREE_NEXT(p)) progress_target++;
    for (TREE_START(p); p; TREE_NEXT(p)) {
        progress_made++;
        if (progress_made % 10 == 0)
            progress_bar(3, ((float) progress_made)/((float) progress_target));
        if (pn_get_node_type(p) == ROUTINE_NT) {
            ph_create_from_preamble(p);
            finish_this_session_of_parsing();
        }
    }
}
```

The function traverse_for_phrases is invoked by a command in a `.i6t` template file.

§7. Just before noon.

```
void register_phrase_meanings(void) {
    advance_phrase_time_to(PRE_NOON_PHT);
    ph_To_register_all();
}
```

The function register_phrase_meanings is invoked by a command in a `.i6t` template file.

§8. **Early afternoon.** The three main routines here have to happen in sequence in order to work as intended; the PushDir warning is less crucial.

Note that the PHRCDs have to be parsed in source text appearance order (the order which `LOOP_OVER` works in) so that the back reference "doing it" (meaning, the most recently mentioned action) can work.

```
void parse_rule_parameters(void) {
    phrase *ph;
    advance_phrase_time_to(EARLY_AFTERNOON_PHT);
    LOOP_OVER(ph, phrase) {
        current_sentence = ph->declaration_node;
        phsf_select_stack_frame_of(ph);
        ph->runtime_context_data = phrcd_from_phud(&(ph->usage_data));
        phsf_remove_nonphrase_stack_frame();
        finish_this_session_of_parsing();
    }
}
void add_rules_to_rulebooks(void) {
    advance_phrase_time_to(EARLY_AFTERNOON_PHT);
    br_add_rules_to_rulebooks();
}
void parse_rule_placements(void) {
    parse_node *p;
```

```
    advance_phrase_time_to(EARLY_AFTERNOON_PHT);
    for (TREE_START(p); p; TREE_NEXT(p))
        if ((pn_get_node_type(p) == SENTENCE_NT) &&
            (p->down) &&
            (pn_get_node_type(p->down) == VERB_NT)) {
            prevailing_mood = pn_int_annotation(p->down, verbal_certainty_ANNOT);
            switch(pn_int_annotation(p->down, verb_id_ANNOT)) {
                case IN_RULEBOOK_VB:
                    place_in_rulebook(p->down->next, p->down->next->next, TRUE);
                    pn_annotate_int(p, sentence_unparsed_ANNOT, TRUE);
                    break;
                case NOT_IN_RULEBOOK_VB:
                    place_in_rulebook(p->down->next, p->down->next->next, FALSE);
                    pn_annotate_int(p, sentence_unparsed_ANNOT, TRUE);
                    break;
            }
            finish_this_session_of_parsing();
        }
}
```

The function parse_rule_parameters is invoked by a command in a `.i6t` template file.
The function add_rules_to_rulebooks is invoked by a command in a `.i6t` template file.
The function parse_rule_placements is invoked by a command in a `.i6t` template file.

§9. Mid-afternoon.

```
void compile_phrases(OUTPUT_STREAM) {
    int i = 0, max_i = 0;
    phrase *ph;
    rulebook *rb;
    advance_phrase_time_to(MID_AFTERNOON_PHT);
    LOOP_OVER(ph, phrase)
        if (ph->compilation_needed)
            max_i++;
    ph_To_compile_all(OUT, &i, max_i);
    WRITE("! Definitions of rule phrases\n\n");
    LOOP_OVER(rb, rulebook)
        rb_compile_rule_phrases(rb, OUT, &i, max_i);
    if (i < max_i) {
        booked_rule *br;
        WRITE("! Definitions of displaced and unbooked rules\n\n");
        LOOP_OVER(br, booked_rule) {
            ph = br_get_rule(br);
            if (ph)
                ph_compile(ph, OUT, &i, max_i, br_stvol_accessible_here(br));
        }
    }
    if (i < max_i) {
        WRITE("! Definitions of miscellaneous remaining rules\n\n");
        LOOP_OVER(ph, phrase)
            ph_compile(ph, OUT, &i, max_i, NULL);
    }
}
```

The function compile_phrases is invoked by a command in a .i6t template file.

§**10. Evening.** These miscellaneous compilations can happen in any order, as suits the needs of the Main.i6t script.

```
void resolve_predeclared_booked_rules(OUTPUT_STREAM) {
    advance_phrase_time_to(EVENING_PHT);
    br_resolve_predeclared_booked_rules(OUT);
}
void compile_adjectival_phrases(OUTPUT_STREAM) {
    advance_phrase_time_to(EVENING_PHT);
    aph_compile_support_code(OUT);
}
void compile_TimedEventsTable_array(OUTPUT_STREAM) {
    advance_phrase_time_to(EVENING_PHT);
    tiph_compile_TimedEventsTable_array(OUT);
}
void compile_TimedEventTimesTable_array(OUTPUT_STREAM) {
    advance_phrase_time_to(EVENING_PHT);
    tiph_compile_TimedEventTimesTable_array(OUT);
}
void compile_rulebooks_array_array(OUTPUT_STREAM) {
    advance_phrase_time_to(EVENING_PHT);
    rb_compile_rulebooks_array_array(OUT);
}
void compile_rulebooks(OUTPUT_STREAM) {
    advance_phrase_time_to(EVENING_PHT);
    rb_compile_rulebooks(OUT);
}
void compile_RulebookNames_array(OUTPUT_STREAM) {
    advance_phrase_time_to(EVENING_PHT);
    rb_compile_RulebookNames_array(OUT);
}
void compile_rule_printing_switch(OUTPUT_STREAM) {
    advance_phrase_time_to(EVENING_PHT);
    br_compile_rule_printing_switch(OUT);
}
```

The function resolve_predeclared_booked_rules is invoked by a command in a .i6t template file.
The function compile_adjectival_phrases is invoked by a command in a .i6t template file.
The function compile_TimedEventsTable_array is invoked by a command in a .i6t template file.
The function compile_TimedEventTimesTable_array is invoked by a command in a .i6t template file.
The function compile_rulebooks_array_array is invoked by a command in a .i6t template file.
The function compile_rulebooks is invoked by a command in a .i6t template file.
The function compile_RulebookNames_array is invoked by a command in a .i6t template file.
The function compile_rule_printing_switch is invoked by a command in a .i6t template file.

Phrases 12/ph

Purpose

To create one `phrase` object for each phrase declaration in the source text.

12/ph.§1-2 Problem messages for complex say constructions

Definitions

¶1. This is the largest structure, in terms of number of fields, found anywhere in NI. To keep some kind of control over this complexity, it is divided into sub-structures handling different purposes, and which are available at different times. Of particular note is the stack frame: for a phrase, this is a function of the type data, but it is also possible to generate a stack frame in a free-standing way in order to compile code which does not derive from a phrase declaration (for instance, to compile a routine to print a piece of text which contains substitutions).

These structures are, as far as possible, private to their own sections. Within this chapter, all sections have access to the substructure elements of `phrase`. Outside this chapter, only the PHTD substructure is directly visible, and then only to Chapter 7 (on type-checking). The other elements of `phrase` are private.

The substructures themselves are private to their sections except that `type_data` is shared with two sections from Chapter 7, and that `ph_runtime_context_data` is shared between two sections of Chapter 12 (since a PHRCD is essentially generated from a PHUD, there is no sensible way to separate code which performs this generation into a PHRCD-owned part and a PHUD-owned part).

```
typedef struct phrase {
    struct parse_node *declaration_node;                    ROUTINE_NT node where declared
    int inline_wn;                                  word number of inline I6 definition, or -1 if not inline
    struct ph_type_data type_data;
    struct ph_usage_data usage_data;
    struct ph_runtime_context_data runtime_context_data;
    struct ph_stack_frame stack_frame;
    struct ph_options_data options_data;
    int compilation_needed;                          do we still need to compile an I6 routine for this?
    char ph_I6_identifier[32];                     name of I6 routine, or empty string for inline phrases
    struct phrase *next_in_logical_order;                                  for "to..." phrases only
    int sequence_count;                                          within the logical order list, from 0
    int ph_documentation_symbol_wn;                              cross-reference with documentation
    MEMORY_MANAGEMENT
} phrase;
```

The structure phrase is shared with 7/vasp, 7/tc, 7/inv, 11/av, 12/cs, 12/phrcd, 12/phtd, 12/phsf, 12/cinv, 12/ambig, 12/cph, 12/toph, 12/phin, 12/tiph and 12/br.

```
phrase *ph_new(ph_usage_data *phud, ph_type_data *phtd, ph_options_data *phod,
    int d, int inline_wn) {
    phrase *ph = CREATE(phrase);
    LOGIF(PHRASE_CREATIONS, "Creating phrase: <$W>\n$U",
        phud_get_preamble_w1(phud), phud_get_preamble_w2(phud), phud);
    if ((current_sentence == NULL) ||
        (pn_get_node_type(current_sentence) != ROUTINE_NT))
        internal_error("Phrase creation other than at a ROUTINE_NT node");
```

```
    ph->declaration_node = current_sentence;
    ph->usage_data = *phud;
    ph->type_data = *phtd;
    ph->stack_frame = phtd_to_stack_frame(phtd);
    if (phod_allows_options(phod))
        phsf_options_parameter_is_needed(&(ph->stack_frame));
    ph->runtime_context_data = phrcd_new();
    ph->options_data = *phod;
    ph->next_in_logical_order = NULL;
    ph->sequence_count = -1;
    ph->inline_wn = inline_wn;
    if (inline_wn >= 0) {
        char *inline_defn = lw_array[inline_wn].lw_text;
        if (strlen(inline_defn) >= MAX_INLINE_DEFN_LENGTH) {
            LOG("Inline definition: <$W>\n", inline_wn, inline_wn);
            sentence_problem(_P_(C12InlineTooLong),
                "the inline definition of this 'to...' phrase is too long",
                "using a quantity of Inform 6 code which exceeds the fairly "
                "small limit allowed. You will need either to write the "
                "phrase definition in Inform 7, or to call an I6 routine "
                "which you define elsewhere with an 'Include ...'.");
            inline_defn[MAX_INLINE_DEFN_LENGTH-1] = 0;
        }
        if (phud_get_RAM(phud) != PHRASE_DECLARATION_RAM) {
            sentence_problem(_P_(C12InlineRule),
                "only 'to...' phrases can be given inline Inform 6 definitions",
                "and in particular rules and adjective definitions can't.");
        }
        ph->compilation_needed = FALSE;
        ph->ph_I6_identifier[0] = 0;
    } else {
        ph->compilation_needed = TRUE;
        sprintf(ph->ph_I6_identifier, "R_%d", ph->allocation_id);
    }
    ph->ph_documentation_symbol_wn = d;
    return ph;
}
void log_phrase(phrase *ph) {
    if (ph == NULL) { LOG("RULE:NULL"); return; }
    LOG("%s", ph->ph_I6_identifier);
    phud_log_rule_name(&(ph->usage_data));
}
char *ph_get_I6_representation(phrase *ph) {
    return ph->ph_I6_identifier;
}
void ph_write_HTML_representation(phrase *ph, char *to) {
    phtd_write_HTML_representation(&(ph->type_data), to);
}
void log_phrase_briefly(phrase *ph) {
    log_ph_type_data_briefly(&(ph->type_data));
}
```

```
void ph_compile(phrase *ph, OUTPUT_STREAM, int *i, int max_i,
    stacked_variable_owner_list *legible) {
    if (ph->compilation_needed) {
        compile_phrase(OUT, ph, legible);
        ph->compilation_needed = FALSE;
        (*i)++;
        progress_bar(4, ((float) (*i))/((float) max_i));
    }
}
```

The function log_phrase is called from 2/dl and 12/toph.

The function ph_get_I6_representation is called from 12/cinv, 12/cph, 12/toph, 12/def, 12/tiph and 12/br.

The function ph_write_HTML_representation is called from 2/prob2, 12/phtd and 12/toph.

The function log_phrase_briefly is called from 7/inv.

The function ph_compile is called from 12/cs, 12/toph and 12/br.

§1. Problem messages for complex say constructions.

```
void add_say_construction_to_error(int ssp_tok) {
    issue_problem_segment(" %P(The construction I'm thinking of is '");
    add_scte_list(ssp_tok, SSP_START);
    issue_problem_segment(" ... ");
    add_scte_list(ssp_tok, SSP_MIDDLE);
    issue_problem_segment(" ... ");
    add_scte_list(ssp_tok, SSP_END);
    issue_problem_segment("'.)");
}
void add_scte_list(int ssp_tok, int list_pos) {
    phrase *ph; int ct = 0, w1, w2;
    LOOP_OVER(ph, phrase)
        if (phtd_ssp_matches(&(ph->type_data), ssp_tok, list_pos, &w1, &w2)) {
            quote_words(3, w1, w2);
            if (ct++ == 0) issue_problem_segment("[%3]"); else issue_problem_segment("/[%3]");
        }
}
```

The function add_say_construction_to_error is called from 12/cinv.

§2. The following routine is the one which constructs a phrase object from the parse-tree representation of a declaration in the source text. This is the only routine in NI which contains the definition of what an inline declaration looks like:

A phrase is inline if and only if its definition consists of a single invocation which is given as verbatim I6.

```
void ph_create_from_preamble(parse_node *p) {
    phrase *current_phrase;
    ph_usage_data phud;
    ph_type_data phtd;
    ph_options_data phod;
    int x1, x2, ow1 = -1, ow2 = -1, inline_flag = FALSE, inline_wn = -1, ds = -1;
    if ((p == NULL) || (pn_get_node_type(p) != ROUTINE_NT))
        internal_error("ph_create_from_preamble other than at a ROUTINE_NT node");
    if ((p->down) && (p->down->next == NULL) && [[p->down == OPENI6 ...]]) {
        inline_flag = TRUE;
        inline_wn = p->down->word_ref1 + 1;
```

```
    }
    phud = phud_new(p->word_ref1, p->word_ref2, FALSE);
    x1 = phud_get_preamble_w1(&phud); x2 = phud_get_preamble_w2(&phud);
    switch (phud_get_RAM(&phud)) {
        case PHRASE_DECLARATION_RAM:
            ds = dref_position_of_symbol(&x1, &x2, FALSE);
            phtd = phtd_parse(x1, x2, &ow1, &ow2, inline_flag);
            break;
        case DEFINING_AN_ADJECTIVE_RAM:
            phtd = phtd_new(DECIDES_CONDITION_MOR);
            break;
        default:
            phtd = phtd_new(DECIDES_NOTHING_AND_RETURNS_MOR);
            break;
    }
    if (ow1 >= 0) phod = phod_parse_declaration(ow1, ow2);
    else phod = phod_new();
    current_phrase = ph_new(&phud, &phtd, &phod, ds, inline_wn);
    switch (phud_get_RAM(&(current_phrase->usage_data))) {
        case PHRASE_DECLARATION_RAM:
            ph_To_make(current_phrase);
            break;
        case DEFINING_AN_ADJECTIVE_RAM: {
            int cw1, cw2;
            join_phrase_to_adjective(p, current_phrase, &cw1, &cw2);
            phsf_add_it_variable(&(current_phrase->stack_frame), cw1, cw2, kova(OBJECT_TY));
            break;
        }
        case RULE_IN_RULEBOOK_RAM:
        case RULE_NOT_IN_RULEBOOK_RAM:
            phud_create_primary_booking(&(current_phrase->usage_data),
                current_phrase);
            break;
    }
}
```

The function ph_create_from_preamble is called from 12/cs.

Phrase Usage 12/phud

Purpose

To parse the preamble of a phrase declaration to a `ph_usage_data` (PHUD) structure containing a still mostly textual representation of the conditions for its usage; to issue problem messages diagnosing bad syntax in these preambles; later, to parse further in order to convert the PHUD to a no longer textual PHRCD structure; finally, to use a PHUD to create a suitable primary booked rule (BR) for any phrase destined to be a rule, employing any predeclared BR for the phrase which may exist.

12/phud.§1 Construction; §2 How the PHUD translates into a PHRCD; §3 Problem messages for misphrased rule preambles; §4-5 Activation of rules

---

Definitions

¶1. "Attachment" is the process by which a phrase is placed into the data structures of the I6 program being compiled by NI. There are four ways this can be done:

```
define AS_YET_UNKNOWN_RAM 0                          used only temporarily during header parsing
define PHRASE_DECLARATION_RAM 1                      "To award (some - number) points: ..."
define RULE_IN_RULEBOOK_RAM 2                        "Before taking a container, ..."
define RULE_NOT_IN_RULEBOOK_RAM 3                    "At 9 PM: ...", "This is the zap rule: ..."
define DEFINING_AN_ADJECTIVE_RAM 4                   "Definition: a container is roomy if..."

typedef struct ph_usage_data {
    int full_preamble_w1, full_preamble_w2;          e.g. to identify nameless rules in the log
    int rule_attachment_mode;                        see below: basically, how the phrase is invoked
    int timing_of_event;                             one of two values defined below; or a specific time
    int explicit_name_w1, explicit_name_w2;          if a named rule, this is its name
    int whenwhile_w1, whenwhile_w2;                  when/while for action/activity rulebooks
    int rule_preamble_w1, rule_preamble_w2;
    int rule_parameter_w1, rule_parameter_w2;        text of object or action parameter
    struct specification *during_scene_spec;         what scene is currently under way
    int event_name_w1, event_name_w2;
    struct rulebook *owning_rulebook;                the primary booking for the phrase will be here
    int owning_rulebook_placement;                   ...and with this placement value: see Rulebooks
    int index_with_data_type;                        UNKNOWN, or the data type which created this
} ph_usage_data;
```

The structure ph_usage_data is private to this section.

¶2. When the following flag is set, we are parsing the header of a phrase to see if it might be a rule being attached to a rulebook, such as "Before taking a fish, ...". The flag is used to tell the parser that "in..." and "when..." have special meanings to do with providing circumstances for the rule to apply – this considerably cuts down on awkward ambiguities.

```
int rule_circumstances_mode = FALSE;
```

```
void log_ph_usage_data(ph_usage_data *phud) {
    char *ram = "<UNKNOWN>";
    switch(phud->rule_attachment_mode) {
        case AS_YET_UNKNOWN_RAM: ram = "AS_YET_UNKNOWN_RAM"; break;
        case DEFINING_AN_ADJECTIVE_RAM: ram = "DEFINING_AN_ADJECTIVE_RAM"; break;
        case RULE_NOT_IN_RULEBOOK_RAM: ram = "RULE_NOT_IN_RULEBOOK_RAM"; break;
        case PHRASE_DECLARATION_RAM: ram = "PHRASE_DECLARATION_RAM"; break;
        case RULE_IN_RULEBOOK_RAM: ram = "RULE_IN_RULEBOOK_RAM"; break;
    }
    LOG("PHUD: <$W>: rule attachment mode %s\n",
        phud->full_preamble_w1, phud->full_preamble_w2, ram);
    if (phud->explicit_name_w1 >= 0)
        LOG("  Explicit name: <$W>\n",
            phud->explicit_name_w1, phud->explicit_name_w2);
    if (phud->rule_preamble_w1 >= 0)
        LOG("  Rule preamble: <$W>\n",
            phud->rule_preamble_w1, phud->rule_preamble_w2);
    if (phud->rule_parameter_w1 >= 0)
        LOG("  Rule parameter: <$W>\n",
            phud->rule_parameter_w1, phud->rule_parameter_w2);
    if (phud->whenwhile_w1 >= 0)
        LOG("  When/while text: <$W>\n",
            phud->whenwhile_w1, phud->whenwhile_w2);
    if (phud->event_name_w1 >= 0)
        LOG("  Event name: <$W>\n",
            phud->event_name_w1, phud->event_name_w2);
    if (phud->timing_of_event != NOT_A_TIMED_EVENT)
        LOG("  Timed event: at %d\n", phud->timing_of_event);
    if (phud->during_scene_spec)
        LOG("  During scene: <$S>\n", phud->during_scene_spec);
    if (phud->owning_rulebook) {
        char *place = "<UNKNOWN>";
        LOG("  Owned by rulebook: ");
        rb_log_name_only(phud->owning_rulebook);
        switch(phud->owning_rulebook_placement) {
            case MIDDLE_PLACEMENT: place = "MIDDLE_PLACEMENT"; break;
            case FIRST_PLACEMENT: place = "FIRST_PLACEMENT"; break;
            case LAST_PLACEMENT: place = "LAST_PLACEMENT"; break;
        }
        LOG("\n  Placement: %s\n", place);
    }
    if (phud->index_with_data_type != UNKNOWN)
        LOG("  Index under kind of value: $s\n", phud->index_with_data_type);
}
```

```
void phud_log_rule_name(ph_usage_data *phud) {
    if (phud->explicit_name_w1 < 0) {
        if (phud->full_preamble_w1 >= 0)
            LOG("\"$W\"", phud->full_preamble_w1, phud->full_preamble_w2);
        else LOG("(nameless)");
    } else LOG("($W)", phud->explicit_name_w1, phud->explicit_name_w2);
}

void phud_write_I6_comment_describing(ph_usage_data *phud, OUTPUT_STREAM) {
    WRITE("! ");
    print_raw_text_within_i6_literal(OUT, phud->full_preamble_w1, phud->full_preamble_w2);
    WRITE(":\n");
}

int phud_phrase_index_area(ph_usage_data *phud) {
    if (phud->rule_attachment_mode != PHRASE_DECLARATION_RAM)
        return -1;
    return phud->index_with_data_type;
}

void phud_print_preamble_to(ph_usage_data *phud, OUTPUT_STREAM) {
    print_raw_text_to_file(phud->full_preamble_w1, phud->full_preamble_w2, OUT);
}
```

The function log_ph_usage_data is called from 2/dl.

The function phud_log_rule_name is called from 12/ph.

The function phud_write_I6_comment_describing is called from 12/cph and 12/br.

The function phud_phrase_index_area is called from 12/phin.

The function phud_print_preamble_to is called from 12/tiph.

§1. **Construction.** The following construction function produces a phud given the text of a phrase's preamble. It is designed so that it can be used as often as necessary in "coarse mode", but should be run once and once only on any given phrase when in "fine mode".

In its coarse mode, it uses only lexical data, and thus can work very early in NI's run – before it is known what the rulebooks, scenes and so forth will be called, for instance. Coarse mode does not provide detailed usage information for `RULE_IN_RULEBOOK_RAM` rules, but it does get everything else right. (This is only possible because an ambiguity was removed in December 2006, removing the possibility of "when" introducing rules of two different RAMs – see below.)

```
int no_now_phrases = 0;

ph_usage_data phud_new(int w1, int w2, int coarse_mode) {
    ph_usage_data phud;
    rulebook_match rm;
    int i, rw1, rw2, without_when_w1 = -1, without_when_w2 = -1;
    phud.full_preamble_w1 = w1; phud.full_preamble_w2 = w2;
    phud.rule_attachment_mode = AS_YET_UNKNOWN_RAM;
    phud.explicit_name_w1 = -1; phud.explicit_name_w2 = -1;
    phud.rule_preamble_w1 = -1; phud.rule_preamble_w2 = -1;
    phud.rule_parameter_w1 = -1; phud.rule_parameter_w2 = -1;
    phud.whenwhile_w1 = -1; phud.whenwhile_w2 = -1;
    phud.during_scene_spec = NULL;
    phud.event_name_w1 = -1; phud.event_name_w2 = -1;
    phud.timing_of_event = NOT_A_TIMED_EVENT;
    phud.owning_rulebook = NULL;
    phud.owning_rulebook_placement = MIDDLE_PLACEMENT;
    phud.index_with_data_type = UNKNOWN;
```

```
if [[w1, w2 == definition]] {
    phud.rule_attachment_mode = DEFINING_AN_ADJECTIVE_RAM;
    return phud;
}
if [[w1, w2 == this is the ... rule --> w1, w2]] {
    phud.rule_attachment_mode = RULE_NOT_IN_RULEBOOK_RAM;
    phud.explicit_name_w1 = w1;
    phud.explicit_name_w2 = w2+1;
    phud_verify_rule_name(phud.explicit_name_w1, phud.explicit_name_w2);
    return phud;
}
if [[w1, w2 == at ...]] {
    phud.rule_attachment_mode = RULE_NOT_IN_RULEBOOK_RAM;
    if (is_kova(is_a_literal(w1+1, w2), TIME_TY)) {
        phud.timing_of_event = last_literal_evaluated;
    } else {
        if [[w1, w2 == at the time when ... --> w1, w2]] {
            [[w1, w2 == the ... --> w1, w2]];
            phud.event_name_w1 = w1;
            phud.event_name_w2 = w2;
        } else {
        sentence_problem(_P_(C12AtWithoutTime),
            "'at' what time? No description of a time is given",
            "which means that this rule can never have effect. "
            "(The convention is that any rule beginning 'At' "
            "is a timed one. The time can either be a fixed time, "
            "as in 'At 11:10 AM: ...', or the time when some named "
            "event takes place, as in 'At the time when the clock "
            "chimes: ...'.)");
        }
        phud.timing_of_event = NO_FIXED_TIME;
    }
    return phud;
}
if [[w1, w2 == to]] {
    sentence_problem(_P_(C12BareTo),
        "'to' what? No name is given",
        "which means that this would not define a new phrase.");
}
if [[w1, w2 == to ...]] {
    phud.rule_attachment_mode = PHRASE_DECLARATION_RAM;
    if [[w1, w2 == ... DASH ###]] {
        if (coarse_mode == FALSE)
            phud.index_with_data_type = atoi(lw_array[w2].lw_rawtext);
        w2 -= 2;
    }
    if [[w1, w2 == to now ***]] {
        if ((coarse_mode) && (no_now_phrases++ == 1)) {
            sentence_problem(_P_(C12RedefinedNow),
                "creating new variants on 'now' is not allowed",
                "because 'now' plays a special role in the language. "
                "It has a wide-ranging ability to make a condition "
```

```
                    "become immediately true. (To give it wider abilities, "
                    "the idea is to create new relations.)");
            }
        }
        phud.rule_preamble_w1 = w1;
        phud.rule_preamble_w2 = w2;
        return phud;
    }
    phud.rule_attachment_mode = RULE_IN_RULEBOOK_RAM;
    if ((w2 >= w1+6) && [[w1, w2 == ... rule CLOSEBRACKET]]) {
        int x;
        for (x=w1+1; x<=w2-3; x++) {
            if [[x, w2 == OPENBRACKET this is the ...]] {
                phud.explicit_name_w1 = x+4;
                phud.explicit_name_w2 = w2-1;
                phud_verify_rule_name(
                    phud.explicit_name_w1, phud.explicit_name_w2);
                w2 = x-1;
                break;
            }
        }
    }
    if (coarse_mode) return phud;
    i = is_word_intermediate(during_V, w1, w2);
    if (i>=0) {
        quantity *q;
        specification *spec = parse_expression(i+1, w2, VALUE_EXPCON);
        if (((q = spec_get_constant_quantity_if_any(spec)) && (qty_is_a_scene_name(q))) ||
            ((species_is(spec, DESCRIPTION_SPC)) && (is_kova(spec_get_described_kov(spec), SCENE_TY))))
{
            phud.during_scene_spec = spec;
            w2 = i-1;
        }
    }
    i = is_word_intermediate(while_V, w1, w2);
    if (i<0) i = is_word_intermediate(when_V, w1, w2);
    if (i>=0) {
        rm = rb_match_from_description(w1, w2);
        without_when_w1 = w1;
        without_when_w2 = w2;
        phud.whenwhile_w1 = i;                                  include word "when/while" for indexing
        phud.whenwhile_w2 = w2;
        w2 = i-1;
    }
    LookForRulebook:
    rw1 = w1; rw2 = w2;
    [[rw1, rw2 == ... rule --> rw1, rw2]];
    rm = rb_match_from_description(rw1, rw2);
    if ((rm.matched_rulebook == NULL) &&
        [[rw1, rw2 == rule ... --> rw1, rw2]]) {
        rm = rb_match_from_description(rw1, rw2);
        if ((rm.matched_rulebook == NULL) &&
```

```
        [[rw1, rw2 == for ... --> rw1, rw2]])
        rm = rb_match_from_description(rw1, rw2);
}
if ((rm.matched_rulebook == NULL) && (without_when_w1 >= 0)) {
    w1 = without_when_w1; w2 = without_when_w2;
    without_when_w1 = -1; without_when_w2 = -1;
    phud.whenwhile_w1 = -1; phud.whenwhile_w2 = -1;
    goto LookForRulebook;
}
phud.owning_rulebook = rm.matched_rulebook;
if (phud.owning_rulebook == NULL) {
    if [[rw1, rw2 == when ***]] {
        quote_source(1, current_sentence);
        handmade_problem(_P_(C12BadRulePreambleWhen));
        issue_problem_segment(
            "The punctuation makes me think %1 should be a definition "
            "of a phrase or a rule, but it doesn't begin as it should, "
            "with either 'To' (e.g. 'To flood the riverplain:'), 'Definition:', "
            "a name for a rule (e.g. 'This is the devilishly cunning rule:'), "
            "'At' plus a time (e.g. 'At 11:12 PM:' or 'At the time when "
            "the clock chimes:') or the name of a rulebook. %P"
            "As your rule begins with 'When', it may be worth noting that in "
            "December 2006 the syntax used by Inform for timed events changed: "
            "the old syntax 'When the sky falls in:' to create a named "
            "event, the sky falls in, became 'At the time when the sky "
            "falls in:'. This was changed to avoid confusion with rules "
            "relating to when scenes begin or end. %P"
            "Or perhaps you meant to say that something would only happen "
            "when some condition held. Inform often allows this, but the "
            "'when...' part tends to be at the end, not up front - for "
            "instance, 'Understand \"blue\" as the deep crevasse when the "
            "location is the South Pole.'");
        issue_problem_end();
    } else
        sentence_problem(_P_(C12BadRulePreamble),
            "the punctuation here ':' makes me think this should be a definition "
            "of a phrase and it doesn't begin as it should",
            "with either 'To' (e.g. 'To flood the riverplain:'), 'Definition:', "
            "a name for a rule (e.g. 'This is the devilishly cunning rule:'), "
            "'At' plus a time (e.g. 'At 11:12 PM:' or 'At the time when "
            "the clock chimes') or the name of a rulebook followed by some "
            "description of the current action (e.g. 'Instead of taking "
            "something:').");
} else {
    int bridge_wn = rw1 + rm.match_length;
    if ((rm.article_used == DEF_ART) &&
        (rm.placement_requested == MIDDLE_PLACEMENT)) {
        sentence_problem(_P_(C12RuleWithDefiniteArticle),
            "a rulebook can contain any number of rules",
            "so (e.g.) 'the before rule: ...' is disallowed; you should "
            "write 'a before rule: ...' instead.");
    }
    phud.owning_rulebook_placement = rm.placement_requested;
```

```
        if [[word bridge_wn == of/for]] bridge_wn++;
        else if [[word bridge_wn == rule]] {
            bridge_wn++;
            if [[word bridge_wn == about/for/on]] bridge_wn++;
        }
        phud.rule_parameter_w1 = bridge_wn;
        phud.rule_parameter_w2 = w2;
        if (phud.rule_parameter_w1 > phud.rule_parameter_w2) {
            phud.rule_parameter_w1 = -1;
            phud.rule_parameter_w2 = -1;
        }
    }
    phud.rule_preamble_w1 = w1;
    phud.rule_preamble_w2 = w2;
    return phud;
}
int C12RuleWithComma_issued_at = -1;
void phud_verify_rule_name(int w1, int w2) {
    int i;
    for (i=w1; i<=w2; i++)
        if [[word i == COMMA]] {
            if (C12RuleWithComma_issued_at != w1) {
                C12RuleWithComma_issued_at = w1;
                sentence_problem(_P_(C12RuleWithComma),
                    "a rule name is not allowed to contain a comma",
                    "because this leads to too much ambiguity later on.");
            }
        }
}
int phud_get_preamble_w1(ph_usage_data *phud) {
    if (phud->rule_attachment_mode == PHRASE_DECLARATION_RAM)
        return phud->rule_preamble_w1;
    return phud->full_preamble_w1;
}
int phud_get_preamble_w2(ph_usage_data *phud) {
    if (phud->rule_attachment_mode == PHRASE_DECLARATION_RAM)
        return phud->rule_preamble_w2;
    return phud->full_preamble_w2;
}
int phud_get_RAM(ph_usage_data *phud) {
    return phud->rule_attachment_mode;
}
int phud_get_timing_of_event(ph_usage_data *phud) {
    return phud->timing_of_event;
}
void phud_set_italicised_index_text(ph_usage_data *phud, booked_rule *br) {
    if ((phud->whenwhile_w1 >= 0) &&
        (phud->whenwhile_w1 == phud->rule_parameter_w2 + 1)) {
        br_set_italicised_index_text(br,
            phud->rule_parameter_w1, phud->whenwhile_w2);
    } else {
        if (phud->whenwhile_w1 >= 0)
```

```
            br_set_italicised_index_text(br,
                phud->whenwhile_w1, phud->whenwhile_w2);
        else
            br_set_italicised_index_text(br,
                phud->rule_parameter_w1, phud->rule_parameter_w2);
    }
}
```

The function phud_new is called from 12/ph.
The function phud_verify_rule_name is called from 12/br.
The function phud_get_preamble_w1 is called from 12/ph.
The function phud_get_preamble_w2 is called from 12/ph.
The function phud_get_RAM is called from 12/ph.
The function phud_get_timing_of_event is called from 12/tiph.

§2. How the PHUD translates into a PHRCD.

```
ph_runtime_context_data phrcd_from_phud(ph_usage_data *phud) {
    ph_runtime_context_data phrcd = phrcd_new();
    phrcd.compile_for_rulebook = NULL;
    if (phud->owning_rulebook) {
        phrcd.compile_for_rulebook = phud->owning_rulebook;
        if ((rb_runs_during_activities(phud->owning_rulebook) == FALSE) &&
            (rb_focus(phud->owning_rulebook) == ACTION_FOCUS)) {
            if ((phud->rule_parameter_w1 >= 0) && (phud->whenwhile_w1 >= 0)) {
                phud->rule_parameter_w2 = phud->whenwhile_w2;
                phud->whenwhile_w1 = -1;
                phud->whenwhile_w2 = -1;
            }
        }
    }
    if (phud->rule_attachment_mode == RULE_NOT_IN_RULEBOOK_RAM)
        phrcd.permit_all_outcomes = TRUE;
    if (phud->rule_attachment_mode != RULE_IN_RULEBOOK_RAM) return phrcd;
    if (phud->rule_parameter_w1 >= 0) {
        int aw1 = phud->rule_parameter_w1, aw2 = phud->rule_parameter_w2;
        action_pattern ap = new_action_pattern();
        rulebook *rule_usage = phud->owning_rulebook;
        if (rb_focus(rule_usage) == ACTION_FOCUS) {
            rule_circumstances_mode = TRUE;
            permit_trying_omission = TRUE;
            phsf_set_stvol(current_stack_frame(), all_action_processing_vars);
            ap = parse_action_pattern(aw1, aw2, IS_TENSE);
            phsf_remove_nonphrase_stack_frame();
            rule_circumstances_mode = FALSE;
            permit_trying_omission = FALSE;
            if (ap_is_valid(&ap) == FALSE) {
                log_ph_usage_data(phud);
                issue_bad_rule_action_problem(&ap, aw1, aw2);
            }
        } else {
            ap = parse_parametric_action_pattern(aw1, aw2, IS_TENSE);
```

```
            if (ap_is_valid(&ap) == FALSE) {
                log_ph_usage_data(phud);
                issue_bad_rule_parameter_problem(&ap, aw1, aw2);
            }
        }
        phrcd.ap = ap;
    }
    if (phud->whenwhile_w1 >= 0) {
        phrcd.activity_context_w1 = phud->whenwhile_w1+1;
        phrcd.activity_context_w2 = phud->whenwhile_w2;
    }
    if (phud->during_scene_spec) phrcd.during_scene = phud->during_scene_spec;
    return phrcd;
}
```

The function phrcd_from_phud is called from 12/cs.

§**3. Problem messages for misphrased rule preambles.** This is one of the commonest-seen problem messages, so it's quite adaptive to what can superficially be detected of the actual problem.

```
int nap_error_explained = FALSE;
void issue_bad_rule_action_problem(action_pattern *ap,
    int action_text_w1, int action_text_w2) {
    action_name *an;
    LOG("Bad action pattern: $W = $A\nPAP failure reason: %d\n",
        action_text_w1, action_text_w2, ap, pap_failure_reason);
    if [[action_text_w1, action_text_w2 == in the presence of ...]] {
        handmade_problem(_P_(C12NonActionInPresenceOf));
        quote_source(1, current_sentence);
        quote_words(2, action_text_w1, action_text_w2);
        issue_problem_segment(
            "You wrote %1, but 'in the presence of...' is a clause which can "
            "only be used to talk about an action: so, for instance, 'waiting "
            "in the presence of...' is needed. "
            "This problem arises especially with 'every turn' rules, where "
            "'every turn in the presence of...' looks plausible but doesn't "
            "work. This could be fixed by writing 'Every turn doing something "
            "in the presence of...', but a neater solution talks about the "
            "current situation instead: 'Every turn when the player can "
            "see...'.");
        issue_problem_end();
        return;
    }
    if [[action_text_w1, action_text_w2 == in ...]] {
        handmade_problem(_P_(C12NonActionIn));
        quote_source(1, current_sentence);
        quote_words(2, action_text_w1, action_text_w2);
        issue_problem_segment(
            "You wrote %1, but 'in...' used in this way should really belong "
            "to an action: for instance, 'Before waiting in the Library'. "
            "Rules like 'Every turn in the Library' don't work, because "
            "'every turn' is not an action; what's wanted is 'Every turn "
            "when in the Library'.");
```

```
    issue_problem_end();
    return;
}
quote_source(1, current_sentence);
quote_words(2, action_text_w1, action_text_w2);
switch(pap_failure_reason) {
    case MIXEDNOUNS_PAPF:
        handmade_problem(_P_(C12APWithDisjunction));
        issue_problem_segment(
            "You wrote %1, which seems to introduce a rule, but the "
            "circumstances ('%2') seem to be too general for me to "
            "understand in a single rule. I can understand a choice of "
            "of actions, in a list such as 'taking or dropping the ball', "
            "but there can only be one set of noun(s) supplied. So 'taking "
            "the ball or taking the bat' is disallowed. You can get around "
            "this by using named actions ('Taking the ball is being "
            "mischievous. Taking the bat is being mischievous. Instead of "
            "being mischievous...'), or it may be less bother just to "
            "write more than one rule.");
        issue_problem_end();
        break;
    case WHEN_PAPF:
        handmade_problem(_P_(C12APWithBadWhen));
        quote_text(3, "so I am unable to accept this rule.");
        if (is_word_intermediate_inc(nothing_V,
            action_text_w1, action_text_w2) >= 0) {
            quote_text(3, "perhaps because 'nothing' tends "
                "not to be allowed in Inform conditions? "
                "(Whereas 'no thing' is usually allowed.)");
        }
        if (is_word_intermediate_inc(nowhere_V,
            action_text_w1, action_text_w2) >= 0) {
            quote_text(3, "perhaps because 'nowhere' tends "
                "not to be allowed in Inform conditions? "
                "(Whereas 'no room' is usually allowed.)");
        }
        issue_problem_segment(
            "You wrote %1, which seems to introduce a rule taking effect "
            "only '%2'. But this condition did not make sense, %3");
        issue_problem_end();
        break;
    case NOPARTICIPLE_PAPF:
        handmade_problem(_P_(C12APWithNoParticiple));
        issue_problem_segment(
            "You wrote %1, which seems to introduce a rule taking effect "
            "only '%2'. But this does not look like an action, since "
            "there is no sign of a participle ending '-ing' (as in "
            "'taking the brick', say) - which makes me think I have "
            "badly misunderstood what you intended.");
        issue_problem_end();
        break;
    case IMMISCIBLE_PAPF:
        handmade_problem(_P_(C12APWithImmiscible));
```

```
    issue_problem_segment(
        "You wrote %1, which seems to introduce a rule taking effect "
        "only '%2'. But this is a combination of actions which cannot "
        "be mixed. The only alternatives where 'or' is allowed are "
        "cases where a choice of actions is given but applying to "
        "the same objects in each case. (So 'taking or dropping the "
        "CD' is allowed, but 'dropping the CD or inserting the CD "
        "into the jewel box' is not, because the alternatives there "
        "would make different use of objects from each other.)");
    issue_problem_end();
    break;
default: {
    int at1, at2;
    quote_words(2, action_text_w1, action_text_w2);
    if (pap_failure_reason == WHENOKAY_PAPF)
        quote_text(3,
            "The part after 'when' (or 'while') was fine, "
            "but the earlier words");
    else quote_text(3, "But that");
    handmade_problem(_P_(C12APUnknown));
    issue_problem_segment(
        "You wrote %1, which seems to introduce a rule taking effect "
        "only if the action is '%2'. %3 did not make sense as a "
        "description of an action.");
    if (ap_trying_omission_position >= 0) {
        quote_words(4, action_text_w1, ap_trying_omission_position-1);
        quote_words(5, ap_trying_omission_position, action_text_w2);
        issue_problem_segment(
            " (I wonder if it is possible that this arose because I "
            "read '%4' as a description of the performer(s) of the "
            "action '%5'?)");
    }
    LOOP_OVER(an, action_name) {
        int x1 = an->word_ref1, x2 = an->word_ref2;
        if (action_text_w2-action_text_w1 < x2-x1) {
            if (compare_word_range(action_text_w1, action_text_w2,
                x1, x1 + action_text_w2 - action_text_w1)) {
                quote_words(3, x1, x2);
                issue_problem_segment(
                    " I notice that there's an action called '%3', "
                    "though: perhaps this is what you meant?");
                break;
            }
        }
    }
    at1 = action_text_w1; at2 = action_text_w2;
    if ([[at1, at2 == doing something/anything other than ... --> at1, at2]]
        || [[at1, at2 == doing something/anything except ... --> at1, at2]]) {
        issue_problem_segment(
            " This looks like a list of actions to avoid: ");
        nap_error_explained = FALSE;
        issue_ap_error_bit(at1, at2);
        issue_problem_segment(
```

```
                    " so");
            } else
            if (is_list_divided(action_text_w1, action_text_w2, LOOK_FOR_OR)) {
                issue_problem_segment(
                    " Looking at this as a list of alternative actions: ");
                nap_error_explained = FALSE;
                issue_ap_error_bit(action_text_w1, action_text_w2);
                issue_problem_segment(
                    " so");
            }
            issue_problem_segment(
                " I am unable to place this rule into any rulebook.");
            issue_problem_end();
            break;
        }
    }
}
void issue_ap_error_bit(int w1, int w2) {
    action_pattern ap; int i;
    if [[w1, w2 == ... when ... : i]] w2 = i-1;
    if (is_list_divided(w1, w2, LOOK_FOR_OR)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        issue_ap_error_bit(lw1, lw2);
        issue_ap_error_bit(rw1, rw2);
        return;
    }
    ap = parse_action_pattern(w1, w2, IS_TENSE);
    LOG("AP clause $W = $A\n", w1, w2, &ap);
    quote_words(4, w1, w2);
    if (ap_is_valid(&ap) == FALSE) {
        issue_problem_segment("'%4' did not make sense; ");
        return;
    }
    if (ap_is_request(&ap)) {
        issue_problem_segment(
            "'%4' would make sense as an action on its own, but 'or' can't "
            "be used in combination with 'asking... to try...' actions; ");
        return;
    }
    if (nap_by_name(w1, w2)) {
        if (nap_error_explained == FALSE)
            issue_problem_segment(
                "'%4' only made sense as a named kind of action, which can "
                "be used on its own but not in an action list; ");
        else
            issue_problem_segment(
                "'%4' is another named kind of action; ");
        nap_error_explained = TRUE;
    } else issue_problem_segment("'%4' was okay; ");
}
void issue_bad_rule_parameter_problem(action_pattern *ap,
    int parameter_w1, int parameter_w2) {
    LOG("Pseudo-AP: $A\nOffending parameter spec: <$W>\n",
```

```
        ap, parameter_w1, parameter_w2);
    quote_source(1, current_sentence);
    quote_words(2, parameter_w1, parameter_w2);
    handmade_problem(_P_(C12BadParameter));
    issue_problem_segment(
        "You wrote %1, but the description of the thing(s) to which the rule "
        "applies ('%2') did not make sense, and should have been something "
        "like 'the perspex box' or 'a closed container'.");
    issue_problem_end();
}
```

§4. Activation of rules.

```
void phud_create_primary_booking(ph_usage_data *phud, phrase *ph) {
    int rw1, rw2;
    booked_rule *br, *br_found = NULL;
    if (phud->event_name_w1 >= 0) {
        rw1 = phud->event_name_w1; rw2 = phud->event_name_w2;
    } else {
        rw1 = phud->explicit_name_w1; rw2 = phud->explicit_name_w2;
    }
    if (rw1 >= 0) {
        [[rw1, rw2 == the ... --> rw1, rw2]];
        br_found = br_by_name(rw1, rw2);
    }
    if (br_found) {
        phrase *ph_found = br_get_rule(br_found);
        if ((ph_found) && (ph_found != ph)) {
            quote_source(1, current_sentence);
            quote_words(2, rw1, rw2);
            handmade_problem(_P_(C12DuplicateRuleName));
            issue_problem_segment(
                "You wrote %1, but this would give a name ('%2') to a "
                "new rule which already belongs to an existing one.");
            issue_problem_end();
            LOG("Duplicate names <$W>\n", rw1, rw2);
        }
        br = br_found;
    } else {
        br = br_new(ph);
        if (rw1 >= 0) br_set_name(br, rw1, rw2);
    }
    br_set_primary_rule(br, ph);
    phud_set_italicised_index_text(phud, br);
}
```

The function phud_create_primary_booking is called from 12/ph.

§**5.** Having long ago decided where and how to place the phrase into a rulebook, we finally get the opportunity to do this. The BR supplied must be the one generated from the PHUD elsewhere.

```
void phud_place_in_rulebook(ph_usage_data *phud, booked_rule *br) {
    if (phud->rule_attachment_mode == RULE_IN_RULEBOOK_RAM) {
        rulebook *original_owner = phud->owning_rulebook;
        if (rb_requires_specific_action(original_owner)) {
            int w1 = phud->rule_parameter_w1, w2 = phud->rule_parameter_w2;
            int waiver = FALSE;
            action_name_list *anl;
            action_name *an;
            if (w1 >= 0) {
                int e1 = w1, e2 = w2, i, x1, x2;
                [[e1, e2 == ... when ... : i --> e1, e2 ... x1, x2]];
                [[e1, e2 == ... while ... : i --> e1, e2 ... x1, x2]];
                for (i=e1; i<=e2; i++)
                    if (nap_by_name(i, e2))
                        goto NotSingleAction;
                anl = anl_extract_actions_only(e1, e2);
                an = anl_get_single_action(anl);
                br_set_marked_for_anyone(br, anl_get_explicit_anyone_flag(anl));
            } else {
                anl = NULL;
                an = NULL;
                waiver = TRUE;
                if (original_owner == built_in_rulebooks[CHECK_RB]) waiver = FALSE;
                if (original_owner == built_in_rulebooks[CARRY_OUT_RB]) waiver = FALSE;
                if (original_owner == built_in_rulebooks[REPORT_RB]) waiver = FALSE;
            }
            LOGIF(RULE_ATTACHMENTS, "BR is: $b\n AN is: $l\n", br, an);
            if ((an == NULL) && (waiver == FALSE)) {
                int x;
                an = act_longest_null(w1, w2, IS_TENSE, &x);
            }
            if ((an == NULL) && (waiver == FALSE)) {
                NotSingleAction:
                quote_source(1, current_sentence);
                quote_words(2, w1, w2);
                handmade_problem(_P_(C12MultipleCCR));
                issue_problem_segment(
                    "You wrote %1, but the situation this refers to ('%2') is "
                    "not a single action. Rules in the form of 'check', 'carry "
                    "out' and 'report' are tied to specific actions, and must "
                    "give a single explicit action name - even if they then go "
                    "on to very complicated conditions about any nouns also "
                    "involved. So 'Check taking something: ...' is fine, "
                    "but not 'Check taking or dropping something: ...' or "
                    "'Check doing something: ...' - the former names two "
                    "actions, the latter none.");
                issue_problem_end();
            } else {
                if (original_owner == built_in_rulebooks[CHECK_RB]) {
                    phud->owning_rulebook =
```

```
                    an_get_fragmented_rulebook(an, built_in_rulebooks[CHECK_RB]);
                } else
                if (original_owner == built_in_rulebooks[CARRY_OUT_RB]) {
                    phud->owning_rulebook =
                        an_get_fragmented_rulebook(an, built_in_rulebooks[CARRY_OUT_RB]);
                } else
                if (original_owner == built_in_rulebooks[REPORT_RB]) {
                    phud->owning_rulebook =
                        an_get_fragmented_rulebook(an, built_in_rulebooks[REPORT_RB]);
                } else {
                    phud->owning_rulebook =
                        an_switch_fragmented_rulebook(an, original_owner);
                }
                if (original_owner != phud->owning_rulebook)
                    LOGIF(RULE_ATTACHMENTS, "Rerouting $b to $K\n", br,
                        phud->owning_rulebook);
            }
        }
        rb_attach_rule(phud->owning_rulebook, br,
            phud->owning_rulebook_placement, 0, NULL);
    }
}

void phud_predeclare_name_in(parse_node *p) {
    ph_usage_data phud = phud_new(p->word_ref1, p->word_ref2, TRUE);
    if (phud.explicit_name_w1 >= 0) {
        booked_rule *br = br_new(NULL);
        br_predeclare(br, phud.explicit_name_w1, phud.explicit_name_w2);
    }
}
```

The function phud_place_in_rulebook is called from 12/br.

The function phud_predeclare_name_in is called from 12/cs.

Phrase Runtime Context Data 12/phrcd

Purpose

To create, logically compare the specificity of, and compile I6 tests which check the current validity of, the circumstances in which a rule phrase should be used. All interesting PHRCD structures are manufactured from PHUDs, however, and the code for that is found in the Phrase Usage section.

12/phrcd.§1 Specificity of phrase runtime contexts

Definitions

¶1. Runtime context data is the context in which a phrase is allowed to run. It is deduced from the PHUD (and the code is inherited from 12/phud); it is tested at runtime in the phrase's defining routine.

```
typedef struct ph_runtime_context_data {
    int activity_context_w1, activity_context_w2;          happens only while any activities go on?
    struct specification *during_scene;                    ...happens only during any particular scene?
    struct action_pattern ap;                              happens only if the action matches this pattern?
    int always_test_actor;                                 ...even if no AP was given, test that actor is player?
    int never_test_actor;                                  ...for instance, for a parametrised rather than action rulebook
    struct rulebook *compile_for_rulebook;                 ...used for the default outcome
    int permit_all_outcomes;                               waive the usual restrictions on rule outcomes
} ph_runtime_context_data;
```

The structure ph_runtime_context_data is shared with 12/phud.

```
ph_runtime_context_data phrcd_new(void) {
    ph_runtime_context_data phrcd;
    phrcd.activity_context_w1 = -1;
    phrcd.activity_context_w2 = -1;
    phrcd.during_scene = NULL;
    phrcd.ap = new_action_pattern();
    phrcd.always_test_actor = FALSE;
    phrcd.never_test_actor = FALSE;
    phrcd.permit_all_outcomes = FALSE;
    phrcd.compile_for_rulebook = NULL;
    return phrcd;
}
scene *phrcd_get_scene(ph_runtime_context_data *phrcd) {
    if (phrcd == NULL) return NULL;
    if (family_is(phrcd->during_scene, VALUE_FMY)) {
        quantity *q = spec_get_constant_quantity_if_any(phrcd->during_scene);
        if ((q) && (qty_is_a_scene_name(q))) return read_scene_from_quantity(q);
    }
    return NULL;
}
int phrcd_within_action_context(ph_runtime_context_data *phrcd,
    action_name *an) {
    if (phrcd == NULL) return FALSE;
    return ap_within_action_context(&(phrcd->ap), an);
}
```

```
action_name *phrcd_required_action(ph_runtime_context_data *phrcd) {
    if (ap_is_valid(&(phrcd->ap)))
        return ap_required_action(&(phrcd->ap));
    return NULL;
}
void phrcd_set_always_test_actor(ph_runtime_context_data *phrcd) {
    phrcd->always_test_actor = TRUE;
}
void phrcd_clear_always_test_actor(ph_runtime_context_data *phrcd) {
    phrcd->always_test_actor = FALSE;
}
void phrcd_set_never_test_actor(ph_runtime_context_data *phrcd) {
    phrcd->never_test_actor = TRUE;
}
void phrcd_suppress_action_testing(ph_runtime_context_data *phrcd) {
    ap_suppress_action_testing(&(phrcd->ap));
}
int outcome_restrictions_waived(void) {
    if (phrase_being_compiled == NULL) return FALSE;
    if (phrase_being_compiled->runtime_context_data.permit_all_outcomes)
        return TRUE;
    return FALSE;
}
int phrcd_is_for_rulebook(ph_runtime_context_data *phrcd) {
    if (phrcd->compile_for_rulebook) return TRUE;
    return FALSE;
}
```

The function phrcd_new is called from 12/ph and 12/phud.
The function phrcd_get_scene is called from 12/br.
The function phrcd_within_action_context is called from 12/br.
The function phrcd_required_action is called from 12/br.
The function phrcd_set_always_test_actor is called from 12/br.
The function phrcd_clear_always_test_actor is called from 12/br.
The function phrcd_set_never_test_actor is called from 12/br.
The function phrcd_suppress_action_testing is called from 12/br.
The function outcome_restrictions_waived is called from 12/rb.

§1. **Specificity of phrase runtime contexts.** The following is one of NI's standardised comparison routines, which takes a pair of objects A, B and returns 1 if A makes a more specific description than B, 0 if they seem equally specific, or −1 if B makes a more specific description than A. This is transitive, and intended to be used in sorting algorithms.

```
int compare_specificity_of_phrcd(ph_runtime_context_data *rcd1,
    ph_runtime_context_data *rcd2) {
    specification *sc1 = NULL, *sc2 = NULL;
    int al1 = -1, al2 = -1;
    action_pattern *ap1 = NULL, *ap2 = NULL;
    int rct1, rct2;
    int rv;
    if (rcd1) {
        ap1 = &(rcd1->ap);
        sc1 = rcd1->during_scene;
```

```
        al1 = rcd1->activity_context_w1;
    }
    if (rcd2) {
        ap2 = &(rcd2->ap);
        sc2 = rcd2->during_scene;
        al2 = rcd2->activity_context_w1;
    }
    c_s_stage_law = "I - Number of aspects constrained";
    rct1 = ap_count_aspects(ap1); rct2 = ap_count_aspects(ap2);
    if (al1 >= 0) rct1++; if (al2 >= 0) rct2++;
    if (sc1) rct1++; if (sc2) rct2++;
    if (rct1 > rct2) return 1;
    if (rct1 < rct2) return -1;
    if ((sc1) && (sc2)) {
        rv = compare_specificity_of_spec(sc1, sc2);
        if (rv != 0) return rv;
    }
    c_s_stage_law = "II - When/while requirement";
    if ((al1 >= 0) && (al2 == -1)) return 1;
    if ((al1 == -1) && (al2 >= 0)) return -1;
    c_s_stage_law = "III - Action requirement";
    rv = compare_specificity_of_ap(ap1, ap2);
    if (rv != 0) return rv;
    c_s_stage_law = "IV - Scene requirement";
    if ((sc1 != NULL) && (sc2 == NULL)) return 1;
    if ((sc1 == NULL) && (sc2 != NULL)) return -1;
    return 0;
}
void phrcd_compile_test_head(ph_runtime_context_data *phrcd, OUTPUT_STREAM,
    char *identifier, phrase *ph) {
    int tests = 0;
    if (phrcd->during_scene) {
        code_indent(OUT, 1);
        if (family_is(phrcd->during_scene, VALUE_FMY)) {
            quantity *q = spec_get_constant_quantity_if_any(phrcd->during_scene);
            if (qty_is_a_scene_name(q)) {
                scene *sc = read_scene_from_quantity(q);
                WRITE("if (scene_status-->%d == 1) { ! Runs only during scene\n",
                    sc->allocation_id);
            } else internal_error("non-scene in during");
        } else {
            WRITE("if (DuringSceneMatching(");
            coerce_DESCRIPTION_to_VALUE(phrcd->during_scene, TRUE);
            spec_compile(OUT, phrcd->during_scene);
            WRITE(")) { ! Runs only during scene\n");
        }
        tests++;
    }
    if (ap_is_valid(&(phrcd->ap))) {
        code_indent(OUT, 1);
```

```
        WRITE("if (");
        if (phrcd->never_test_actor)
            compile_action_pattern_match(OUT, phrcd->ap, TRUE);
        else
            compile_action_pattern_match(OUT, phrcd->ap, FALSE);
        WRITE(") { ! Runs only when pattern matches\n");
        tests++;
    } else {
        if (phrcd->always_test_actor == TRUE) {
            code_indent(OUT, 1);
            WRITE("if (actor == player) {\n");
            tests++;
        }
    }
    if (phrcd->activity_context_w1 >= 0) {
        activity_list *avl =
            parse_activity_list(phrcd->activity_context_w1, phrcd->activity_context_w2, TRUE);
        if (avl == NULL)
            sentence_problem(_P_(C12BadWhenWhile),
                "I don't understand the 'when/while' clause",
                "which should name activities or conditions.");
        else {
            code_indent(OUT, 1);
            WRITE("if ");
            compile_activity_list(OUT, avl);
            WRITE(" { ! Runs only while condition holds\n");
            avl_annotate_for_cross_references(avl, ph);
            tests++;
        }
    }
    if (tests > 0) {
        code_indent(OUT, 1);
        WRITE("if (debug_rules) DB_Rule(%s, %d);\n", identifier, ph->allocation_id);
    }
}
void phrcd_compile_test_tail(ph_runtime_context_data *phrcd,
    OUTPUT_STREAM, char *identifier, phrase *ph) {
    int tests = 0;
    if (phrcd->compile_for_rulebook)
        rb_compile_default_outcome(phrcd->compile_for_rulebook, OUT);
    if (phrcd->during_scene) {
        code_indent(OUT, 1);
        WRITE("} ! Runs only during scene\n");
        tests++;
    }
    if (ap_is_valid(&(phrcd->ap))) {
        code_indent(OUT, 1);
        WRITE("} ! Runs only when pattern matches\n");
        tests++;
    } else {
        if (phrcd->always_test_actor == TRUE) {
            code_indent(OUT, 1);
```

```
            WRITE("} ! Runs only when actor is the player\n");
            tests++;
        }
    }
    if (phrcd->activity_context_w1 >= 0) {
        code_indent(OUT, 1);
        WRITE("} ! Runs only while condition holds\n");
        tests++;
    }
    if (tests > 0) {
        code_indent(OUT, 1);
        WRITE("else if (debug_rules > 1) DB_Rule(%s, %d, true);\n",
            identifier, ph->allocation_id);
    }
}
```

The function compare_specificity_of_phrcd is called from 12/br.
The function phrcd_compile_test_head is called from 12/cph.
The function phrcd_compile_test_tail is called from 12/cph.

Phrase Type Data 12/phtd

Purpose

To create, manage, compare the logical specificity of, and assist excerpt parsing concerning, the type of a To phrase. This involves whether it is void, determines a condition or returns a value (and if so, what kind of value); and also what parameters it takes (possibly values, possible other types used by inline definitions) and their types in turn.

12/phtd.§8 Comparison of PHTDs; §9 Parsing of phrases; §10 Adding invocations to phrase SPs

---

Definitions

¶1. Phrases declared with the words "To say..." define new text substitution. The first to be defined is special: it will be "to say [value]". Others which implement control structures also need special treatment.

```
define NOT_A_SAY_PHRASE 0                    needs to be 0 so that say_phrase can be a C condition
define A_MISCELLANEOUS_SAY_PHRASE 1
define THE_PRIMORDIAL_SAY_PHRASE 2

define NO_SAY_CS 0
define IF_SAY_CS 1
define OTHERWISE_SAY_CS 2
define END_IF_SAY_CS 3
```

¶2. The `manner_of_return` field for a phrase's type shows what it does. The exceptional case is the last, `DECIDES_NOTHING_AND_RETURNS_MOR`, which marks out a phrase which exits the phrase it is part of the definition of – like the keyword `return` in a C function.

We enumerate from 2 here so that `UNKNOWN` can be used as a "don't know what the manner is" value.

```
define DECIDES_NOTHING_MOR 2                                   e.g., "award 4 points"
define DECIDES_VALUE_MOR 3                                     e.g., "square root of 16"
define DECIDES_CONDITION_MOR 4                                 e.g., "a random chance of 1 in 3 succeeds"
define DECIDES_NOTHING_AND_RETURNS_MOR 5                       e.g., "continue the action"
```

¶3. Only in the case of DECIDES_VALUE_MOR does the phrase produce data which can be used as a value, and in this event the type-checking code needs to know what the data type of the value is. Arithmetic phrases are polymorphic in this respect: that is, the type of the result of "X plus Y" depends on the types of X and Y. Other phrases always have definite data types, though, and these are recorded in the `return_type` field.

```
define MAX_TOKENS_PER_PHRASE 10
define MAX_WORDS_PER_PHRASE 32                                  the most the excerpt parser can hold anyway

typedef struct ph_type_data {
    int register_w1, register_w2;                               words used to register the excerpt meaning
    int manner_of_return;                                       one of the *_MOR values – see above
    struct specification *return_type;                          well, if not arithmetical
    int words[MAX_WORDS_PER_PHRASE];                            the "word sequence": see above
    int no_words;                                               length of the word sequence
    int tokens_w1[MAX_TOKENS_PER_PHRASE], tokens_w2[MAX_TOKENS_PER_PHRASE];   names
    struct specification *tokens_spec[MAX_TOKENS_PER_PHRASE];   type expected
    struct kind_of_value *tokens_template[MAX_TOKENS_PER_PHRASE];   for variable types
    int no_tokens;
    int say_phrase;                                             one of the three values above
    int say_phrase_running_on;                                  ignore implied newlines in previous invocation
    int say_phrase_stream_position;                             special position in say stream, if any
    int say_phrase_stream_token_at;                             word number of say stream token name
    int say_phrase_stream_closing_token_at;                     ditto for choice of ending
    int say_control_structure;                                  one of the four values above
    int let_phrase;                                             is this "let A be B", where A is as yet unknown?
    int includes_a_QUOT_token;                                  is one of the token types VALUE/QUOT?
    int block_follows;                                          for inline phrases only: followed by a begin... end block?
    int only_in_block_wn;                                       -1, or the name of the block this must occur inside
    int only_in_loop;                                           if TRUE, then the phrase can only be used in a loop body
    int arithmetical_operation;                                 -1, or one of the five operators – see Types
    int polymorphism_exception;                                 -1, or an exception to the type returned rules
    int assignment_exception;                                   TRUE if this has to be typechecked as an assignment
    int invoked_inline_not_as_call;                             TRUE, FALSE or NOT APPLICABLE for non-phrases
} ph_type_data;
```

The structure ph_type_data is shared with 7/tc, 7/inv, 12/phsf and 12/cinv.

¶4. A phrase whose invocations are compiled inline, like "award (some - a number) points", does have token names with scope local to its own schematic definition – short-lived as that is. It also has phrase options; but it has no local variables of its own, as such, since it has to live inside another routine. Instead, it has the right to claim new locals as needed: for instance, a phrase for a loop will need to claim at least one local as a loop counter, and indeed some loops claim two. The following variable holds the local variable number of whatever was most recently created.

```
int phrase_made_local = -1;
```

¶5. The say stream is the sequence of successive invocations in a single say, or in a piece of text with substitutions. Some say phrases are compound in that they must occur in a given sequence (for instance, say end if makes sense only if it follows a matching say if earlier in the stream, though for various reasons this isn't actually how that is implemented).

The following markers are used to identify such say phrases:

```
define SSP_NONE 0
define SSP_START 1
define SSP_MIDDLE 2
define SSP_END 3
```

¶6. The following codes describe the possible ways routines can return values: not at all, the standard way, or polymorphically – which is used for arithmetic operations, whose KOV is determined by the KOVs of the operands. These are not type IDs and cannot be used as such.

```
define NO_RETURN_RTY   1
define POLYMORPHIC_RTY 2
define STANDARD_RTY 3
```

---

§**1.**

```
char *describe_manner_of_return(int mor) {
    switch (mor) {
        case DECIDES_NOTHING_MOR: return "no value resulting";
        case DECIDES_VALUE_MOR: return "a phrase to decide a value";
        case DECIDES_CONDITION_MOR: return "a phrase to make a decision";
        case DECIDES_NOTHING_AND_RETURNS_MOR:
            return "a phrase providing an outcome to a rulebook";
    }
    return "some phrase";                                   should never actually be needed
}

ph_type_data phtd_new(int mor) {
    ph_type_data phtd;
    phtd.register_w1 = -1; phtd.register_w2 = -1;
    phtd.manner_of_return = mor;
    phtd.return_type = new_UNKNOWN_spec();
    phtd.no_words = 0;
    phtd.no_tokens = 0;
    phtd.let_phrase = FALSE;
    phtd.say_phrase = NOT_A_SAY_PHRASE;
    phtd.say_phrase_running_on = FALSE;
    phtd.say_control_structure = NO_SAY_CS;
    phtd.say_phrase_stream_position = SSP_NONE;
    phtd.say_phrase_stream_token_at = -1;
    phtd.say_phrase_stream_closing_token_at = -1;
    phtd.invoked_inline_not_as_call = NOT_APPLICABLE;
    phtd.block_follows = FALSE;
    phtd.includes_a_QUOT_token = FALSE;
    phtd.only_in_block_wn = -1;
    phtd.only_in_loop = FALSE;
```

```
    phtd.assignment_exception = FALSE;
    return phtd;
}
void log_ph_type_data(ph_type_data *phtd) {
    LOG("PHTD: register as <$W>\n  manner of return %d\n",
        phtd->register_w1, phtd->register_w2,
        phtd->manner_of_return);
    if (spec_is_UNKNOWN(phtd->return_type) == FALSE)
        LOG("  returns value of kind $S\n", phtd->return_type);
    if (phtd->no_words > 0) {
        int i;
        LOG("  ");
        for (i=0; i<phtd->no_words; i++)
            if (phtd->words[i] < MAX_TOKENS_PER_PHRASE) LOG("#%d ", phtd->words[i]);
            else LOG("%s ", lw_array[phtd->words[i]].lw_text);
        LOG("(%d words)\n", phtd->no_words);
    }
    if (phtd->no_tokens > 0) {
        int i;
        for (i=0; i<phtd->no_tokens; i++)
            LOG("  #%d: \"$W\" = $S\n", i,
                phtd->tokens_w1[i], phtd->tokens_w2[i], phtd->tokens_spec[i]);
    }
    if (phtd->includes_a_QUOT_token) LOG("  includes a QUOT token\n");
    if (phtd->block_follows) LOG("  block follows\n");
    if (phtd->let_phrase) LOG("  let phrase\n");
    if (phtd->only_in_block_wn >= 0)
        LOG("  may only be used in '%s'\n", lw_array[phtd->only_in_block_wn].lw_text);
    if (phtd->only_in_loop)
        LOG("  may only be used in loop body\n");
    switch (phtd->invoked_inline_not_as_call == TRUE) {
        case TRUE: LOG("  invoked inline\n"); break;
        case FALSE: LOG("  invoked by I6 function call\n"); break;
        case NOT_APPLICABLE: LOG("  never invoked\n"); break;
    }
    switch (phtd->say_phrase) {
        case NOT_A_SAY_PHRASE: break;
        case A_MISCELLANEOUS_SAY_PHRASE: LOG("  A_MISCELLANEOUS_SAY_PHRASE\n"); break;
        case THE_PRIMORDIAL_SAY_PHRASE: LOG("  THE_PRIMORDIAL_SAY_PHRASE\n"); break;
        default: LOG("  <invalid say phrase status>\n"); break;
    }
    if (phtd->say_phrase_running_on)
        LOG("  running on from previous say invocation without implied newline\n");
    switch (phtd->say_control_structure) {
        case NO_SAY_CS: break;
        case IF_SAY_CS: LOG("  IF_SAY_CS\n"); break;
        case END_IF_SAY_CS: LOG("  END_IF_SAY_CS\n"); break;
        case OTHERWISE_SAY_CS: LOG("  OTHERWISE_SAY_CS\n"); break;
        default: LOG("  <invalid say phrase status>\n"); break;
    }
}
void log_ph_type_data_briefly(ph_type_data *phtd) {
    LOG("\"$W\"", phtd->register_w1, phtd->register_w2);
```

```
    switch(phtd->manner_of_return) {
        case DECIDES_CONDITION_MOR:
            LOG("(=condition)"); break;
        case DECIDES_VALUE_MOR:
            LOG("(=$S)", phtd->return_type); break;
    }
}

int phtd_safe_for_runtime_resolution(ph_type_data *phtd) {
    int i;
    for (i=0; i<phtd->no_tokens; i++) {
        if (spec_get_storage_form(phtd->tokens_spec[i]) == PROPERTY_VALUE_SPC)
            return FALSE;
    }
    return TRUE;
}

int phtd_is_phrase_to_decide_if(ph_type_data *phtd) {
    if (phtd->manner_of_return == DECIDES_CONDITION_MOR) return TRUE;
    return FALSE;
}

int phtd_determine_return_modifier(ph_type_data *phtd) {
    if (phtd->manner_of_return != DECIDES_VALUE_MOR) return NO_RETURN_RTY;
    if (phtd->arithmetical_operation >= 0) return POLYMORPHIC_RTY;
    if (phtd->polymorphism_exception >= 0) return POLYMORPHIC_RTY;
    return STANDARD_RTY;
}

kind_of_value *phtd_get_return_kov(ph_type_data *phtd) {
    kind_of_value *kov;
    if ((species_is(phtd->return_type, DESCRIPTION_SPC))
        && (spec_get_described_kind(phtd->return_type)))
        kov = kovko(spec_get_described_kind(phtd->return_type));
    else
        kov = spec_evaluates_to(phtd->return_type);
    return kov;
}

specification *phtd_get_return_TS(ph_type_data *phtd) {
    return phtd->return_type;
}

void phtd_compile_default_return(OUTPUT_STREAM, ph_type_data *phtd,
    ph_runtime_context_data *phrcd) {
    if (species_is(phtd->return_type, CONSTANT_SPC)) {
        WRITE("return ");
        compile_default_value(OUT, spec_get_kind_of_value(phtd->return_type),
            -1, -1, "return value");
        WRITE(";");
        return;
    }
    if (spec_get_described_kind(phtd->return_type)) {
        WRITE("return ");
        compile_default_value(OUT, kovko(spec_get_described_kind(phtd->return_type)),
            -1, -1, "return value");
        WRITE(";");
        return;
```

```
    }
    WRITE("rfalse;");
}

int phtd_get_no_tokens(ph_type_data *phtd) {
    return phtd->no_tokens;
}

int phtd_is_a_say_X_phrase(ph_type_data *phtd) {
    if (phtd->say_phrase == NOT_A_SAY_PHRASE) return FALSE;
    if ((phtd->no_words == 2) && (phtd->words[1] < MAX_TOKENS_PER_PHRASE))
        return TRUE;
    return FALSE;
}

int phtd_ssp_matches(ph_type_data *phtd, int ssp_tok, int list_pos,
    int *w1, int *w2) {
    int this_tok = phtd->say_phrase_stream_token_at;
    int this_pos = phtd->say_phrase_stream_position;
    if (this_tok == -1) return FALSE;
    if (this_pos != list_pos) return FALSE;
    if (compare_words(ssp_tok, this_tok) == FALSE) return FALSE;
    *w1 = phtd->register_w1 + 1;
    *w2 = phtd->register_w2;
    return TRUE;
}

int phtd_is_a_let_V_phrase(ph_type_data *phtd) {
    if (phtd->let_phrase) {
        specification *first_operand = phtd->tokens_spec[0];
        if ((species_is(first_operand, NONLOCAL_VARIABLE_SPC)) ||
            (species_is(first_operand, NEW_LOCAL_VARIABLE_NAME_SPC)))
            return TRUE;
    }
    return FALSE;
}

int phtd_is_any_let_phrase(ph_type_data *phtd) {
    return phtd->let_phrase;
}

void phtd_write_HTML_representation(ph_type_data *phtd, char *to) {
    int j=0, f = FALSE, s = FALSE;
    to[0] = 0;
    if ((phtd->no_words > 0) && [[word phtd->words[0] == say]]) {
        j++; s = TRUE;
        sprintf(to, "say \"[");
    }
    for (; j<phtd->no_words; j++) {
        specification *spec;
        if (f) sprintf(to+strlen(to), " "); f = TRUE;
        if (phtd->words[j] < 10) {
            spec = phtd->tokens_spec[phtd->words[j]];
            if (s == FALSE) sprintf(to+strlen(to), "(");
            sprintf(to+strlen(to), "<i>");
            write_specification_out_in_English(spec, to+strlen(to));
            sprintf(to+strlen(to), "</i>");
            if (s == FALSE) sprintf(to+strlen(to), ")");
```

```
        } else sprintf(to+strlen(to),
            "%s", lw_array[phtd->words[j]].lw_rawtext);
    }
    if (phtd->block_follows) {
        sprintf(to+strlen(to),
            "; ...; end %s", lw_array[phtd->words[0]].lw_text);
    }
    if (s) {
        sprintf(to+strlen(to), "]\"");
    }
}
```

The function describe_manner_of_return is called from 12/cinv.
The function phtd_new is called from 12/ph.
The function log_ph_type_data is called from 2/dl.
The function log_ph_type_data_briefly is called from 12/ph.
The function phtd_safe_for_runtime_resolution is called from 12/ambig.
The function phtd_is_phrase_to_decide_if is called from 12/cinv.
The function phtd_determine_return_modifier is called from 7/vasp.
The function phtd_get_return_kov is called from 7/vasp and 7/tc.
The function phtd_get_return_TS is called from 12/cinv.
The function phtd_compile_default_return is called from 12/cph.
The function phtd_get_no_tokens is called from 7/inv.
The function phtd_is_a_say_X_phrase is called from 7/tc and 12/phin.
The function phtd_ssp_matches is called from 12/ph.
The function phtd_is_a_let_V_phrase is called from 12/phin.
The function phtd_is_any_let_phrase is called from 7/tc.
The function phtd_write_HTML_representation is called from 12/ph.

§**2.** This is a rather cool depiction used in Problem messages:

```
void inv_write_HTML_representation(invocation *inv, char *to) {
    phrase *ph = inv->phrase_invoked;
    ph_type_data *phtd = &(ph->type_data);
    int j=0, f = FALSE, s = FALSE;
    to[0] = 0;
    if ((phtd->no_words > 0) && [[word phtd->words[0] == say]]) {
        j++; s = TRUE;
        sprintf(to, "say \"[");
    }
    for (; j<phtd->no_words; j++) {
        if (f) sprintf(to+strlen(to), " "); f = TRUE;
        if (phtd->words[j] < 10) {
            specification *spec = phtd->tokens_spec[phtd->words[j]];
            specification *found = inv_get_token_as_parsed(inv, phtd->words[j]);
            if (s == FALSE) sprintf(to+strlen(to), "(");
            print_modest_sized_text_to_string(found->word_ref1, found->word_ref2, to+strlen(to));
            sprintf(to+strlen(to), " - <i>");
            write_specification_out_in_English(spec, to+strlen(to));
            sprintf(to+strlen(to), "</i>");
            if (s == FALSE) sprintf(to+strlen(to), ")");
            note_inv_token_text(found);
        } else sprintf(to+strlen(to), "%s", lw_array[phtd->words[j]].lw_rawtext);
```

```
    }
    if (phtd->block_follows) sprintf(to+strlen(to), ": ...");
    if (s) sprintf(to+strlen(to), "]\"");
}

void phtd_write_index_representation(ph_type_data *phtd, phrase *ph) {
    char phrase_buffer[1024];
    if (phtd->manner_of_return == DECIDES_CONDITION_MOR)
        INDEX("<i>if</i> ");
    ph_write_HTML_representation(ph, phrase_buffer);
    INDEX("%s", phrase_buffer);
    if (spec_is_UNKNOWN(phtd->return_type)) {
        if (phtd->manner_of_return == DECIDES_CONDITION_MOR)
            INDEX(" ...");
    } else {
        kind_of_value *kov;
        INDEX(" ... <i>");
        if (spec_get_described_kind(phtd->return_type)) kov = kovko(spec_get_described_kind(phtd->return_type));
        else kov = spec_get_kind_of_value(phtd->return_type);
        index_lowercase_kov(kov);
        INDEX("</i>");
    }
}
void phtd_write_I6_comment_describing(ph_type_data *phtd, OUTPUT_STREAM) {
    int j;
    WRITE("! ");
    for (j=0; j<phtd->no_words; j++) {
        if (phtd->words[j] < 10)
            WRITE("#%d ", phtd->words[j]);
        else
            WRITE("%s ", lw_array[phtd->words[j]].lw_text);
    }
    WRITE(":\n");
}
```

The function inv_write_HTML_representation is called from 2/prob2.
The function phtd_write_index_representation is called from 12/phin.

§**3.** NB: on some newer platforms (such as Android), `inline` is a reserved word to `gcc`, so we mustn't use it as a variable name.

```
ph_type_data phtd_parse(int x1, int x2, int *ow1, int *ow2, int as_inline) {
    int cw, allow_comma = TRUE, brace_level, i;
    ph_type_data phtd = phtd_new(DECIDES_NOTHING_MOR);
    phtd.invoked_inline_not_as_call = as_inline;
    if ([[x1, x2 == to ...]] == FALSE)
        internal_error("phtd_parse called on non-'to' clause");
    x1 = phtd_parse_return_data(&phtd, x1, x2);
    x2 = phtd_parse_miscellaneous_data(&phtd, x1, x2);
    if [[x1, x2 == if/while/unless ...]] allow_comma = FALSE;
    if [[x1, x2 == otherwise/else if/unless ...]] allow_comma = FALSE;
    cw = -1; brace_level = 0;
```

```
    for (i=x1; i<=x2; i++) {
        if [[word i == OPENBRACE]] brace_level++;
        if [[word i == CLOSEBRACE]] brace_level--;
        if ([[word i == COMMA]] && (brace_level == 0) && (i>x1) && (i<x2)) { cw = i; break; }
    }
    if ((allow_comma) && (cw >= 0)) {
        if [[x1, x2 == say ...]] {
            sentence_problem(_P_(C12SayWithPhraseOptions),
                "phrase options are not allowed for 'say' phrases",
                "because the commas would lead to ambiguous sentences, and "
                "because the content of a substitution is intended to be "
                "something conceptually simple and not needing clarification.");
        }
        *ow1 = cw+1; *ow2 = x2;
        x2 = cw-1;
    }
    phtd.register_w1 = x1; phtd.register_w2 = x2;
    phtd_parse_call_data(&phtd, x1, x2);
    return phtd;
}
```

The function phtd_parse is called from 12/ph.

§**4.** Miscellaneous annotations at the back of the declaration (which are removed when noticed), and spotting suggestive words at the beginning (which are not).

```
int first_say_made = FALSE;
int no_lets_made = 0;
int phtd_parse_miscellaneous_data(ph_type_data *phtd, int w1, int w2) {
    phtd->arithmetical_operation = read_arithmetic_op_number(&w1, &w2);
    phtd->polymorphism_exception = read_polymorphism(&w1, &w2);
    phtd->assignment_exception = read_assignment(&w1, &w2);
    if ([[w1, w2 == let ...]] && (no_lets_made < 5)) {
        phtd->let_phrase = TRUE;
        no_lets_made++;
    }
    if [[w1, w2 == say ...]] {
        phtd->say_phrase = A_MISCELLANEOUS_SAY_PHRASE;
        if (first_say_made == FALSE) {
            phtd->say_phrase = THE_PRIMORDIAL_SAY_PHRASE;
            first_say_made = TRUE;
        }
        if [[w1, w2 == say otherwise/else]]
            phtd->say_control_structure = OTHERWISE_SAY_CS;
        if [[w1, w2 == say if/unless ...]]
            phtd->say_control_structure = IF_SAY_CS;
        if [[w1, w2 == say end if/unless]]
            phtd->say_control_structure = END_IF_SAY_CS;
        if [[w1, w2 == ... DASHDASH running on --> w1, w2]]
            phtd->say_phrase_running_on = TRUE;
        if [[w1, w2 == ... DASHDASH beginning ###]] {
```

```
            phtd->say_phrase_stream_position = SSP_START;
            phtd->say_phrase_stream_token_at = w2;
            w2 -= 3;
        } else
        if [[w1, w2 == ... DASHDASH continuing ###]] {
            phtd->say_phrase_stream_position = SSP_MIDDLE;
            phtd->say_phrase_stream_token_at = w2;
            w2 -= 3;
        } else
        if [[w1, w2 == ... DASHDASH ending ### with marker ###]] {
            phtd->say_phrase_stream_position = SSP_END;
            phtd->say_phrase_stream_token_at = w2-3;
            phtd->say_phrase_stream_closing_token_at = w2;
            w2 -= 6;
        } else
        if [[w1, w2 == ... DASHDASH ending ###]] {
            phtd->say_phrase_stream_position = SSP_END;
            phtd->say_phrase_stream_token_at = w2;
            w2 -= 3;
        }
    }
    if [[w1, w2 == ... DASHDASH end --> w1, w2]]
        phtd->block_follows = TRUE;
    if [[w1, w2 == ... DASHDASH in loop]] {
        phtd->only_in_loop = TRUE;
        w2 -= 3;
    }
    if [[w1, w2 == ... DASHDASH in ###]] {
        phtd->only_in_block_wn = w2;
        w2 -= 3;
    }
    return w2;
}
int read_arithmetic_op_number(int *w1, int *w2) {
    int x1 = *w1, x2 = *w2;
    if (x2 < x1+4) return -1;
    x1 = x2-4;
    if ([[x1, x2 == OPENBRACKET arithmetic operation ### CLOSEBRACKET]]
        && (is_kova(is_a_literal(x1+3, x1+3), NUMBER_TY))) {
        *w2 -= 5;
        return last_literal_evaluated;
    }
    return -1;
}
int read_assignment(int *w1, int *w2) {
    int x1 = *w1, x2 = *w2;
    if (x2 < x1+3) return FALSE;
    x1 = x2-3;
    if [[x1, x2 == OPENBRACKET assignment operation CLOSEBRACKET]] {
        *w2 -= 4;
        return TRUE;
    }
```

```
    return FALSE;
}
int read_polymorphism(int *w1, int *w2) {
    int x1 = *w1, x2 = *w2;
    if (x2 < x1+6) return -1;
    x1 = x2-6;
    if ([[x1, x2 == OPENBRACKET amended kind of value ### CLOSEBRACKET]]
        && (is_kova(is_a_literal(x1+5, x1+5), NUMBER_TY))) {
        *w2 -= 7;
        return last_literal_evaluated;
    }
    return -1;
}
```

§**5.** Here we parse the call data: the fixed words used and the bracketed tokens which become call parameters when the PHTD is used to build the stack frame for the phrase.

```
void phtd_parse_call_data(ph_type_data *phtd, int w1, int w2) {
    int i, j;
    phtd->no_tokens = 0;
    phtd->no_words = 0;
    for (i=w1; i<=w2; i++) {
        int word_to_add;
        if [[word i == CLOSEBRACKET]] {
            sentence_problem(_P_(C12TokenWithoutOpenBracket),
                "a close bracket ')' appears here with no matching open '('",
                "so I can't see what is meant to be the fixed text and "
                "what is meant to be changeable. The idea is to put "
                "brackets around whatever varies from one usage to "
                "another: for instance, 'To contribute (N - a number) "
                "dollars: ...'.");
            return;
        }
        if [[word i == OPENBRACKET]] {
            int no_match = FALSE;
            phtd->tokens_w1[phtd->no_tokens] = i+1;
            if (i+1 > w2) no_match = TRUE;
            else {
                while ((i<=w2) && ([[word i == CLOSEBRACKET]] == FALSE))
                    i++;
                if (i > w2) no_match = TRUE;
            }
            if (no_match) {
                sentence_problem(_P_(C12TokenWithoutCloseBracket),
                    "the opening bracket '(' has no matching close bracket ')'",
                    "so I can't see what is meant to be the fixed text and "
                    "what is meant to be changeable. The idea is to put "
                    "brackets around whatever varies from one usage to "
                    "another: for instance, 'To contribute (N - a number) "
                    "dollars: ...'.");
                return;
            }
```

```
phtd->tokens_w2[phtd->no_tokens] = i-1;
if (phtd->tokens_w1[phtd->no_tokens] >
    phtd->tokens_w2[phtd->no_tokens]) {
    sentence_problem(_P_(C12TokenWithEmptyBrackets),
        "nothing is between the opening bracket '(' and its "
        "matching close bracket ')'",
        "so I can't see what is meant to be the fixed text and "
        "what is meant to be changeable. The idea is to put "
        "brackets around whatever varies from one usage to "
        "another: for instance, 'To contribute (N - a number) "
        "dollars: ...'.");
    return;
}
phtd->tokens_spec[phtd->no_tokens] = new_actual_CONSTANT_spec(NULL);
phtd->tokens_template[phtd->no_tokens] = NULL;

j = is_word_intermediate(OPENBRACKET_V,
    phtd->tokens_w1[phtd->no_tokens],
    phtd->tokens_w2[phtd->no_tokens]);
if (j >= 0) {
    sentence_problem(_P_(C12TokenWithNestedBrackets),
        "the text inside the brackets '(' and ')' itself contains "
        "another open bracket '('",
        "which is not allowed.");
    return;
}

j = is_word_intermediate(HYPHEN_V, phtd->tokens_w1[phtd->no_tokens],
    phtd->tokens_w2[phtd->no_tokens]);
if (j >= 0) {
    int x1 = j+1, x2 = phtd->tokens_w2[phtd->no_tokens];
    int k = is_word_intermediate(EQUALS_V, x1, x2);
    if (k >= 0) {
        recognise_type_template_variables = TRUE;
        specification *spec = parse_expression(k+1, x2, DESCRIPTIVE_TYPE_EXPCON);
        recognise_type_template_variables = FALSE;
        if (spec_is_generic_CONSTANT(spec)) {
            phtd->tokens_template[phtd->no_tokens] =
                kov_when_VALUE_is_evaluated(spec);
        } else {
            LOG("Failed on template: $W\n$X", k+1, x2, spec);
            sentence_problem(_P_(C12UnknownKOVTemplate),
                "the kind-of-value template description following "
                "the equals sign is not one I recognise",
                "and if this does not make sense then suffice to "
                "say that the equals sign '=' should not be used "
                "in the name of a kind of value.");
            return;
        }
        x2 = k-1;
    }
    specification *spec = parse_expression(x1, x2, DESCRIPTIVE_TYPE_EXPCON);
    if (spec_is_UNKNOWN(spec)) {
        quote_source(1, current_sentence);
        quote_source(2, current_sentence);
```

```
                    handmade_problem(_P_(C12BadTypeIndication));
                    issue_problem_segment(
                        "In %1, the type indication '%2' is something I don't recognise, "
                        "but should be one of the following possibilities: the name of a kind, "
                        "for instance 'a room', perhaps specified further by some property, "
                        "say 'a lighted room'; or of a kind of value, such as 'number'.");
                    issue_problem_end();
                    return;
                }
                if ((species_is(spec, DESCRIPTION_SPC)) && (spec_get_described_object(spec))) {
                    if ((spec_get_proposition(spec) == NULL) &&
                        (number_of_adjectives_applied_to(spec) == 0) && (spec_get_described_kind(spec)
== NULL)) {
                        world_object *wo = spec_get_described_object(spec);
                        spec = world_object_to_OBJECT_spec(wo);
                    } else {
                        sentence_problem(_P_(C12TokenWithStipulations),
                            "a bracketed token must be a general description",
                            "or else a single specific name, but not a specific "
                            "name with stipulations attached.");
                        return;
                    }
                }
                phtd->tokens_spec[phtd->no_tokens] = spec;
                phtd->tokens_w2[phtd->no_tokens] = j-1;
            } else {
                if (compare_word(phtd->tokens_w1[phtd->no_tokens], called_V))
                    continue;
                sentence_problem(_P_(C12TokenMisunderstood),
                    "the brackets '(' and ')' here neither say that something "
                    "varies but has a given type, nor specify a called name",
                    "so I can't make sense of them. For a 'To...' phrase, "
                    "brackets like this are used with a hyphen dividing "
                    "the name for a varying value and the kind it has: for "
                    "instance, 'To contribute (N - a number) dollars: ...'. "
                    "Rules, on the other hand, use brackets to give names "
                    "to things or rooms found when matching conditions: for "
                    "instance, 'Instead of opening a container in the presence "
                    "of a man (called the box-watcher): ...'");
                return;
            }
            word_to_add = phtd->no_tokens++;
        } else {
            word_to_add = i;
        }
        if (phtd->no_words >= MAX_WORDS_PER_PHRASE) {
            sentence_problem(_P_(C12PhraseTooLong),
                "this phrase has too many words",
                "and needs to be simplified.");
            return;
        }
        phtd->words[phtd->no_words++] = word_to_add;
    }
```

```
    for (i=0; i<phtd->no_tokens; i++)
        if (is_kova(spec_get_kind_of_value(phtd->tokens_spec[i]), QUOT_TY))
            phtd->includes_a_QUOT_token = TRUE;
}
```

§**6.** Here we parse the return data, which is decided by the first words of the phrase preamble.

Note that "to decide on X", "to decide yes" and "to decide no" are special cases: phrases defined in the Standard Rules which control the behaviour of "to decide" phrases. This is why they are exceptions to the syntactic rules below.

```
int no_truth_state_returns = 0;
int phtd_parse_return_data(ph_type_data *phtd, int x1, int x2) {
    x1++;                                                                    skip "to"
    if [[x1, x2 == decide yes/no]] return x1;
    if [[x1, x2 == decide on ...]] return x1;
    if [[x1, x2 == decide whether/if ... --> x1, x2]] {
        phtd->manner_of_return = DECIDES_CONDITION_MOR;
        [[x1, x2 == the ... --> x1, x2]];
        return x1;
    }
    if [[x1, x2 == decide what/which ... --> x1, x2]] {
        int y, t1, t2;
        phtd->manner_of_return = DECIDES_VALUE_MOR;
        if [[x1, x2 == ... is ... : y --> t1, t2 ... x1, x2]] {
            [[x1, x2 == the ... --> x1, x2]];
            phtd->return_type = parse_expression(t1, t2, DESCRIPTIVE_TYPE_EXPCON);
            if (spec_is_UNKNOWN(phtd->return_type)) {
                quote_source(1, current_sentence);
                quote_source(2, current_sentence);
                handmade_problem(_P_(C12UnknownValueToDecide));
                issue_problem_segment(
                    "In %1, the type indication '%2' is something I don't recognise, "
                    "but should be one of the following possibilities: the name of a kind, "
                    "for instance 'a room', perhaps specified further by some property, "
                    "say 'a lighted room'; or of a kind of value, such as 'number'.");
                issue_problem_end();
            } else if ((spec_is_generic_CONSTANT(phtd->return_type) == FALSE) &&
                ((spec_is_DESCRIPTION_of_kind(phtd->return_type) == FALSE))) {
                LOG("$W produced $S\n", t1, t2, phtd->return_type);
                sentence_problem(_P_(C12NonValueToDecide),
                    "phrases to decide what ... must decide a value",
                    "and although this is a construction Inform recognises, "
                    "it isn't a value. For instance, 'To decide which "
                    "phrase is ...' is not allowed, whereas 'To decide "
                    "which number is ...' would be fine, because a number "
                    "is a value and a phrase is not.");
            } else if (spec_is_generic_CONSTANT_of_kova(phtd->return_type,
                TRUTH_STATE_TY)) {
                if (no_truth_state_returns++ > 0)
                sentence_problem(_P_(C12TruthStateToDecide),
                    "phrases are not allowed to decide a truth state",
                    "and should be defined with the form "
```

```
                    "'To decide if ...' rather than "
                    "'To decide what truth state is ...'.");
            }
        }
        return x1;
    }
    phtd->manner_of_return = DECIDES_NOTHING_MOR;
    return x1;
}
```

§**7.** Not parsing, but deducing: the code which builds a stack frame for a phrase on the basis of its type data. (Options data is not considered here: that must be added separately.)

```
ph_stack_frame phtd_to_stack_frame(ph_type_data *phtd) {
    int i;
    ph_stack_frame phsf;
    phsf = phsf_new();
    phsf_set_kov_returned(&phsf, spec_get_kind_of_value(phtd->return_type));
    for (i=0; i<phtd->no_tokens; i++) {
        kind_of_value *kov;
        specification *spec = phtd->tokens_spec[i];
        if (species_is(spec, DESCRIPTION_SPC)) {
            if (spec_get_described_kov(spec)) kov = spec_get_described_kov(spec);
            else kov = kova(OBJECT_TY);
        } else
            kov = spec_get_kind_of_value(spec);
        phsf_add_call_parameter(&phsf, phtd->tokens_w1[i], phtd->tokens_w2[i], kov);
    }
    return phsf;
}
```

The function phtd_to_stack_frame is called from 12/ph.

§**8. Comparison of PHTDs.** This is used when sorting "to..." phrases in order of logical priority: see Phrases for the return codes.

```
int phtd_comparison(ph_type_data *phtd1, ph_type_data *phtd2) {
    int i, flag1, flag2, fw1, fw2, vw1, vw2;
    if (phtd1 == phtd2) return EQUAL_PH;
    if ((phtd1->only_in_loop) && (phtd2->only_in_loop == FALSE))
        return BEFORE_PH;
    if ((phtd2->only_in_loop) && (phtd1->only_in_loop == FALSE))
        return AFTER_PH;
    if ((phtd1->only_in_loop) && (phtd2->only_in_loop))
        return AFTER_PH;
    if ((phtd1->only_in_block_wn >= 0) && (phtd2->only_in_block_wn == 0))
        return BEFORE_PH;
    if ((phtd2->only_in_block_wn >= 0) && (phtd1->only_in_block_wn == 0))
        return AFTER_PH;
    if ((phtd1->only_in_block_wn >= 0) && (phtd2->only_in_block_wn >= 0))
        return AFTER_PH;
    fw1 = 0; vw1 = 0;
```

```
fw2 = 0; vw2 = 0;
for (i=0; i<phtd1->no_words; i++)
    if (phtd1->words[i] < 10) vw1++; else fw1++;
for (i=0; i<phtd2->no_words; i++)
    if (phtd2->words[i] < 10) vw2++; else fw2++;
if (fw1>fw2) return BEFORE_PH;
if (fw1<fw2) return AFTER_PH;
if (vw1>vw2) return BEFORE_PH;
if (vw1<vw2) return AFTER_PH;
for (i=0; i<phtd1->no_words; i++) {
    if (phtd1->words[i] < 10) {
        if (phtd1->words[i] != phtd2->words[i])
            return AFTER_PH;
    } else {
        int x;
        if (phtd2->words[i] < 10) return BEFORE_PH;
        x = strcmp(lw_array[phtd1->words[i]].lw_rawtext, lw_array[phtd2->words[i]].lw_rawtext);
        if (x > 0) return AFTER_PH;
        if (x < 0) return BEFORE_PH;
    }
}
for (i=0, flag1 = TRUE, flag2 = TRUE; i<phtd1->no_tokens; i++) {
    specification *spec1 = phtd1->tokens_spec[i];
    specification *spec2 = phtd2->tokens_spec[i];
    int skip_detailed_look = FALSE;
    if (((spec_is_generic_CONSTANT(spec1)) || (species_is(spec1, DESCRIPTION_SPC)))
        && (spec_is_actual_CONSTANT(spec2))) {
        flag1 = FALSE; skip_detailed_look = TRUE;
    }
    if (((spec_is_generic_CONSTANT(spec2)) || (species_is(spec2, DESCRIPTION_SPC)))
        && (spec_is_actual_CONSTANT(spec1))) {
        flag2 = FALSE; skip_detailed_look = TRUE;
    }
    if (skip_detailed_look) continue;
    int a = spec_is_evaluating(spec1);
    int b = spec_is_evaluating(spec2);
    if (spec_get_storage_form(spec1) == PROPERTY_VALUE_SPC) a = FALSE;
    if (spec_get_storage_form(spec2) == PROPERTY_VALUE_SPC) b = FALSE;
    if (species_is(spec1, DESCRIPTION_SPC)) a = TRUE;
    if (species_is(spec2, DESCRIPTION_SPC)) b = TRUE;
    if ((a) && (b)) {
        if (can_we_match_value_descriptions(spec1, spec2) != ALWAYS_MATCH) flag1 = FALSE;
        if (can_we_match_value_descriptions(spec2, spec1) != ALWAYS_MATCH) flag2 = FALSE;
    } else {
        if (spec_get_family(spec1) != spec_get_family(spec2)) {
            flag1 = FALSE; flag2 = FALSE;
        }
    }
}
if (flag1) return SUBSCHEMA_PH;
if (flag2) return SUPERSCHEMA_PH;
return INCOMPARABLE_PH;
```

```
}
```

The function phtd_comparison is called from 12/toph.

§9. Parsing of phrases.

```
void phtd_register_excerpt(ph_type_data *phtd, phrase *ph) {
    if (phtd->register_w1 < 0) return;
    LOGIF(PHRASEBOOK, "Register phrase with type:\n$h", phtd);
    switch(phtd->manner_of_return) {
        case DECIDES_NOTHING_MOR:
            if (phtd->say_phrase != NOT_A_SAY_PHRASE)
                register_phrasal(SAY_PHRASE_MC, ph,
                    phtd->register_w1+1, phtd->register_w2);
            else
                register_phrasal(VOID_PHRASE_MC, ph,
                    phtd->register_w1, phtd->register_w2);
            if (phtd->block_follows)
                register_reworded_meaning(END_PHRASE_MC, 0,
                    end_V, 0, phtd->register_w1, phtd->register_w1, 0,
                    STORE_POINTER_phrase(ph));
            break;
        case DECIDES_CONDITION_MOR:
            register_phrasal(COND_PHRASE_MC, ph,
                phtd->register_w1, phtd->register_w2);
            break;
        case DECIDES_VALUE_MOR:
            register_phrasal(VALUE_PHRASE_MC, ph,
                phtd->register_w1, phtd->register_w2);
            break;
    }
}
phrase *last_phrase_where_rp_problemed = NULL;
void register_phrasal(int phrase_mc, phrase *ph, int w1, int w2) {
    int i;
    LOGIF(PHRASEBOOK, "register phrasal on <$W>\n", w1, w2);
    for (i=w1; i<=w2; i++) {
        char *p = lw_array[i].lw_text;
        int j, w3;
        if (i<w2) {
            w3 = i+1;
            if [[i, w3 == CLOSEBRACKET OPENBRACKET]] {
                if (ph != last_phrase_where_rp_problemed)
                sentence_problem(_P_(C12AdjacentTokens),
                    "phrases can't be defined so that they have two bracketed "
                    "varying elements immediately next to each other",
                    "but instead need at least one fixed word in between. "
                    "Thus 'To combine (X - a number) (Y - a number)' is not "
                    "allowed, but 'To combine (X - a number) with (Y - a "
                    "number)' works because of the 'with' dividing the "
                    "bracketed terms X and Y.");
                last_phrase_where_rp_problemed = ph;
            }
```

```
        }
        for (j=0; p[j]; j++) {
            if ((j>0) && (p[j-1] != '/') && (p[j] == '/') &&
                (p[j+1]) && (p[j+1] != '/')) {
                int k, aw1, aw2, bw1, bw2;
                char stack_copy_a[1024], stack_copy_b[1024], *slashless;
                stack_copy_a[0] = 0;
                stack_copy_b[0] = 0;
                for (k=w1; k<i; k++) {
                    sprintf(stack_copy_a+strlen(stack_copy_a), "%s ", lw_array[k].lw_text);
                    sprintf(stack_copy_b+strlen(stack_copy_b), "%s ", lw_array[k].lw_text);
                }
                slashless = stack_copy_a+strlen(stack_copy_a);
                for (k=0; k<j; k++)
                    sprintf(stack_copy_a+strlen(stack_copy_a), "%c", p[k]);
                if ((strcmp(slashless, "say") == 0) && (i == w1) &&
                    (phrase_mc != SAY_PHRASE_MC)) {
                    if (ph != last_phrase_where_rp_problemed)
                    sentence_problem(_P_(C12SaySlashed),
                        "'say' is not allowed as the first word of a phrase",
                        "even when presented as one of a number of slashed "
                        "alternatives. (This is because 'say' is reserved for "
                        "creating text substitutions.)");
                    last_phrase_where_rp_problemed = ph;
                }
                if (strcmp(slashless, "--") == 0) *slashless = 0;
                sprintf(stack_copy_a+strlen(stack_copy_a), " ");
                if (strcmp(p+j+1, "--") != 0) {
                    for (k=j+1; p[k]; k++)
                        sprintf(stack_copy_b+strlen(stack_copy_b), "%c", p[k]);
                    sprintf(stack_copy_b+strlen(stack_copy_b), " ");
                }
                for (k=i+1; k<=w2; k++) {
                    sprintf(stack_copy_a+strlen(stack_copy_a), "%s ", lw_array[k].lw_text);
                    sprintf(stack_copy_b+strlen(stack_copy_b), "%s ", lw_array[k].lw_text);
                }
                aw1 = lexer_wordcount;
                feed_into_lexer(stack_copy_a, TRUE, FALSE);
                aw2 = lexer_wordcount-1;
                bw1 = lexer_wordcount;
                feed_into_lexer(stack_copy_b, TRUE, FALSE);
                bw2 = lexer_wordcount-1;
                register_phrasal(phrase_mc, ph, aw1, aw2);
                register_phrasal(phrase_mc, ph, bw1, bw2);
                return;
            }
        }
    }
    register_excerpt_meaning(phrase_mc, 0, w1, w2, STORE_POINTER_phrase(ph));
}
```

The function phtd_register_excerpt is called from 12/toph.

§**10. Adding invocations to phrase SPs.** Try to complete a half-made specification by matching its text against the lexical pattern of the given phrase, which is stored in the PHTD; add an invocation to its list, ignoring all type checking at this stage.

```
int parse_against_PHTD(specification *spec, phrase *phm,
    meaning_list *ml) {
    int i, j, l, mw1, mw2, ow1 = -1, ow2 = -1, instead_flag;
    int token_values_w1[15], token_values_w2[15], no_t;
    int multiplicity_count;
    int argument_count;
    int invocation_count;
    ph_type_data *phtd = &(phm->type_data);
    invocation *inv;

    if (phtd->invoked_inline_not_as_call == NOT_APPLICABLE)
        return FALSE;

    mw1 = spec->word_ref1;
    mw2 = spec->word_ref2;
    instead_flag = FALSE;

    if (ml) {
        if (mw1<0) {
            ml_get_text(ml, &mw1, &mw2);
            LOGIF(PHRASE_COMPILATION,
                "Dredging up <$W> from: $m\n", mw1, mw2, ml);
        }
        no_t = 0;
        ml = ml_down(ml);
        if (ml && (ml_production(ml) == PHR_OPT_PRODUCTION)) {
            ml_get_text(ml, &ow1, &ow2);
            LOGIF(PHRASE_COMPILATION,
                "Thereby dredging up options <$W>\n", ow1, ow2);
            ml = ml_right(ml);
        }
        for (; ml; ml = ml_right(ml)) {
            if (ml_production(ml) == UNPARSED_PRODUCTION) {
                ml_get_text(ml, &(token_values_w1[no_t]),
                    &(token_values_w2[no_t]));
                no_t++;
            } else internal_error("Unexpected production in phrase args");
        }
        goto ParametersParsed;
    }

    if (mw1<0) return FALSE;

    if (phod_allows_options(&(phm->options_data))) {
        j = mw1;
        while ((j <= mw2) && ([[word j == COMMA]] == FALSE)) j++;
        if (j <= mw2) {
            ow1 = j+1; ow2 = mw2; mw2 = j-1;
            LOGIF(PHRASE_COMPILATION,
                "Options at <$W>\n", ow1, ow2);
            if (ow2 < ow1) return FALSE;
        }
    }
```

```
j = mw1;
if (j<0) internal_error("j is invalid in phrase parsing");
for (l=0, no_t=0; l<phtd->no_words; l++) {
    if (j > mw2) return FALSE;
    if (phtd->words[l] >= 10) {
        if (strcmp(lw_array[j++].lw_rawtext,
            lw_array[phtd->words[l]].lw_rawtext) != 0)
            return FALSE;
    } else {
        if (l == phtd->no_words-1) {
            token_values_w1[no_t] = j;
            token_values_w2[no_t++] = mw2;
            j = mw2+1;
        } else {
            int bracket_level = 0;
            token_values_w1[no_t] = j;
            while ((bracket_level != 0) ||
                    (strcmp(lw_array[j].lw_rawtext,
                    lw_array[phtd->words[l+1]].lw_rawtext) != 0)) {
                if [[word j == OPENBRACKET]]  bracket_level++;
                if [[word j == CLOSEBRACKET]] bracket_level--;
                j++;
                if (j > mw2) return FALSE;
            }
            if (j == token_values_w1[no_t]) return FALSE;
            token_values_w2[no_t++] = j-1;
        }
    }
}
if (j != mw2+1) return FALSE;

ParametersParsed:

multiplicity_count = 1;
argument_count = 1;
invocation_count = 0;
while (multiplicity_count > 0) {
    inv = new_invocation();
    set_phrase_invoked(inv, phm);
    if (ow1 >= 0) inv_set_phrase_options(inv, ow1, ow2);

    for (i=0; i<no_t; i++) {
        int w1 = token_values_w1[i];
        int w2 = token_values_w2[i];
        int context, pto;
        int x1 = w1, x2 = w2;
        kind_of_value *save_noun_context;
        w1 = w2+1;
        if (w1 <= w2) {
            multiplicity_count++;
            argument_count++;
        }
        if (i >= MAX_TOKENS_PER_PHRASE)
            fatal_error("MAX_TOKENS_PER_PHRASE exceeded");
        [[x1, x2 == the ... --> x1, x2]];
        if (family_is(phtd->tokens_spec[i], CONDITION_FMY)) {
```

```
        if (species_is(phtd->tokens_spec[i], DESCRIPTION_SPC))
            context = VALUE_EXPCON;
        else
            context = CONDITION_EXPCON;
    } else if (family_is(phtd->tokens_spec[i], COMMAND_FMY)) {
        if (species_is(phtd->tokens_spec[i], TRY_ACTION_SPC)) {
            pto = permit_trying_omission;
            permit_trying_omission = TRUE;
            action_pattern ap = parse_action_pattern(x1, x2, IS_TENSE);
            permit_trying_omission = pto;
            specification *as_parsed;
            if (ap_is_valid(&ap))
                as_parsed = new_TEST_ACTION_spec(store_ap(ap));
            else
                as_parsed = new_UNKNOWN_spec();
            as_parsed->word_ref1 = x1;
            as_parsed->word_ref2 = x2;
            inv_set_token_as_parsed(inv, i, as_parsed);
            continue;
        }
        context = VOID_EXPCON;
    } else context = VALUE_EXPCON;

    pto = permit_trying_omission;
    permit_trying_omission = FALSE;
    save_noun_context = probable_noun_phrase_context;
    probable_noun_phrase_context = NULL;
    if (spec_is_generic_CONSTANT(phtd->tokens_spec[i]))
        probable_noun_phrase_context =
            kov_when_VALUE_is_evaluated(phtd->tokens_spec[i]);

    specification *as_parsed = parse_expression(x1, x2, context);
    if (phtd->includes_a_QUOT_token) {
        if ((spec_is_CONSTANT_of_kova(as_parsed, TEXT_TY)) ||
            (spec_is_CONSTANT_of_kova(as_parsed, TEXT_ROUTINE_TY))) {
            spec_set_kind_of_value(as_parsed, kova(QUOT_TY));
        }
    }
    inv_set_token_as_parsed(inv, i, as_parsed);
    permit_trying_omission = pto;
    probable_noun_phrase_context = save_noun_context;
}
multiplicity_count--;
inv_clear_flag(inv, UNPROVEN_INVFLAG);
if (no_t > 0) inv_set_flag(inv, UNPROVEN_INVFLAG);
inv_set_instead_flag(inv, instead_flag);
inv_set_group(inv, invocation_count++);
inv_set_word_range(inv, mw1, mw2);

if (phtd->say_phrase == A_MISCELLANEOUS_SAY_PHRASE) {
    if ((no_t == 1) &&
        (spec_is_CONSTANT_of_kova(inv_get_token_as_parsed(inv, 0), TEXT_TY))) {
        continue;
    }
}
```

```
        if ((no_t == 1) && (spec_is_actual_CONSTANT(inv_get_token_as_parsed(inv, 0)))) {
            kind_of_value *kov_found = spec_get_kind_of_value(inv_get_token_as_parsed(inv, 0));
            kind_of_value *kov_wanted = spec_get_kind_of_value(phtd->tokens_spec[0]);
            if ((kov_found) && (kov_wanted) &&
                (kov_can_be_stored_in_variables(kov_found))) {
                if ((can_we_cast_kovs(kov_found, kov_wanted) == NEVER_MATCH) &&
                    (can_we_cast_kovs(kov_wanted, kov_found) == NEVER_MATCH)) {
                    if ((multiplicity_count > 0) || (number_of_adjectives_applied_to(spec) > 0)) {
                        continue;
                    }
                }
            }
        }
        add_to_invocation_list(spec_invocation_list(spec), inv);
    }
    return TRUE;
}
```

The function parse_against_PHTD is called from 5/mlc.

Phrase Options 12/phod

Purpose

To create and subsequently parse against the list of phrase options with which the user can choose to invoke a To phrase.

Definitions

¶1. A "phrase option" is a sort of modifier tacked on to the instruction to do something, changing how it works but not enough to merit an entirely new phrase. It's like an argument passed to a routine which specifies optional behaviour, and indeed that will be how it is compiled. Individual phrase options are stored in a very rudimentary structure at present, but we take the trouble to hold them in a structure because we may one day wish to use them in diagnosing errors.

The end of the header (the chin? the neck?) is an optional comma and then a list of phrase options. Like the token names, phrase option names have local scope (which is why they are here and not in the excerpts database). Unlike them, they are not valid as values, which is where parsing time is eaten up: only as conditions, which are parsed much less often (since a condition is not also a value in Inform 7). That means we need not go to the trouble of hashing the names, saving nuisance.

```
typedef struct phrase_option {
    int word_ref1, word_ref2;                                   text of name
} phrase_option;
```

The structure phrase_option is private to this section.

¶2. The packet of these associated with a phrase is stored in the PHOD structure.

```
define MAX_OPTIONS_PER_PHRASE 16                    because held in a 16-bit Z-machine bitmap

typedef struct ph_options_data {
    struct phrase_option *options_permitted[MAX_OPTIONS_PER_PHRASE];    see below
    int no_options_permitted;
    int options_w1, options_w2;                     the text declaring the whole set of options
    int multiple_options_permitted;                 can be combined, or mutually exclusive?
} ph_options_data;
```

The structure ph_options_data is private to this section.

```
phrase_option *po_new(int w1, int w2) {
    phrase_option *po;
    po = CREATE(phrase_option);
    po->word_ref1 = w1; po->word_ref2 = w2;
    return po;
}

ph_options_data phod_new(void) {
    ph_options_data phod;
    phod.no_options_permitted = 0;
    phod.multiple_options_permitted = FALSE;
    phod.options_w1 = -1; phod.options_w2 = -1;
    return phod;
}

ph_options_data phod_parse_declaration(int w1, int w2) {
```

```
    ph_options_data phod = phod_new();
    phod.options_w1 = w1; phod.options_w2 = w2;
    while (w1 <= w2) {
        int opt_w1 = w1;
        while ((w1<=w2) && ([[word w1 == or/andSLASHor/COMMA]] == FALSE)) w1++;
        phod_add_phrase_option(&phod, opt_w1, w1-1);
        [[w1, w2 == COMMA ... --> w1, w2]];
        if [[w1, w2 == or ... --> w1, w2]] continue;
        if [[w1, w2 == andSLASHor ... --> w1, w2]]
            phod.multiple_options_permitted = TRUE;
    }
    return phod;
}

int too_many_POs_error = FALSE;
void phod_add_phrase_option(ph_options_data *phod, int w1, int w2) {
    LOGIF(PHRASE_CREATIONS, "Adding phrase option <$W>\n", w1, w2);
    if (phod->no_options_permitted >= MAX_OPTIONS_PER_PHRASE) {
        if (too_many_POs_error == FALSE)
            sentence_problem(_P_(C12TooManyPhraseOptions),
                "a phrase is only allowed to have 16 different options",
                "so either some of these will need to go, or you may want to "
                "consider breaking up the phrase into simpler ones whose usage "
                "is easier to describe.");
        too_many_POs_error = TRUE;
        return;
    }
    too_many_POs_error = FALSE;                          so that the problem can recur on later phrases
    phod->options_permitted[phod->no_options_permitted++] = po_new(w1, w2);
}

int phod_allows_options(ph_options_data *phod) {
    if (phod->no_options_permitted > 0) return TRUE;
    return FALSE;
}

int phod_parse(ph_options_data *phod, int w1, int w2) {
    int i;
    for (i=0; i<phod->no_options_permitted; i++) {
        phrase_option *po = phod->options_permitted[i];
        if (compare_word_range(w1, w2, po->word_ref1, po->word_ref2))
            return (1 << i);
    }
    return -1;
}

void phod_index(ph_options_data *phod) {
    int i;
    for (i=0; i<phod->no_options_permitted; i++) {
        phrase_option *po = phod->options_permitted[i];
        INDEX("    ");
        if (i==0) INDEX("<i>optionally</i> ");
        else {
            if (i == phod->no_options_permitted-1) {
                if (phod->multiple_options_permitted)
                    INDEX("<i>and/or</i> ");
```

```
                else INDEX("<i>or</i> ");
            }
        }
        print_raw_text_to_file(po->word_ref1, po->word_ref2, ifl);
        if (i < phod->no_options_permitted-1) INDEX(",");
        INDEX("<br>\n");
    }
}

int phod_check_supplied_options(ph_options_data *phod, phrase *ph,
    int silently, invocation *inv, int w1, int w2) {
    int obits;
    if (is_list_divided(w1, w2, LOOK_FOR_AND)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        if (phod->multiple_options_permitted == FALSE) {
            if (silently) return FALSE;
            quote_source(1, current_sentence);
            quote_words(2, w1, w2);
            quote_phrase(3, ph);
            quote_words(4, phod->options_w1, phod->options_w2);
            handmade_problem(_P_(C12PhraseOptionsExclusive));
            issue_problem_segment(
                "You wrote %1, supplying the options '%2' to "
                "the phrase '%3', but the options listed for this phrase ('%4') are "
                "mutually exclusive.");
            issue_problem_end();
            return FALSE;
        }
        if (phod_check_supplied_options(phod, ph, silently, inv, lw1, lw2) == FALSE)
            return FALSE;
        if (phod_check_supplied_options(phod, ph, silently, inv, rw1, rw2) == FALSE)
            return FALSE;
        return TRUE;
    }
    obits = phod_parse(phod, w1, w2);
    if (obits == -1) {
        if (silently) return FALSE;
        quote_source(1, current_sentence);
        quote_words(2, w1, w2);
        quote_phrase(3, ph);
        quote_words(4, phod->options_w1, phod->options_w2);
        if (phod->no_options_permitted > 1) {
            handmade_problem(_P_(C12NotAPhraseOption));
            issue_problem_segment(
                "You wrote %1, but '%2' is not one of the options allowed on "
                "the end of the phrase '%3'. (The options allowed are: '%4'.)");
            issue_problem_end();
        } else {
            handmade_problem(_P_(C12NotTheOnlyPhraseOption));
            issue_problem_segment(
                "You wrote %1, but the only option allowed on the end of the "
                "phrase '%3' is '%4', so '%2' is not something I know how to "
                "deal with.");
            issue_problem_end();
```

```
        }
        return FALSE;
    }
    inv_set_phrase_options_bitmap(inv, obits);
    return TRUE;
}
```

The function phod_new is called from 12/ph.

The function phod_parse_declaration is called from 12/ph.

The function phod_allows_options is called from 12/ph, 12/phtd and 12/toph.

The function phod_parse is called from 12/toph.

The function phod_index is called from 12/phin.

The function phod_check_supplied_options is called from 12/toph.

Stack Frames 12/phsf

Purpose

To create, and parse local value names within the context of, stack frames. A stack frame is a context in which sequences of void-phrase expression evaluations can be performed, that is, in which procedural code can be compiled. Each phrase compiled to an I6 routine needs such a context, but a stack frame can also arise when other routines not derived from phrases are being compiled. In this section we also manage the beginning and ending of code blocks (as in "If ... begin; ...; end if").

12/phsf.§1 Parsing; §2 The current stack frame; §3-5 The current block stack

---

Definitions

¶1. There are three sets of names with scope local to the definition of a given phrase: the names of its tokens, of its phrase options, and of its loop variables and "let" values. The header for a phrase declaration will be something like

To award (some - number) points: ...

and the *words sequence* of this phrase is "to", "award", token 0, "points": a sequence of length 4. If the above text occurred at words 561 to 569, the words sequence would be 561, 562, 0, 568. In the words sequence, numbers above 10 are taken as word numbers in the source; numbers from 0 to 9 indicate the presence of a *token*, where some textual content is substituted. For each token, we record its name ("some"), a hash code for its name – since names like this one will need to parsed against frequently when compiling the phrase's definition routine – and a specification which indicates the type of expression we expect to be find in this position.

Note that this is a full SP, not a mere data type. If the phrase is to compile to an I6 routine, then this will indeed be `VALUE/D` for some data type `D`. But if it is to compile inline, then a much wider range of possibilities presents itself: a condition, or a sentence capable of being asserted with "now", a phrase, etc.

¶2. Supposing that the phrase compiles to a routine, it will use local I6 variables to store the token values and a bitmap of the phrase option settings; and also for any local variables needed in the routine for loop variables, "let" values, etc. In effect, NI is in the same position as a traditional compiler which must allocate its values – of which there might be any number – to registers in a CPU, of which there are only a finite number. Here, our registers are the 15 local variables allowed in any single Z-machine routine: I6's restriction of only 15 locals for any single function is because it performs no "register allocation" and maps its local variables directly onto Z-machine locals.

NI, on the other hand, has a less literal mapping. These variables are allocated and deallocated as needed in the course of compiling the routine: for instance, a loop variable is allocated at the start of a loop and deallocated at the end. It then vanishes as an I7 name, but the I6 local variable holding it may be re-used to hold another I7 value further down the same routine.

The most complicated form such an I6 function can take resembles the following example:

```
[ R_314 t_0 t_1 phrase_options t_2 t_3 t_4 ct_0 ct_1;
```

where there are two tokens passed to the routine (`t_0` and `t_1`), and there are phrase options (whose bitmap is in `phrase_options`), and at the busiest point in the routine three temporary values are needed at once (`t_2`, `t_3`, `t_4`), and finally table-row selection will be going on, so that we need to record a choice of table (`ct_0`) and row number within that table (`ct_1`). A typical invocation of this example phrase would compile to I6 like so:

```
R_314(20, O31_black_marble_slab, 16);
```

It is perhaps worth stopping to ask ourselves why it is helpful for values to be in local variables. Unlike traditional register allocation, this is not done for any speed gain – in the Z-machine, there is no particular

advantage to a local vs a global variable. The actual reason is to place a value in the stack frame for the current routine. For instance, each phrase must have its own "currently selected table row": it would not do for one phrase which uses another one to find that the table row had been deselected as a side-effect. So `ct_0` and `ct_1` have to be locals, not globals. In a few cases, we can get something into the stack frame without having to use up a precious local: we do this trick to hold the current state of conditional text printing brought about by "say if" text substitutions, where the value in question is a global variable which is pushed onto the Z-machine stack, then overwritten, used for a whole, and then restored to its original value by pulling from the stack. But this trick is only possible if we are absolutely sure that we will be in a position to pull the value from the stack when we are done: and with `ct_0` and `ct_1`, we have no such certainty.

```
typedef struct local_variable {
    int allocated;                                  currently in use at this point in the routine
    int varname_w1, varname_w2;                     name of local variable
    int name_hash;                                  hash code for this name
    int block_scope;                                scope of a local - block depth, or -1 for throughout
    struct kind_of_value *variable_data_type;       data type for the contents
} local_variable;

typedef struct ph_stack_frame {
    int no_call_parameters;                         local variables which are call parameters
    struct local_variable call_parameters[15];      a Z-machine imposed limit
    int no_locals;                                  local variables which are not call parameters
    struct local_variable locals[15];               a Z-machine imposed limit
    int ct_locals_needed;                           do we need table-row selection, and thus ct_0, ct_1?
    int scene_locals_needed;                        similarly for scene end checking?
    int no_callings;                                number of "(called X)" variables created here
    int phrase_options_variable_needed;             to hold options called with
    int it_variable_exists;                         used for adjective definitions
    int it_pseudonym_w1, it_pseudonym_w2;           ditto
    int determines_past_conditions;                 or rather, in the present, but for future use
    struct stacked_variable_owner_list *local_stvol;  those in scope here
    struct pointer_allocation *allocated_pointers;
    struct kind_of_value *kov_returned;             or NULL for no return value
} ph_stack_frame;

typedef struct ph_stack_frame_box {
    struct ph_stack_frame boxed_phsf;
    MEMORY_MANAGEMENT
} ph_stack_frame_box;

typedef struct pointer_allocation {
    char allocation_code[128];
    char local_reference_code[128];
    char schema_for_promotion[128];
    struct pointer_allocation *next_in_frame;
    int offset_index;
    MEMORY_MANAGEMENT
} pointer_allocation;

typedef struct block_stack {
    int phrase_block_sp;                            stack pointer for the block stack
    struct phrase *phrase_blocks_owned_by[MAX_BLOCK_NESTING];  e.g., "if" or "while"
    struct parse_node *phrase_blocks_start_at[MAX_BLOCK_NESTING];  where block begins
    struct kind_of_value *phrase_blocks_kov[MAX_BLOCK_NESTING];  for a switch statement
    int otherwise_count[MAX_BLOCK_NESTING];         number of unconditional else/otherwises yet
```

```
    int restore_ct_variables[MAX_BLOCK_NESTING];                    at the end of this loop body
} block_stack;
```

The structure local_variable is private to this section.
The structure ph_stack_frame is private to this section.
The structure ph_stack_frame_box is private to this section.
The structure pointer_allocation is private to this section.
The structure block_stack is private to this section.

¶3. One further name is sometimes local to a phrase: or rather, not so much the name as its possessive form. This is "its", "his", "her" or "their", and in some definitions this can stand for "the thing to which the definition applies". The following variable holds `-1` if this is disallowed, or else the local variable number referred to by these pronouns if it is allowed.

```
int enable_its = -1;
```

```
local_variable lvar_new(int w1, int w2, kind_of_value *kov) {
    local_variable lvar;
    lvar.allocated = FALSE;
    if [[w1, w2 == the ***]] {
        if (w1 == w2) {
            sentence_problem(_P_(C12CalledThe),
                "please don't call a temporary value 'the'",
                "it's just a gratuitously annoying thing to do. I don't think "
                "people realise how hard it is to be an IF compiler, sometimes.");
        } else w1++;
    }
    lvar.varname_w1 = w1; lvar.varname_w2 = w2;
    lvar.name_hash = 0;
    if (w1 >= 0) lvar.name_hash = hash_code_from_excerpt(w1, w2);
    lvar.block_scope = -1;                                    universal scope throughout routine
    lvar.variable_data_type = kov;
    return lvar;
}
void log_local_variable(local_variable *lvar) {
    if (lvar->allocated == FALSE) { LOG("LV<unallocated>"); return; }
    if (lvar->varname_w1 >= 0) {
        LOG("LV\"$W\"", lvar->varname_w1, lvar->varname_w2);
    } else { LOG("LV<nameless>"); }
    LOG("-$u", lvar->variable_data_type);
}
void lvar_deallocate(local_variable *lvar) {
    lvar->allocated = FALSE;
    lvar->varname_w1 = -1; lvar->varname_w2 = -1; lvar->name_hash = 0;
    lvar->block_scope = -1;
}
int phsf_add_call_parameter(ph_stack_frame *phsf,
    int w1, int w2, kind_of_value *kov) {
    local_variable lvar = lvar_new(w1, w2, kov);
    lvar.allocated = TRUE;
```

```
    phsf->call_parameters[phsf->no_call_parameters] = lvar;
    LOGIF(LOCAL_VARIABLES, "Allocating call parameter t_%d as $k\n",
        phsf->no_call_parameters, &(phsf->call_parameters[phsf->no_call_parameters]));
    return phsf->no_call_parameters++;
}

ph_stack_frame phsf_new(void) {
    ph_stack_frame phsf;
    phsf.no_call_parameters = 0;
    phsf.no_locals = 0;
    phsf.ct_locals_needed = FALSE;
    phsf.scene_locals_needed = FALSE;
    phsf.phrase_options_variable_needed = FALSE;
    phsf.it_variable_exists = FALSE;
    phsf.determines_past_conditions = FALSE;
    phsf.no_callings = 0;
    phsf.local_stvol = NULL;
    phsf.it_pseudonym_w1 = -1;
    phsf.it_pseudonym_w2 = -1;
    phsf.allocated_pointers = NULL;
    phsf.kov_returned = NULL;
    return phsf;
}

void phsf_set_kov_returned(ph_stack_frame *phsf, kind_of_value *kov) {
    phsf->kov_returned = kov;
}

void phsf_options_parameter_is_needed(ph_stack_frame *phsf) {
    phsf->phrase_options_variable_needed = TRUE;
}

stacked_variable_owner_list *phsf_get_stvol(ph_stack_frame *phsf) {
    return phsf->local_stvol;
}

void phsf_set_stvol(ph_stack_frame *phsf, stacked_variable_owner_list *stvol) {
    phsf->local_stvol = stvol;
}

int last_local_made = -1;
int phsf_add_local(ph_stack_frame *phsf, local_variable lvar) {
    int i, new_l = -1;
    if (phsf == NULL)
        internal_error("tried to add variable where no stack frame exists");
    for (i=0; i<phsf->no_locals; i++)
        if (phsf->locals[i].allocated == FALSE) { new_l = i; break; }
    if (new_l < 0) {
        if (phsf->no_call_parameters + phsf->no_locals == 15) {
            sentence_problem(_P_(C12TooManyLocals),
                "there are too many temporarily-named values in this phrase",
                "which may be a sign that it is complicated enough to need "
                "breaking up into smaller phrases making use of each other. "
                "For reasons to do with the construction of IF story files, "
                "there can never be more than 15 temporary values at a time, "
                "and that has to include both values created in the "
                "declaration of a phrase (e.g. the 'N' in 'To deduct (N - "
                "a number) points: ...', or the 'watcher' in 'Instead of "
```

```
                "taking something in the presence of a man (called the "
                "watcher): ...'), and also values created with 'let' or "
                "'repeat' (each 'repeat' loop claiming two such values) - "
                "not to mention one or two values occasionally needed to "
                "work with Tables. Because of all this, it's best to keep the "
                "complexity to a minimum within any single phrase.");
            return 0;
        }
        new_l = phsf->no_locals++;
    }
    lvar.allocated = TRUE;
    phsf->locals[new_l] = lvar;
    warn_expression_cache();
    LOGIF(LOCAL_VARIABLES, "Allocating t_%d as $k\n",
        phsf->no_call_parameters + new_l, &(phsf->locals[new_l]));
    last_local_made = phsf->no_call_parameters + new_l;
    return last_local_made;
}
void cls_var(OUTPUT_STREAM, int pos, local_variable lvar) {
    WRITE("(LocalParking-->%d=t_%d)*0+", pos, pos);
}
ph_stack_frame *phsf_compile_local_storage(OUTPUT_STREAM, ph_stack_frame *phsf) {
    ph_stack_frame_box *phsfb;
    int i, j=0;
    for (i=0; i<phsf->no_call_parameters; i++)
        cls_var(OUT, j++, phsf->call_parameters[i]);
    for (i=0; i<phsf->no_locals; i++)
        cls_var(OUT, j++, phsf->locals[i]);
    if (phsf->ct_locals_needed) {
        WRITE("(LocalParking-->%d=ct_0)*0+", j++);
        WRITE("(LocalParking-->%d=ct_1)*0+", j++);
    }
    phsfb = CREATE(ph_stack_frame_box);
    phsfb->boxed_phsf = *phsf;
    return &(phsfb->boxed_phsf);
}
void clr_var(OUTPUT_STREAM, int pos, local_variable lvar) {
    WRITE("t_%d=LocalParking-->%d;\n", pos, pos);
}
void phsf_compile_local_retrieval(OUTPUT_STREAM, ph_stack_frame *phsf) {
    int i, j=0;
    for (i=0; i<phsf->no_call_parameters; i++)
        clr_var(OUT, j++, phsf->call_parameters[i]);
    for (i=0; i<phsf->no_locals; i++)
        clr_var(OUT, j++, phsf->locals[i]);
    if (phsf->ct_locals_needed) {
        WRITE("ct_0=LocalParking-->%d;\n", j++);
        WRITE("ct_1=LocalParking-->%d;\n", j++);
    }
}
void phsf_copy_locals(ph_stack_frame *phsf_to, ph_stack_frame *phsf_from) {
```

```
    int i;
    for (i=0; i<phsf_from->no_locals; i++)
        phsf_to->locals[phsf_to->no_locals++] =
            phsf_from->locals[i];
    for (i=0; i<phsf_from->no_call_parameters; i++)
        phsf_to->call_parameters[phsf_to->no_call_parameters++] =
            phsf_from->call_parameters[i];
    if (phsf_from->ct_locals_needed) phsf_to->ct_locals_needed = TRUE;
    if (phsf_from->scene_locals_needed) phsf_to->scene_locals_needed = TRUE;
    phsf_to->local_stvol = phsf_from->local_stvol;
}
pointer_allocation *phsf_add_allocation(kind_of_value *kov, char *proto) {
    ph_stack_frame *phsf = current_stack_frame();
    pointer_allocation *pall;
    if (phsf == NULL) {
        LOG("Tried to allocate: $u\n", kov);
        internal_error("tried to allocate block KOV outside all stack frames");
    }
    pall = CREATE(pointer_allocation);
    pall->next_in_frame = phsf->allocated_pointers;
    phsf->allocated_pointers = pall;
    if (pall->next_in_frame == NULL) pall->offset_index = 0;
    else pall->offset_index = pall->next_in_frame->offset_index + 1;
    TEMPORARY_STREAM;
    compile_heap_allocation(TEMP, kov, 0, FALSE);
    truncated_strcpy(pall->allocation_code, STREAM_TEXT(TEMP), 127);
    CLOSE_TEMPORARY_STREAM;
    sprintf(pall->local_reference_code, "(blockv_stack-->(I7BASPL+%d))",
        pall->offset_index);
    if (proto == NULL) pall->schema_for_promotion[0] = 0;
    else {
        int i, j;
        for (i=0, j=0; proto[i]; i++) {
            if ((proto[i] == '*') && (proto[i+1] == '#') && (proto[i+2] == '#')) {
                sprintf(pall->schema_for_promotion + j,
                    pall->local_reference_code);
                j = strlen(pall->schema_for_promotion);
                i+=2;
                continue;
            }
            pall->schema_for_promotion[j++] = proto[i];
        }
        pall->schema_for_promotion[j++] = 0;
    }
    return pall;
}
char *pall_get_local_reference(pointer_allocation *pall) {
    return pall->local_reference_code;
}
char *pall_get_expanded_schema(pointer_allocation *pall) {
    return pall->schema_for_promotion;
}
```

The function log_local_variable is called from 2/dl.
The function phsf_add_call_parameter is called from 5/bp and 12/phtd.
The function phsf_new is called from 12/phtd.
The function phsf_set_kov_returned is called from 12/phtd.
The function phsf_options_parameter_is_needed is called from 12/ph.
The function phsf_get_stvol is called from 5/tandv.
The function phsf_set_stvol is called from 12/phud and 12/cph.
The function phsf_compile_local_storage is called from 7/vasp.
The function phsf_compile_local_retrieval is called from 10/str.
The function phsf_copy_locals is called from 10/str.
The function phsf_add_allocation is called from 7/data, 7/cfsp and 12/cinv.
The function pall_get_local_reference is called from 7/cfsp and 12/cinv.
The function pall_get_expanded_schema is called from 7/data.

§**1. Parsing.** Locally defined values are parsed by hand, not via the dictionaries, because they come and go on the breeze: although our symbols table is well-adapted to many things, it is not a true hash, and there is a real reward for not allowing it to grow unnecessarily. So we do not want to register the locals as meanings. We do use the excerpt hashing to make the parse reasonably quick, though: and there are only ever at most 15 cases to check.

```
int phsf_parse_name(ph_stack_frame *phsf, int w1, int w2) {
    int i, h;
    if ((phsf->it_variable_exists) && (w1 == w2)) {
        if [[w1, w2 == it/he/she/they/him/her/them]]
            return 0;
    }
    if [[w1, w2 == the ***]] {
        w1++; if (w1 > w2) return -1;
    }
    if (phsf->it_pseudonym_w1 >= 0) {
        if (compare_word_range(w1, w2, phsf->it_pseudonym_w1, phsf->it_pseudonym_w2))
            return 0;
    }
    h = hash_code_from_excerpt(w1, w2);
    for (i=0; i<phsf->no_call_parameters; i++)
        if (h == phsf->call_parameters[i].name_hash)
            if (compare_word_range(w1, w2,
                phsf->call_parameters[i].varname_w1,
                phsf->call_parameters[i].varname_w2)) return i;
    for (i=phsf->no_locals-1; i>=0; i--)
        if ((phsf->locals[i].allocated) && (h == phsf->locals[i].name_hash))
            if (compare_word_range(w1, w2,
                phsf->locals[i].varname_w1,
                phsf->locals[i].varname_w2))
                    return i + phsf->no_call_parameters;
    return -1;
}
int parse_name_local_to_current_stack_frame(int w1, int w2) {
    ph_stack_frame *phsf = current_stack_frame();
    if (phsf == NULL) return -1;
    return phsf_parse_name(phsf, w1, w2);
}
void declare_local_names_in_current_stack_frame(equation *eqn) {
```

```
    ph_stack_frame *phsf = current_stack_frame();
    if (phsf == NULL) return;
    int i;
    for (i=0; i<phsf->no_call_parameters; i++)
        equation_declare_local(eqn,
            phsf->call_parameters[i].varname_w1,
            phsf->call_parameters[i].varname_w2,
            phsf->call_parameters[i].variable_data_type);
    for (i=phsf->no_locals-1; i>=0; i--)
        equation_declare_local(eqn,
            phsf->locals[i].varname_w1,
            phsf->locals[i].varname_w2,
            phsf->locals[i].variable_data_type);
}

int parse_name_local_to_inline_stack_frame(phrase *ph, int w1, int w2) {
    ph_stack_frame *phsf = &(ph->stack_frame);
    if (phsf == NULL) return -1;
    return phsf_parse_name(phsf, w1, w2);
}

int stack_frame_contains_locals(void) {
    ph_stack_frame *phsf = current_stack_frame();
    if (phsf == NULL) return FALSE;
    if (phsf->no_locals > 0) return TRUE;
    return FALSE;
}

int phsf_contains_local_quantities(ph_stack_frame *phsf) {
    if (phsf == NULL) return FALSE;
    if ((phsf->no_locals > 0)
        || (phsf->no_call_parameters > 0)
        || (phsf->phrase_options_variable_needed)) return TRUE;
    return FALSE;
}

void describe_repetition_local(OUTPUT_STREAM, int lnum) {
    ph_stack_frame *phsf = current_stack_frame();
    if (phsf == NULL) internal_error(
        "tried to ensure ct locals where no stack frame exists");
    lnum -= phsf->no_call_parameters;
    if ((lnum < 0) || (lnum > phsf->no_locals)) {
        if (problem_count == 0) internal_error("lnum out of range");
        return;
    }
    WRITE("[repetition with ");
    print_raw_text_to_file(phsf->locals[lnum].varname_w1, phsf->locals[lnum].varname_w2, OUT);
    WRITE(" set to \", (%s) t_%d, \"]^\";\n",
        kov_get_name_of_printing_rule(phsf->locals[lnum].variable_data_type),
        lnum + phsf->no_call_parameters);
}

void we_need_ct_locals(void) {
    ph_stack_frame *phsf = current_stack_frame();
    if (phsf == NULL) return;                    to allow early parsing of expressions before the SF is ready
    phsf->ct_locals_needed = TRUE;
    LOGIF(LOCAL_VARIABLES, "Stack frame acquires CT locals\n");
```

```
}
void we_need_scene_locals(void) {
    ph_stack_frame *phsf = current_stack_frame();
    if (phsf == NULL) internal_error(
        "tried to ensure scene locals where no stack frame exists");
    phsf->scene_locals_needed = TRUE;
    LOGIF(LOCAL_VARIABLES, "Stack frame acquires scene locals\n");
}
int do_we_need_ct_locals(void) {
    ph_stack_frame *phsf = current_stack_frame();
    if (phsf == NULL) return FALSE;
    if (phsf->ct_locals_needed) return TRUE;
    return FALSE;
}
void phsf_determines_the_past(void) {
    ph_stack_frame *phsf = current_stack_frame();
    if (phsf == NULL) internal_error(
        "tried to determine past where no stack frame exists");
    phsf->determines_past_conditions = TRUE;
    LOGIF(LOCAL_VARIABLES, "Stack frame determines past\n");
}
int phsf_used_for_past_tense(void) {
    ph_stack_frame *phsf = current_stack_frame();
    if (phsf == NULL) return FALSE;
    if (phsf->determines_past_conditions) return TRUE;
    return FALSE;
}
int make_new_local(int w1, int w2, kind_of_value *kov) {
    return phsf_add_local(current_stack_frame(), lvar_new(w1, w2, kov));
}
void phsf_add_it_variable(ph_stack_frame *phsf, int cw1, int cw2, kind_of_value *kov) {
    if (cw1 == -1) {
        int it_wn = lexer_wordcount;
        feed_into_lexer(" it ", FALSE, FALSE);
        vocab_identify_word_range(it_wn, it_wn);
        cw1 = it_wn; cw2 = it_wn;
        phsf->it_variable_exists = TRUE;
    }
    phsf_add_call_parameter(phsf, cw1, cw2, kov);
}
void phsf_rename_it_variable(ph_stack_frame *phsf, int ritw1, int ritw2) {
    phsf->it_pseudonym_w1 = ritw1;
    phsf->it_pseudonym_w2 = ritw2;
}
void phsf_restore_it_variable(ph_stack_frame *phsf) {
    phsf->it_pseudonym_w1 = -1;
    phsf->it_pseudonym_w2 = -1;
}
int ensure_called_local(int w1, int w2, kind_of_value *kov) {
    int i;
    ph_stack_frame *phsf = current_stack_frame();
```

```
    if (phsf == NULL) internal_error(
        "tried to ensure local variable where no stack frame exists");
    if (is_word_intermediate_inc(DASH_V, w1, w2) >= 0) {
        sentence_problem(_P_(C12CalledWithDash),
            "a '(called ...)' name is not allowed to include a hyphen",
            "since this would look misleadingly like a declaration of "
            "kind of value it has.");
        return 0;
    }
    [[w1, w2 == the ... --> w1, w2]];
    for (i=phsf->no_locals-1; i>=0; i--)
        if ((phsf->locals[i].allocated) && (phsf->locals[i].varname_w1 >= 0))
            if (compare_word_range(w1, w2,
                phsf->locals[i].varname_w1, phsf->locals[i].varname_w2)) {
                    if (kov) {
                        LOG("Retyping called var %d ($W) to $u\n", i, w1, w2, kov);
                        phsf->locals[i].variable_data_type = kov;
                    }
                    return i + phsf->no_call_parameters;
                }
    phsf->no_callings++;
    if (kov == NULL) kov = kova(OBJECT_TY);
    return make_new_local(w1, w2, kov);
}
int last_mnc_w1 = -1, last_mnc_w2 = -1;
void make_necessary_callings(int w1, int w2) {
    int i = w1, j, k;
    if ((w1 >= last_mnc_w1) && (w2 <= last_mnc_w2)) return;
    last_mnc_w1 = w1; last_mnc_w2 = w2;
    while ((j=is_word_intermediate(called_V, i, w2)) >= 0) {
        if ([[word j-1 == OPENBRACKET]] == FALSE) { i=j; continue; }
        k=j+1;
        while (k<=w2) {
            if [[word k == CLOSEBRACKET]] {
                ensure_called_local(j+1, k-1, kova(OBJECT_TY));
                break;
            }
            k++;
        }
        i=k;
    }
}
void set_KOV_of_local_variable(int index, kind_of_value *kov) {
    ph_stack_frame *phsf = current_stack_frame();
    local_variable *lvar;
    if (phsf == NULL) internal_error(
        "Tried to find data type of local variable where no stack frame exists");
    if (index < 0)
        internal_error("tried to find data type of -ve local");
    if (index >= phsf->no_call_parameters + phsf->no_locals)
        internal_error("tried to find data type of oversized local");
```

```
    if (index < phsf->no_call_parameters) lvar = &(phsf->call_parameters[index]);
    else lvar = &(phsf->locals[index - phsf->no_call_parameters]);
    lvar->variable_data_type = kov;
}
kind_of_value *kov_of_local(int index) {
    ph_stack_frame *phsf = current_stack_frame();
    local_variable *lvar;
    if (phsf == NULL) internal_error(
        "Tried to find data type of local variable where no stack frame exists");
    if (index < 0)
        internal_error("tried to find data type of -ve local");
    if (index >= phsf->no_call_parameters + phsf->no_locals)
        internal_error("tried to find data type of oversized local");
    if (index < phsf->no_call_parameters) lvar = &(phsf->call_parameters[index]);
    else lvar = &(phsf->locals[index - phsf->no_call_parameters]);
    return lvar->variable_data_type;
}
kind_of_value *unproblematic_data_type_of_local(int index) {
    ph_stack_frame *phsf = current_stack_frame();
    if (phsf == NULL) return NULL;
    if (index < 0) return NULL;
    if (index >= phsf->no_call_parameters + phsf->no_locals) return NULL;
    return kov_of_local(index);
}
void relinquish_locals_of_scope(int s) {
    ph_stack_frame *phsf = current_stack_frame();
    int i;
    if (phsf == NULL)
        internal_error("relinquishing locals where no stack frame exists");
    for (i=0; i<phsf->no_locals; i++) {
        local_variable *lvar = &(phsf->locals[i]);
        if ((lvar->allocated) && (lvar->block_scope == s)) {
            LOGIF(LOCAL_VARIABLES,
                "De-allocating t_%d = $k at end of block level %d\n",
                i + phsf->no_call_parameters, lvar, s);
            lvar_deallocate(lvar);
        }
    }
    warn_expression_cache();
}
int local_whose_scope_is_block(int s) {
    ph_stack_frame *phsf = current_stack_frame();
    int i;
    if (phsf == NULL) return -1;
    for (i=0; i<phsf->no_locals; i++)
        if ((phsf->locals[i].allocated) && (phsf->locals[i].block_scope == s))
            return i + phsf->no_call_parameters;
    return -1;
}
void set_scope_of_local_to_current_block(int i) {
    ph_stack_frame *phsf = current_stack_frame();
```

```
    int s = current_block_level();
    if (phsf == NULL)
        internal_error("setting scope of local where no stack frame exists");
    if (i < phsf->no_call_parameters)
        internal_error("i too low in scoping local");
    if (i >= phsf->no_call_parameters + phsf->no_locals)
        internal_error("i too big in scoping local");
    phsf->locals[i - phsf->no_call_parameters].block_scope = s;
    LOGIF(LOCAL_VARIABLES, "Setting scope of t_%d = $k to block level %d\n",
        i, &(phsf->locals[i - phsf->no_call_parameters]), s);
}

void set_kov_of_local(int i, kind_of_value *kov) {
    ph_stack_frame *phsf = current_stack_frame();
    if (phsf == NULL)
        internal_error("setting kov of local where no stack frame exists");
    if (i < phsf->no_call_parameters)
        internal_error("i too low in set-kov local");
    if (i >= phsf->no_call_parameters + phsf->no_locals)
        internal_error("i too big in set-kov local");
    phsf->locals[i - phsf->no_call_parameters].variable_data_type = kov;
}

int shell_routines_counter = 0;
void phsf_compile_i6_local_declarations(ph_stack_frame *phsf, OUTPUT_STREAM) {
    if (phsf == NULL) {
        WRITE(";\n");
        return;
    }
    if (phsf->allocated_pointers) {
        pointer_allocation *pall; int i, j, rp_flag = FALSE, xsu;
        if (kov_uses_pointer_values(phsf->kov_returned)) rp_flag = TRUE;
        if (rp_flag) WRITE("I7RBLK ");
        phsf_compile_i6_local_varnames(phsf, OUT, FALSE, TRUE);
        INDENT;
        for (pall=phsf->allocated_pointers, i=0; pall; pall=pall->next_in_frame, i++)
            WRITE("blockv_stack-->(blockv_sp+%d) = %s;\n",
                pall->offset_index, pall->allocation_code);
        WRITE("blockv_sp = blockv_sp + %d;\n", i);
        if (rp_flag) {
            WRITE("I7RBLK = BlkValueCopy(I7RBLK, R_SHELL_%d(blockv_sp-%d",
                shell_routines_counter, i);
            phsf_compile_i6_local_varnames(phsf, OUT, TRUE, TRUE);
            WRITE("));\n");
            xsu = 0;
        } else {
            WRITE("blockv_stack-->(blockv_sp++) = R_SHELL_%d(blockv_sp-%d",
                shell_routines_counter, i);
            phsf_compile_i6_local_varnames(phsf, OUT, TRUE, TRUE);
            WRITE(");\n");
            xsu = 1;
        }
        WRITE("blockv_sp = blockv_sp - %d;\n", i+xsu);
        for (pall=phsf->allocated_pointers, j=i+xsu; pall; pall=pall->next_in_frame, j--)
            WRITE("BlkFree(blockv_stack-->(blockv_sp+%d));\n", pall->offset_index);
```

```
        if (rp_flag) WRITE("return I7RBLK;\n");
        else WRITE("return blockv_stack-->(blockv_sp+%d);\n", i);
        OUTDENT; WRITE("];\n");
        WRITE("[ R_SHELL_%d I7BASPL ", shell_routines_counter++);
    }
    phsf_compile_i6_local_varnames(phsf, OUT, FALSE, FALSE);
}
void phsf_compile_i6_local_varnames(ph_stack_frame *phsf, OUTPUT_STREAM,
    int commad, int call_pars_only) {
    int j, lines = 0;
    for (j=0; j<phsf->no_call_parameters; j++) {
        WRITE("%s\n    t_%d", (commad)?",":"", j);
        if (commad == FALSE) {
            WRITE(" ! Call parameter '");
            print_raw_text_to_file(
                phsf->call_parameters[j].varname_w1,
                phsf->call_parameters[j].varname_w2,
                OUT);
            WRITE("' = ");
            kov_write_source_name_to_file(OUT, phsf->call_parameters[j].variable_data_type);
        }
        lines++;
    }
    if (phsf->phrase_options_variable_needed > 0) {
        WRITE("%s\n    phrase_options ! Bitmap of options called with",
            (commad)?",":"");
        lines++;
    }
    if (call_pars_only) {
        if (lines > 0) WRITE("\n    ");
        if (commad == FALSE) WRITE(";");
        return;
    }
    for (j=0; j<phsf->no_locals; j++) {
        WRITE("%s\n    t_%d ! Local variable e.g. '",
            (commad)?",":"",
            j+phsf->no_call_parameters);
        print_raw_text_to_file(phsf->locals[j].varname_w1, phsf->locals[j].varname_w2, OUT);
        WRITE("' = ");
        kov_write_source_name_to_file(OUT, phsf->locals[j].variable_data_type);
        lines++;
    }
    if (phsf->ct_locals_needed) {
        WRITE("%s\n    ct_0%s ct_1 ! Used for table searches",
            (commad)?",":"", (commad)?",":"");
        lines++;
    }
    if (phsf->scene_locals_needed) {
        WRITE("%s\n    chs%s sc%s ch ! Used for scene searches",
            (commad)?",":"", (commad)?",":"", (commad)?",":"");
        lines++;
    }
    if (lines > 0) WRITE("\n    ");
```

```
    if (commad) return;
    WRITE(";\n");
}
```

The function parse_name_local_to_current_stack_frame is called from 5/tandv and 10/eqns.
The function declare_local_names_in_current_stack_frame is called from 10/eqns.
The function parse_name_local_to_inline_stack_frame is called from 12/cinv.
The function phsf_contains_local_quantities is called from 10/str.
The function describe_repetition_local is called from 12/cph.
The function we_need_ct_locals is called from 6/simp, 7/stsp, 10/libp, 11/ap and 12/cinv.
The function we_need_scene_locals is called from 9/scene.
The function do_we_need_ct_locals is called from 5/mlc, 6/cdefp and 11/chron.
The function phsf_determines_the_past is called from 11/chron.
The function phsf_used_for_past_tense is called from 6/atoms.
The function make_new_local is called from 7/tc and 12/cinv.
The function phsf_add_it_variable is called from 5/aph, 11/los and 12/ph.
The function phsf_rename_it_variable is called from 12/def.
The function phsf_restore_it_variable is called from 12/def.
The function ensure_called_local is called from 5/mlc, 6/atoms, 6/defer and 11/ap.
The function make_necessary_callings is called from 5/candp.
The function set_KOV_of_local_variable is called from 10/eqns.
The function kov_of_local is called from 7/tc and 10/eqns.
The function unproblematic_data_type_of_local is called from 7/stsp.
The function local_whose_scope_is_block is called from 12/cph.
The function set_scope_of_local_to_current_block is called from 6/defer and 12/cinv.
The function set_kov_of_local is called from 12/cinv.
The function phsf_compile_i6_local_declarations is called from 12/cph.

§**2. The current stack frame.** At any given time, a single stack frame is valid for local variable names and phrase option names used as conditions. This might be the stack frame of the current "To..." phrase being defined, or it might be a free-standing stack frame which is "nonphrasal", meaning, not a `.stack_frame` element of any phrase structure. Stack frames change during compilation and are liable to be destroyed when compilation is complete, so pointers to them must not be stored for later use.

```
int nonphrasal_stack_frame_is_current = FALSE;
ph_stack_frame nonphrasal_stack_frame;

ph_stack_frame *current_stack_frame(void) {
    if (phrase_being_compiled) return &(phrase_being_compiled->stack_frame);
    if (nonphrasal_stack_frame_is_current) return &nonphrasal_stack_frame;
    return NULL;
}

void phsf_select_stack_frame_of(phrase *ph) {
    if (ph == NULL)
        internal_error("can't select null stack frame");
    if ((phrase_being_compiled) && (phrase_being_compiled != ph))
        internal_error("can't select new stack frame while phrase open");
    if (nonphrasal_stack_frame_is_current)
        internal_error("can't select new stack frame while nonphrasal open");
    phrase_being_compiled = ph;
}

ph_stack_frame *phsf_create_nonphrase_stack_frame(void) {
    if (phrase_being_compiled)
        internal_error("can't create nonphrasal stack frame while phrase open");
```

```
    if (nonphrasal_stack_frame_is_current)
        internal_error("can't nest nonphrasal stack frames");
    nonphrasal_stack_frame = phsf_new();
    nonphrasal_stack_frame_is_current = TRUE;
    return &nonphrasal_stack_frame;
}

void phsf_remove_nonphrase_stack_frame(void) {
    nonphrasal_stack_frame = phsf_new();                    to prevent accidental lucky misuse
    nonphrasal_stack_frame_is_current = FALSE;
    phrase_being_compiled = NULL;
    finish_this_session_of_parsing();
}
```

The function current_stack_frame is called from 5/tandv, 5/mlc, 7/vasp, 10/str, 10/list, 11/ap, 12/phud and 12/cph.
The function phsf_select_stack_frame_of is called from 12/cs.
The function phsf_create_nonphrase_stack_frame is called from 5/aph, 6/cdefp, 9/scene, 10/str, 11/chron, 11/ap, 11/los, 13/gl and 13/test.
The function phsf_remove_nonphrase_stack_frame is called from 5/aph, 6/cdefp, 9/scene, 10/str, 11/chron, 11/ap, 11/los, 12/cs, 12/phud, 13/gl and 13/test.

§**3. The current block stack.** During code compilation, we must keep track of statement blocks: those defined by begin... end keywords. We need these partly to ensure that the control flow is correct, partly to deallocate local variables when they are no longer needed (for instance, a repeat variable does not exist outside its loop).

In principle, this information belongs to the current stack frame, since it's within the context of a stack frame that code is compiled. But it would be wasteful to store arrays for statement blocks inside every stack frame structure, because in practice we only compile within one stack frame at a time, and we finish each before beginning the next. So we store the block stack in the only instance of a private structure.

```
define MAX_BLOCK_NESTING 20

block_stack current_block_stack;

int block_stack_active = FALSE;

void begin_code_blocks(void) {
    if (block_stack_active)
        internal_error("tried to begin block stack already in use");
    current_block_stack.phrase_block_sp = 0;
    block_stack_active = TRUE;
    LOGIF(LOCAL_VARIABLES, "Block stack now active\n");
}

void end_code_blocks(void) {
    if (block_stack_active == FALSE) internal_error("block stack inactive");
    while (current_block_stack.phrase_block_sp > 0) {
        current_block_stack.phrase_block_sp--;
        current_sentence =
            current_block_stack.phrase_blocks_start_at[
                current_block_stack.phrase_block_sp];
        sentence_problem(_P_(C12BeginWithoutEnd),
            "the definition of the phrase ended with no matching 'end' for "
            "this 'begin'",
            "bearing in mind that every begin must have a matching end, and "
            "that the one most recently begun must be the one first to end. For "
            "instance, 'if ... begin' must have a matching 'end if'.");
```

```
    }
    block_stack_active = FALSE;
    LOGIF(LOCAL_VARIABLES, "Block stack now inactive\n");
}

int next_block_protects_ctv = -1;
void open_code_block(phrase *owner) {
    if (block_stack_active == FALSE) internal_error("block stack inactive");
    if (current_block_stack.phrase_block_sp == MAX_BLOCK_NESTING)
        sentence_problem(_P_(C12BlockNestingTooDeep),
            "compound phrases have gone too deep",
            "perhaps because many have begun but not been properly ended?");
    else {
        current_block_stack.otherwise_count[
            current_block_stack.phrase_block_sp] = 0;
        current_block_stack.restore_ct_variables[
            current_block_stack.phrase_block_sp] = -1;
        current_block_stack.phrase_blocks_kov[
            current_block_stack.phrase_block_sp] = NULL;
        current_block_stack.phrase_blocks_start_at[
            current_block_stack.phrase_block_sp] = current_sentence;
        current_block_stack.phrase_blocks_owned_by[
            current_block_stack.phrase_block_sp++] = owner;
        if (next_block_protects_ctv >= 0) {
            int var1 = next_block_protects_ctv, var2 = var1 + 1;
            current_block_stack.restore_ct_variables[
                current_block_stack.phrase_block_sp-1] = var1;
            set_scope_of_local_to_current_block(var1);
            set_scope_of_local_to_current_block(var2);
        }
    }
    next_block_protects_ctv = -1;
}

void close_code_block(void) {
    if (block_stack_active == FALSE) internal_error("block stack inactive");
    if (current_block_stack.phrase_block_sp == 0)
        internal_error("code block underflow");
    current_block_stack.phrase_block_sp--;
    relinquish_locals_of_scope(current_block_stack.phrase_block_sp);
}
```

The function begin_code_blocks is called from 5/aph, 9/scene, 10/str, 11/los, 12/cph and 13/gl.
The function end_code_blocks is called from 5/aph, 9/scene, 10/str, 11/los, 12/cph and 13/gl.
The function open_code_block is called from 12/cinv.
The function close_code_block is called from 7/cmsp.

§**4.** If there is no current block stack, we behave as if the block stack were empty, but (as long as nobody tries to open or close any blocks) no internal errors are issued. This allows the typechecker to run even when there is no current block stack, which is important when typechecking an expression whose evaluation requires the use of a phrase.

```
int current_block_level(void) {
    if (block_stack_active == FALSE) return 0;
    return current_block_stack.phrase_block_sp;
}
int ct_variables_saved_in_this_block(void) {
    if (block_stack_active == FALSE) return -1;
    if (next_block_protects_ctv >= 0) return next_block_protects_ctv;
    if (current_block_stack.phrase_block_sp == 0) return -1;
    return current_block_stack.restore_ct_variables[
        current_block_stack.phrase_block_sp-1];
}
void save_ct_variables_in_current_loop(void) {
    if (block_stack_active == FALSE) return;
    int var1 = make_new_local(-1, -1, kova(NUMBER_TY));
    make_new_local(-1, -1, kova(NUMBER_TY));
    next_block_protects_ctv = var1;
}
int increment_otherwise_count(void) {
    if (block_stack_active == FALSE) return 0;
    if (current_block_stack.phrase_block_sp == 0) return 0;
    return current_block_stack.otherwise_count[
        current_block_stack.phrase_block_sp-1]++;
}
int read_otherwise_count(void) {
    if (block_stack_active == FALSE) return 0;
    if (current_block_stack.phrase_block_sp == 0) return 0;
    return current_block_stack.otherwise_count[
        current_block_stack.phrase_block_sp-1];
}
void set_kov_for_current_block(kind_of_value *kov) {
    if (block_stack_active == FALSE) return;
    if (current_block_stack.phrase_block_sp == 0) return;
    current_block_stack.phrase_blocks_kov[
        current_block_stack.phrase_block_sp-1] = kov;
}
kind_of_value *kov_for_current_block(void) {
    if (block_stack_active == FALSE) return NULL;
    if (current_block_stack.phrase_block_sp == 0) return NULL;
    return current_block_stack.phrase_blocks_kov[
        current_block_stack.phrase_block_sp-1];
}
char *name_of_current_block(void) {
    if (block_stack_active == FALSE) return NULL;
    if (current_block_stack.phrase_block_sp == 0) return NULL;
    return lw_array[
        current_block_stack
            .phrase_blocks_owned_by[current_block_stack.phrase_block_sp-1]
```

```
                ->type_data.words[0]
        ].lw_text;
}

int inside_a_loop_body(void) {
    int i;
    if (block_stack_active == FALSE) return FALSE;
    for (i = current_block_stack.phrase_block_sp-1; i >= 0; i--) {
        vocabulary_entry *incipit =
            lw_array[current_block_stack.phrase_blocks_owned_by[i]->type_data.words[0]].lw_identity;
        if (incipit == repeat_V) return TRUE;
        if (incipit == while_V) return TRUE;
    }
    return FALSE;
}

parse_node *start_of_current_block(void) {
    if (block_stack_active == FALSE) return NULL;
    if (current_block_stack.phrase_block_sp == 0) return NULL;
    return current_block_stack.phrase_blocks_start_at[
        current_block_stack.phrase_block_sp-1];
}
```

The function current_block_level is called from 12/cph.
The function ct_variables_saved_in_this_block is called from 7/cmsp and 12/cinv.
The function save_ct_variables_in_current_loop is called from 12/cinv.
The function increment_otherwise_count is called from 7/cmsp.
The function read_otherwise_count is called from 7/cmsp.
The function set_kov_for_current_block is called from 12/cinv.
The function kov_for_current_block is called from 7/tc.
The function name_of_current_block is called from 7/tc.
The function inside_a_loop_body is called from 7/tc.
The function start_of_current_block is called from 7/tc.

§5. An example: how we compile a routine to test a single condition.

```
void compile_condition_routine(OUTPUT_STREAM, char *rname,
    int w1, int w2, bp_term_details par1, bp_term_details par2) {
    specification *spec;
    ph_stack_frame *phsf;

    phsf = phsf_create_nonphrase_stack_frame();
    OUT = begin_compiling_phrase(OUT);
    begin_code_blocks();

    bptd_add_as_call_parameter(OUT, phsf, par1);
    bptd_add_as_call_parameter(OUT, phsf, par2);

    enable_its = 0;
    spec = parse_expression(w1, w2, CONDITION_EXPCON);
    if (validate_when(spec) == FALSE) {
        LOG("Error at: $S", &spec);
        sentence_problem(_P_(C12BadRelationCondition),
            "the condition defining this relation makes no sense to me",
            "although the definition was properly formed - it is only "
            "the part after 'when' which I can't follow.");
    } else {
```

```
        INDENT;
        WRITE("return "); spec_compile(OUT, spec); WRITE(";\n");
        OUTDENT; WRITE("];\n");
    }
    OUT = write_routine_header();
    WRITE("[ %s ", rname);
    copy_compiled_phrase();
    end_code_blocks();
    phsf_remove_nonphrase_stack_frame();
}
```

The function compile_condition_routine is called from 5/rel.

Stacked Variables 12/stv

Purpose

To permit quantities to belong to rulebooks, and to have limited name-scope.

Definitions

```
typedef struct stacked_variable {
    int word_ref1, word_ref2;                          text of the name
    struct parse_node *assigned_at;                    sentence assigning it
    struct quantity *underlying_quantity;              the quantity in question
    int owner_id;                                      who owns this
    int offset_in_owning_frame;                        word offset of storage (counts from 0)
    int mw1, mw2;                                      matching text (relevant for action variables only)
    MEMORY_MANAGEMENT
} stacked_variable;

typedef struct stacked_variable_list {
    struct stacked_variable *the_stv;                  the STV
    struct stacked_variable_list *next;                in linked list
    MEMORY_MANAGEMENT
} stacked_variable_list;

typedef struct stacked_variable_owner {
    int no_stvs;
    int recognition_id;
    struct stacked_variable_list *list_of_stvs;
    MEMORY_MANAGEMENT
} stacked_variable_owner;

typedef struct stacked_variable_owner_list {
    struct stacked_variable_owner *stvo;               the STO
    struct stacked_variable_owner_list *next;          in linked list
    MEMORY_MANAGEMENT
} stacked_variable_owner_list;

int max_frame_size_needed = 0;
```

The structure stacked_variable is private to this section.
The structure stacked_variable_list is private to this section.
The structure stacked_variable_owner is private to this section.
The structure stacked_variable_owner_list is private to this section.

```
void stv_compile_lvalue(OUTPUT_STREAM, stacked_variable *stv) {
    if ((stv->owner_id == ACTION_PROCESSING_RB) && (stv->offset_in_owning_frame == 0)) {
        WRITE("actor");
        return;
    }
    WRITE("(MStack-->MstVO(%d,%d))",
        stv->owner_id, stv->offset_in_owning_frame);
}

void stv_compile_rvalue_to_text(char *x, stacked_variable *stv) {
    if ((stv->owner_id == ACTION_PROCESSING_RB) && (stv->offset_in_owning_frame == 0)) {
        sprintf(x, "actor");
```

```
        return;
    }
    sprintf(x, "(MStack-->MstVON(%d,%d))",
        stv->owner_id, stv->offset_in_owning_frame);
}

int stv_get_owner_id(stacked_variable *stv) {
    return stv->owner_id;
}

int stv_get_offset(stacked_variable *stv) {
    return stv->offset_in_owning_frame;
}

quantity *stv_get_quantity(stacked_variable *stv) {
    if (stv == NULL) return NULL;
    return stv->underlying_quantity;
}

void stv_set_matching_text(stacked_variable *stv, int w1, int w2) {
    stv->mw1 = w1; stv->mw2 = w2;
}

void stv_get_matching_text(stacked_variable *stv, int *w1, int *w2) {
    *w1 = stv->mw1; *w2 = stv->mw2;
}

stacked_variable *stvo_parse_match_clause(stacked_variable_owner *stvo,
    int w1, int w2) {
    stacked_variable_list *stvl;
    for (stvl = stvo->list_of_stvs; stvl; stvl = stvl->next) {
        int kw1 = stvl->the_stv->mw1, kw2 = stvl->the_stv->mw2;
        if (w2-w1 < kw2-kw1) continue;
        if (compare_word_range(kw1, kw2, w1, w1+kw2-kw1))
            return stvl->the_stv;
    }
    return NULL;
}

stacked_variable_owner *stvo_new(int id) {
    stacked_variable_owner *stvo = CREATE(stacked_variable_owner);
    stvo->recognition_id = id;
    stvo->no_stvs = 0;
    stvo->list_of_stvs = NULL;
    return stvo;
}

int stvo_empty(stacked_variable_owner *stvo) {
    if (stvo->no_stvs == 0) return TRUE;
    return FALSE;
}

stacked_variable *stvo_add(stacked_variable_owner *stvo, int w1, int w2,
    kind_of_value *kov, world_object *kind_required) {
    stacked_variable *stv = CREATE(stacked_variable);
    quantity *q;
    [[w1, w2 == the ... --> w1, w2]];
    specification *spec = parse_expression(w1, w2, TYPE_OR_VALUE_EXPCON);
    if (spec_is_UNKNOWN(spec) == FALSE) {
        LOG("Offending SP: $X", spec);
```

```
        quote_source(1, current_sentence);
        quote_words(2, w1, w2);
        handmade_problem(_P_(C12StackedVariableForbidden));
        issue_problem_segment(
            "You wrote %1, but '%2' already has a meaning, which this would "
            "clash with.");
        issue_problem_end();
    }
    if (kind_required) kov = kovko(kind_required);
    q = new_stacked_quantity(w1, w2, kov, stv);
    stv->word_ref1 = w1;
    stv->word_ref2 = w2;
    stv->underlying_quantity = q;
    stv->owner_id = stvo->recognition_id;
    stv->offset_in_owning_frame = stvo->no_stvs++;
    stv->assigned_at = current_sentence;
    stv->mw1 = -1; stv->mw2 = -1;
    stvo->list_of_stvs = stvl_add_to_list(stvo->list_of_stvs, stv);
    if (stvo->no_stvs > max_frame_size_needed)
        max_frame_size_needed = stvo->no_stvs;
    return stv;
}

stacked_variable_owner_list *stvol_add_owner(stacked_variable_owner_list *stvol,
    stacked_variable_owner *stvo) {
    stacked_variable_owner_list *nstvol = CREATE(stacked_variable_owner_list),
        *ostvol = stvol;
    nstvol->next = NULL;
    nstvol->stvo = stvo;
    if (stvol == NULL) return nstvol;
    while (stvol->next) stvol = stvol->next;
    stvol->next = nstvol;
    return ostvol;
}

stacked_variable_owner_list *stvol_append(stacked_variable_owner_list *stvol,
    stacked_variable_owner_list *extras) {
    stacked_variable_owner_list *new_head = stvol, *new_tail = new_head;
    while ((new_tail) && (new_tail->next)) new_tail = new_tail->next;
    while (extras) {
        stacked_variable_owner_list *nstvol = CREATE(stacked_variable_owner_list);
        *nstvol = *extras;
        nstvol->next = NULL;
        if (new_head == NULL) new_head = nstvol; else new_tail->next = nstvol;
        new_tail = nstvol;
        extras = extras->next;
    }
    return new_head;
}

int stvl_length(stacked_variable_list *stvl) {
    int l = 0;
    while (stvl) {
        l++;
        stvl = stvl->next;
    }
```

```
    return l;
}
void stvo_index(stacked_variable_owner *stvo) {
    stacked_variable_list *stvl;
    for (stvl=stvo->list_of_stvs; stvl; stvl = stvl->next)
        if ((stvl->the_stv) && (stvl->the_stv->underlying_quantity))
            qty_index_single(stvl->the_stv->underlying_quantity);
}
stacked_variable *stvol_parse(stacked_variable_owner_list *stvol, int w1, int w2) {
    if ((w1<0) || (w2<w1)) return NULL;
    [[w1, w2 == the ... --> w1, w2]];
    while (stvol) {
        stacked_variable *stv = NULL;
        if (stvol->stvo) stv = stvl_parse(stvol->stvo->list_of_stvs, w1, w2);
        if (stv) return stv;
        stvol = stvol->next;
    }
    return NULL;
}
stacked_variable *stvl_parse(stacked_variable_list *stvl, int w1, int w2) {
    while (stvl) {
        if (compare_word_range(stvl->the_stv->word_ref1, stvl->the_stv->word_ref2, w1, w2))
            return stvl->the_stv;
        stvl = stvl->next;
    }
    return NULL;
}
stacked_variable_list *stvl_add_to_list(stacked_variable_list *stvl,
    stacked_variable *stv) {
    stacked_variable_list *nstvl = CREATE(stacked_variable_list), *ostvl = stvl;
    nstvl->the_stv = stv;
    nstvl->next = NULL;
    if (stvl == NULL) return nstvl;
    while (stvl->next) stvl = stvl->next;
    stvl->next = nstvl;
    return ostvl;
}
int stvl_compile_frame_creator(OUTPUT_STREAM, stacked_variable_owner *stvo,
    char *name_prototype, int name_id) {
    int i;
    stacked_variable_list *stvl;
    if (stvo == NULL) return 0;
    WRITE("[ ");
    WRITE(name_prototype, name_id);
    WRITE(" pos state;\n"); INDENT;
    WRITE("if (state == 1) {\n"); INDENT;
    for (i=0, stvl = stvo->list_of_stvs; stvl; i++, stvl = stvl->next) {
        quantity *q = stv_get_quantity(stvl->the_stv);
        kind_of_value *kov = qty_kind_of_value(q);
        WRITE("MStack-->pos = ");
        if (kov_uses_pointer_values(kov)) compile_heap_allocation(OUT, kov, 1, FALSE);
        else compile_quantity_initial(OUT, q);
```

```
        WRITE("; pos++;\n");
    }
    OUTDENT;
    WRITE("} else {\n");
    INDENT;
    for (i=0, stvl = stvo->list_of_stvs; stvl; i++, stvl = stvl->next) {
        quantity *q = stv_get_quantity(stvl->the_stv);
        kind_of_value *kov = qty_kind_of_value(q);
        if (kov_uses_pointer_values(kov)) WRITE("BlkFree(MStack-->pos);\n");
        WRITE("pos++;\n");
    }
    OUTDENT; WRITE("}\n");
    WRITE("return %d;\n", i);
    OUTDENT; WRITE("];\n");
    return i;
}
```

The function stv_compile_lvalue is called from 9/qty.
The function stv_compile_rvalue_to_text is called from 11/ap.
The function stv_get_owner_id is called from 11/ap.
The function stv_get_offset is called from 11/ap.
The function stv_get_quantity is called from 5/tandv.
The function stv_set_matching_text is called from 11/act.
The function stvo_parse_match_clause is called from 11/act.
The function stvo_new is called from 11/act, 11/av and 12/rb.
The function stvo_empty is called from 11/act, 11/av and 12/rb.
The function stvo_add is called from 11/act, 11/av and 12/rb.
The function stvol_add_owner is called from 11/act and 12/rb.
The function stvol_append is called from 12/br.
The function stvo_index is called from 11/act.
The function stvol_parse is called from 5/tandv.
The function stvl_compile_frame_creator is called from 11/act, 11/av and 12/rb.

Compile Invocations 12/cinv

Purpose

Here we generate Inform 6 code to execute the phrase(s) called for by an invocation list.

Template interpreter commands

```
2   {-callv:compile_list_together_routines}
21  {-callv:compile_resolver_routines}
```

Definitions

¶1. The arguments packet. Just a data structure used in assembling phrases for compilation. Unlike most NI structures, these do not hang around in memory for any length of time, and are not worth further comment.

```
typedef struct args_packet {
    int args_count;                                  number of arguments to phrase
    struct specification *args[32];                  what they are
    struct kind_of_value *pointer_kov[32];           what pointer kinds of value, if they are
} args_packet;
```

The structure args_packet is private to this section.

¶2. List together routines. Another uninteresting data structure. One of these is created for each distinct routine needed as a value of the I6 property `list_together` at run-time.

```
typedef struct list_together_routine {
    int articles_bit;                                if false, add NOARTICLE_BIT to the I6 listing style
    MEMORY_MANAGEMENT
} list_together_routine;
```

The structure list_together_routine is private to this section.

¶3.

define `MAX_INLINE_DEFN_LENGTH 1024`

---

§**1. Compilation of invocations.** Recall that a phrase SP contains a list of invocations, batched into groups. Each group contains a range of possible things to do, to be distinguished by run-time type checking: only one will ever happen. (So for unambiguous code, a group consists of a single invocation.) In this section we take such a SP and compile the I6 code which carries it out. There are three levels: bottom level, a single invocation; middle level, a group of invocations; and top level, the entire list.

In fact, before we can get to the actual compilation, we need to supply some support infrastructure for the extraneous material which can also sometimes be compiled in the course of an invocation.

§**2. List together routines.** These are minimal data structures. Each time we request an LTR we get a fresh numerical ID `N`, and the I6 source code then contains a routine with name `LTR_N`. The source for this routine is always one of only two possibilities, so it looks rather odd to keep generating fresh copies. We do this so that the routine addresses can be used as distinct values of the `list_together` property at run-time, because the I6 library uses distinctness of this property value to determine whether things are or are not listed together.

```
int new_list_together(int include_articles) {
    list_together_routine *ltr = CREATE(list_together_routine);
    ltr->articles_bit = include_articles;
    return ltr->allocation_id;
}
void compile_list_together_routines(OUTPUT_STREAM) {
    list_together_routine *ltr;
    LOOP_OVER(ltr, list_together_routine) {
        WRITE("[ LTR_%d;\n", ltr->allocation_id); INDENT;
        WRITE("if (inventory_stage == 1) {\n"); INDENT;
        if (ltr->articles_bit)
            WRITE("c_style = c_style | (ENGLISH_BIT);\n");
        else
            WRITE("c_style = c_style | (ENGLISH_BIT + NOARTICLE_BIT);\n");
        WRITE("c_style = c_style &~ (NEWLINE_BIT + INDENT_BIT);\n");
        OUTDENT; WRITE("}\n");
        WRITE("rfalse;\n");
        OUTDENT; WRITE("];\n");
    }
}
```

The function compile_list_together_routines is invoked by a command in a `.i6t` template file.

§**3. Function calls for phrases.** When an invocation of a phrase is compiled, it either becomes a function call to a function like `R_102`, or else it is compiled using an "inline definition" into other I6 statements. The following routine compiles the `R_102(...)` function call needed for non-inline-definitions, then. But it is also used for calls like `Resolver_3(...)` used to perform run-time type checking; and for that, we need to know where the phrase came from in the source text, so that an informative run-time problem message can be produced if that type-checking should fail.

```
void compile_function_call(OUTPUT_STREAM, source_location *where_from,
    args_packet *argsp, int resolver, char *identifier, int phrase_options,
    kind_of_value *block_type_returned, kind_of_value *block_resolved,
    int within_resolver) {
    int k, ar = 0;
    BEGIN_COMPILATION_MODE;
    COMPILATION_MODE_ENTER(DEREFERENCE_POINTERS_CMODE);
    WRITE("(%s(", identifier);
    if (block_type_returned) {
        if (ar++ > 0) WRITE(",");
        if (within_resolver) WRITE("rvblock");
        else {
            pointer_allocation *pall =
                phsf_add_allocation(block_type_returned, NULL);
            WRITE("%s", pall_get_local_reference(pall));
        }
```

```
    }
    for (k=0; k<argsp->args_count; k++) {
        int promote = FALSE, cmode = 0;
        if (ar++ > 0) WRITE(",");
        if ((TEST_COMPILATION_MODE(INSIDE_RESOLVER_CMODE) == FALSE) &&
            (argsp->pointer_kov[k] != NULL) &&
            (kov_compare(argsp->pointer_kov[k],
                spec_evaluates_to(argsp->args[k])) == FALSE))
            promote = TRUE;
        if (promote) {
            pointer_allocation *pall = phsf_add_allocation(argsp->pointer_kov[k], NULL);
            WRITE("BlkValueCast(%s, %d, %d, ",
                pall_get_local_reference(pall),
                kov_I6_ID(argsp->pointer_kov[k]),
                kov_I6_ID(spec_evaluates_to(argsp->args[k])));
            cmode = PERMIT_LOCALS_IN_TEXT_CMODE;
        }
        BEGIN_COMPILATION_MODE;
        COMPILATION_MODE_ENTER(cmode);
        spec_compile(OUT, argsp->args[k]);
        if (promote) WRITE(")");
        END_COMPILATION_MODE;
    }
    if (phrase_options != -1) {
        if (ar++ > 0) WRITE(",");
        WRITE("%d", phrase_options);
    }
    if (resolver) {
        if (ar++ > 0) WRITE(",");
        if (block_resolved) {
            pointer_allocation *pall = phsf_add_allocation(block_resolved, NULL);
            WRITE("%s", pall_get_local_reference(pall));
            if (ar++ > 0) WRITE(", ");
        }
        WRITE("\"source\", %d", where_from->line_number);
    }
    WRITE("))");
    END_COMPILATION_MODE;
}
```

§4. **Compiling single invocations.** Strictly speaking, the following routine does not quite compile the contents of an invocation: this is because it ignores the values of the tokens as parsed, and draws the necessary arguments from a "packet" passed alongside the invocation. Now in fact this packet will in almost all cases be the same as the set of values of tokens as parsed, but not when we are in a type-resolution routine; so it seems wisest to abstract the arguments like this.

```
int compile_single_invocation(OUTPUT_STREAM, invocation *inv,
    source_location *where_from, args_packet *mock_argsp, int within_resolver) {
    BEGIN_COMPILATION_MODE;
    int rv = compile_single_invocation_dash(OUT, inv, where_from, mock_argsp, within_resolver);
    END_COMPILATION_MODE;
    return rv;
}

int compile_single_invocation_dash(OUTPUT_STREAM, invocation *inv,
    source_location *where_from, args_packet *mock_argsp, int within_resolver) {
    phrase *ph = inv->phrase_invoked;
    int singleton_block = FALSE;
    int phrase_spec_specific_type = UNKNOWN;
    int dereference_pointer = TRUE, i;
    STREAM_MUST_BE_IN_MEMORY(OUT);
    LOGIF(INVOCATIONS, "Compiling single invocation: $e\n", inv);
    int start_position = STREAM_EXTENT(OUT);
    for (i=0; i<mock_argsp->args_count; i++) {
        kind_of_value *kov = NULL;
        if (spec_is_generic_CONSTANT(ph->type_data.tokens_spec[i]))
            kov = spec_get_kind_of_value(ph->type_data.tokens_spec[i]);
        else kov = spec_evaluates_to(ph->type_data.tokens_spec[i]);
        LOGIF(INVOCATIONS, "Argument %d ($S) when eval is $u\n",
            i, ph->type_data.tokens_spec[i], kov);
        mock_argsp->pointer_kov[i] = NULL;
        if (kov_uses_pointer_values(kov)) {
            mock_argsp->pointer_kov[i] = kov;
            LOGIF(INVOCATIONS, "And argument %d has pointer kov $u\n", i, kov);
        }
    }
    ⟨CSI - Start of phrase 5⟩;
    if (ph->type_data.invoked_inline_not_as_call == TRUE) {
        ⟨CSI - Inline 6⟩;
    } else {
        ⟨CSI - By function call 13⟩;
    }
    if (inv->phrase_invoked->type_data.block_follows) {
        if (ph->type_data.invoked_inline_not_as_call == TRUE) {
            WRITE("{");
            open_code_block(ph);
            LOGIF(INVOCATIONS, "Opening code block for $e\n", inv);
            if (inv_get_number_tokens(inv) > 0) {
                kind_of_value *kov = spec_evaluates_to(inv_get_token_as_parsed(inv, 0));
                LOGIF(INVOCATIONS, "Setting kov as $u\n", kov);
                set_kov_for_current_block(kov);
            }
```

```
        } else {
            sentence_problem(_P_(C12NonInlineBeginEnd),
                "phrases can only be used in definitions when they are given "
                "inline Inform 6 translations",
                "and if that doesn't make sense then you probably shouldn't be "
                "trying to do things like this.");
            return phrase_spec_specific_type;
        }
    }
    ⟨CSI - End of phrase 14⟩;
    if (phrase_spec_specific_type != UNKNOWN) {
        LOGIF(INVOCATIONS, "Single invocation return manner: %d\n",
            phrase_spec_specific_type);
    }
    return phrase_spec_specific_type;
}
```

§**5.** Phrases to decide a condition must compile to valid I6 conditions, which we need to ensure there are round brackets around.

⟨CSI - Start of phrase 5⟩ ≡

```
    switch (ph->type_data.manner_of_return) {
        case DECIDES_CONDITION_MOR:
            WRITE("(");
            break;
    }
```

This code is used in §4.

§**6.** CSI: Inline, the new criminal forensics show. Jack Corrizo, lonely but brilliant scene of crime officer, tells it like it is, and as serial killers stalk the troubled city of Inline, Missouri, ... Oh, very well: this is the code which turns an inline phrase definition into I6 code. That sounds like an elementary matter of copying it out, but we need to expand material in curly braces, according to what amounts to a mini-language. The exact definition of this mini-language was the subject of some speculation in the early days of the I7 Public Beta: but the exotic features it contains were never meant to be used anywhere except by the Standard Rules.

As the following shows, the inline definition is copied literally except for commands inside `{` and `}`, except that `{` followed by a space is treated as literal and not a command opening.

⟨CSI - Inline 6⟩ ≡

```
{
    char inline_definition[MAX_INLINE_DEFN_LENGTH];
    int j, tok, definition_length;
    ⟨CSI - I - Read inline definition 7⟩;
    definition_length = strlen(inline_definition);
    for (j=0; j<definition_length; j++) {
        if ((inline_definition[j] == '{') &&
            (inline_definition[j+1] != ' ') && (inline_definition[j+1] != '\t') &&
            (inline_definition[j+1] != '\n') && (inline_definition[j+1] != '|')) {
            char substitution[MAX_INLINE_DEFN_LENGTH];
            char command[MAX_INLINE_DEFN_LENGTH];
            char *extremal_property_name = NULL;
```

```
            int extremal_property_sign = 0;
            ⟨CSI - I - Parse braced command 8⟩;
            ⟨CSI - I - Obey substitutionless commands 9⟩;
            feed_into_lexer(substitution, FALSE, FALSE);
            ⟨CSI - I - Obey non-token substitutions 10⟩;
            tok = parse_name_local_to_inline_stack_frame(ph,
                lexer_feed_w1, lexer_feed_w2);
            if ((tok >= 0) && (tok < inv_get_number_tokens(inv))) {
                ⟨CSI - I - Obey token substitutions 11⟩;
            }
            ⟨CSI - I - Error in braced command 12⟩;
        } else {
            WRITE("%c", inline_definition[j]);
        }
    }
}
```

This code is used in §4.

§**7.** Extract the phrase's inline definition, and its return type.

```
⟨CSI - I - Read inline definition 7⟩ ≡
{
    int outcome_mor = ph_To_inline_read_outcome_mor(ph);
    if (outcome_mor != UNKNOWN) phrase_spec_specific_type = outcome_mor;
    ph_To_inline_write_definition(ph, inline_definition);
}
```

This code is used in §6.

§**8.** A general braced command can be any of the following.

```
{-command}
{some text}
{-annotation:some text}
{-annotation<property name:some text}
{-annotation>property name:some text}
```

We parse this with the command or annotation in `command`, the "some text" in `substitution`, and the property name (if given) in `extremal_property_name`.

```
⟨CSI - I - Parse braced command 8⟩ ≡
{
    int k;
    for (k=0; (k<MAX_INLINE_DEFN_LENGTH-1)&&(inline_definition[j+1+k] != '}'); k++)
        substitution[k] = inline_definition[j+1+k];
    j += k+1;
    substitution[k] = 0;
    command[0] = 0;
    if (substitution[0] == '-') {
        for (k=1; (substitution[k]) && (substitution[k] != ':'); k++)
            command[k-1] = substitution[k];
        command[k-1] = 0;
        if (substitution[k] == 0) substitution[0] = 0;
```

```
        else {
            int k2; k++;
            for (k2=k; substitution[k2]; k2++)
                substitution[k2-k] = substitution[k2];
            substitution[k2-k] = 0;
        }
    }
    for (k=0; command[k]; k++) {
        if (command[k] == '>') extremal_property_sign = 1;
        if (command[k] == '<') extremal_property_sign = -1;
        if (extremal_property_sign != 0) {
            command[k] = 0;
            extremal_property_name = command+k+1;
            break;
        }
    }
}
```

This code is used in §6.

§**9.** The raw commands, which have no text supplied. (Strictly speaking, the `label` and `next-label` commands do optionally take a substitution, but in almost all cases this is the empty text.)

⟨CSI - I - Obey substitutionless commands 9⟩ ≡

```
    if (command[0]) {
        if (strcmp(command, "delete") == 0) {
            STREAM_BACKSPACE(OUT);
            continue;
        }
        if (strcmp(command, "erase") == 0) {
            STREAM_ERASE_BACK_TO(start_position);
            continue;
        }
        if (strcmp(command, "label") == 0) {
            lns_write(substitution, OUT, TRUE);
            continue;
        }
        if (strcmp(command, "next-label") == 0) {
            lns_write(substitution, OUT, FALSE);
            continue;
        }
        if (strcmp(command, "segment-count") == 0) {
            WRITE("%d", inv->ssp_segment_count);
            continue;
        }
        if (strcmp(command, "final-segment-marker") == 0) {
            if (inv->ssp_closing_segment_wn == -1) WRITE("NULL");
            else {
                char plenty[1024];
                print_text_to_string(inv->ssp_closing_segment_wn,
                    inv->ssp_closing_segment_wn, plenty);
                WRITE("%s", plenty);
            }
            continue;
```

```
}
if (strcmp(command, "counter") == 0) {
    WRITE("%d", lns_read_counter(substitution, FALSE));
    continue;
}
if (strcmp(command, "advance-counter") == 0) {
    WRITE("%d", lns_read_counter(substitution, TRUE));
    continue;
}
if (strcmp(command, "zero-counter") == 0) {
    lns_zero_counter(substitution);
    continue;
}
if (strcmp(command, "allocate-storage") == 0) {
    lns_allocate_storage(substitution, 1);
    continue;
}
if (strcmp(command, "list-together") == 0) {
    WRITE("LTR_%d", new_list_together(FALSE));
    continue;
}
if (strcmp(command, "articled-list-together") == 0) {
    WRITE("LTR_%d", new_list_together(TRUE));
    continue;
}
if (strcmp(command, "open-brace") == 0) {
    WRITE("{");
    continue;
}
if (strcmp(command, "close-brace") == 0) {
    WRITE("}");
    continue;
}
if (strcmp(command, "rescale-times") == 0) {
    kind_of_value *kovx = spec_evaluates_to(inv_get_token_as_parsed(inv, 0));
    kind_of_value *kovy = spec_evaluates_to(inv_get_token_as_parsed(inv, 1));
    kov_rescale_multiplication(OUT, kovx, kovy);
    continue;
}
if (strcmp(command, "rescale-divide") == 0) {
    kind_of_value *kovx = spec_evaluates_to(inv_get_token_as_parsed(inv, 0));
    kind_of_value *kovy = spec_evaluates_to(inv_get_token_as_parsed(inv, 1));
    kov_rescale_division(OUT, kovx, kovy);
    continue;
}
if (strcmp(command, "rescale-root") == 0) {
    kind_of_value *kov = spec_evaluates_to(inv_get_token_as_parsed(inv, 0));
    kov_rescale_root(OUT, kov, 2);
    continue;
}
if (strcmp(command, "rescale-cuberoot") == 0) {
    kind_of_value *kov = spec_evaluates_to(inv_get_token_as_parsed(inv, 0));
    kov_rescale_root(OUT, kov, 3);
```

```
            continue;
        }
        if (strcmp(command, "require-ctvs") == 0) {
            we_need_ct_locals();
            continue;
        }
        if (strcmp(command, "push-ctvs") == 0) {
            we_need_ct_locals();
            WRITE("@push ct_0; @push ct_1;");
            save_ct_variables_in_current_loop();
            continue;
        }
        if (strcmp(command, "ct-v0") == 0) {
            int var0 = ct_variables_saved_in_this_block();
            if (var0 >= 0) WRITE("t_%d", var0);
            else internal_error("unable to form ct variable 0");
            continue;
        }
        if (strcmp(command, "ct-v1") == 0) {
            int var1 = ct_variables_saved_in_this_block() + 1;
            if (var1 >= 1) WRITE("t_%d", var1);
            else internal_error("unable to form ct variable 1");
            continue;
        }
        if (strcmp(command, "assignment") == 0) {
            pcalc_term pt1 = term_new_constant(mock_argsp->args[0]);
            pcalc_term pt2 = term_new_constant(mock_argsp->args[1]);
            kind_of_value *kov1 = spec_evaluates_to(mock_argsp->args[0]);
            kind_of_value *kov2 = spec_evaluates_to(mock_argsp->args[1]);
            int storage_class = spec_get_storage_form(mock_argsp->args[0]);
            char *prototype = kov_interpret_store(storage_class, kov1, kov2);
            sch_expand(sch_new(prototype), OUT, &pt1, &pt2);
            continue;
        }
        if (strcmp(command, "solve-equation") == 0) {
            compile_solution(OUT,
                mock_argsp->args[0]->word_ref1, mock_argsp->args[0]->word_ref2,
                EQUATION_spec_to_equation(mock_argsp->args[1]));
            WRITE(";\n");
            continue;
        }
    }
    if (strcmp(command, "pointer-to-new") == 0) {
        kind_of_value *kov = kova(type_ID_get_ID_for_source_name(substitution));
        if (kov == NULL) continue;
        pointer_allocation *pall = phsf_add_allocation(kov, NULL);
        WRITE("%s", pall_get_local_reference(pall));
        continue;
    }
```

This code is used in §6.

§**10.** The only cases where the text in a substitution can be other than the name of one of the tokens: "phrase options", and the name of any individual phrase option.

⟨CSI - I - Obey non-token substitutions 10⟩ ≡

```
{ int popt;
    if [[lexer_feed_w1, lexer_feed_w2 == phrase options]] {
        WRITE("%d", inv_get_phrase_options_bitmap(inv));
        continue;
    }
    popt = ph_To_parse_phrase_option_used(ph, lexer_feed_w1, lexer_feed_w2);
    if (popt >= 0) {
        if (((inv_get_phrase_options_bitmap(inv)) & popt) != 0) WRITE("true");
        else WRITE("false");
        continue;
    }
}
```

This code is used in §6.

§**11.** At this point, the substitution text is the name of token `tok`. Usually we compile the value of that argument as drawn from the arguments packet, but the presence of annotations can change what we do.

⟨CSI - I - Obey token substitutions 11⟩ ≡

```
{
    int allow_command = FALSE;
    specification *type_needed = ph->type_data.tokens_spec[tok];
    specification *type_supplied = mock_argsp->args[tok];
    if ((family_is(type_needed, COMMAND_FMY)) &&
        (species_is(type_needed, TRY_ACTION_SPC) == FALSE)) {
        singleton_block = TRUE;
        WRITE("{ ");
    }
    if (strcmp(command, "pointer-to") == 0) {
        kind_of_value *kov_needed =
            spec_get_kind_of_value(type_needed);
        kind_of_value *kov_found =
            spec_evaluates_to(type_supplied);
        LOG("pointer-to: needed kov $u, found kov $u\n",
            kov_needed, kov_found);
        if (kov_allow_word_as_pointer(kov_needed, kov_found)) {
            pointer_allocation *pall = phsf_add_allocation(kov_needed, NULL);
            WRITE("BlkValueCast(%s,%d,%d,",
                pall_get_local_reference(pall),
                kov_I6_ID(kov_needed), kov_I6_ID(kov_found));
            BEGIN_COMPILATION_MODE;
            COMPILATION_MODE_ENTER(PERMIT_LOCALS_IN_TEXT_CMODE);
            spec_compile(OUT, type_supplied);
            END_COMPILATION_MODE;
            WRITE(")");
            continue;
        }
        dereference_pointer = FALSE;
        allow_command = TRUE;
```

```
} else dereference_pointer = TRUE;
if (strcmp(command, "do-not-dereference") == 0) {
    dereference_pointer = FALSE;
    allow_command = TRUE;
}
if (strcmp(command, "allocate-storage-for") == 0) {
    kind_of_value *kov = KOV_of_recently_created_local;
    if (kov == NULL) {
        LOG("tok=%d, data at tok = $S\n", tok, type_needed);
        internal_error("-allocate-storage-for applied to nonvalue");
    }
    if (kov_uses_pointer_values(kov) == FALSE)
        internal_error("-allocate-storage-for applied to word value");
    WRITE("%s", pall_get_local_reference(
        phsf_add_allocation(kov, NULL)));
    continue;
}
if (strcmp(command, "allow-stack-frame-access") == 0) {
    COMPILATION_MODE_ENTER(PERMIT_LOCALS_IN_TEXT_CMODE);
    allow_command = TRUE;
}
if (strcmp(command, "default-value-for") == 0) {
    LOGIF(INVOCATIONS, "-default-value-for tok=%d, data at tok = $S\n",
        tok, type_supplied);
    compile_default_value(OUT,
        spec_get_kind_of_value(type_supplied), -1, -1, "token");
    continue;
}
if (species_is(type_needed, NEW_LOCAL_VARIABLE_NAME_SPC)) {
    specification *local_t =
        new_LOCAL_VARIABLE_spec(lexer_feed_w1, lexer_feed_w2, phrase_made_local);
    BEGIN_COMPILATION_MODE;
    if (dereference_pointer == FALSE) COMPILATION_MODE_EXIT(DEREFERENCE_POINTERS_CMODE);
    spec_compile(OUT, local_t);
    END_COMPILATION_MODE;
    int pw = ph->type_data.words[0];
    if [[word pw == repeat]]
        set_scope_of_local_to_current_block(phrase_made_local);
    continue;
}
if (strcmp(command, "check-return-type") == 0) {
    int allow_me;
    specification *need =
        phtd_get_return_TS(&(phrase_being_compiled->type_data));
    specification *returning = type_supplied;
    if (spec_is_UNKNOWN(need)) {
        if ((phtd_is_phrase_to_decide_if(&(phrase_being_compiled->type_data))) &&
            (is_kova(spec_evaluates_to(returning), TRUTH_STATE_TY))) {
            LOGIF(INVOCATIONS,
                "Allowing TRUTH_STATE_TY value as return in phrase to decide if\n");
        } else {
            if (phtd_is_phrase_to_decide_if(&(phrase_being_compiled->type_data))) {
```

```
            LOGIF(INVOCATIONS, "Inside phrase to decide if.\n");
        }
        LOGIF(INVOCATIONS,
            "KOV returned is $u\n", spec_evaluates_to(returning));
        quote_source(1, current_sentence);
        quote_kind_of(2, returning);
        handmade_problem(_P_(C12RedundantReturnKOV));
        issue_problem_segment(
            "You wrote %1 as the outcome of a phrase, %2, "
            "but in the definition of something which was not a phrase "
            "to decide a value.");
        issue_problem_end();
    }
    allow_me = ALWAYS_MATCH;                                                    to recover
} else allow_me = can_we_match_value_descriptions(returning, need);
switch(allow_me) {
    case ALWAYS_MATCH: {
        kind_of_value *kov_needed = spec_evaluates_to(need);
        kind_of_value *kov_found = spec_evaluates_to(returning);
        if (kov_allow_word_as_pointer(kov_needed, kov_found)) {
            LOGIF(INVOCATIONS,
                "check-return-type: needed kov $u, found kov $u\n",
                kov_needed, kov_found);
            pointer_allocation *pall = phsf_add_allocation(kov_needed, NULL);
            WRITE("BlkValueCast(%s,%d,%d,",
                pall_get_local_reference(pall),
                kov_I6_ID(kov_needed), kov_I6_ID(kov_found));
            BEGIN_COMPILATION_MODE;
            COMPILATION_MODE_ENTER(PERMIT_LOCALS_IN_TEXT_CMODE);
            spec_compile(OUT, type_supplied);
            END_COMPILATION_MODE;
            WRITE(")");
            continue;
        }
        spec_compile(OUT, returning);
        break;
    }
    case SOMETIMES_MATCH:
        if (spec_get_described_kind(need)) {
            WRITE("CheckKindReturned(");
            spec_compile(OUT, returning);
            WRITE(",%s)", wo_get_I6_representation(spec_get_described_kind(need)));
            break;
        }                                                   and otherwise fall into...
    case NEVER_MATCH:
        quote_source(1, current_sentence);
        quote_kind_of(2, returning);
        quote_kind_of(3, need);
        handmade_problem(_P_(C12ReturnWrongKind));
        issue_problem_segment(
            "You wrote %1 as the outcome of a phrase to decide a "
            "value, but this was the wrong kind of value: %2 rather "
            "than %3.");
```

```
            issue_problem_end();
            break;
    }
    continue;
}
if (strcmp(command, "loop-over") == 0) {
    int v = phrase_made_local;
    int protected_copy_of_v =
        make_new_local(-1, -1, kova(OBJECT_TY));
    compile_repeat_over_S(OUT,
        type_supplied, v, protected_copy_of_v);
    continue;
}
if (strcmp(command, "loop-over-list") == 0) {
    int v = phrase_made_local;
    int index_in_list =
        make_new_local(-1, -1, kova(NUMBER_TY));
    compile_loop_over_list_S(OUT,
        type_supplied, v, index_in_list);
    continue;
}
if (strcmp(command, "loop-over-domain") == 0) {
    int v = phrase_made_local;
    int index_in_list =
        make_new_local(-1, -1, kova(NUMBER_TY));
    compile_loop_over_domain_S(OUT,
        type_supplied, v, index_in_list);
    continue;
}
if (strcmp(command, "domain-kov") == 0) {
    WRITE("%d", kov_I6_ID(spec_get_described_kov(type_supplied)));
    continue;
}
if (strcmp(command, "bind-variable") == 0) {
    compile_test_of_subst__v(OUT,
        type_supplied);
    continue;
}
if (strcmp(command, "number-of") == 0) {
    compile_number_of_S(OUT,
        type_supplied);
    continue;
}
if (strcmp(command, "random-of") == 0) {
    compile_random_of_S(OUT,
        type_supplied);
    continue;
}
if (strcmp(command, "total-of") == 0) {
    if (spec_is_CONSTANT_of_kova(mock_argsp->args[0], PROPERTY_TY) == FALSE)
        sentence_problem(_P_(BelievedImpossible),          type-checking seems to prevent this now
            "this uses 'total ... of' in a way too complicated to handle",
            "since it isn't as simple as working out the total of a single "
```

```
            "property's value. For instance, 'let X be the total weight of "
            "the men' is allowed, but not 'let X be the total height times "
            "width of the rectangles'.");
    else
        compile_total_of_S(OUT,
            PROPERTY_spec_to_property_name(mock_argsp->args[0]),
            mock_argsp->args[1]);
    continue;
}
if ((strcmp(command, "extremal") == 0) &&
    (extremal_property_sign != 0) &&
    (extremal_property_name[0])) {
    property_name *prn;
    feed_into_lexer(extremal_property_name, FALSE, FALSE);
    LOG("EPN: <%s>, %d, %d, <$W>\n",
        extremal_property_name, lexer_feed_w1, lexer_feed_w2,
        lexer_feed_w1, lexer_feed_w2);
    prn = parse_property_name(lexer_feed_w1, lexer_feed_w2);
    if (prn) {
        compile_extremal_of_S(OUT,
            type_supplied, prn, extremal_property_sign);
        continue;
    }
}
if (strcmp(command, "convert-adjectival-constants") == 0) {
    quantity *q = spec_get_constant_quantity_if_any(type_supplied);
    allow_command = TRUE;
    if (q) {
        property_name *pn = kov_get_coinciding_property(qty_kind_of_value(q));
        if (pn != NULL) {
            WRITE("%s", prn_get_i6_identifier(pn));
            continue;
        }
    }
}
if (strcmp(command, "adjective-definition") == 0) {
    property_name *prn = spec_get_property_name_if_any(type_supplied);
    if (prn) {
        if (prn_condition_of_which_object(prn) == NULL) WRITE("0");
        else {
            adjectival_phrase *aph =
                aph_parse(type_supplied->word_ref1, type_supplied->word_ref2);
            if (aph)
                WRITE("Adj_%d_t%d_v%d",
                    aph->allocation_id, NOW_ADJECTIVE_TRUE_TASK, OBJECT_TY);
            else
                internal_error("can't reconstitute adjective definition");
        }
    }
    continue;
}
if ((spec_is_CONSTANT_of_kova(type_needed, UNDERSTANDING_TY)) &&
        ((spec_is_CONSTANT_of_kova(type_supplied, TEXT_TY)) ||
```

```
            (spec_is_CONSTANT_of_kova(type_supplied, TEXT_ROUTINE_TY)))) {
        spec_set_kind_of_value(type_supplied, kova(UNDERSTANDING_TY));
    }
    if ((command[0]) && (allow_command == FALSE)) {
        quote_source(1, current_sentence);
        quote_text(2, command);
        handmade_problem(_P_(C12BadInlineTag));
        issue_problem_segment(
            "I attempted to compile %1 using its inline definition, "
            "but this contained the invalid annotation '%2'.");
        issue_problem_end();
        continue;
    }
    BEGIN_COMPILATION_MODE;
    if (dereference_pointer == FALSE) COMPILATION_MODE_EXIT(DEREFERENCE_POINTERS_CMODE);
    spec_compile(OUT, type_supplied);
    END_COMPILATION_MODE;
    continue;
}
```

This code is used in §6.

§**12.** Only meddling charlatans will ever see this message. And they deserve it.

⟨CSI - I - Error in braced command 12⟩ ≡

```
    sentence_problem(_P_(C12BadInlineExpansion),
        "when I expanded that command using its inline definition, "
        "I ran into an expansion written in braces { and } which "
        "was not one of the names of tokens from its specification",
        "so I don't know how to compile this. What you've defined "
        "here is not a phrase which will compile to an I6 "
        "routine, but rather a phrase which will compile 'inline' "
        "directly into the body of other routines: this happened "
        "because it was a 1-command definition, and that command "
        "was an I6 insertion. The ability to write inline phrases "
        "is really intended only for the Standard Rules and other "
        "low-level system extensions, and it is (intentionally) "
        "limited in its flexibility. A good rule of thumb is: if "
        "you can define a phrase without using I6 insertions, "
        "always do so.");
```

This code is used in §6.

§**13.** Compiling an invocation by a function call is simple:

⟨CSI - By function call 13⟩ ≡
```
{ int options_supplied = inv_get_phrase_options_bitmap(inv);
    kind_of_value *rp_kov = NULL;
    if (inv->phrase_options_invoked == NULL) options_supplied = -1;
    if (kov_uses_pointer_values(spec_get_kind_of_value(ph->type_data.return_type)))
        rp_kov = spec_get_kind_of_value(ph->type_data.return_type);
    compile_function_call(OUT,
        where_from, mock_argsp, FALSE, ph_get_I6_representation(ph),
        options_supplied, rp_kov, NULL, within_resolver);
}
```
This code is used in §4.

§**14.** Here we complete the I6 code for the invocation. Two crucial rules of I7 are contained here: the one which prints a new-line if we say text which ends with punctuation of the right kind, and the one which causes the routine to return `true` in the event of the "instead" keyword being used. The rest of this stuff is just tidying up.

⟨CSI - End of phrase 14⟩ ≡
```
    switch (ph->type_data.manner_of_return) {
        case DECIDES_CONDITION_MOR:
            WRITE(")");
            break;
        case DECIDES_NOTHING_MOR:
            if (ph->type_data.invoked_inline_not_as_call == FALSE)
                WRITE(";");
            if (inv_implies_newline(inv)) {
                if ((mock_argsp->args_count > 0)
                    && (spec_is_CONSTANT_of_kova(mock_argsp->args[0], TEXT_TY))
                    && (text_ending_sentence(
                        mock_argsp->args[0]->word_ref1)))
                    WRITE(" new_line;");
            }
            if (inv_test_flag(inv, INSTEAD_INVFLAG)) {
                WRITE(" rtrue;");
                phrase_spec_specific_type = DECIDES_NOTHING_AND_RETURNS_MOR;
            }
            if (singleton_block)
                WRITE(" }");
    }
```
This code is used in §4.

§15. Middle level: compile invocation blocks. The block is a sublist of the list of invocations in a SP, and it is marked here by the first entry in the sublist (`tref1`) and the first entry beyond the sublist (`tref2`), which is null if the sublist extends to the end of the list.

The block sublist will consist of at least one invocation, and all of these except possibly the last one will be "unproven" by the type-checking apparatus – that is, will not be safe to execute without run-time checking. We must execute exactly one invocation in the block – the first one which is found to be type-safe – or else produce a run-time error message.

It follows that there is only one case where no checking is needed: when the list consists of a single proven invocation. This we compile directly to the I6 stream. In all other cases, we compile a function call to a "resolver routine" to the I6 stream, delegating the choice to an external routine: and we will probably have to compile this routine, too, unless the decision on this block is one that we recognise from an earlier invocation block (in what may be another setting entirely).

```
STREAM resolve_f_struct;
STREAM *resolve_f = NULL;

void compile_invocation_block(OUTPUT_STREAM, specification *spec_found,
    int iref1, int iref2, int compiling_last_inv_group) {
    STREAM_MUST_BE_IN_MEMORY(OUT);
    int first_pos, last_pos, size;                          group boundaries (inclusive)
    invocation *first_inv, *last_inv;                       invocations at these boundaries
    int pos, resolving_by_routine;
    ⟨CIB - Find group boundaries 16⟩;
    if (inv_test_flag(first_inv, SAVE_SELF_INVFLAG)) WRITE("@push self; ");
    LOGIF(INVOCATIONS,
        "Compiling invocation group %d (C%d to C%d) (%s)\n",
        inv_get_group(first_inv), first_pos, last_pos,
        (compiling_last_inv_group)?"final":"intermediate");
    resolving_by_routine = FALSE;
    if (inv_test_flag(first_inv, UNPROVEN_INVFLAG)) {
        resolving_by_routine = TRUE;
        ⟨CIB - Call and create resolver routine 17⟩;
    }
    INVOCATION_VARIABLE(inv);
    LOOP_THROUGH_PART_OF_INVOCATION_LIST(inv, spec_invocation_list(spec_found), pos, first_pos, size)
{
        phrase *ph = inv->phrase_invoked;
        args_packet mock_argsp;
        int returned_in_manner, i;
        LOGIF(INVOCATIONS, "C%d: $e\n", pos, inv);
        mock_argsp.args_count = inv_get_number_tokens(inv);
        for (i=0; i<mock_argsp.args_count; i++) {
            mock_argsp.args[i] = inv_get_token_as_parsed(inv, i);
            mock_argsp.pointer_kov[i] = NULL;
        }
        if (resolving_by_routine) {
            ⟨CIB - Compile invocation as line in resolver routine 18⟩;
        } else {
            ⟨CIB - Compile invocation directly to I6 stream 19⟩;
        }
        if (returned_in_manner != UNKNOWN) {
```

```
            LOGIF(INVOCATIONS, "C%d: returned in manner %d\n", pos,
                returned_in_manner);
            LOGIF(INVOCATIONS, "vs Phrase being compiled: %d\n",
                phrase_being_compiled->type_data.manner_of_return);
            if ((returned_in_manner !=
                     phrase_being_compiled->type_data.manner_of_return)
                && (phrase_being_compiled->type_data.manner_of_return
                    != DECIDES_NOTHING_AND_RETURNS_MOR)) {
                quote_source(1, current_sentence);
                quote_text(2, describe_manner_of_return(returned_in_manner));
                quote_text(3, describe_manner_of_return(
                    phrase_being_compiled->type_data.manner_of_return));
                handmade_problem(_P_(C12WrongEndToPhrase));
                issue_problem_segment(
                    "The line %1 seems to be a way that the "
                    "phrase you're defining can come to an end, with %2, "
                    "but it should always end up with %3.");
                issue_problem_end();
                return;
            }
        }
    }
    if (resolving_by_routine) {
        ⟨CIB - Finish resolver routine 20⟩;
    }
    PullSelfIfNecessary:
    if (inv_test_flag(first_inv, SAVE_SELF_INVFLAG)) WRITE("@pull self; ");
    if ((compiling_last_inv_group) &&
        (first_inv->phrase_invoked->type_data.say_phrase)) {
        WRITE(" .");
        lns_write("Say", OUT, TRUE);
        WRITE(";");
        WRITE(" .");
        lns_write("SayX", OUT, TRUE);
        WRITE(";");
        if (inv_test_flag(first_inv, INSTEAD_INVFLAG)) WRITE(" rtrue;");
    }
}
```

§**16.** The group is a sublist of a linked list of references, each of which points to an invocation. Here we determine the first and last invocations in the sublist, and their position numbers in the full list.

⟨CIB - Find group boundaries 16⟩ ≡
```
first_pos = iref1; last_pos = iref2 - 1; size = last_pos - first_pos + 1;
last_inv = NULL; first_inv = NULL;
INVOCATION_VARIABLE(latest_inv_found);
pos = 0;
LOOP_THROUGH_INVOCATION_LIST(latest_inv_found, spec_invocation_list(spec_found)) {
    if (pos == first_pos) first_inv = latest_inv_found;
    if (pos == last_pos) last_inv = latest_inv_found;
    pos++;
}
if ((first_pos<0) || (last_pos<0) || (last_pos < first_pos))
    internal_error("Bad positions in compile invocation block range");
if ((first_inv == NULL) || (last_inv == NULL))
    internal_error("Bad invs in compile invocation block range");
```
This code is used in §15.

§**17.** We get to here if the first invocation in the group is unproven, meaning that at compile time it was impossible to determine whether it was type-safe to execute. We must therefore compile code to determine this at run-time.

A "resolver routine" abstracts the choice to be made at run-time. Recall that out of all the invocations in the group, at most one is supposed to happen, and we have to choose which at run-time (or, if none happens, produce a run-time error). Each such decision is handled by a routine with a name like `Resolver_2`. (The suffix, here 2, is called the "ambiguity number".) We cache these routines because some decisions occur very commonly, and we don't want to compile more resolver routines than we have to.

The code we immediately compile, then, is a call to this resolver routine, which goes into the I6 stream. If the ambiguity is one that has been cached already, i.e., for which a resolver routine already exists, then that is all that we need to do; so we exit the CIB routine. Otherwise, we need to compile a new resolver routine: these are written serially to a file with handle `resolve_f`, which points to a temporary file created if and when needed.

Two points need to be remembered when compiling the function call, which looks the easiest thing but is not:

(i) The arguments used to call the resolver routine are always those tokens provided by the first invocation in the list. (The type-checker will not allow us to have a list of unproven invocations in which the individual entries have different, or a different number of, token values: so we could just as easily take the arguments from any of the invocations in the list – the result would be the same.)

(ii) All of this code has to be able to work recursively. It might be that the various tokens used as arguments to the resolver routine will themselves require resolution to evaluate. Because of this, we force the type-checker to sort out each of the arguments before beginning the resolver routine header; otherwise we might end up with messily interleaved resolver routines in the `resolve_f` file.

⟨CIB - Call and create resolver routine 17⟩ ≡
```
{
    int r = find_ambiguity_number(spec_found, first_pos, last_pos);
    int i, argc;
    int newly_made = FALSE;
    phrase *ph = first_inv->phrase_invoked;
    args_packet mock_argsp;
    char identifier[32];
```

```
if (r == NONDETERMINABLE) {
    sentence_problem(_P_(BelievedImpossible),
        "something I am trying to translate here has an ambiguous type",
        "meaning that I can't tell what kind of value or phrase "
        "it might be. This is a Problem message which only "
        "turns up when I'm fairly lost, which is why there's "
        "not much clue about the trouble.");
    return;
}
if (r<0) {
    newly_made = TRUE; r = (-r)-1;
}
LOGIF(INVOCATIONS, "Preparing function call to Resolver_%d\n", r);
BEGIN_COMPILATION_MODE;
COMPILATION_MODE_ENTER(DO_NOT_CREATE_LOCAL_VARS_CMODE);
argc = inv_get_number_tokens(first_inv);
mock_argsp.args_count = argc;
for (i=0; i<argc; i++) {
    mock_argsp.args[i] = inv_get_token_as_parsed(first_inv, i);
    if (mock_argsp.args[i] == NULL) internal_error("null mock argument");
    kind_of_value *kov = NULL;
    if (spec_is_generic_CONSTANT(ph->type_data.tokens_spec[i]))
        kov = spec_get_kind_of_value(ph->type_data.tokens_spec[i]);
    else kov = spec_evaluates_to(ph->type_data.tokens_spec[i]);
    LOGIF(INVOCATIONS, "Resolver argument %d ($S) when eval is $u\n",
        i, ph->type_data.tokens_spec[i], kov);
    mock_argsp.pointer_kov[i] = NULL;
    if (kov_uses_pointer_values(kov)) {
        mock_argsp.pointer_kov[i] = kov;
        LOGIF(INVOCATIONS, "Resolver argument %d has pointer kov $u\n", i, kov);
    }
}
for (i=0; i<argc; i++)
    if (spec_is_phrasal(mock_argsp.args[i]))
        typecheck_without_expectations(mock_argsp.args[i]);
END_COMPILATION_MODE;
sprintf(identifier, "Resolver_%d", r);
kind_of_value *resolved_into = spec_get_kind_of_value(ph->type_data.return_type);
if (kov_uses_pointer_values(resolved_into) == FALSE)
    resolved_into = NULL;
compile_function_call(OUT, &(lw_array[spec_found->word_ref1].lw_source),
    &mock_argsp, TRUE, identifier, -1, NULL, resolved_into, FALSE);
if (ph->type_data.manner_of_return == DECIDES_NOTHING_MOR) WRITE(";");
if (newly_made == FALSE) goto PullSelfIfNecessary;
LOGIF(INVOCATIONS,
    "Writing head of new ambiguity routine Resolver_%d\n", r);
if (resolve_f == NULL) {
    resolve_f = &resolve_f_struct;
    if (STREAM_OPEN_IN_MEMORY(resolve_f) == FALSE)
        fatal_error("Out of memory: unable to allocate for resolvers");
}
STREAM_WRITE(resolve_f, "[ Resolver_%d", r);
```

```
    for (i=0; i<argc; i++) STREAM_WRITE(resolve_f, " t_%d", i);
    if (resolved_into) STREAM_WRITE(resolve_f, " rvblock");
    STREAM_WRITE(resolve_f, " filename line;\n");
}
```

This code is used in §15.

§**18.** The structure of a resolver routine is simple: the header; then a list of I6 statements in the form "if run-time type checking works for this invocation, carry it out and return", one for each invocation in the group; then the footer.

Recall that the type-checker nulls out `token_check_to_do` fields as it proves them safe: therefore, the non-null ones left to us here are the ones we have to check.

We compile the invocation to one of the following forms, the first for unproven invocations, the second for proven ones (as the last in the list may well be):

`if (`...run-time-checking...`) { `...execute and return...`}`

...execute and return...

If the phrase invoked returns a value, we return that value as part of "execute and return"; if it is void, we perform it and return `true`. Either way, "...execute and return..." leaves the resolver routine if it is reached. Thus we can only fall through to subsequent invocations if the invocation was unproven and the run-time checking failed.

⟨CIB - Compile invocation as line in resolver routine 18⟩ ≡

```
{
    int i, condition_attached = FALSE;
    BEGIN_COMPILATION_MODE;
    COMPILATION_MODE_EXIT(DEREFERENCE_POINTERS_CMODE);
    COMPILATION_MODE_ENTER(INSIDE_RESOLVER_CMODE);
    for (i=0; i<inv_get_number_tokens(inv); i++) {
        specification *check_against = inv_get_token_check_to_do(inv, i);
        if (check_against != NULL) {
            if (condition_attached == FALSE) {
                STREAM_WRITE(resolve_f, "if (");
            } else STREAM_WRITE(resolve_f, " && ");
            condition_attached = TRUE;
            if (species_is(check_against, DESCRIPTION_SPC)) {
                STREAM_WRITE(resolve_f, "(");
                specification *spec = new_LOCAL_VARIABLE_spec(-1, -1, i);
                compile_test_if_var_matches_description(resolve_f, spec, check_against);
                STREAM_WRITE(resolve_f, ")");
                continue;
            }
            if (spec_is_generic_CONSTANT(check_against)) {
                kind_of_value *kov = spec_get_kind_of_value(check_against);
                if (is_kova(kov, OBJECT_TY)) {
                    STREAM_WRITE(resolve_f, "(metaclass(t_%d) == Object)", i);
                    continue;
                }
                if (is_kova(kov, TEXT_TY)) {
                    STREAM_WRITE(resolve_f, "(metaclass(t_%d) == String)", i);
                    continue;
                }
                if ((is_kova(kov, RULE_TY)) ||
```

```
                (is_kova(kov, TEXT_ROUTINE_TY))) {
                STREAM_WRITE(resolve_f, "(metaclass(t_%d) == Routine)", i);
                continue;
            }
            LOG("Error on: $X", check_against);
            internal_error("bad const check-against in run-time type check");
        }
        if (family_is(check_against, VALUE_FMY)) {
            STREAM_WRITE(resolve_f, "(t_%d == ", i);
            spec_compile(resolve_f, check_against);
            STREAM_WRITE(resolve_f, ")");
            continue;
        }
        if (species_is(check_against, NONLOCAL_VARIABLE_SPC)) {
            STREAM_WRITE(resolve_f, "(t_%d == ", i);
            spec_compile(resolve_f, check_against);
            STREAM_WRITE(resolve_f, ")");
            continue;
        }
        LOG("Error on: $X", check_against);
        internal_error("bad check-against in run-time type check");
    }
}
if (condition_attached) STREAM_WRITE(resolve_f, ") { ");
if (ph->type_data.manner_of_return != DECIDES_NOTHING_MOR) STREAM_WRITE(resolve_f, "return ");
for (i=0; i<inv_get_number_tokens(inv); i++)
    mock_argsp.args[i] = new_LOCAL_VARIABLE_spec(-1, -1, i);
returned_in_manner =
    compile_single_invocation(resolve_f, inv,
        &(lw_array[spec_found->word_ref1].lw_source), &mock_argsp, TRUE);
if (ph->type_data.manner_of_return != DECIDES_NOTHING_MOR) STREAM_WRITE(resolve_f, ";");
else STREAM_WRITE(resolve_f, " rtrue;");
if (condition_attached) STREAM_WRITE(resolve_f, " }");
STREAM_WRITE(resolve_f, "\n");
END_COMPILATION_MODE;
}
```

This code is used in §15.

§**19.** The much simpler alternative, if no resolver routine is being made:

⟨CIB - Compile invocation directly to I6 stream 19⟩ ≡

```
returned_in_manner = compile_single_invocation(OUT,
    inv, &(lw_array[spec_found->word_ref1].lw_source), &mock_argsp, FALSE);
```

This code is used in §15.

§**20.** The resolver routine footer can only be reached if all of the invocations fail run-time checking, so unless we can prove that the last invocation will always happen, we need to print a run-time error message if execution gets there.

⟨CIB - Finish resolver routine 20⟩ ≡

```
    if (inv_test_flag(last_inv, UNPROVEN_INVFLAG)) {
        LOGIF(INVOCATIONS, "Final inv unproven, so resolver falls to RTE\n");
        STREAM_WRITE(resolve_f, "ArgumentTypeFailed(filename, line);\n");
    }
    STREAM_WRITE(resolve_f, "  rfalse; ];\n");
```

This code is used in §15.

§**21.** Of course, we also have to merge these resolver routines into the main code, once all passion is spent.

```
void compile_resolver_routines(OUTPUT_STREAM) {
    if (resolve_f) {
        STREAM_COPY(OUT, resolve_f);
        STREAM_CLOSE(resolve_f);
        resolve_f = NULL;
    }
}
```

The function compile_resolver_routines is invoked by a command in a `.i6t` template file.

§**22. Top level: compile all the invocations of a phrase.** We do a little initialisation if the list turns out to be a set of things, one said after another, because special rules apply to "say": but otherwise we pretty much simply find each group in turn and pass it down to the middle level.

define MAX_COMPLEX_SAY_DEPTH 32 *and it would be terrible coding style to approach this*

```
void compile_invocational_phrase(OUTPUT_STREAM, specification *spec_found) {
    int say_initialised = FALSE;
    int group_count, say_if_nesting = 0, problem_issued = FALSE,
        it_was_not_worth_adding = it_is_not_worth_adding;
    int SSP_stack[MAX_COMPLEX_SAY_DEPTH];
    invocation *SSP_invocations[MAX_COMPLEX_SAY_DEPTH];
    int SSP_sp = 0, i;
    it_is_not_worth_adding = TRUE;
    INVOCATION_VARIABLE(inv);
    LOOP_THROUGH_INVOCATION_LIST(inv, spec_invocation_list(spec_found)) {
        if (inv->phrase_invoked == NULL) internal_error("Invocation of null phrase");
        if (inv->phrase_invoked->type_data.say_phrase) {
            int ssp_tok = inv->phrase_invoked->type_data.say_phrase_stream_token_at;
            int ssp_ctok = inv->phrase_invoked->type_data.say_phrase_stream_closing_token_at;
            int ssp_pos = inv->phrase_invoked->type_data.say_phrase_stream_position;
            switch (ssp_pos) {
                case SSP_NONE:
                    break;
                case SSP_START:
                    SSP_invocations[SSP_sp] = inv;
                    SSP_stack[SSP_sp++] = ssp_tok;
                    break;
                case SSP_MIDDLE:
                    if ((SSP_sp > 0) && (SSP_stack[SSP_sp-1] != -1) &&
```

```
            (compare_words(SSP_stack[SSP_sp-1], ssp_tok))) {
            SSP_invocations[SSP_sp-1]->ssp_segment_count++;
            inv->ssp_segment_count =
                SSP_invocations[SSP_sp-1]->ssp_segment_count;
            break;
        }
        if (problem_issued) break;
        LOG("SSP_sp = %d\n", SSP_sp);
        for (i=0; i<SSP_sp; i++) {
            LOG("SSP level %d: $W and $e\n",
                i, SSP_stack[i], SSP_stack[i], SSP_invocations[i]);
        }
        quote_source(1, current_sentence);
        quote_words(2, inv->invocation_w1, inv->invocation_w2);
        handmade_problem(_P_(C12ComplicatedSayStructure));
        issue_problem_segment(
            "In the text at %1, the text substitution '[%2]' ought "
            "to occur as the middle part of its construction, "
            "but it appears to be on its own.");
        add_say_construction_to_error(ssp_tok);
        issue_problem_end();
        problem_issued = TRUE;
        break;
    case SSP_END:
        if ((SSP_sp > 0) && (SSP_stack[SSP_sp-1] != -1) &&
            (compare_words(SSP_stack[SSP_sp-1], ssp_tok))) {
            SSP_invocations[SSP_sp-1]->ssp_segment_count++;
            SSP_invocations[SSP_sp-1]->ssp_closing_segment_wn = ssp_ctok;
            inv->ssp_segment_count =
                SSP_invocations[SSP_sp-1]->ssp_segment_count;
            SSP_sp--;
            break;
        }
        if (problem_issued) break;
        LOG("SSP_sp = %d\n", SSP_sp);
        for (i=0; i<SSP_sp; i++) {
            LOG("SSP level %d: $W and $e\n",
                i, SSP_stack[i], SSP_stack[i], SSP_invocations[i]);
        }
        quote_source(1, current_sentence);
        quote_words(2, inv->invocation_w1, inv->invocation_w2);
        handmade_problem(_P_(C12ComplicatedSayStructure2));
        issue_problem_segment(
            "In the text at %1, the text substitution '[%2]' ought "
            "to occur as the ending part of its construction, "
            "but it appears to be on its own.");
        add_say_construction_to_error(ssp_tok);
        issue_problem_end();
        problem_issued = TRUE;
        break;
}
switch (inv->phrase_invoked->type_data.say_control_structure) {
    case IF_SAY_CS:
```

```
    if (say_if_nesting > 0) {
        if (problem_issued == FALSE)
        sentence_problem(_P_(C12SayIfNested),
            "a second '[if ...]' text substitution occurs inside "
            "an existing one",
            "which makes this text too complicated. While a "
            "single text can contain more than one '[if ...]', "
            "this can only happen if the old if is finished "
            "with an '[end if]' or the new one is written "
            "'[otherwise if]'. If you need more complicated "
            "variety than this allows, the best approach is "
            "to define a new text substitution of your own "
            "('To say fiddly details: ...') and then use it "
            "in this text by including the '[fiddly details]'.");
        problem_issued = TRUE;
    } else {
        say_if_nesting++;
        SSP_invocations[SSP_sp] = NULL;
        SSP_stack[SSP_sp++] = -1;
    }
    break;
case OTHERWISE_SAY_CS:
    if (say_if_nesting == 0) {
        if (problem_issued == FALSE)
        sentence_problem(_P_(C12SayOtherwiseWithoutIf),
            "an '[otherwise]' text substitution occurs where "
            "there appears to be no [if ...]",
            "which doesn't make sense - there is nothing for "
            "it to be otherwise to.");
        problem_issued = TRUE;
    }
    if ((SSP_sp > 0) && (SSP_stack[SSP_sp-1] != -1)) {
        if (problem_issued == FALSE) {
            quote_source(1, current_sentence);
            quote_words(2, inv->invocation_w1, inv->invocation_w2);
            handmade_problem(_P_(C12ComplicatedSayStructure5));
            issue_problem_segment(
                "In the text at %1, the '[%2]' ought "
                "to occur inside an [if ...], but is cut off because "
                "it has been interleaved with a complicated say "
                "construction.");
            add_say_construction_to_error(SSP_stack[SSP_sp-1]);
            issue_problem_end();
        }
        problem_issued = TRUE;
    }
    break;
case END_IF_SAY_CS:
    if (say_if_nesting == 0) {
        if (problem_issued == FALSE)
        sentence_problem(_P_(C12SayEndIfWithoutIf),
            "an '[end if]' text substitution occurs where "
            "there appears to be no [if ...]",
```

```
                        "which doesn't make sense - there is nothing for "
                        "it to end.");
                    problem_issued = TRUE;
                } else {
                    if ((SSP_sp > 0) && (SSP_stack[SSP_sp-1] != -1)) {
                        if (problem_issued == FALSE) {
                            quote_source(1, current_sentence);
                            quote_words(2, inv->invocation_w1, inv->invocation_w2);
                            handmade_problem(_P_(C12ComplicatedSayStructure4));
                            issue_problem_segment(
                                "In the text at %1, the '[%2]' is cut off from "
                                "its [if ...], because it has been interleaved "
                                "with a complicated say construction.");
                            add_say_construction_to_error(SSP_stack[SSP_sp-1]);
                            issue_problem_end();
                        }
                        problem_issued = TRUE;
                    } else {
                        say_if_nesting--;
                        SSP_sp--;
                    }
                }
                break;
        }
        if (say_initialised == FALSE) {
            say_initialised = TRUE;
            WRITE("say__p=1;");
        }
    }
}
if (SSP_sp > 0) {
    if ((SSP_sp == 1) && (SSP_stack[0] == -1)) {
        an if without an end if, which uniquely is legal
    } else {
        LOG("SSP_sp = %d\n", SSP_sp);
        for (i=0; i<SSP_sp; i++) {
            LOG("SSP level %d: $W and $e\n",
                i, SSP_stack[i], SSP_stack[i], SSP_invocations[i]);
        }
        if (problem_issued == FALSE) {
            invocation *stinv = NULL;
            int ssp_tok = -1;
            for (i=0; i<SSP_sp; i++)
                if (SSP_invocations[i]) {
                    stinv = SSP_invocations[i];
                    ssp_tok = SSP_stack[i];
                }
            if (stinv) {
                quote_source(1, current_sentence);
                quote_words(2, stinv->invocation_w1, stinv->invocation_w2);
                handmade_problem(_P_(C12ComplicatedSayStructure3));
                issue_problem_segment(
                    "In the text at %1, the text substitution '[%2]' seems "
```

```
                    "to start a complicated say construction, but it "
                    "doesn't have a matching end.");
                if (ssp_tok >= 0) add_say_construction_to_error(ssp_tok);
                issue_problem_end();
                problem_issued = TRUE;
            }
        }
    }
    it_is_not_worth_adding = it_was_not_worth_adding;
    ⟨Compile invocation blocks for each numbered group in turn 23⟩;
}
```

The function compile_invocational_phrase is called from 7/vasp, 7/cosp and 7/cmsp.

§23.

⟨Compile invocation blocks for each numbered group in turn 23⟩ ≡

```
    group_count = -1;
    INVOCATION_VARIABLE(inv);
    int i = 0;
    LOOP_THROUGH_INVOCATION_LIST(inv, spec_invocation_list(spec_found)) {
        if (inv_get_group(inv) > group_count) {
            group_count = inv_get_group(inv);
            ⟨Compile the invocation block for this new group 24⟩;
        }
        i++;
    }
```

This code is used in §22.

§24.

⟨Compile the invocation block for this new group 24⟩ ≡

```
    int compiling_last_inv_group = FALSE;
    int iref1 = i, iref2 = i;
    invocation *end = NULL;
    INVOCATION_VARIABLE(look);
    LOOK_AHEAD_THROUGH_INVOCATION_LIST(look, inv, spec_invocation_list(spec_found)) {
        if (inv_get_group(look) != group_count) break;
        end = look;
        iref2++;
    }
    if (look == NULL)
        compiling_last_inv_group = TRUE;
    else {
        if ((look->phrase_invoked->type_data.say_phrase) &&
            (look->phrase_invoked->type_data.say_phrase_running_on)) {
            LOOK_AHEAD_THROUGH_INVOCATION_LIST(look, inv, spec_invocation_list(spec_found)) {
                inv_set_flag(look, SUPPRESS_IMPLIED_NEWLINES_INVFLAG);
                if (look == end) break;
            }
        }
    }
```

```
    if ((inv->phrase_invoked->type_data.say_phrase) &&
        (inv->phrase_invoked->type_data.say_control_structure == NO_SAY_CS))
        WRITE("ParaContent(); ");
    compile_invocation_block(OUT, spec_found, iref1, iref2, compiling_last_inv_group);
```

This code is used in §23.

§25.

```
void compile_loop_over_list_S(OUTPUT_STREAM, specification *spec, int v1, int v2) {
    kind_of_value *kov = spec_evaluates_to(spec);
    LOG("compile_loop_over_list_S: $S, $u, $u\n", spec, kov, kovcon_get_base(kov));
    set_scope_of_local_to_current_block(v1);
    set_kov_of_local(v1, kovcon_get_base(kov));
    set_scope_of_local_to_current_block(v2);
    BEGIN_COMPILATION_MODE;
    COMPILATION_MODE_EXIT(DEREFERENCE_POINTERS_CMODE);
    WRITE("for (t_%d=1, t_%d=LIST_OF_TY_GetItem(", v2, v1);
    spec_compile(OUT, spec);
    WRITE(", t_%d, true): t_%d<=LIST_OF_TY_GetLength(", v2, v2);
    spec_compile(OUT, spec);
    WRITE("): t_%d++, t_%d=LIST_OF_TY_GetItem(", v2, v1);
    spec_compile(OUT, spec);
    WRITE(", t_%d, true))", v2);
    END_COMPILATION_MODE;
}
void compile_loop_over_domain_S(OUTPUT_STREAM, specification *spec, int v1, int v2) {
    kind_of_value *kov = spec_get_described_kov(spec);
    specification *localv = new_LOCAL_VARIABLE_spec(-1, -1, v1);
    char val_var[8], ix_var[8];
    LOG("compile_loop_over_domain_S: $S, $u\n", spec, kov);
    set_scope_of_local_to_current_block(v1);
    set_kov_of_local(v1, kov);
    set_scope_of_local_to_current_block(v2);
    sprintf(val_var, "t_%d", v1);
    sprintf(ix_var, "t_%d", v2);
    BEGIN_COMPILATION_MODE;
    COMPILATION_MODE_EXIT(DEREFERENCE_POINTERS_CMODE);
    i6_schema loop_schema;
    if (kov_write_loop_schema(&loop_schema, kov, FALSE))
        sch_expand_textual(&loop_schema, OUT, val_var, ix_var);
    else
        sentence_problem(_P_(C12BadRepeatDomain),
            "this describes a collection of values which can't be repeated through",
            "because the possible range is too large (or has no sensible ordering). "
            "For instance, you can 'repeat with D running through doors' because "
            "there are only a small number of doors and they can be put in order "
            "of creation. But you can't 'repeat with N running through numbers' "
            "because numbers are without end.");
    spec_convert_docket_to_proposition(spec);
    if (spec_get_proposition(spec)) {
        WRITE("if (");
```

```
        compile_test_of_proposition(OUT, localv, spec_get_proposition(spec));
        WRITE(") ");
    }
    END_COMPILATION_MODE;
}
```

Runtime Ambiguities 12/ambig

Purpose

To keep track of runtime ambiguities likely to arise, classifying them into equivalent questions, in order to minimise the amount of run-time checking code which will be needed.

Definitions

¶1. When runtime ambiguities are allowed to persist, special resolution routines must be compiled to sort them out. The following structure caches details of previous similar resolutions, to minimise the quantity of code produced, and a global variable counts distinct "resolver routines" as they are compiled.

```
define LARGEST_CACHED_INVOCATION_GROUP 16                    larger groups are legal, but not cached

typedef struct runtime_ambiguity {
    int resolver_routine_number;                              N such that this caches Resolver_N
    int no_invs;                                              number of invocations in group
    invocation invs[LARGEST_CACHED_INVOCATION_GROUP];         and what they are
    MEMORY_MANAGEMENT
} runtime_ambiguity;

int no_runtime_ambiguities = 0;
```

The structure runtime_ambiguity is private to this section.

---

§**1.** We are given an invocation group for a given SP, and must return (i) an ID number for the resolver routine needed to handle this invocation group, and also (ii) whether the routine has already been compiled or not. It would be valid to simply always return with a new ID number, always requiring a resolver routine to be newly compiled: but it would be inefficient. So we cache all previous decisions on groups of size 16 or fewer (it is vanishingly unlikely that a duplicate will ever occur on two different invocation groups of size 16 or more, so this is no real loss).

We return two pieces of information, which we compile into a single integer: if the routine returns a non-negative number, the routine already exists and this is its ID number; if it returns a negative number X, then the routine does not already exist and should be created with new ID number $-X-1$. In the event of an error, we return the following non-value.

```
define NONDETERMINABLE -1000000

int find_ambiguity_number(specification *spec_found, int first_pos, int last_pos) {
    int group_size = last_pos - first_pos + 1;
    int k, p, cached;
    runtime_ambiguity *rta;
    LOGIF(INVOCATIONS, "Finding ambiguity number for $S group (invs %d to %d)\n",
        spec_found, first_pos, last_pos);
    ⟨FAN - Validate group as determinable 2⟩;
    cached = FALSE;
    if (group_size < LARGEST_CACHED_INVOCATION_GROUP) {
        ⟨FAN - Seek in cache 3⟩;
        cached = TRUE;
        ⟨FAN - Place in cache 4⟩;
    }
    LOGIF(INVOCATIONS, "Allocating new runtime ambiguity number %d (%s)\n",
        no_runtime_ambiguities, (cached)?"cached":"uncached");
```

```
    no_runtime_ambiguities++;
    return -no_runtime_ambiguities;
}
```

The function find_ambiguity_number is called from 12/cinv.

§**2.** We look for certain bad cases here, returning "nondeterminable" if we find them. This enables routines higher up to generate a problem message, since such bad cases occur only due to errors in the source text.

⟨FAN - Validate group as determinable 2⟩ ≡

```
    INVOCATION_VARIABLE(inv1);
    LOOP_THROUGH_PART_OF_INVOCATION_LIST(inv1, spec_invocation_list(spec_found), p, first_pos, group_size)
{
        LOGIF(INVOCATIONS, "Validating %d: $e\n", p, inv1);
        phrase *ph = inv1->phrase_invoked;
        if (ph == NULL) continue;
        if (phtd_safe_for_runtime_resolution(&(ph->type_data)) == FALSE) {
            LOGIF(INVOCATIONS, "Impossible: not determinable at runtime\n");
            return NONDETERMINABLE;
        }
        for (k=0; k<inv_get_number_tokens(inv1); k++) {
            specification *spec1 = inv_get_token_check_to_do(inv1, k);
            if (spec_get_storage_form(spec1) == PROPERTY_VALUE_SPC) {
                LOGIF(INVOCATIONS, "Impossible: not determinable at runtime\n");
                return NONDETERMINABLE;
            }
        }
    }
```

This code is used in §1.

§**3.** For each cached group of the same length, we compare invocations. If we can find a perfect match, we return its resolver routine number, always a non-negative integer.

⟨FAN - Seek in cache 3⟩ ≡

```
    LOOP_OVER(rta, runtime_ambiguity) {
        if (group_size != rta->no_invs) continue;
        INVOCATION_VARIABLE(inv1);
        int j = 0;
        LOOP_THROUGH_PART_OF_INVOCATION_LIST(inv1, spec_invocation_list(spec_found), p, first_pos, group_size)
{
            invocation *inv2 = &(rta->invs[j++]);
            if (inv1->phrase_invoked != inv2->phrase_invoked)
                goto NotThisAmbiguity;
            if (inv_get_phrase_options_bitmap(inv1) != inv_get_phrase_options_bitmap(inv2))
                goto NotThisAmbiguity;
            for (k=0; k<inv_get_number_tokens(inv1); k++) {
                specification *spec1 = inv_get_token_check_to_do(inv1, k);
                specification *spec2 = inv_get_token_check_to_do(inv2, k);
                if ((spec1 == NULL) && (spec2 != NULL)) goto NotThisAmbiguity;
                if ((spec1 != NULL) && (spec2 == NULL)) goto NotThisAmbiguity;
            }
        }
        LOGIF(INVOCATIONS, "Found in cache of runtime ambiguities: %d\n",
```

```
            rta->resolver_routine_number);
        return rta->resolver_routine_number;
        NotThisAmbiguity: ;
    }
```

This code is used in §1.

§**4.** If it hasn't been found, then we will be returning `no_runtime_ambiguities` as the ID number for the resolver routine; we cache this particular invocation group for future use, assigning it to this ID number.

⟨FAN - Place in cache 4⟩ ≡

```
    rta = CREATE(runtime_ambiguity);
    rta->resolver_routine_number = no_runtime_ambiguities;
    rta->no_invs = group_size;
    int j = 0;
    INVOCATION_VARIABLE(inv1);
    LOOP_THROUGH_PART_OF_INVOCATION_LIST(inv1, spec_invocation_list(spec_found), p, first_pos, group_size)
{
        rta->invs[j++] = *inv1;
    }
```

This code is used in §1.

Compile Phrases 12/cph

Purpose

For each phrase whose declaration involves a list of expressions to be evaluated in void context – that is, whose declaration is not an inline piece of I6 – we compile a suitable I6 routine. We also provide some utilities for other sections of NI needing to compile I6 routines not derived directly from phrases.

12/cph.§3 A whole little compiler

Definitions

¶1. We need to keep track of the current phrase being compiled since it provides the local scope for the names of local variables and the like. Note that one phrase definition cannot contain another, so there is never any need to recursively compile phrases.

```
phrase *phrase_being_compiled = NULL;                    phrase whose definition is being compiled
```

```
void code_indent(OUTPUT_STREAM, int indent_level) {
    while (indent_level-- > 0) WRITE("    ");
}
void code_indent_to_block_level(OUTPUT_STREAM) {
    int indent_level = current_block_level() + 2;
    code_indent(OUT, indent_level);
}
void compile_i6_divider_comment(OUTPUT_STREAM) {
    WRITE("! ------------------------------------------------"
                "------------------------------------------------\n");
}
```

The function code_indent is called from 12/phrcd and 12/rb.

The function compile_i6_divider_comment is called from 12/br and 12/rb.

§**1.** We have a problem in that we don't know how many locals a routine we will need until later, but we have to declare them in the routine's header. So we cheat, by compiling the routine code to a temporary file, then writing the header, then copying over the code.

```
STREAM *of_holder;
STREAM temporary_of_struct; STREAM *temporary_of = &temporary_of_struct;
STREAM *begin_compiling_phrase(OUTPUT_STREAM) {
    if (STREAM_OPEN_IN_MEMORY(temporary_of) == FALSE)
        fatal_error("Out of memory: can't allocate for phrase definition");
    of_holder = OUT; OUT = temporary_of;
    return OUT;
}
STREAM *write_routine_header(void) {
    return of_holder;
}
void copy_compiled_phrase(void) {
    ph_stack_frame *phsf = current_stack_frame();
    phsf_compile_i6_local_declarations(phsf, of_holder);
    STREAM_COPY(of_holder, temporary_of);
    STREAM_CLOSE(temporary_of);
}
```

The function begin_compiling_phrase is called from 5/aph, 9/scene, 10/str, 11/los, 12/phsf and 13/gl.
The function write_routine_header is called from 5/aph, 9/scene, 10/str, 11/los, 12/phsf and 13/gl.
The function copy_compiled_phrase is called from 5/aph, 9/scene, 10/str, 11/los, 12/phsf and 13/gl.

§**2.** The following routines have no effect on the functionality of the I6 compiled: they generate either comments helping to show how the I6 matches the original I7 source text, or else debugging print statements to the same effect, so that the testing command RULES ALL will be able to display to the user what is going on at run-time.

```
int statement_count;
void compile_I6_text_for_COMMAND_node(OUTPUT_STREAM, parse_node *p,
    int write_debugging_matter) {
    int j, out_loud = TRUE;
    if ((write_debugging_matter == FALSE) || [[p == end/else/otherwise ***]] ||
        (pn_int_annotation(p, suppress_debug_text_ANNOT)))
        out_loud = FALSE;
    out_loud = FALSE;
    if (pn_get_node_type(p) != COMMAND_NT) internal_error("not a COMMAND node");
    code_indent_to_block_level(OUT);
    if (out_loud == FALSE) WRITE("! ");
    if (out_loud) {
        WRITE("  #ifndef MEMORY_ECONOMY; ");
        WRITE("if (debug_rules>1) print \"  ");
    }
    WRITE("[%d: ", statement_count++);
    for (j=p->word_ref1; j<=p->word_ref2; j++) {
        isn_compile_string(OUT, lw_array[j].lw_text, ISN_FLATTEN_NEWLINES);
        if (j<p->word_ref2) WRITE(" ");
    }
    if (out_loud)
        WRITE("]^\"; #endif;\n");
    else
        WRITE("]\n");
    if ((out_loud) && (p->next) && [[p->next == otherwise/else ***]])
        compile_I6_text_for_COMMAND_node(OUT, p->next, TRUE);
}
void compile_more_I6_text_for_COMMAND_node(OUTPUT_STREAM, parse_node *p,
    int write_debugging_matter) {
    if (write_debugging_matter) {
        if [[p == repeat ***]] {
            int lnum;
            code_indent_to_block_level(OUT);
            if (memory_economy_in_force == FALSE) {
                WRITE("  if (debug_rules>1) print \"  ");
                lnum = local_whose_scope_is_block(current_block_level()-1);
                if (lnum >= 0) describe_repetition_local(OUT, lnum);
                else WRITE("[repetition through row \", ct_1, \"]^\";\n");
            }
        }
        if [[p == end repeat]] {
            code_indent_to_block_level(OUT);
```

```
            if (memory_economy_in_force == FALSE) {
                WRITE("  if (debug_rules>1) print \"  ");
                WRITE("[%d: end repeat]^\";\n", statement_count-1);
            }
        }
    }
}
void compile_phrase(OUTPUT_STREAM, phrase *ph, stacked_variable_owner_list *legible) {
    parse_node *p;
    heading *definition_area;
    extension_file *definition_extension;
    int write_debugging_matter = TRUE;
    if ((ph->declaration_node == NULL) ||
        (pn_get_node_type(ph->declaration_node) != ROUTINE_NT) ||
        (ph->declaration_node->word_ref1 < 0))
        internal_error("tried to compile phrase with bad ROUTINE node");
    definition_area = heading_of(lw_array[ph->declaration_node->word_ref1].lw_source);
    definition_extension = hd_get_extension_containing(definition_area);
    phrase_being_compiled = ph;
    phsf_set_stvol(&(ph->stack_frame), legible);
    if (definition_extension) {
        write_debugging_matter = FALSE;
        ef_write_I6_comment_describing(definition_extension, OUT);
    }
    phud_write_I6_comment_describing(&(ph->usage_data), OUT);
    OUT = begin_compiling_phrase(OUT);
    begin_code_blocks();
    LOGIF(PHRASE_COMPILATION, "Compiling phrase:\n$T", ph->declaration_node);
    current_sentence = ph->declaration_node;
    phrcd_compile_test_head(&(ph->runtime_context_data), OUT,
        ph_get_I6_representation(ph), ph);
    statement_count = 1;
    for (p = ph->declaration_node->down; p; p = p->next) {
        current_sentence = p;
        code_indent_to_block_level(OUT);
        WRITE("! phrase %d\n", statement_count);
        compile_I6_text_for_COMMAND_node(OUT, p, write_debugging_matter);
        code_indent_to_block_level(OUT);
        compile_To_phrase(OUT, p->word_ref1, p->word_ref2);
        WRITE("\n");
        compile_more_I6_text_for_COMMAND_node(OUT, p, write_debugging_matter);
        STREAM_FLUSH(dl);                                    In case of any accident
        finish_this_session_of_parsing();
    }
    end_code_blocks();
    phrcd_compile_test_tail(&(ph->runtime_context_data), OUT,
        ph_get_I6_representation(ph), ph);
    OUT = write_routine_header();
    STREAM_WRITE(of_holder, "[ %s ", ph_get_I6_representation(ph));
```

```
    copy_compiled_phrase();
    code_indent(OUT, 1);
    phtd_compile_default_return(OUT, &(phrase_being_compiled->type_data),
        &(ph->runtime_context_data));
    WRITE("\n];\n");
    phrase_being_compiled = NULL;
    finish_this_session_of_parsing();
}
```

The function compile_phrase is called from 12/ph.

§**3. A whole little compiler.** This routine sits at the summit of a mountain of code: it takes a use of a phrase through every stage from raw text to final compiled code.

```
specification *match_To_phrase = NULL;
void compile_To_phrase(OUTPUT_STREAM, int w1, int w2) {
    if (match_To_phrase == NULL) match_To_phrase = new_COMMAND_spec();
    int initial_problem_count = problem_count;
    LOGIF(EXPRESSIONS, "\n-- -- Evaluating <$W> -- --\n(a) Parsing:\n", w1, w2);
    specification *spec_found = parse_expression(w1, w2, VOID_EXPCON);
    if (initial_problem_count == problem_count) {
        LOGIF(EXPRESSIONS, "(b) Type checking as $S:\n$X", match_To_phrase, spec_found);
        typecheck(spec_found, match_To_phrase);
    }
    if (initial_problem_count == problem_count) {
        LOGIF(EXPRESSIONS, "(c) Compilation:\n");
        if (spec_has_invocation_list(spec_found))
            LOGIF(INVOCATIONS, "Invocation list to be compiled:\n$E\n",
                spec_invocation_list(spec_found));
        spec_compile(OUT, spec_found);
    }
    if (initial_problem_count == problem_count) {
        LOGIF(EXPRESSIONS, "-- -- Completed -- --\n");
    } else {
        LOGIF(EXPRESSIONS, "-- -- Failed -- --\n");
    }
}
```

The function compile_To_phrase is called from 10/str.

To Phrases 12/toph

Purpose

The first of four ways phrases are invoked: in the definitions of other phrases, and in I7 expressions and conditions. Here a phrase is used much as a function would be used in a C-like language.

12/toph.§3-5 Logical priority of To phrases; §6 Registering and compiling To phrases; §7 Phrase option parsing

---

§**1.** Some "To..." phrases have a body consisting of a list of invocations of other "to..." phrases, but others have inline definitions in terms of I6 code. The following routine accesses this definition:

```
void ph_To_inline_write_definition(phrase *ph, char *to) {
    if (ph->inline_wn < 0)
        internal_error("tried to access inline definition of non-inline phrase");
    strcpy(to, lw_array[ph->inline_wn].lw_text);
}
```

The function ph_To_inline_write_definition is called from 12/cinv.

§**2.** Inline phrase outcomes are one of the oldest type-checking features in NI, but they fell into abeyance due to a bug and also due to confusion about the right way to classify To phrases, so that between about 3Kxx and 4Sxx they did nothing. They now work, though. The idea is that certain inline definitions can mark themselves to be included only in To phrases of the right sort: it makes no sense to respond "yes" to a phrase "To decide what number is...", for instance. The following routine looks at the tail of the inline definition to parse the appropriate phrase type.

This syntax is intentionally undocumented, and is reserved for use by the Standard Rules only: hence the internal error rather than problem message for cases where it's misphrased.

```
int ph_To_inline_read_outcome_mor(phrase *ph) {
    parse_node *p = ph->declaration_node->down;
    int w1, w2;
    if (p == NULL)
        internal_error("tried to access inline definition of phrase with no body");
    if (ph->inline_wn < 0)
        internal_error("tried to access inline definition of non-inline phrase");
    [[w1, w2 <-- p]];
    LOGIF(INVOCATIONS, "form of inline: $W\n", w1, w2);
    if (w2 == w1+1) return UNKNOWN;
    w1 += 2;
    if [[w1, w2 == DASH in to only]] return DECIDES_NOTHING_MOR;
    if [[w1, w2 == DASH in to decide if only]] return DECIDES_CONDITION_MOR;
    if [[w1, w2 == DASH in to decide only]] return DECIDES_VALUE_MOR;
    LOG("Tail: $W\n", w1, w2);
    internal_error("unknown inline definition tail");
    return UNKNOWN;
}
```

The function ph_To_inline_read_outcome_mor is called from 12/cinv.

§**3. Logical priority of To phrases.** "To" phrases are insertion-sorted, as they are defined, into a linked list held in logical priority order. This essentially means that two lexically indistinguishable phrases (e.g., "admire (OC - an open container)" and "admire (C - a container)") are placed such that the more specific, in type-checking terms, comes first (the open container case being the more specific). The purpose of this list is to ensure that excerpt meanings for phrase definitions are registered in logical priority order, because the excerpt parser prefers earlier registrations to later ones in case of ambiguity.

Note that the following sort algorithm affects only "to..." phrases, and therefore has no effect on rule ordering within rulebooks.

§**4.** The system for deciding which of two phrases is logically prior, if either is. This is not quite compatible with the other comparison routines in NI (for comparing action patterns, SPs, etc.) because it returns a wider variety of values:

```
define BEFORE_PH -3
define SUBSCHEMA_PH -1
define EQUAL_PH 0
define SUPERSCHEMA_PH 1
define INCOMPARABLE_PH 2
define AFTER_PH 3

int ph_To_compare(phrase *ph1, phrase *ph2) {
    int r = phtd_comparison(&(ph1->type_data), &(ph2->type_data));
    if (dl_this(PHRASE_COMPARISONS_DA)) {
        char tell_debug[1024];
        ph_write_HTML_representation(ph1, tell_debug);
        LOG("Phrase comparison <%s>\n  ", tell_debug);
        switch(r) {
            case INCOMPARABLE_PH: LOG("~~"); break;
            case SUBSCHEMA_PH: LOG("<="); break;
            case SUPERSCHEMA_PH: LOG(">="); break;
            case EQUAL_PH: LOG("=="); break;
            case BEFORE_PH: LOG("<"); break;
            case AFTER_PH: LOG(">"); break;
        }
        ph_write_HTML_representation(ph2, tell_debug);
        LOG(" <%s>\n", tell_debug);
    }
    return r;
}
```

§**5.** The following routine takes a phrase and makes it officially a To phrase, which in particular means adding it to the list of To phrases in logical order.

```
struct phrase *first_in_logical_order = NULL;
void ph_To_make(phrase *ph) {
    phrase *previous_phrase = NULL;
    phrase *current_phrase = first_in_logical_order;
    sprintf(ph_get_I6_representation(ph), "PHR_%d", ph->allocation_id);
    if (first_in_logical_order == NULL) { first_in_logical_order = ph; return; }
    while ((current_phrase != NULL) &&
            (ph_To_compare(ph, current_phrase) >= 0)) {
        previous_phrase = current_phrase;
        current_phrase = current_phrase->next_in_logical_order;
    }
    if (previous_phrase == NULL) {
        ph->next_in_logical_order = first_in_logical_order;
        first_in_logical_order = ph;
        return;
    }
    previous_phrase->next_in_logical_order = ph;
    ph->next_in_logical_order = current_phrase;
}
```

The function ph_To_make is called from 12/ph.

§**6. Registering and compiling To phrases.** These are the only places where the logical precedence list is directly used, but registration of the excerpts in precedence order ensures that this ordering has a profound effect on expression parsing throughout NI.

Compilation in precedence order is by contrast done for purely cosmetic reasons, that is, to make the I6 source code more legible.

```
void ph_To_register_all(void) {
    phrase *ph;
    int c = 0;
    for (ph = first_in_logical_order; ph; ph = ph->next_in_logical_order) {
        current_sentence = ph->declaration_node;
        phtd_register_excerpt(&(ph->type_data), ph);
        ph->sequence_count = c++;
    }
}
int ph_sequence_count(phrase *ph) {
    if (ph == NULL) internal_error("Sought sequence count for null phrase");
    if (ph->sequence_count == -1) {
        log_phrase(ph);
        internal_error("Sequence count not ready");
    }
    return ph->sequence_count;
}
void ph_To_compile_all(OUTPUT_STREAM, int *i, int max_i) {
    phrase *ph;
    if (first_in_logical_order) {
        WRITE("\n\n! Definitions of non-inline \"To...\" phrases\n\n");
```

```
        for (ph = first_in_logical_order; ph; ph = ph->next_in_logical_order)
            ph_compile(ph, OUT, i, max_i, NULL);
    }
}
```

The function ph_To_register_all is called from 12/cs.
The function ph_sequence_count is called from 7/inv.
The function ph_To_compile_all is called from 12/cs.

§**7. Phrase option parsing.** These indirections are provided so that the implementation of phrase options is confined to the current Chapter.

```
int ph_To_allows_options(phrase *ph) {
    return phod_allows_options(&(ph->options_data));
}
int ph_To_parse_phrase_option_used(phrase *ph, int w1, int w2) {
    return phod_parse(&(ph->options_data), w1, w2);
}
int ph_To_check_supplied_options(phrase *ph,
    int silently, invocation *inv, int w1, int w2) {
    return phod_check_supplied_options(&(ph->options_data), ph, silently,
        inv, w1, w2);
}
```

The function ph_To_allows_options is called from 5/parse.
The function ph_To_parse_phrase_option_used is called from 5/candp and 12/cinv.
The function ph_To_check_supplied_options is called from 7/tc.

Phrasebook Index 12/phin

Purpose

To compile the HTML page for the Phrasebook index.

Template interpreter commands

```
0   {-callv:index_page_Phrasebook}
```

```
int say_extra_lines = TRUE;
void index_single_phrase(phrase *phm) {
    if (say_extra_lines) {
        say_extra_lines = FALSE;
        INDEX("say (<i>some text with substitutions</i>)");
        index_doc_link("ph_say");
        INDEX("<br>say \"[<i>a value of some sort</i>]\"");
        INDEX("<br>");
    }
    phtd_write_index_representation(&(phm->type_data), phm);
    index_link(lw_array[phm->declaration_node->word_ref1].lw_source);
    if (phm->ph_documentation_symbol_wn >= 0) {
        index_doc_link(lw_array[phm->ph_documentation_symbol_wn].lw_rawtext);
    }
    INDEX("<br>\n");
}
void index_page_Phrasebook(void) {
    phrase *phm;
    int pass, count, say_disclaimer = FALSE, just_made_heading = FALSE, subx1, subx2;
    heading *definition_area, *current_area;
    extension_file *ext = NULL;
    for (pass=1; pass<=2; pass++) {
        int no_on_line = 0;
        current_area = NULL; subx1 = -1; subx2 = -1;
        int slice;
        for (slice=0; slice<NUMBER_CREATED(extension_file)+1; slice++) {
        LOOP_OVER(phm, phrase) {
            int index_this = FALSE;
            if (phud_phrase_index_area(&(phm->usage_data)) == UNKNOWN)
                index_this = TRUE;
            if (phtd_is_a_let_V_phrase(&(phm->type_data)))
                index_this = FALSE;
            if (phtd_is_a_say_X_phrase(&(phm->type_data))) {
                index_this = FALSE;
                if ((pass == 2) && (say_disclaimer == FALSE)) {
                    say_disclaimer = TRUE;
                    say_extra_lines = TRUE;
                }
            }
```

```
if (index_this == FALSE) continue;
definition_area =
    heading_of(lw_array[phm->declaration_node->word_ref1].lw_source);
if (hd_indexed(definition_area) == FALSE) continue;
extension_file *this_extension =
    hd_get_extension_containing(definition_area);
if (slice == NUMBER_CREATED(extension_file)) {
    if (this_extension != NULL) continue;
} else {
    if ((this_extension == NULL) || (this_extension->allocation_id != slice)) continue;
}
if (pass >= 1) {
    if (ext != this_extension) {
        ext = this_extension;
        if (ext == NULL) {
            INDEX("<p>");
            if (pass == 2) INDEX("<hr>");
            INDEX("<b>Defined in the source</b>");
            INDEX("<br>");
            just_made_heading = TRUE; no_on_line = 0;
        } else
        if (ext != standard_rules_extension) {
            INDEX("<p>");
            if (pass == 2) INDEX("<hr>");
            INDEX("<b>From the extension ");
            ef_write_name_to_file(ext, ifl);
            INDEX(" by ");
            ef_write_author_to_file(ext, ifl);
            INDEX("</b><br>");
            just_made_heading = TRUE; no_on_line = 0;
        }
    }
}
if (definition_area != current_area) {
    int x1, x2, j;
    if (pass == 2) INDEX("<p>");
    if (pass == 2) index_anchor_numbered(phm->allocation_id);
    if (pass == 2) INDEX("<b>");
    hd_get_text_of_heading(definition_area, &x1, &x2);
    if (x1 >= 0) {
        if [[x1, x2 == ... HYPHEN ... : j]] {
            x1 = j+1;
            if [[x1, x2 == ... HYPHEN ... : j]] {
                if ((subx1 < 0) ||
                    (compare_word_range(x1, j-1, subx1, subx2) == FALSE)) {
                    if (pass == 2) INDEX("<p><hr>");
                    INDEX("<p><b>");
                    print_raw_text_to_file(x1, j-1, ifl);
                    INDEX("</b><br>");
                    if (pass == 1) no_on_line = 0;
                }
                subx1 = x1; subx2 = j-1;
                x1 = j+1;
```

```
                }
            }
            if ((pass == 1) && (no_on_line > 0)) INDEX(", ");
            no_on_line++;
            print_raw_text_to_file(x1, x2, ifl);
        } else {
            /* if ((just_made_heading == FALSE) &&
                (hd_in_same_file(definition_area, current_area) == FALSE))
                INDEX("<p>"); */
            if ((pass == 1) && (no_on_line > 0)) INDEX(", ");
            no_on_line++;
            INDEX("Miscellaneous");
        }
        just_made_heading = FALSE;
        if (pass == 2) INDEX("</b>");
        if (pass == 1) index_below_link_numbered(phm->allocation_id);
        if (pass == 2) INDEX("<br>");
    }
    current_area = definition_area;
    if (pass == 1) continue;
    index_single_phrase(phm);
    phod_index(&(phm->options_data));
}
}
count = 0;
LOOP_OVER(phm, phrase) {
    int d = phud_phrase_index_area(&(phm->usage_data));
    if ((d == UNKNOWN) || (d == -1)) continue;
    count++;
}
if (count >= 1) {
    int disclaimed = FALSE, k;
    if (pass == 1) INDEX("<p>");
    if (pass == 2) INDEX("<p><hr><p>");
    if (pass == 2) index_anchor("NEWKINDS");
    INDEX("<b>Brought into being by new kinds of value</b>");
    if (pass == 1) index_below_link("NEWKINDS");
    INDEX("<br>");
    if (pass == 2)
    LOOP_OVER(phm, phrase) {
        int d = phud_phrase_index_area(&(phm->usage_data));
        if ((d == UNKNOWN) || (d == -1)) continue;
        if (d != LOWEST_DESIGNED_TY) {
            if (disclaimed) continue;
            disclaimed = TRUE;
            INDEX("<i>together with similar constructions for ");
            LOOP_OVER_DESIGNED_TYPE_IDS(k) {
                if (k == LOWEST_DESIGNED_TY) continue;
                index_lowercase_dtid(k);
                if (k < HIGHEST_DESIGNED_TY-1) INDEX(", ");
                if (k == HIGHEST_DESIGNED_TY-1) INDEX(" and ");
            }
            INDEX("</i><br>");
```

```
                    continue;
                }
                index_single_phrase(phm);
                phod_index(&(phm->options_data));
            }
        }
        if (pass == 1) {
            INDEX("<p><hr><p>");
            INDEX("<small>Not all of the phrases listed below have blue help ");
            INDEX("icons, as many occur in clumps of similar phrases which ");
            INDEX("share the same documentation. Try the nearest icon ");
            INDEX("above.</small><p>");
        }
    }
    INDEX("<p><hr><p>");
    index_lexicon();
}
```

The function index_page_Phrasebook is invoked by a command in a `.i6t` template file.

Adjectival Definitions 12/def

Purpose

The second of four ways phrases are invoked: as definitions of adjectives which can be used as unary predicates in the calculus. (And we also look after adjectives arising from I6 or I7 conditions, and from I6 routines.)

Template interpreter commands

```
0   {-callv:traverse_for_adjective_definitions}
```

Definitions

¶1. A typical example would be:

Definition: A container is significant if it contains a clue.

Here the domain of the definition is "container", and the meaning assigned to "significant" is an I7 condition; but we can also make it an I6 condition, or (a tidier way to express the same thing) delegate it to an I6 routine. Phrases enter only when we define an adjective with an explicit, though nameless, I7 rule:

Definition: A container (called the sac) is significant: if the sac contains a clue, decide yes; ...

That makes four distinct kinds of adjective, but all share the following structure to hold details of their specific meanings:

```
typedef struct definition {
    struct parse_node *definition_node;          current sentence: where the word "Definition" is
    struct parse_node *node;                                  where the actual definition is
    int format;                          +1 to go by condition, -1 to negate it, 0 to use routine
    int condition_w1, condition_w2;                text of condition to match, if +1 or -1
    int domain_calling_w1, domain_calling_w2;        what if anything the term is called
    struct adjective_meaning *am_of_def;                          which adjective meaning
    MEMORY_MANAGEMENT
} definition;
```

The structure definition is private to this section.

```
void traverse_for_adjective_definitions(void) {
    parse_node *p;
    for (TREE_START(p); p; TREE_NEXT(p))
        if (pn_get_node_type(p) == ROUTINE_NT) {
            if [[p == definition]] {
                int a1, a2, cond_w1 = -1, cond_w2 = -1, i, is_word, the_format;
                int adj_name_w1, adj_name_w2, dom_name_w1, dom_name_w2, called_w1, called_w2;
                int neg_name_w1, neg_name_w2;
                parse_node *q = p->down;
                if (q == NULL) {
                    if ((p->next == NULL) || (pn_get_node_type(p->next) != ROUTINE_NT)) {
                        sentence_problem(_P_(BelievedImpossible),
                            "don't leave me in suspense",
                            "write a definition after 'Definition:'!");
                        continue;
                    }
                    q = p->next; p->next = q->next; p->down = q->down; q->next = NULL;
```

```
                }
                the_format = 0;
                if [[q == ... if ... : i --> a1, a2 ... cond_w1, cond_w2]] the_format = 1;
                else if [[q == ... unless ... : i --> a1, a2 ... cond_w1, cond_w2]] the_format = -1;
                else if (q->down) {
                    a1 = q->word_ref1; a2 = q->word_ref2; cond_w1 = -1; cond_w2 = -1;
                } else {
                    GenericDefnError:
                    LOG("Definition tree:\n$T\n", p);
                    definition_problem(_P_(C12DefinitionWithoutCondition),
                        q, "a definition must take the form "
                        "'Definition: a ... is ... if/unless ...' or "
                        "else 'Definition: a ... is ...: ...'",
                        "but I can't make this fit either shape.");
                    continue;
                }
                is_word = is_word_intermediate(is_V, a1, a2);
                if (is_word<0) is_word = is_word_intermediate(are_V, a1, a2);
                if (is_word<0) goto GenericDefnError;
                adj_name_w1 = is_word+1; adj_name_w2 = a2;
                dom_name_w1 = a1; dom_name_w2 = is_word-1;
                called_w1 = -1; called_w2 = -1;
                word_range_ends_with_calling(dom_name_w1, dom_name_w2,
                    &dom_name_w1, &dom_name_w2, &called_w1, &called_w2);
                neg_name_w1 = -1; neg_name_w2 = -1;
                if ([[adj_name_w1, adj_name_w2 == ... rather ... : i ]] &&
                    [[i, adj_name_w2 == rather than ... --> neg_name_w1, neg_name_w2]])
                    adj_name_w2 = i-1;
                adjective_meaning *am = am_parse(q, the_format, adj_name_w1, adj_name_w2,
                    dom_name_w1, dom_name_w2, cond_w1, cond_w2, called_w1, called_w2);
                if (am == NULL) internal_error("unclaimed adjective definition");
                if (neg_name_w1 >= 0) {
                    adjective_meaning *neg = am_negate(am);
                    am_declare(neg, neg_name_w1, neg_name_w2);
                }
                if (the_format != 0)  p->down = NULL;
            }
        }
}
```

The function traverse_for_adjective_definitions is invoked by a command in a `.i6t` template file.

§**1.** Condition adjectives.

```
definition *def_new(parse_node *q) {
    definition *def = CREATE(definition);
    def->node = q;
    def->format = 0;
    def->condition_w1 = -1; def->condition_w2 = -1;
    def->domain_calling_w1 = -1; def->domain_calling_w2 = -1;
    def->definition_node = current_sentence;
    return def;
}
```

```
adjective_meaning *CONDITION_KADJ_parse(parse_node *q,
    int sense,
    int adj_name_w1, int adj_name_w2,
    int dom_name_w1, int dom_name_w2,
    int cond_w1, int cond_w2,
    int called_w1, int called_w2) {
    if (sense == 0) return NULL;
    definition *def = def_new(q);
    adjective_meaning *am = am_new(CONDITION_KADJ, STORE_POINTER_definition(def),
        q->word_ref1, q->word_ref2);
    def->condition_w1 = cond_w1; def->condition_w2 = cond_w2;
    def->format = sense;
    def->domain_calling_w1 = called_w1; def->domain_calling_w2 = called_w2;
    def->am_of_def = am;
    am_declare(am, adj_name_w1, adj_name_w2);
    am_pass_task_to_support_routine(am, TEST_ADJECTIVE_TASK);
    am_set_domain_from_word_range(am, dom_name_w1, dom_name_w2);
    return am;
}
void CONDITION_KADJ_compiling_soon(adjective_meaning *am, definition *def, int T) {
}
int CONDITION_KADJ_compile(definition *def, int T, OUTPUT_STREAM, ph_stack_frame *phsf) {
    switch (T) {
        case TEST_ADJECTIVE_TASK:
            if (OUT) {
                if (def->domain_calling_w1 >= 0) {
                    LOGIF(QUANTITY_CREATIONS, "Renaming 'it' as $W\n",
                        def->domain_calling_w1, def->domain_calling_w2);
                    phsf_rename_it_variable(phsf,
                        def->domain_calling_w1, def->domain_calling_w2);
                }
                if (def->condition_w1 >= 0) {
                    specification *spec =
                        parse_expression(def->condition_w1, def->condition_w2, CONDITION_EXPCON);
                    if (validate_when(spec) == FALSE) {
                        LOG("Error on: $X", spec);
                        definition_problem(_P_(C12DefinitionBadCondition),
                            def->node,
                            "that condition makes no sense to me",
                            "although the preamble to the definition was properly "
                            "written. There must be something wrong after 'if'.");
                    } else {
                        WRITE("(");
                        if (def->format == -1) WRITE("~~(");
                        spec_compile(OUT, spec);
                        if (def->format == -1) WRITE(")");
                        WRITE(")");
                    }
                }
                if (def->domain_calling_w1 >= 0) phsf_restore_it_variable(phsf);
            }
            return TRUE;
```

```
        case NOW_ADJECTIVE_TRUE_TASK: return FALSE;
        case NOW_ADJECTIVE_FALSE_TASK: return FALSE;
    }
    return FALSE;
}
int CONDITION_KADJ_assert(definition *def,
    world_object *wo_to_assert_on, specification *val_to_assert_on, int parity) {
    return FALSE;
}
int CONDITION_KADJ_index(definition *def) {
    return FALSE;
}
```

The function CONDITION_KADJ_parse is called from 5/aph.
The function CONDITION_KADJ_compiling_soon is called from 5/aph.
The function CONDITION_KADJ_compile is called from 5/aph.
The function CONDITION_KADJ_assert is called from 5/aph.
The function CONDITION_KADJ_index is called from 5/aph.

§2. Phrase adjectives.

```
void join_phrase_to_adjective(parse_node *p, phrase *ph, int *cw1, int *cw2) {
    definition *def;
    *cw1 = -1; *cw2 = -1;
    if (ph == NULL) return;
    LOOP_OVER(def, definition)
        if ((def->definition_node == p) && (am_get_form(def->am_of_def) == PHRASE_KADJ)) {
            i6_schema *sch = am_set_i6_schema(def->am_of_def, TEST_ADJECTIVE_TASK, FALSE);
            sch_write_to_existing_1(sch, "(%s(*1))", ph_get_I6_representation(ph));
            *cw1 = def->domain_calling_w1; *cw2 = def->domain_calling_w2;
        }
}
adjective_meaning *PHRASE_KADJ_parse(parse_node *q,
    int sense,
    int adj_name_w1, int adj_name_w2,
    int dom_name_w1, int dom_name_w2,
    int cond_w1, int cond_w2,
    int called_w1, int called_w2) {
    if (sense != 0) return NULL;
    definition *def = def_new(q);
    adjective_meaning *am = am_new(PHRASE_KADJ, STORE_POINTER_definition(def),
        q->word_ref1, q->word_ref2);
    def->domain_calling_w1 = called_w1; def->domain_calling_w2 = called_w2;
    def->am_of_def = am;
    am_declare(am, adj_name_w1, adj_name_w2);
    am_pass_task_to_support_routine(am, TEST_ADJECTIVE_TASK);
    am_set_domain_from_word_range(am, dom_name_w1, dom_name_w2);
    return am;
}
void PHRASE_KADJ_compiling_soon(adjective_meaning *am, definition *def, int T) {
}
int PHRASE_KADJ_compile(definition *def, int T, OUTPUT_STREAM, ph_stack_frame *phsf) {
```

```
    return FALSE;
}
int PHRASE_KADJ_assert(definition *def,
    world_object *wo_to_assert_on, specification *val_to_assert_on, int parity) {
    return FALSE;
}
int PHRASE_KADJ_index(definition *def) {
    return FALSE;
}
```

The function join_phrase_to_adjective is called from 12/ph.
The function PHRASE_KADJ_parse is called from 5/aph.
The function PHRASE_KADJ_compiling_soon is called from 5/aph.
The function PHRASE_KADJ_compile is called from 5/aph.
The function PHRASE_KADJ_assert is called from 5/aph.
The function PHRASE_KADJ_index is called from 5/aph.

§3. I6-defined routine adjectives.

```
adjective_meaning *I6_ROUTINE_KADJ_parse(parse_node *q,
    int sense,
    int adj_name_w1, int adj_name_w2,
    int dom_name_w1, int dom_name_w2,
    int cond_w1, int cond_w2,
    int called_w1, int called_w2) {
    if (sense != 1) return NULL;
    if (called_w1 >= 0) return NULL;
    if (![[cond_w1, cond_w2 == i6 routine ### says so OPENBRACKET ... CLOSEBRACKET]]) return NULL;
    if (vocab_test_flags(cond_w1+2, TEXT_MC) == 0) return NULL;
    dequote_word(cond_w1+2);
    definition *def = def_new(q);
    adjective_meaning *am = am_new(I6_ROUTINE_KADJ, STORE_POINTER_definition(def),
        cond_w1+7, cond_w2-1);
    def->am_of_def = am;
    am_declare(am, adj_name_w1, adj_name_w2);
    am_pass_task_to_support_routine(am, TEST_ADJECTIVE_TASK);
    am_set_domain_from_word_range(am, dom_name_w1, dom_name_w2);
    i6_schema *sch = am_set_i6_schema(am, TEST_ADJECTIVE_TASK, TRUE);
    sch_write_to_existing_1(sch, "(%s(*1))", lw_array[cond_w1+2].lw_text);
    return am;
}
void I6_ROUTINE_KADJ_compiling_soon(adjective_meaning *am, definition *def, int T) {
}
int I6_ROUTINE_KADJ_compile(definition *def, int T, OUTPUT_STREAM, ph_stack_frame *phsf) {
    return FALSE;
}
int I6_ROUTINE_KADJ_assert(definition *def,
    world_object *wo_to_assert_on, specification *val_to_assert_on, int parity) {
    return FALSE;
}
int I6_ROUTINE_KADJ_index(definition *def) {
    return FALSE;
}
```

The function I6_ROUTINE_KADJ_parse is called from 5/aph.
The function I6_ROUTINE_KADJ_compiling_soon is called from 5/aph.
The function I6_ROUTINE_KADJ_compile is called from 5/aph.
The function I6_ROUTINE_KADJ_assert is called from 5/aph.
The function I6_ROUTINE_KADJ_index is called from 5/aph.

§4. I6-defined condition adjectives.

```
adjective_meaning *I6_CONDITION_KADJ_parse(parse_node *q,
    int sense,
    int adj_name_w1, int adj_name_w2,
    int dom_name_w1, int dom_name_w2,
    int cond_w1, int cond_w2,
    int called_w1, int called_w2) {
    if (sense != 1) return NULL;
    if (called_w1 >= 0) return NULL;
    if (![[cond_w1, cond_w2 == i6 condition ### says so OPENBRACKET ... CLOSEBRACKET]]) return NULL;
    if (vocab_test_flags(cond_w1+2, TEXT_MC) == 0) return NULL;
    dequote_word(cond_w1+2);
    definition *def = def_new(q);
    adjective_meaning *am = am_new(I6_CONDITION_KADJ, STORE_POINTER_definition(def),
        cond_w1+7, cond_w2-1);
    def->am_of_def = am;
    am_declare(am, adj_name_w1, adj_name_w2);
    am_pass_task_to_support_routine(am, TEST_ADJECTIVE_TASK);
    am_set_domain_from_word_range(am, dom_name_w1, dom_name_w2);
    i6_schema *sch = am_set_i6_schema(am, TEST_ADJECTIVE_TASK, FALSE);
    sch_write_to_existing_1(sch, "(%s)", lw_array[cond_w1+2].lw_text);
    return am;
}
void I6_CONDITION_KADJ_compiling_soon(adjective_meaning *am, definition *def, int T) {
}
int I6_CONDITION_KADJ_compile(definition *def, int T, OUTPUT_STREAM, ph_stack_frame *phsf) {
    return FALSE;
}
int I6_CONDITION_KADJ_assert(definition *def,
    world_object *wo_to_assert_on, specification *val_to_assert_on, int parity) {
    return FALSE;
}
int I6_CONDITION_KADJ_index(definition *def) {
    return FALSE;
}
```

The function I6_CONDITION_KADJ_parse is called from 5/aph.
The function I6_CONDITION_KADJ_compiling_soon is called from 5/aph.
The function I6_CONDITION_KADJ_compile is called from 5/aph.
The function I6_CONDITION_KADJ_assert is called from 5/aph.
The function I6_CONDITION_KADJ_index is called from 5/aph.

Timed Phrases 12/tiph

Purpose

The third of four ways phrases are invoked: as timed events, which need no special NI data structure and are simply compiled into a pair of timetable I6 arrays to be processed at run-time.

Definitions

¶1. The timing of an event records the time at which a phrase should spontaneously happen. This is ordinarily a time value, in minutes from 12 midnight, for a phrase happening at a specific time – for instance, one defined as "At 9:00 AM: ..." But two values are special:

define NOT_A_TIMED_EVENT -1 *as for the vast majority of phrases*
define NO_FIXED_TIME -2 *for phrases like "When the clock strikes: ..."*

---

§**1.** Timed events are stored in two simple arrays, processed by run-time code.

```
void tiph_compile_TimedEventsTable_array(OUTPUT_STREAM) {
    phrase *ph;
    int i, t, when_count = 0;
    WRITE("Array TimedEventsTable table");
    LOOP_OVER(ph, phrase) {
        t = phud_get_timing_of_event(&(ph->usage_data));
        if (t == NOT_A_TIMED_EVENT) continue;
        if (t == NO_FIXED_TIME) when_count++;
        else WRITE(" %s", ph_get_I6_representation(ph));
    }
    for (i=0; i<when_count+1; i++) WRITE(" 0 0");
    WRITE(";\n");
}
void tiph_compile_TimedEventTimesTable_array(OUTPUT_STREAM) {
    phrase *ph;
    int i, t, when_count = 0;
    WRITE("Array TimedEventTimesTable table");
    LOOP_OVER(ph, phrase) {
        t = phud_get_timing_of_event(&(ph->usage_data));
        if (t == NOT_A_TIMED_EVENT) continue;
        if (t == NO_FIXED_TIME) when_count++;
        else WRITE(" %d", t);
    }
    for (i=0; i<when_count+1; i++) WRITE(" 0 0");
    WRITE(";\n");
}
void tiph_index_rules(void) {
    phrase *ph;
    int t, when_count = 0, tt_count = 0;
    LOOP_OVER(ph, phrase) {
        t = phud_get_timing_of_event(&(ph->usage_data));
        if (t == NOT_A_TIMED_EVENT) continue;
        if (t == NO_FIXED_TIME) {
            if (when_count == 0)
```

```
                INDEX("<p><i>Events with no specific time</i><br>");
            when_count++;
            INDEX("        ");
            phud_print_preamble_to(&(ph->usage_data), ifl);
            if ((ph->declaration_node != NULL) &&
                (ph->declaration_node->word_ref1 >= 0))
                index_link(lw_array[ph->declaration_node->word_ref1].lw_source);
            INDEX("<br>");
        }
    }
    LOOP_OVER(ph, phrase) {
        t = phud_get_timing_of_event(&(ph->usage_data));
        if (t == NOT_A_TIMED_EVENT) continue;
        if (t != NO_FIXED_TIME) {
            if (tt_count == 0)
                INDEX("<p><i>Timetable</i><br>");
            tt_count++;
            INDEX("        ");
            phud_print_preamble_to(&(ph->usage_data), ifl);
            if ((ph->declaration_node != NULL) &&
                (ph->declaration_node->word_ref1 >= 0))
                index_link(lw_array[ph->declaration_node->word_ref1].lw_source);
            INDEX("<br>");
        }
    }
    INDEX("<p>");
}
```

The function tiph_compile_TimedEventsTable_array is called from 12/cs.
The function tiph_compile_TimedEventTimesTable_array is called from 12/cs.
The function tiph_index_rules is called from 12/rb.

Rules 12/br

Purpose

The fourth of four ways phrases are invoked: as rules capable of being placed in rulebooks. Each such phrase is pointed to by one or more `booked_rule` structures, which are analogous to looseleaf pages for use in ring-binders: a page can either be left looseleaf, or can be bound into a single rulebook. NI requires that phrases be able to belong to multiple rulebooks, and this is achieved by creating a different BR for each such membership. Here we create and manage both individual BRs and linked lists of BRs in logical specificity order.

12/br.§4 Variables accessible from here; §5 Placing primary BRs into rulebooks; §6 Run-time representation; §7 Specificity of rules; §8 Printing rule names at run time; §9-10 Lists of booked rules; §11 Compilation and other serial operations on lists

---

Definitions

¶1. A booked rule is a wrapper structure: it's a pointer to a phrase together with some cross-references to make compilation and error message production easier. We indirect through booked rules, rather than pointing directly to phrases, not only for this convenience but also to ensure that the same phrase may be booked into several rulebooks at once.

Booked rules are also used to keep track of rule names early in NI's run. NI basically sorts out the data structures first, then attends to the rules, but this goes awry if the data structures – tables, for instance – include references to specific rules by name, thus using them as data. Such rules are therefore "predeclared", a process which really only records their names, early in NI's run. (For similar timing-of-parsing reasons we also record the usage details of the rule from its header, even though this eventually just duplicates what will appear in the relevant `phrase` structure.)

The field `I6_value_of_rule` is generated by compiled a `VALUE/RULE` SP with this booked rule as its reference, and copying the resultant I6 code into the field's text. We need this so that we can be certain of the I6 value to compare against when we need this rule as a constant in the I6 code.

```
typedef struct booked_rule {
    struct booked_rule *next_rule;                          linked list of entries in rulebook
    int next_rule_specificity;                              1 more specific than following, 0 equal, −1 less
    char *next_rule_specificity_law;                        description of reason
    char *next_rule_specificity_lawname;                    name of Law used to sort
    struct phrase *the_rule;                                the rule being booked
    int placement;                                          one of three placement values
    int primary_booking;                                    TRUE if this results from the main preamble booking
    int predeclared;                                        as a name with value not known?
    int word_ref1, word_ref2;                               text of this predeclared name
    int italicised_index_text_w1, italicised_index_text_w2; when indexing a rulebook
    int marked_for_anyone;
    struct stacked_variable_owner_list *listed_stv_owners;  making vars visible here
    char I6_value_of_rule[32];                              an I6 identifier, something like "R_341"
    MEMORY_MANAGEMENT
} booked_rule;
```

The structure booked_rule is private to this section.

¶2. When booked rules are gathered into linked lists of booked rules, they are positioned using "placements". Ordinarily they go somewhere in the middle, but declarations are allowed to specify that they must occur at the front or back, e.g.:

The first reaching inside rule: ...

The `owning_rulebook_placement` field is therefore always one of the following three values:

```
define MIDDLE_PLACEMENT 0                    most rules are somewhere in the middle
define FIRST_PLACEMENT 1
define LAST_PLACEMENT 2
```

¶3. We specify the way we want to add a BR to a list of BRs using the following enumerated values, which handle requirements like "before the awkward noises rule".

```
define NO_SIDE -2                                              a dummy value
define BEFORE_SIDE -1                                          before a reference BR
define IN_SIDE 0                                               without reference to any other BR
define AFTER_SIDE 1                                            after a reference BR
define INSTEAD_SIDE 2                                          in place of reference BR
```

---

§**1.** Booked rules can be created either from an I7 `phrase`, possibly initially `NULL` for a predeclared rule; or from the name of an I6 routine in the I6 library, in which case no I7 `phrase` is needed since the compiled form of the rule is already available.

```
booked_rule *br_new(phrase *ph) {
    booked_rule *br = CREATE(booked_rule);
    br->the_rule = ph;
    br->I6_value_of_rule[0] = 0;
    br->word_ref1 = -1;
    br->word_ref2 = -1;
    br->italicised_index_text_w1 = -1;
    br->italicised_index_text_w2 = -1;
    br->predeclared = FALSE;
    br->primary_booking = FALSE;
    br->next_rule_specificity = 0;
    br->next_rule_specificity_law = NULL;
    br->next_rule_specificity_lawname = NULL;
    br->listed_stv_owners = NULL;
    br->marked_for_anyone = FALSE;
    br->next_rule = NULL;
    br->placement = MIDDLE_PLACEMENT;
    return br;
}
booked_rule *new_br_from_i6(int w1, int w2, int i6_wn, int copy_flag) {
    booked_rule *br = br_new(NULL);
    if (copy_flag) {
        booked_rule *br2 = br_by_name(w1, w2);
        if (br2) br_copy(br, br2);
        else sentence_problem(_P_(BelievedImpossible),          there is no sensible way to test this
            "the concise action syntax of the Standard Rules has gone wrong",
            "which suggests that perhaps you are tampering with the very "
```

```
            "forces of creation. Fool! Know ye not the perils that await "
            "the unwary?");
    } else {
        if (i6_wn < 0) internal_error("no such rule name text");
        br_set_name(br, w1, w2);
        strcpy(br->I6_value_of_rule, lw_array[i6_wn].lw_text);
        LOGIF(LIBRARY_RULES, "New library rule: $b = %s\n",
            br, br->I6_value_of_rule);
    }
    return br;
}
void log_booked_rule(booked_rule *br) {
    if (br == NULL) { LOG("BR:<null-booked-rule>"); return; }
    LOG("BR%d", br->allocation_id);
    if (br->predeclared) LOG("p");
    switch (br->placement) {
        case MIDDLE_PLACEMENT: LOG("m"); break;
        case FIRST_PLACEMENT: LOG("f"); break;
        case LAST_PLACEMENT: LOG("l"); break;
        default: LOG("?"); break;
    }
    if (br->the_rule) LOG("[$R]", br->the_rule); else LOG("[-]");
}
```

The function br_new is called from 12/phud and 12/rps.
The function new_br_from_i6 is called from 12/cs.
The function log_booked_rule is called from 2/dl.

§**2.** We cannot copy a br simply by writing, say, `*to = *from` because this would also copy the hidden linked list references generated by the memory manager, and because we need to avoid similar duplication of the rulebook linked list placements, too. So we copy each element over by hand.

```
void br_copy(booked_rule *to, booked_rule *from) {
    to->the_rule = from->the_rule;
    strcpy(to->I6_value_of_rule, from->I6_value_of_rule);
    to->word_ref1 = from->word_ref1;
    to->word_ref2 = from->word_ref2;
    to->italicised_index_text_w1 = from->italicised_index_text_w1;
    to->italicised_index_text_w2 = from->italicised_index_text_w2;
    to->predeclared = from->predeclared;
    to->listed_stv_owners = stvol_append(NULL, from->listed_stv_owners);
    to->primary_booking = FALSE;                                    a copy cannot also be primary
    to->next_rule = NULL;
    to->placement = MIDDLE_PLACEMENT;
}
```

The function br_copy is called from 12/rps.

§**3.** Access to miscellaneous elements of the BR structure.

```
void br_set_name(booked_rule *br, int w1, int w2) {
    br->word_ref1 = w1;
    br->word_ref2 = w2;
    phud_verify_rule_name(w1, w2);
    register_excerpt_meaning(RULE_MC, 0, w1, w2, STORE_POINTER_booked_rule(br));
}
booked_rule *br_by_name(int w1, int w2) {
    meaning_list *ml;
    [[w1, w2 == the ... --> w1, w2]];
    ml = SP_excerpt(RULE_MC, w1, w2);
    if (ml) return RETRIEVE_POINTER_booked_rule(em_data(ml_meaning(ml)));
    return NULL;
}
void br_set_always_test_actor(booked_rule *br) {
    if (br->the_rule)
        phrcd_set_always_test_actor(&(br->the_rule->runtime_context_data));
}
void br_set_never_test_actor(booked_rule *br) {
    if (br->the_rule)
        phrcd_set_never_test_actor(&(br->the_rule->runtime_context_data));
}
void br_set_marked_for_anyone(booked_rule *br, int to) {
    br->marked_for_anyone = to;
}
void br_suppress_action_testing(booked_rule *br) {
    if (br->the_rule)
        phrcd_suppress_action_testing(&(br->the_rule->runtime_context_data));
}
void br_copy_actor_test_flags(booked_rule *br_to, booked_rule *br_from) {
    if (((br_from->the_rule == NULL) && (br_to->the_rule)) ||
        ((br_from->marked_for_anyone) && (br_to->marked_for_anyone == FALSE))) {
        phrcd_clear_always_test_actor(&(br_to->the_rule->runtime_context_data));
        phrcd_set_never_test_actor(&(br_to->the_rule->runtime_context_data));
    }
}
void br_set_italicised_index_text(booked_rule *br, int w1, int w2) {
    br->italicised_index_text_w1 = w1; br->italicised_index_text_w2 = w2;
}
void br_set_primary_rule(booked_rule *br, phrase *ph) {
    if (br == NULL) internal_error("tried to set rule of null br");
    br->the_rule = ph;
    br->primary_booking = TRUE;
}
phrase *br_get_rule(booked_rule *br) {
    if (br == NULL) return NULL;
    return br->the_rule;
}
void br_predeclare(booked_rule *br, int w1, int w2) {
    br->predeclared = TRUE;
```

```
    br_set_name(br, w1, w2);
}
```

The function br_set_name is called from 12/phud.
The function br_by_name is called from 12/phud and 12/rps.
The function br_set_always_test_actor is called from 12/rb.
The function br_set_never_test_actor is called from 12/rb.
The function br_set_marked_for_anyone is called from 12/phud.
The function br_suppress_action_testing is called from 12/rb.
The function br_copy_actor_test_flags is called from 12/rb.
The function br_set_italicised_index_text is called from 12/phud.
The function br_set_primary_rule is called from 12/phud.
The function br_get_rule is called from 12/cs and 12/phud.
The function br_predeclare is called from 12/phud.

§4. Variables accessible from here.

```
void br_acquire_stvol(booked_rule *br, stacked_variable_owner_list *stvol) {
    br->listed_stv_owners = stvol_append(br->listed_stv_owners, stvol);
}
stacked_variable_owner_list *br_stvol_accessible_here(booked_rule *br) {
    return br->listed_stv_owners;
}
void br_acquire_action_variables(booked_rule *br) {
    br_acquire_stvol(br, all_nonempty_stacked_action_vars);
    if (all_action_processing_vars == NULL) internal_error("APROC not ready");
    br_acquire_stvol(br, all_action_processing_vars);
}
```

The function br_acquire_stvol is called from 12/rb.
The function br_stvol_accessible_here is called from 12/cs.
The function br_acquire_action_variables is called from 12/rb.

§5. Placing primary BRs into rulebooks. Any BR marked as being the primary BR arising from a phrase is here officially placed into the rulebook it was declared as belonging to. Note that non-primary BRs (those arising, for instance, from declarations that a given rule name corresponds to a given I6 library routine name) are left alone.

Rules are insertion-sorted into rulebooks in `LOOP_OVER` order, which coincides with declaration order. This is important because it is creation order which the rule-sorting code falls back on when it can see no other justification for placing one rule either side of another.

```
void br_add_rules_to_rulebooks(void) {
    booked_rule *br;
    LOOP_OVER(br, booked_rule)
        if (br->primary_booking) {
            phrase *ph = br->the_rule;
            current_sentence = ph->declaration_node;
            phud_place_in_rulebook(&(ph->usage_data), br);
        }
}
```

The function br_add_rules_to_rulebooks is called from 12/cs.

§**6. Run-time representation.** In I6 code, a rule is compiled to the name of the routine implementing it, an I6 value of metaclass `Routine`. Note that two different BRs which point to the same `phrase` (or, though this is less likely, that provide two different I7 name aliases for the same I6 library routine) will compile to the same I6 value. This is essential, as it ensures that a phrase can indeed be used as a rule in more than one rulebook.

Note that a predeclared rule will be created with the I6 identifier `BR_21`, say, but will later (when phrase creation catches up) pick up a name like `R_306`. For consistency, these values must be equal in I6, and we ensure this by compiling I6 constant declarations such as:

```
Constant BR_21 = R_306;
```

```
void br_compile(OUTPUT_STREAM, booked_rule *br) {
    if (br->the_rule)
        WRITE("%s", ph_get_I6_representation(br->the_rule));
    else {
        if (br->I6_value_of_rule[0])
            WRITE("%s", br->I6_value_of_rule);
        else internal_error("tried to compile nameless booked rule");
        WRITE("RB    }
}
```

```
void br_resolve_predeclared_booked_rules(OUTPUT_STREAM) {
    booked_rule *br;
    LOOP_OVER(br, booked_rule)
        if ((br->predeclared) && (br->the_rule))
            WRITE("Constant BR_%d = %s;\n", br->allocation_id,
                ph_get_I6_representation(br->the_rule));
}
```

The function br_compile is called from 7/vasp.

The function br_resolve_predeclared_booked_rules is called from 12/cs.

§**7. Specificity of rules.** The following is one of NI's standardised comparison routines, which takes a pair of objects A, B and returns 1 if A makes a more specific description than B, 0 if they seem equally specific, or −1 if B makes a more specific description than A. This is transitive, and intended to be used in sorting algorithms.

```
int compare_specificity_of_br(booked_rule *br1, booked_rule *br2, int dflag) {
    phrase *ph1, *ph2;
    ph_runtime_context_data *rcd1 = NULL, *rcd2 = NULL;
    int rv;
    if ((br1 == NULL) || (br2 == NULL))
        internal_error("Compared specificity of null BR");
    if (dflag) {
        LOG("Comparing specificity of rules:\n(1) $b\n(2) $b\n", br1, br2);
    }
    ph1 = br1->the_rule;
    ph2 = br2->the_rule;
    if (ph1) rcd1 = &(ph1->runtime_context_data);
    if (ph2) rcd2 = &(ph2->runtime_context_data);
    rv = compare_specificity_of_phrcd(rcd1, rcd2);
    if (dflag) {
        if (rv != 0) {
            LOG("Decided by Law %s that ", c_s_stage_law);
```

```
        } else {
            LOG("Decided that ");
        }
        switch(rv) {
            case -1: LOG("(2) is more specific than (1)\n"); break;
            case 0: LOG("they are equally specific\n"); break;
            case 1: LOG("(1) is more specific than (2)\n"); break;
        }
    }
    return rv;
}
```

§8. Printing rule names at run time.

```
void br_compile_rule_printing_switch(OUTPUT_STREAM) {
    int j, k, cas; booked_rule *br; int brc=0;
    LOOP_OVER(br, booked_rule) {
        if (brc++ == 100000) internal_error("too many brs");
        j = -1; k = -1;
        if (br->word_ref1 >= 0) {
            [[j, k <-- br]];
            [[j, k == the ... --> j, k]];
        } else {
            if ((br->the_rule == NULL) ||
                (br->the_rule->declaration_node == NULL) ||
                (br->the_rule->declaration_node->down == NULL) ||
                (br->the_rule->declaration_node->down->word_ref1 < 0))
                continue;
        }
        WRITE("\t\tif (R == ");
        if (br->I6_value_of_rule[0])
            WRITE("%s", br->I6_value_of_rule);
        else
            WRITE("%s", ph_get_I6_representation(br->the_rule));
        WRITE(") { print \"");
        if (j >= 0) {
            print_raw_text_within_i6_literal(OUT, j, k); cas = 1;
        } else {
            if (br->the_rule->declaration_node) {
                [[j, k <-- br->the_rule->declaration_node]];
                [[j, k == the ... --> j, k]];
                print_raw_text_within_i6_literal(OUT, j, k);
                cas = 2;
            } else {
                if (br->I6_value_of_rule[0] != 0)
                    WRITE("%s", br->I6_value_of_rule);
                else
                    WRITE("%s", ph_get_I6_representation(br->the_rule));
                cas = 3;
            }
        }
        WRITE("\"; return; } ! %d, case %d\n", br->allocation_id, cas);
    }
```

```
}
```

The function br_compile_rule_printing_switch is called from 12/cs.

§**9. Lists of booked rules.** Booked rules are, in effect, two classes in one: the individual booked rules, and the linked list of an ordered sequence of the same. Methods dealing with lists of booked rules are distinguished by the `br_list` prefix.

BR lists are changed only by creation (naturally) and by the process of adding a rule to a BR list. The following invariants are preserved:

(a) The list head is a dummy BR which has never been the subject of any addition operation and has never moved.

(b) The only `FIRST_PLACEMENT` entries in the list immediately follow the list head. Those which were added explicitly as first-placed are in reverse order of addition to the list.

(c) The only `LAST_PLACEMENT` entries in the list are at the end. Those which were added explicitly as last-placed are in order of addition to the list.

(d) If R and S are middle-placed rules which were placed in the list within the same range (say, both anywhere, or both "after T" or "before U") and R precedes S, then either R is more specific than S, or they are equally specific and R was added to the list before S.

```
booked_rule *br_list_new(void) {
    return br_new(NULL);
}

int br_list_is_empty(booked_rule *list_head, scene *context) {
    booked_rule *br;
    if (list_head == NULL) return TRUE;
    br = list_head->next_rule;
    while (br) {
        if (context == NULL) return FALSE;
        if ((br->the_rule) &&
            (phrcd_get_scene(&(br->the_rule->runtime_context_data)) == context))
            return FALSE;
        br = br->next_rule;
    }
    return TRUE;
}

int br_list_is_empty_of_i7_rules(booked_rule *list_head) {
    booked_rule *br;
    if (list_head == NULL) return TRUE;
    br = list_head->next_rule;
    while (br) {
        if (br->the_rule) return FALSE;
        br = br->next_rule;
    }
    return TRUE;
}

int br_list_contains(booked_rule *list_head, booked_rule *to_find) {
    booked_rule *br;
    if (list_head == NULL) return FALSE;

    br = list_head->next_rule;
    while ((br) && (br_test_eventual_I6_routine_equal(br, to_find) == FALSE))
        br = br->next_rule;
    if (br) return TRUE;
    return FALSE;
```

```
}
int br_list_contains_ph(booked_rule *list_head, phrase *ph_to_find) {
    booked_rule *br;
    if (list_head == NULL) return FALSE;
    br = list_head->next_rule;
    while ((br) && (br->the_rule != ph_to_find))
        br = br->next_rule;
    if (br) return TRUE;
    return FALSE;
}
int br_test_eventual_I6_routine_equal(booked_rule *br1, booked_rule *br2) {
    if ((br2->the_rule == NULL) && (br1->the_rule != NULL)) return FALSE;
    if ((br1->the_rule == NULL) && (br2->the_rule != NULL)) return FALSE;
    if (br1->the_rule) {
        if (br1->the_rule == br2->the_rule) return TRUE;
    } else {
        if (strcmp(br1->I6_value_of_rule, br2->I6_value_of_rule) == 0)
            return TRUE;
    }
    return FALSE;
}
```

The function br_list_new is called from 12/rb.

The function br_list_is_empty is called from 12/rb.

The function br_list_is_empty_of_i7_rules is called from 12/rb.

The function br_list_contains is called from 12/rb.

The function br_list_contains_ph is called from 12/rb.

§**10.** When rule R is explicitly placed into (the rule list of) rulebook B (by an assertion like "R is listed after S in B", say), there are evidently two possibilities:

(i) R's phrase already occurs somewhere in rulebook B, so that this affects only the ordering of rulebook B. We therefore remove it (so that it does not occur twice in B) and reinsert it within the position range indicated. Note that this process still makes use of logical precedence; it simply confines itself to a narrower range. If R has to occur before S, then R is placed according to logical precedence within the sublist from the head of the list up to just-before-S.

To determine whether or not R's phrase is already in B, we compare their phrases if set, and their I6 equivalents if not. This will work on all non-declared BRs arising from phrases, and all BRs arising from named I6 library equivalents: it would however fail to recognise duplicates among predeclared BRs. However, it's impossible for these to arise in a way which permits duplication, so this does not matter.

Note that we search the entire rulebook for R, not just the valid interval in the rulebook where R might go (e.g., "after S").

(ii) R's phrase does not occur in B. We insert R into rulebook B within the position range indicated.

Each addition leaves the list either the same size or longer by 1.

If we insert a rule as first-placed rule when there already is a first-placed rule, the new one displaces it to go first, but both continue to be labelled as "first-placed", so that subsequent rule insertions of middle-placed rules will still go after both of them. Symmetrically, a second last-placed rule is inserted after any existing one, but both are labelled "last-placed". Because of the range possibility ("after S") we might find ourselves inserting a rule as middle-placed and yet still after a last-placed rule, or before a first-placed one: if so we change its placement to last or first respectively, in order to preserve invariants (b) and (c) above.

There was a small debate on `rec.arts.int-fiction` in February 2009 as to whether a rule placed instead of another rule within the same rulebook should be duplicated, or moved. In builds from 2008 and earlier,

there was duplication, but this broke the clean principle that a rule appears only once per rulebook, and made it difficult to place certain rules with tricky preambles; on the other hand merely moving makes it more difficult to replace a whole run of rules with a single place-holder. Both sides were argued for. In March 2009, it was finally decided to go with moving, not duplication, and to preserve the "only once per rulebook" principle.

```
void br_list_add(booked_rule *list_head, booked_rule *new_rule,
    int placing, int side, booked_rule *ref_rule) {
    booked_rule *pos, *prev, *end_rule, *start_rule, *subseq;
    int dflag;
    if ((side != IN_SIDE) && (ref_rule == NULL))
        internal_error("tried to add before or after non-rule");
    if ((side != IN_SIDE) && (placing != MIDDLE_PLACEMENT))
        internal_error("tried to add before or after but with non-middle placement");
    if ((side == IN_SIDE) && (ref_rule != NULL))
        internal_error("tried to add in middle but with ref rule");
    if (list_head == NULL)
        internal_error("tried to add rule to null list");
    switch(placing) {
        case MIDDLE_PLACEMENT: break;
        case FIRST_PLACEMENT: LOGIF(RULE_ATTACHMENTS, "Placed first\n"); break;
        case LAST_PLACEMENT: LOGIF(RULE_ATTACHMENTS, "Placed last\n"); break;
        default:
            LOG("Invalid placing %d\n", placing);
            internal_error("invalid placing of rule");
    }
    for (prev=list_head, pos=list_head->next_rule; pos; prev=pos, pos=pos->next_rule)
        if (br_test_eventual_I6_routine_equal(pos, new_rule)) {
            if ((side == IN_SIDE) && (placing == MIDDLE_PLACEMENT))
                return;                                   rule is already in rulebook: do nothing
            prev->next_rule = pos->next_rule;
            pos->next_rule = NULL;
            LOGIF(RULE_ATTACHMENTS, "Removing previous entry from rulebook\n");
            break;                                rule can only appear once, so no need to keep checking
        }
    if (side == INSTEAD_SIDE) {
        for (prev=list_head, pos=list_head->next_rule; pos; prev=pos, pos=pos->next_rule)
            if (br_test_eventual_I6_routine_equal(pos, ref_rule)) {
                new_rule->placement = pos->placement;              replace with same placement
                new_rule->next_rule = pos->next_rule;
                prev->next_rule = new_rule;
            }
        return;
    }
    new_rule->placement = placing;
    if (placing == FIRST_PLACEMENT) {                  first in valid interval (must be whole list)
        subseq = list_head->next_rule;
        list_head->next_rule = new_rule;
        new_rule->next_rule = subseq;                    pushes any existing first rule forward
        return;
    }
    if (placing == LAST_PLACEMENT) {                   last in valid interval (must be whole list)
```

```
    prev = list_head;
    while (prev->next_rule != NULL) prev = prev->next_rule;
    prev->next_rule = new_rule;                          pushes any existing last rule backward
    new_rule->next_rule = NULL;
    return;
}
start_rule = list_head;                  valid interval begins one link after the dummy start marker
end_rule = NULL;                         valid interval ends before this rule, or runs to end if NULL
switch(side) {
    case BEFORE_SIDE:
        for (pos=list_head->next_rule; pos; pos=pos->next_rule)
            if (br_test_eventual_I6_routine_equal(pos, ref_rule))
                end_rule = pos;                          insert before: so valid interval ends here
        if (end_rule == NULL) internal_error("can't find end rule");
        break;
    case AFTER_SIDE:
        for (pos=list_head->next_rule; pos; pos=pos->next_rule)
            if (br_test_eventual_I6_routine_equal(pos, ref_rule))
                start_rule = pos;                        insert after: so valid interval begins here
        if (start_rule == list_head) internal_error("can't find start rule");
        break;
}
pos = list_head;
while ((pos != NULL) && (pos != start_rule)) pos = pos->next_rule;
if (pos == NULL) internal_error("can't find start of valid interval");
if (end_rule) {
    pos = list_head;
    while ((pos != NULL) && (pos != end_rule)) pos = pos->next_rule;
    if (pos == NULL) internal_error("can't find end of valid interval");
}
dflag = FALSE; if (dl_this(SPECIFICITIES_DA)) dflag = TRUE;
prev = start_rule;                                   find p ∈ [s, e) and place new rule after p
while ((prev->next_rule != end_rule)
    && (prev->next_rule->placement == FIRST_PLACEMENT))
    prev = prev->next_rule;                  move p forward to final valid first rule (if any exist)
while ((prev->next_rule != end_rule)                                stop before p leaves valid range
    && (prev->next_rule->placement != LAST_PLACEMENT)                    or reaches a last rule
    && (compare_specificity_of_br(prev->next_rule, new_rule,
        dflag) >= 0))                                    or a rule less specific than the new one
    prev = prev->next_rule;
subseq = prev->next_rule;
prev->next_rule = new_rule;
new_rule->next_rule = subseq;
if ((prev != list_head) &&
    (prev->placement == LAST_PLACEMENT))
    new_rule->placement =
        LAST_PLACEMENT;                          happens if valid interval is after a last rule
if ((subseq) &&
    (subseq->placement == FIRST_PLACEMENT))
    new_rule->placement =
        FIRST_PLACEMENT;                         happens if valid interval is before a first rule
```

```
}
void br_list_remove(booked_rule *list_head, booked_rule *ref_rule) {
    booked_rule *br, *prev;
    for (br = list_head->next_rule, prev = list_head; br; prev = br,
        br = br->next_rule) {
        if (br_test_eventual_I6_routine_equal(br, ref_rule)) {
            prev->next_rule = br->next_rule;
            return;
        }
    }
}
```

The function br_list_add is called from 12/rb.
The function br_list_remove is called from 12/rb.

§11. **Compilation and other serial operations on lists.** None of which change the contents in any way.

```
int show_index_links = TRUE;
void br_list_suppress_indexed_links(void) {
    show_index_links = FALSE;
}
void br_list_resume_indexed_links(void) {
    show_index_links = TRUE;
}
void index_rule_name(int w1, int w2) {
    [[w1, w2 == the ... --> w1, w2]];
    print_raw_text_to_file(w1, w2, ifl);
}
int use_numbered_rules = FALSE;
void br_set_numbered_rules(void) {
    use_numbered_rules = TRUE;
}
int br_list_index(booked_rule *list_head, scene *context,
    action_name *action_context, int before_or_after, char *billing, rulebook *owner) {
    booked_rule *br, *prev;
    int count = 0;
    for (br = list_head->next_rule, prev = NULL; br; prev = br, br = br->next_rule) {
        int circs = FALSE;
        int w1 = -1, w2 = -1;
        if (br->the_rule) {
            if ((context) &&
                (phrcd_get_scene(&(br->the_rule->runtime_context_data)) != context))
                continue;
            if ((action_context) &&
                (phrcd_within_action_context(&(br->the_rule->runtime_context_data),
                    action_context) == FALSE))
                continue;
            if (br->word_ref1 < 0) {
                if ((br->the_rule->declaration_node) &&
                    (br->the_rule->declaration_node->down)) {
                    w1 = br->the_rule->declaration_node->down->word_ref1;
```

```
            w2 = br->the_rule->declaration_node->down->word_ref2;
            if [[w1, w2 == abide by i6 ...]] continue;
            if [[w1, w2 == cause the ### to take place]] continue;
        }
        if ((br->italicised_index_text_w1 < 0) && (before_or_after))
            continue;
    }
}
count++;
br_start_index_line(prev, billing);
if (br->italicised_index_text_w1 >= 0) {
    INDEX("<i>");
    index_rule_name(
        br->italicised_index_text_w1, br->italicised_index_text_w2);
    INDEX("</i>  ");
    circs = TRUE;
}
if (br->word_ref1 >= 0) {
    INDEX("<font color=\"#800000\">");
    index_rule_name(br->word_ref1, br->word_ref2);
    INDEX("</font>");
    INDEX("  ");
    char temp[600];
    print_raw_text_to_string_truncated(
        br->word_ref1, br->word_ref2, temp, 250);
    write_javascript_paste(ifl, -1, -1, temp);
    INDEX(" <i>name</i> ");
    sprintf(temp, "The ");
    print_text_to_string_truncated(
        br->word_ref1, br->word_ref2, temp+strlen(temp), 250);
    sprintf(temp+strlen(temp), " is not listed in the ");
    print_text_to_string_truncated(
        owner->word_ref1, owner->word_ref2, temp+strlen(temp), 250);
    sprintf(temp+strlen(temp), " rulebook.\n");
    write_javascript_paste(ifl, -1, -1, temp);
    INDEX(" <i>unlist</i>");
} else {
    if ((circs == FALSE) && (br->the_rule)) {
        parse_node *pn = br->the_rule->declaration_node->down;
        int lines = 0;
        while (pn != NULL) {
            w1 = pn->word_ref1;
            w2 = pn->word_ref2;
            lines++;
            if [[w1, w2 == abide by the ...]] {
                if (lines>1)
                    INDEX("<br>    "
                        "%s    ", billing);
                print_raw_text_to_file(w1+3, w2, ifl);
                pn = pn->next;
            } else {
                INDEX("(");
                print_raw_text_to_file(w1, w2, ifl);
```

```
                        if (pn->next != NULL)
                            INDEX("; ...");
                        INDEX(")");
                        break;
                    }
                }
            }
        }
        if ((br->the_rule != NULL) &&
            (br->the_rule->declaration_node != NULL) &&
            (br->the_rule->declaration_node->word_ref1 >= 0))
            index_link(lw_array[br->the_rule->declaration_node->word_ref1].lw_source);
        if (use_numbered_rules) {
            if (br->the_rule)
                INDEX(" <small>%d</small>", br->the_rule->allocation_id);
            else
                INDEX(" <small>primitive</small>");
        }
        INDEX("</p>\n");
    }
    return count;
}
void br_start_index_line(booked_rule *prev, char *billing) {
    open_html_paragraph(ifl, 2, "hanging");
    if (billing[0]) br_show_linkage_icon(prev);
    INDEX("%s", billing);
    INDEX("    ");
    if (billing[0] == 0) br_show_linkage_icon(prev);
}
void br_show_linkage_icon(booked_rule *prev) {
    char *icon_name = NULL;                                    redundant assignment to appease gcc -O2
    if (show_index_links == FALSE) return;
    if ((prev == NULL) || (prev->next_rule_specificity_law == NULL)) {
        html_icon_with_tooltip(ifl, "rulenone.png", "start of rulebook", NULL);
        return;
    }
    switch (prev->next_rule_specificity) {
        case -1: icon_name = "ruleless.png"; break;
        case 0: icon_name = "ruleequal.png"; break;
        case 1: icon_name = "rulemore.png"; break;
        default: internal_error("unknown rule specificity");
    }
    html_icon_with_tooltip(ifl, icon_name, prev->next_rule_specificity_law,
        prev->next_rule_specificity_lawname);
}
#define RULE_OPTIMISATION_THRESHOLD 20
void br_list_compile(booked_rule *list_head, OUTPUT_STREAM, char *identifier) {
    int countup = 0, group_size = 0;
    booked_rule *br;
    for (br = list_head->next_rule; br; br = br->next_rule) countup++;
    if (countup == 0) {
        WRITE("Constant %s = EMPTY_RULEBOOK;\n", identifier);
```

```
        return;
    }
    WRITE("Array %s -->", identifier);
    if (countup > RULE_OPTIMISATION_THRESHOLD) WRITE(" (-2)");
    for (br = list_head->next_rule; br; br = br->next_rule) {
        specification *spec = booked_rule_to_RULE_spec(br);
        if (countup > RULE_OPTIMISATION_THRESHOLD) {
            if (group_size == 0) {
                action_name *an = br_required_action(br);
                booked_rule *brg = br;
                while ((brg) && (an == br_required_action(brg))) {
                    group_size++;
                    brg = brg->next_rule;
                }
                if (an) WRITE(" ##%s", an_get_I6_representation(an));
                else WRITE(" (-2)");
                if (group_size >= 32) group_size = 31;
                if (group_size > 1) WRITE(" %d", group_size);
            }
            group_size--;
        }
        WRITE(" ");
        TEMPORARY_STREAM;
        spec_compile(TEMP, spec);
        truncated_strcpy(br->I6_value_of_rule, STREAM_TEXT(TEMP), 31);
        STREAM_COPY(OUT, TEMP);
        CLOSE_TEMPORARY_STREAM;
    }
    WRITE(" NULL; ! %d rule(s)\n", countup);
}

action_name *br_required_action(booked_rule *br) {
    phrase *ph = (br->the_rule);
    if (ph) return phrcd_required_action(&(ph->runtime_context_data));
    return NULL;
}

void br_list_log(booked_rule *list_head) {
    booked_rule *br; int t=0, s=0;
    if (list_head == NULL) { LOG("<null-booked-rule-list>\n"); return; }
    for (br = list_head->next_rule; br; br = br->next_rule) t++;
    if (t == 0) { LOG("<empty-booked-rule-list>\n"); return; }
    for (br = list_head->next_rule; br; br = br->next_rule) {
        LOG("  %d/%d. $b", ++s, t, br);
        if (br->the_rule == NULL) LOG(" $W", br->word_ref1, br->word_ref2);
        LOG("\n");
    }
}

void br_list_judge_ordering(booked_rule *list_head) {
    booked_rule *br;
    if (list_head == NULL) return;
    for (br = list_head->next_rule; br; br = br->next_rule) {
        if (br->next_rule) {
            if (br->placement != br->next_rule->placement) {
                switch(br->placement) {
```

```
            case FIRST_PLACEMENT:
                switch(br->next_rule->placement) {
                    case MIDDLE_PLACEMENT:
                        br->next_rule_specificity = 1;
                        br->next_rule_specificity_law =
                            "the rule above was listed as 'first' so "
                            "precedes this one, which wasn't";
                        break;
                    case LAST_PLACEMENT:
                        br->next_rule_specificity = 1;
                        br->next_rule_specificity_law =
                            "the rule above was listed as 'first' so "
                            "precedes this one, which was listed as 'last'";
                        break;
                    default:
                        br_list_log(list_head);
                        internal_error("br list invariant broken");
                        break;
                }
                break;
            case MIDDLE_PLACEMENT:
                switch(br->next_rule->placement) {
                    case LAST_PLACEMENT:
                        br->next_rule_specificity = 1;
                        br->next_rule_specificity_law =
                            "the rule below was listed as 'last' so comes "
                            "after the rule above, which wasn't";
                        break;
                    default:
                        br_list_log(list_head);
                        internal_error("br list invariant broken");
                        break;
                }
                break;
            default:
                br_list_log(list_head);
                internal_error("br list invariant broken");
                break;
        }
    } else {
        int r;
        switch(br->placement) {
            case FIRST_PLACEMENT:
                br->next_rule_specificity = 0;
                br->next_rule_specificity_law =
                    "these rules were both listed as 'first' in this "
                    "rulebook, and since each was made 'first' in turn, "
                    "they appear in reverse order of listing";
                break;
            case MIDDLE_PLACEMENT:
                r = compare_specificity_of_br(br, br->next_rule, FALSE);
                br->next_rule_specificity = r;
                if (r == 0) br->next_rule_specificity_law =
```

```
                        "these rules are equally ranked, so their order is "
                        "determined by which was defined first (or by "
                        "explicit 'listed in' sentences)";
                    else {
                        br->next_rule_specificity_law =
                        "the arrow points from a more specific rule to a "
                        "more general one, as decided by Law";
                        br->next_rule_specificity_lawname = c_s_stage_law;
                    }
                    break;
                case LAST_PLACEMENT:
                    br->next_rule_specificity = 0;
                    br->next_rule_specificity_law =
                        "these rules were both listed as 'last' in this "
                        "rulebook, and since each was made 'last' in turn, "
                        "they appear in order of listing";
                    break;
            }
        }
    } else {
        br->next_rule_specificity = 0;
        br->next_rule_specificity_law = NULL;
    }
  }
}
void br_list_compile_rule_phrases(booked_rule *list_head,
    OUTPUT_STREAM, int *i, int max_i) {
    booked_rule *br; int t=0, s=0;
    if (list_head == NULL) return;
    for (br = list_head->next_rule; br; br = br->next_rule) t++;
    for (br = list_head->next_rule; br; br = br->next_rule) {
        s++;
        if (br->the_rule) {
            WRITE("! Rule %d/%d ", s, t);
            phud_write_I6_comment_describing(&(br->the_rule->usage_data), OUT);
        } else {
            WRITE("! Rule %d/%d: %s\n", s, t, br->I6_value_of_rule);
        }
        if (br->next_rule) {
            if (br->placement != br->next_rule->placement) {
                WRITE("! --- now the ");
                switch(br->next_rule->placement) {
                    case FIRST_PLACEMENT:
                        WRITE("first-placed rules"); break;
                    case MIDDLE_PLACEMENT:
                        WRITE("mid-placed rules"); break;
                    case LAST_PLACEMENT:
                        WRITE("last-placed rules"); break;
                }
                WRITE(" ---\n");
            } else {
                switch(br->next_rule_specificity) {
                    case -1:
```

```
                        WRITE("!   <<< %s <<<\n",
                                br->next_rule_specificity_lawname);
                        break;
                    case 0:
                        WRITE("!   === which is equally specific with ===\n");
                        break;
                    case 1:
                        WRITE("!   >>> %s >>>\n",
                                br->next_rule_specificity_lawname);
                        break;
                }
            }
        }
    }
    compile_i6_divider_comment(OUT);
    s = 0;
    for (br = list_head->next_rule; br; br = br->next_rule) {
        if (br->the_rule) {
            ph_compile(br->the_rule, OUT, i, max_i, br_stvol_accessible_here(br));
        }
    }
}
```

The function br_list_suppress_indexed_links is called from 12/rb.
The function br_list_resume_indexed_links is called from 12/rb.
The function br_set_numbered_rules is called from 10/isin.
The function br_list_index is called from 12/rb.
The function br_list_compile is called from 12/rb.
The function br_list_log is called from 12/rb.
The function br_list_judge_ordering is called from 12/rb.
The function br_list_compile_rule_phrases is called from 12/rb.

Rulebooks 12/rb

Purpose

To create, manage, compile and index rulebooks, the content of which is a linked list of booked rules together with some general conventions as to how they are to be used.

Template interpreter commands

```
4   {-array:rulebook_var_creators}
9   {-routine:RulebookOutcomePrintingRule}
10   {-callv:index_page_Rules}
```

Definitions

¶1. A rulebook consists of some general properties together with a linked list of booked rules, which constitute its entries. Some rulebooks are created explicitly by the user's source text, some are created explicitly by the Standard Rules, while others are "automatically generated" as a result of other creations by NI. For instance, each new scene ending generates a rulebook. There are numerous other examples because a rulebook is the natural and most flexible way to provide a "hook" by which to attach behaviour to a world model.

In some ways phrases and rules are really subsidiary ideas, and the rulebook is the fundamental level of programming in Inform 7. The reason it turns up so late in this chapter of definitions is that all of the other data structures were building up to this one: a `rulebook` contains `booked_rule`s which refer to `phrase`s whose usage is defined with `specification`s, and whose definition is stored in the `parse_node` tree until it can be resolved as a list of `invocation`s.

define `MAX_OUTCOMES_PER_RULEBOOK 10`

```
typedef struct rulebook {
    int word_ref1, word_ref2;                                   Name in source text
    int alt_word_ref1, alt_word_ref2;                           Alternative form of name
    int default_outcome_declared;                               flag: whether author has declared this
    int default_rule_outcome;                                   success, failure or none: see above
    int no_outcomes;
    int default_named_outcome;                                  alternative to the above
    struct rulebook_outcome named_outcomes[MAX_OUTCOMES_PER_RULEBOOK];
    int rulebook_focus;                                         always ACTION_FOCUS or PARAMETER_FOCUS
    int has_no_parameter;                                       if created as NO_FOCUS, this is set
    int rules_always_test_actor;                                for action-tied check, carry out, report
    int runs_during_activities;                                 allow "while..." clauses to name these
    int fragmentation_stem_length;                              0 for all ordinary rulebooks
    struct booked_rule *rule_list;                              linked list of booked rules
    struct placement_affecting *placement_list;                 linked list of explicit placements
    struct stacked_variable_owner *owned_by_rb;                 rulebook variables owned here
    struct stacked_variable_owner_list *accessible_from_rb;     and which can be named here
    char rb_I6_identifier[32];                                  I6 identifier name for array holding contents
    int used_by_future_action_activity;                         like "deciding the scope of something..."
```

```
    int automatically_generated;                    purely so that the index can omit these
    MEMORY_MANAGEMENT
} rulebook;
```

The structure rulebook is private to this section.

¶2. Placements:

```
typedef struct placement_affecting {
    struct parse_node *placement_sentence;
    struct placement_affecting *next;
    MEMORY_MANAGEMENT
} placement_affecting;
```

The structure placement_affecting is private to this section.

¶3. Each rulebook reaches one of three possible outcomes: success, failure and neither (meaning that it proceeded through to the end without any positive or negative news arising). If a rule does apply to the current circumstances (i.e., if the parameter is matched successfully) then it can explicitly choose to produce any of the three outcomes: if it makes no decision, the result is that stored in the rulebook's `default_rule_outcome` field. This is how

> Instead of going north, say "The snow is too deep."
>
> Before going north, say "Well, I'll try..."

behave differently: the instead rule halts the action, because the instead rulebook has a default outcome of `FAILURE_OUTCOME`; the before rule allows it to proceed, because the default is `NO_OUTCOME`.

```
define UNRECOGNISED_OUTCOME -1                      used in parsing only
define NO_OUTCOME 0
define SUCCESS_OUTCOME 1
define FAILURE_OUTCOME 2
```

¶4. However, a rulebook is allowed to give special problem-specific names to its outcomes.

```
typedef struct named_rulebook_outcome {
    int word_ref1, word_ref2;                       Name in source text
    MEMORY_MANAGEMENT
} named_rulebook_outcome;

typedef struct rulebook_outcome {
    struct named_rulebook_outcome *outcome_name;
    int kind_of_outcome;                            one of the three values above
} rulebook_outcome;
```

The structure named_rulebook_outcome is private to this section.
The structure rulebook_outcome is private to this section.

¶5. When a rulebook is being worked through at run-time, a special value or situation normally gives it a focus – something to work on. Rulebooks appear to the Inform user to come in three types:

(0) Action rulebooks, like "Instead of", which take a pattern match on the current action as their focus: for instance, the rule "Instead of eating something portable..." is in the "Instead of eating" rulebook but uses "something portable" as focus.

(1) Parametrised rulebooks, like "reaching inside", take (a pattern match on) a single parameter as their focus. For instance, "A rule for reaching inside the flask..." has the flask as focus.

(2) Rulebooks with no focus, like "turn sequence", are independent of circumstances.

In fact there are only really two types because rulebooks without focus (2) are internally converted to action rulebooks (0), but where the pattern matching is so broad that it matches every action. And in any case we set up the `action_pattern` structure in such a way that parametrised rulebooks (1) are handled almost exactly similarly to action ones as far as NI is concerned.

```
define ACTION_FOCUS 0
define PARAMETER_FOCUS 1
define NO_FOCUS 2
```

¶6. The following is used only to store the result of parsing text as a rulebook name:

```
typedef struct rulebook_match {
    struct rulebook *matched_rulebook;
    int match_length;
    int article_used;
    int placement_requested;
} rulebook_match;
```

The structure rulebook_match is shared with 12/phud.

¶7. The built-in rulebooks. As rulebooks are declared, the first few are quietly copied into a small array: that way, we can always obtain a pointer to, say, the turn sequence rules by looking up `built_in_rulebooks[TURN_SEQUENCE_RB]`. (These pointers are used only in the Rulebooks section.)

```
define MAX_BUILT_IN_RULEBOOKS 32

rulebook *built_in_rulebooks[MAX_BUILT_IN_RULEBOOKS];
struct stacked_variable_owner_list *all_action_processing_vars = NULL;
```

¶8. Many of the standard NI-created rulebooks need to have numbers which are predictable, because they need to be referred to by number by the I6 library code. Because of this, it is important not to change the numbers below without checking the corresponding I6 Constant declarations in the `Rulebooks.i6t` file: the two sets of declarations must exactly match. They must also exactly match the sequence in which these rulebooks are created in the Standard Rules file.

define PROCEDURAL_RB 0 — *Procedural rules*

define STARTUP_RB 1 — *Startup rules*
define TURN_SEQUENCE_RB 2 — *Turn sequence rules*
define SHUTDOWN_RB 3 — *Shutdown rules*
define SCENE_CHANGING_RB 4 — *Scene changing rules*
define WHEN_PLAY_BEGINS_RB 5 — *When play begins*
define WHEN_PLAY_ENDS_RB 6 — *When play ends*
define EVERY_TURN_RB 7 — *Every turn*

define ACTION_PROCESSING_RB 8 — *Action-processing rules*
define SETTING_ACTION_VARIABLES_RB 9 — *Setting action variables rules*
define SPECIFIC_ACTION_PROCESSING_RB 10 — *Specific action-processing rules*
define PLAYERS_ACTION_AWARENESS_RB 11 — *Player's action awareness rules*

define ACCESSIBILITY_RB 12 — *Accessibility rules*
define REACHING_INSIDE_RB 13 — *Reaching inside rules*
define REACHING_OUTSIDE_RB 14 — *Reaching outside rules*
define VISIBILITY_RB 15 — *Visibility rules*

define PERSUASION_RB 16 — *Persuasion rules*
define UNSUCCESSFUL_ATTEMPT_BY_RB 17 — *Unsuccessful attempt by*

define BEFORE_RB 18 — *Before rules*
define INSTEAD_RB 19 — *Instead rules*
define CHECK_RB 20 — *Check*
define CARRY_OUT_RB 21 — *Carry out rules*
define AFTER_RB 22 — *After rules*
define REPORT_RB 23 — *Report*

define DOES_THE_PLAYER_MEAN_RB 24 — *Does the player mean...? rules*

---

§**1. Construction.** Whereas, at run-time, rulebooks are special cases of rules (they have the same kind of value, though their I6 values are such as to make it possible to distinguish them), within NI rulebooks and rules have entirely different data structures. There are two constructor functions: a basic one, used for those created by typical source text, and an advanced one used when rulebooks are automatically created as a result of other structures being built (for instance, scene endings).

```
rulebook *rb_new(int w1, int w2, int parametrisation) {
    rulebook *rb = CREATE(rulebook);
    [[w1, w2 == the ... --> w1, w2]];
    [[w1, w2 == ... rules --> w1, w2]];
    if [[w1, w2 == at ***]] {
        sentence_problem(_P_(C12RulebookWithAt),
            "this would create a rulebook whose name begins with 'at'",
            "which is forbidden since it would lead to ambiguities in "
            "the way people write rules. A rule beginning with 'At' "
            "is one which happens at a given time, whereas a rule "
            "belonging to a rulebook starts with the name of that "
            "rulebook, so a rulebook named 'at ...' would make such "
```

```
        "a rule inscrutable.");
}
if [[w1, w2 == definition ***]] {
    sentence_problem(_P_(C12RulebookWithDefinition),
        "this would create a rulebook whose name begins with 'definition'",
        "which is forbidden since it would lead to ambiguities in "
        "the way people write rules. A rule beginning with 'Definition' "
        "is one which defines an adjective, whereas a rule "
        "belonging to a rulebook starts with the name of that "
        "rulebook, so a rulebook named 'to ...' would make such "
        "a rule inscrutable.");
}
if [[w1, w2 == to ***]] {
    sentence_problem(_P_(C12RulebookWithTo),
        "this would create a rulebook whose name begins with 'to'",
        "which is forbidden since it would lead to ambiguities in "
        "the way people write rules. A rule beginning with 'To' "
        "is one which defines a phrase, whereas a rule "
        "belonging to a rulebook starts with the name of that "
        "rulebook, so a rulebook named 'to ...' would make such "
        "a rule inscrutable.");
}
rb->word_ref1 = w1; rb->word_ref2 = w2;
rb->alt_word_ref1 = -1; rb->alt_word_ref2 = -1;
register_excerpt_meaning(RULEBOOK_MC, 0, w1, w2,
    STORE_POINTER_rulebook(rb));
register_reworded_meaning(RULEBOOK_MC, 0, 0, 0, w1, w2, rules_V,
    STORE_POINTER_rulebook(rb));
register_reworded_meaning(RULEBOOK_MC, 0, 0, 0, w1, w2, rulebook_V,
    STORE_POINTER_rulebook(rb));
isn_compose_identifier(rb->rb_I6_identifier, 'B', rb->allocation_id, w1, w2);
rb->rule_list = br_list_new();
rb->automatically_generated = FALSE;
rb->has_no_parameter = FALSE;
rb->rules_always_test_actor = FALSE;
rb->used_by_future_action_activity = FALSE;
rb->runs_during_activities = FALSE;
if (parametrisation == NO_FOCUS) {
    parametrisation = ACTION_FOCUS;
    rb->has_no_parameter = TRUE;
} else rb->has_no_parameter = FALSE;
rb->rulebook_focus = parametrisation;
rb->fragmentation_stem_length = 0;
rb->default_outcome_declared = FALSE;
rb->default_rule_outcome = NO_OUTCOME;
rb->default_named_outcome = -1;
if (rb->allocation_id == INSTEAD_RB) rb->default_rule_outcome = FAILURE_OUTCOME;
if (rb->allocation_id == AFTER_RB) rb->default_rule_outcome = SUCCESS_OUTCOME;
if (rb->allocation_id == UNSUCCESSFUL_ATTEMPT_BY_RB) rb->default_rule_outcome = SUCCESS_OUTCOME;
rb->no_outcomes = 0;
rb->placement_list = NULL;
```

```
    rb->owned_by_rb = stvo_new(rb->allocation_id);
    rb->accessible_from_rb = stvol_add_owner(NULL, rb->owned_by_rb);
    if (rb->allocation_id < MAX_BUILT_IN_RULEBOOKS)
        built_in_rulebooks[rb->allocation_id] = rb;
    if (rb == built_in_rulebooks[ACTION_PROCESSING_RB])
        all_action_processing_vars = stvol_add_owner(NULL, rb->owned_by_rb);
    return rb;
}
rulebook *rb_new_automatic(int w1, int w2, int parametrisation,
    int oc, int ata, int ubfaa, int rda) {
    rulebook *rb = rb_new(w1, w2, parametrisation);
    rb->automatically_generated = TRUE;
    rb->default_rule_outcome = oc;
    rb->rules_always_test_actor = ata;
    rb->used_by_future_action_activity = ubfaa;
    rb->runs_during_activities = rda;
    return rb;
}
void rb_set_alt_name(rulebook *rb, int aw1, int aw2) {
    rb->alt_word_ref1 = aw1; rb->alt_word_ref2 = aw2;
}
void rb_fragment_by_actions(rulebook *rb, int wn) {
    rb->fragmentation_stem_length = wn;
}
int rb_requires_specific_action(rulebook *rb) {
    if (rb == built_in_rulebooks[CHECK_RB]) return TRUE;
    if (rb == built_in_rulebooks[CARRY_OUT_RB]) return TRUE;
    if (rb == built_in_rulebooks[REPORT_RB]) return TRUE;
    if (rb->fragmentation_stem_length > 0) return TRUE;
    return FALSE;
}
```

The function rb_new is called from 8/creat.

The function rb_new_automatic is called from 9/scene, 11/act and 11/av.

The function rb_set_alt_name is called from 9/scene.

The function rb_fragment_by_actions is called from 11/act.

The function rb_requires_specific_action is called from 12/phud.

§2. **Affected by placements.** Needed to make a useful index.

```
void rb_affected_by_placement(rulebook *rb, parse_node *where) {
    placement_affecting *npl = CREATE(placement_affecting);
    npl->placement_sentence = where;
    npl->next = rb->placement_list;
    rb->placement_list = npl;
}
int rb_no_placements(rulebook *rb) {
    int t = 0;
    placement_affecting *npl = rb->placement_list;
    while (npl) { t++; npl = npl->next; }
    return t;
}
void rb_index_placements(rulebook *rb) {
    placement_affecting *npl = rb->placement_list;
    while (npl) {
        INDEX("      <small><i>NB:</i> ");
        print_text_to_file(npl->placement_sentence->word_ref1, npl->placement_sentence->word_ref2, ifl);
        index_link(lw_array[npl->placement_sentence->word_ref1].lw_source);
        INDEX("</small><br>\n");
        npl = npl->next;
    }
}
```

The function rb_affected_by_placement is called from 12/rps.

§3. **Reading properties of rulebooks.** Those readable from outside the current section.

```
int rb_focus(rulebook *rb) {
    return rb->rulebook_focus;
}
int rb_used_by_future_actions(rulebook *rb) {
    return rb->used_by_future_action_activity;
}
int rb_is_empty(rulebook *rb, scene *context) {
    if (rb == NULL) return TRUE;
    return br_list_is_empty(rb->rule_list, context);
}
int rule_in_rulebook(booked_rule *br, rulebook *rb) {
    if (rb == NULL) return FALSE;
    return br_list_contains(rb->rule_list, br);
}
int rb_runs_during_activities(rulebook *rb) {
    return rb->runs_during_activities;
}
```

The function rb_focus is called from 12/phud.
The function rb_used_by_future_actions is called from 11/av.
The function rb_is_empty is called from 9/tmap and 11/av.
The function rule_in_rulebook is called from 12/rps.
The function rb_runs_during_activities is called from 12/phud.

§4. Rulebook variables.

```
void rb_add_variable(rulebook *rb, parse_node *cnode) {
    specification *spec;
    int nw1, nw2, tw1, tw2, i;
    if (pn_get_node_type(cnode) != PROPERTYCALLED_NT)
        internal_error("rb_add_variable on a node of unknown type");
    [[tw1, tw2 <-- cnode->down]];
    [[nw1, nw2 <-- cnode->down->next]];
    spec = parse_expression(tw1, tw2, TYPE_EXPCON);
    if [[nw1, nw2 == ... and ... : i]] {
        quote_source(1, current_sentence);
        quote_words(2, nw1, nw2);
        handmade_problem(_P_(C12RulebookVariableAnd));
        issue_problem_segment(
            "You wrote %1, which I am reading as a request to make "
            "a new named variable for a rulebook - a value associated "
            "with a rulebook and which has a name. The request seems to "
            "say that the name in question is '%2', but I'd prefer to "
            "avoid 'and' in such names, please.");
        issue_problem_end();
        return;
    }
    if (species_is(spec, DESCRIPTION_SPC)) {
        if ((spec_get_described_kind(spec)) && (number_of_adjectives_applied_to(spec) == 0)) {
            spec = new_generic_CONSTANT_type(kovko(spec_get_described_kind(spec)));
        } else {
            quote_source(1, current_sentence);
            quote_words(2, tw1, tw2);
            handmade_problem(_P_(C12RulebookVariableTooSpecific));
            issue_problem_segment(
                "You wrote %1, which I am reading as a request to make "
                "a new named variable for a rulebook - a value associated "
                "with a rulebook and which has a name. The request seems to "
                "say that the value in question is '%2', but this is too "
                "specific a description. (Instead, a kind of value "
                "(such as 'number') or a kind of object (such as 'room' "
                "or 'thing') should be given. To get a property whose "
                "contents can be any kind of object, use 'object'.)");
            issue_problem_end();
            return;
        }
    }
    if (spec_is_actual_CONSTANT(spec)) {
        LOG("Offending SP: $X", spec);
        quote_source(1, current_sentence);
        quote_words(2, tw1, tw2);
        handmade_problem(_P_(C12RulebookVariableBadKind));
        issue_problem_segment(
            "You wrote %1, but '%2' is not the name of a kind of "
            "value which I know (such as 'number' or 'text').");
        issue_problem_end();
```

```
        return;
    }
    if (is_kova(spec_get_kind_of_value(spec), ANY_VALUE_TY)) {
        quote_source(1, current_sentence);
        quote_words(2, tw1, tw2);
        handmade_problem(_P_(C12RulebookVariableVague));
        issue_problem_segment(
            "You wrote %1, but saying that a variable is a 'value' "
            "does not give me a clear enough idea what it will hold. "
            "You need to say what kind of value: for instance, 'A door "
            "has a number called street address.' is allowed because "
            "'number' is specific about the kind of value.");
        issue_problem_end();
        return;
    }
    stvo_add(rb->owned_by_rb, nw1, nw2, spec_get_kind_of_value(spec), NULL);
}
void rb_make_stvs_accessible(rulebook *rb, stacked_variable_owner *stvo) {
    rb->accessible_from_rb = stvol_add_owner(rb->accessible_from_rb, stvo);
}
void compile_rulebook_var_creators_array(OUTPUT_STREAM) {
    rulebook *rb;
    WRITE("Array rulebook_var_creators -->");
    LOOP_OVER(rb, rulebook) {
        if (stvo_empty(rb->owned_by_rb)) WRITE(" 0");
        else WRITE(" RBSTVC_%d", rb->allocation_id);
    }
    WRITE(" 0;\n");
    LOOP_OVER(rb, rulebook) {
        if (stvo_empty(rb->owned_by_rb) == FALSE)
            stvl_compile_frame_creator(OUT, rb->owned_by_rb,
                " RBSTVC_%d", rb->allocation_id);
    }
}
```

The function rb_add_variable is called from 8/mass.

The function rb_make_stvs_accessible is called from 11/av.

The function compile_rulebook_var_creators_array is invoked by a command in a `.i6t` template file.

§5. Indexing and logging rulebooks.

```
void rb_log_name_only(rulebook *rb) {
    LOG("Rulebook %d ($W)", rb->allocation_id, rb->word_ref1, rb->word_ref2);
}
void log_rulebook(rulebook *rb) {
    rb_log_name_only(rb);
    LOG(": ");
    br_list_log(rb->rule_list);
}
int rb_index(rulebook *rb, char *billing, scene *context,
    action_name *action_context) {
    int before_or_after = FALSE, suppress_outcome = FALSE, t;
    if (rb == NULL) return 0;
    if (billing == NULL) internal_error("No billing for rb index");
    if (billing[0] != 0) {
        if ((action_context) || (br_list_is_empty(rb->rule_list, context)))
            suppress_outcome = TRUE;
    }
    if ((rb == built_in_rulebooks[BEFORE_RB]) ||
        (rb == built_in_rulebooks[AFTER_RB])) before_or_after = TRUE;
    t = br_list_index(rb->rule_list, context, action_context,
        before_or_after, billing, rb);
    rb_outcomes_index(rb, suppress_outcome);
    rb_index_placements(rb);
    return t;
}
void rb_index_action_rules(action_name *an, rulebook *rb, int code, char *desc) {
    int t = 0;
    br_list_suppress_indexed_links();
    if (code >= 0) t += rb_index(built_in_rulebooks[code], desc, NULL, an);
    if (rb) t += rb_index(rb, desc, NULL, NULL);
    br_list_resume_indexed_links();
    if (t > 0) INDEX("<br>");
}
```

The function rb_log_name_only is called from 12/phud.
The function log_rulebook is called from 2/dl.
The function rb_index is called from 9/tmap and 11/av.
The function rb_index_action_rules is called from 11/act.

§**6. Name parsing of rulebooks.** Now we parse the names of rulebooks. The obvious way, using the main excerpt parser, is all very well: but it doesn't contain nearly enough variation for rulebook name clauses used in phrase headers, and a special routine is used for these.

```
rulebook *rb_by_name(int w1, int w2) {
    meaning_list *ml;
    [[w1, w2 == the ... --> w1, w2]];
    ml = SP_excerpt(RULEBOOK_MC, w1, w2);
    if (ml) return RETRIEVE_POINTER_rulebook(em_data(ml_meaning(ml)));
    return NULL;
}

rulebook_match rb_match_from_description(int w1, int w2) {
    int initial_w1 = w1, modifier_words;
    int art = NO_ART, pl = MIDDLE_PLACEMENT;
    rulebook *rb;
    rulebook_match rm;
    if [[w1, w2 == a/an ... --> w1, w2]] art = INDEF_ART;
    else if [[w1, w2 == the ... --> w1, w2]] art = DEF_ART;
    if [[w1, w2 == rule ... --> w1, w2]] {
        [[w1, w2 == for/about/on ... --> w1, w2]];
    } else {
        if [[w1, w2 == first ... --> w1, w2]] pl = FIRST_PLACEMENT;
        else if [[w1, w2 == last ... --> w1, w2]] pl = LAST_PLACEMENT;
    }
    modifier_words = w1 - initial_w1;
    rm.match_length = 0;
    rm.matched_rulebook = NULL;
    rm.article_used = art;
    rm.placement_requested = pl;
    LOOP_OVER(rb, rulebook) {
        int x1 = rb->word_ref1, x2 = rb->word_ref2;
        if ((rb->has_no_parameter) && (w2-w1 != x2-x1)) goto TryAlt;
        if (w2-w1 >= x2-x1) {
            int this_match = x2-x1+1;
            if (compare_word_range(x1, x2, w1, w1+x2-x1) == FALSE) goto TryAlt;
            if (rm.match_length < this_match) {
                rm.match_length = this_match;
                rm.matched_rulebook = rb;
            }
        }
        TryAlt: x1 = rb->alt_word_ref1; x2 = rb->alt_word_ref2;
        if (x1 < 0) continue;
        if ((rb->has_no_parameter) && (w2-w1 != x2-x1)) continue;
        if (w2-w1 >= x2-x1) {
            int this_match = x2-x1+1;
            if (compare_word_range(x1, x2, w1, w1+x2-x1) == FALSE) continue;
            if (rm.match_length < this_match) {
                rm.match_length = this_match;
                rm.matched_rulebook = rb;
            }
        }
    }
```

```
    if (rm.match_length == 0) return rm;
    if (rm.matched_rulebook->fragmentation_stem_length) {
        int w1a = w1 + rm.match_length - 1;
        if (w1a != w2) {
            rm.match_length = rm.matched_rulebook->fragmentation_stem_length;
            w1a = w1 + rm.match_length - 1;
        }
    }
    rm.match_length += modifier_words;
    return rm;
}
```

The function rb_by_name is called from 12/rps.
The function rb_match_from_description is called from 12/phud.

§**7. Rule attachments.** The following routine contains a bit of a surprise: that the act of placing a BR within a given rulebook can change it, by altering the way it acts on its applicability test. This is a device needed to manage the parallel rulebooks for action processing for the main player character and for third parties. Though the code below does not make this apparent, the changes propagate down through the BR to the phrase structure itself. This is necessary because they manifest themselves in the compiled code of the phrase, but it is also unfortunate, because it is possible that the same phrase is used by more than one BR. If it should happen that BRs are created to place the same phrase into two different rulebooks, therefore, and which have different actor-testing settings, the outcome would be confusing. (As unlikely as this seems, it did once happen to a user in beta-testing.)

All work on the sequence of rules in rulebooks is delegated to the sub-section on linked lists of booked rules in the section on Rules.

```
void rb_attach_rule(rulebook *rb, booked_rule *the_new_rule,
    int placing, int side, booked_rule *ref_rule) {
    LOGIF(RULE_ATTACHMENTS, "Attaching booked rule $b at sentence:\n  $T",
        the_new_rule, current_sentence);
    LOGIF(RULE_ATTACHMENTS, "Rulebook before attachment: $K", rb);
    if (ref_rule) {
        LOGIF(RULE_ATTACHMENTS, "With respect to $b\n", ref_rule);
    }
    if (rb == built_in_rulebooks[SETTING_ACTION_VARIABLES_RB]) {
        br_set_never_test_actor(the_new_rule);
    } else {
        if (rb->rules_always_test_actor) {
            LOGIF(RULE_ATTACHMENTS,
                "Setting always test actor for destination rulebook\n");
            br_set_always_test_actor(the_new_rule);
        }
        if (rb->rulebook_focus == PARAMETER_FOCUS){
            LOGIF(RULE_ATTACHMENTS,
                "Setting never test actor for destination rulebook\n");
            br_set_never_test_actor(the_new_rule);
        }
    }
    if (side == INSTEAD_SIDE) {
        LOGIF(RULE_ATTACHMENTS,
            "Copying actor test flags from rule being replaced\n");
```

```
        br_copy_actor_test_flags(the_new_rule, ref_rule);
        LOGIF(RULE_ATTACHMENTS,
            "Copying former rulebook's variable permissions to displaced rule\n");
        br_acquire_stvol(ref_rule, rb->accessible_from_rb);
        if (rb->rulebook_focus == ACTION_FOCUS)
            br_acquire_action_variables(ref_rule);
    }
    br_acquire_stvol(the_new_rule, rb->accessible_from_rb);
    if (rb->rulebook_focus == ACTION_FOCUS)
        br_acquire_action_variables(the_new_rule);
    if (rb->fragmentation_stem_length > 0)
        br_suppress_action_testing(the_new_rule);
    br_list_add(rb->rule_list, the_new_rule, placing, side, ref_rule);
    LOGIF(RULE_ATTACHMENTS, "Rulebook after attachment: $K", rb);
}
void rb_detach_rule(rulebook *rb, booked_rule *the_new_rule) {
    br_list_remove(rb->rule_list, the_new_rule);
}
```

The function rb_attach_rule is called from 12/phud and 12/rps.
The function rb_detach_rule is called from 12/rps.

§**8. Compilation.** We do not actually compile the I6 routines for a rulebook here, but simply act as a proxy. The I6 arrays making the rulebooks available to run-time code are the real outcome of the code in this section.

```
void rb_compile_rule_phrases(rulebook *rb, OUTPUT_STREAM, int *i, int max_i) {
    br_list_judge_ordering(rb->rule_list);
    if (br_list_is_empty_of_i7_rules(rb->rule_list)) return;
    WRITE("\n");
    compile_i6_divider_comment(OUT);
    WRITE("! Rules in rulebook: ");
    print_raw_text_within_i6_literal(OUT, rb->word_ref1, rb->word_ref2);
    WRITE(" (%s)\n", rb->rb_I6_identifier);
    compile_i6_divider_comment(OUT);
    br_list_compile_rule_phrases(rb->rule_list, OUT, i, max_i);
    compile_i6_divider_comment(OUT);
}
void rb_compile_rulebooks_array_array(OUTPUT_STREAM) {
    rulebook *rb;
    WRITE("Array rulebooks_array -->");
    LOOP_OVER(rb, rulebook)
        WRITE(" %s", rb->rb_I6_identifier);
    WRITE(" 0;\n");
}
void rb_compile_rulebooks(OUTPUT_STREAM) {
    rulebook *rb;
    LOOP_OVER(rb, rulebook) {
        LOGIF(COMPILE_RULEBOOKS, "Compiling rulebook: $W = %s\n",
            rb->word_ref1, rb->word_ref2, rb->rb_I6_identifier);
        br_list_compile(rb->rule_list, OUT, rb->rb_I6_identifier);
    }
```

```
}
void rb_compile_RulebookNames_array(OUTPUT_STREAM) {
    rulebook *rb;
    WRITE("Array RulebookNames -->\n"); INDENT;
    LOOP_OVER(rb, rulebook) {
        WRITE("\"");
        print_raw_text_within_i6_literal(OUT, rb->word_ref1, rb->word_ref2);
        WRITE(" rulebook\" ! %d\n", rb->allocation_id);
    }
    OUTDENT; WRITE(";\n\n");
}
```

The function rb_compile_rule_phrases is called from 12/cs.
The function rb_compile_rulebooks_array_array is called from 12/cs.
The function rb_compile_rulebooks is called from 12/cs.
The function rb_compile_RulebookNames_array is called from 12/cs.

§9. Parsing rulebook properties.

```
void rb_parse_properties(rulebook *rb, int w1, int w2) {
    if [[w1, w2 == outcome/outcomes ... --> w1, w2]] {
        rb_parse_outcomes(rb, w1, w2);
        return;
    }
    if [[w1, w2 == default ... --> w1, w2]] {
        int o = parse_outcome(w1, w2);
        if (o == UNRECOGNISED_OUTCOME) {
            sentence_problem(_P_(C12BadDefaultOutcome),
                "the default outcome given for the rulebook isn't what I expected",
                "which would be one of 'default success', 'default failure' or "
                "'default no outcome'.");
            return;
        }
        if (rb->default_outcome_declared) {
            sentence_problem(_P_(C12DefaultOutcomeTwice),
                "the default outcome for this rulebook has already been declared",
                "and this is something which can only be done once.");
            return;
        }
        rb->default_outcome_declared = TRUE;
        rb->default_rule_outcome = o;
        return;
    }
    LOG("RBPP: <$W>\n", w1, w2);
    sentence_problem(_P_(C12NonOutcomeProperty),
        "the only properties of a rulebook are its outcomes",
        "for the time being at least.");
}
int parse_outcome(int b1, int b2) {
    if [[b1, b2 == success]] return SUCCESS_OUTCOME;
    if [[b1, b2 == failure]] return FAILURE_OUTCOME;
    if [[b1, b2 == no outcome]] return NO_OUTCOME;
    return UNRECOGNISED_OUTCOME;
```

```
}
void rb_parse_outcomes(rulebook *rb, int w1, int w2) {
    int i, koo = SUCCESS_OUTCOME;
    named_rulebook_outcome *rbno;
    if (rb->no_outcomes >= MAX_OUTCOMES_PER_RULEBOOK) {
        sentence_problem(_P_(C12TooManyRulebookOutcomes),
            "this rulebook already has the maximum permitted number of outcomes",
            "which is a limit which can't be raised.");
        return;
    }
    if (is_list_divided(w1, w2, LOOK_FOR_AND + LOOK_FOR_OR)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        rb_parse_outcomes(rb, lw1, lw2);
        rb_parse_outcomes(rb, rw1, rw2);
        return;
    }
    if [[word w2 == CLOSEBRACKET]] {
        i = w2-1;
        while ((i>=w1) && ([[word i == OPENBRACKET]] == FALSE)) i--;
        if (i>=w1) {
            int b1 = i+1, b2 = w2-1;
            w2 = i-1;
            if [[word b2 == default]] {
                b2--;
                if [[word b2 == the]] b2--;
                if ((b2>w1) && [[word b2 == DASH]]) {
                    b2--;
                    if (rb->default_named_outcome >= 0) {
                        sentence_problem(_P_(C12DefaultNamedOutcomeTwice),
                            "at most one of the named outcomes from a rulebook "
                            "can be the default",
                            "and here we seem to have two.");
                        return;
                    }
                    rb->default_named_outcome = rb->no_outcomes;
                    if (rb->default_outcome_declared) {
                        sentence_problem(_P_(C12DefaultOutcomeAlready),
                            "the default outcome for this rulebook has already "
                            "been declared",
                            "and this is something which can only be done once.");
                        return;
                    }
                    rb->default_outcome_declared = TRUE;
                } else b2 = w2-1;
            }
            koo = parse_outcome(b1, b2);
            if (koo == UNRECOGNISED_OUTCOME) {
                LOG("BC: <$W>\n", b1, b2);
                sentence_problem(_P_(C12BadOutcomeClarification),
                    "the bracketed clarification isn't what I expected",
                    "which would be one of '(success)', '(failure)' or "
                    "'(no outcome)'.");
                return;
```

```
            }
        }
    }
    rbno = rbno_by_name(w1, w2);
    for (i=0; i<rb->no_outcomes; i++)
        if (rb->named_outcomes[i].outcome_name == rbno) {
            sentence_problem(_P_(C12DuplicateOutcome),
                "this duplicates a previous assignment of the same outcome",
                "and to the same rulebook.");
            return;
        }
    rb->named_outcomes[rb->no_outcomes].outcome_name = rbno;
    rb->named_outcomes[rb->no_outcomes].kind_of_outcome = koo;
    rb->no_outcomes++;
}
named_rulebook_outcome *rbno_by_name(int w1, int w2) {
    named_rulebook_outcome *rbno;
    meaning_list *ml = SP_excerpt(MISCELLANEOUS_MC, w1, w2);
    if ((ml != NULL) && (em_get_secondary_code(ml_meaning(ml)) == RULE_OUTCOME_SMC)) {
        return RETRIEVE_POINTER_named_rulebook_outcome(em_data(ml_meaning(ml)));
    }
    rbno = CREATE(named_rulebook_outcome);
    register_excerpt_meaning(
        MISCELLANEOUS_MC, RULE_OUTCOME_SMC, w1, w2,
            STORE_POINTER_named_rulebook_outcome(rbno));
    rbno->word_ref1 = w1;
    rbno->word_ref2 = w2;
    return rbno;
}
void rb_compile_default_outcome(rulebook *rb, OUTPUT_STREAM) {
    if (rb->default_named_outcome >= 0) {
        rulebook_outcome *rbo = &(rb->named_outcomes[rb->default_named_outcome]);
        switch(rbo->kind_of_outcome) {
            case SUCCESS_OUTCOME:
                WRITE("RulebookSucceeds(true, RBNO_%d); rtrue;\n",
                    rbo->outcome_name->allocation_id);
                break;
            case FAILURE_OUTCOME:
                WRITE("RulebookFails(true, RBNO_%d); rtrue;\n",
                    rbo->outcome_name->allocation_id);
                break;
        }
    } else {
        switch (rb->default_rule_outcome) {
            case FAILURE_OUTCOME:
                code_indent(OUT, 2);
                WRITE("RulebookFails(); rtrue;\n");
                break;
            case SUCCESS_OUTCOME:
                code_indent(OUT, 2);
                WRITE("RulebookSucceeds(); rtrue;\n");
                break;
        }
```

```
    }
}
rulebook_outcome *rbo_from_context(named_rulebook_outcome *rbno) {
    int i;
    phrase *ph = phrase_being_compiled;
    rulebook *rb;
    if (ph == NULL) return NULL;
    LOOP_OVER(rb, rulebook) {
        for (i=0; i<rb->no_outcomes; i++) {
            if (rb->named_outcomes[i].outcome_name == rbno) {
                if (br_list_contains_ph(rb->rule_list, ph))
                    return &(rb->named_outcomes[i]);
            }
        }
    }
    return NULL;
}
int rb_allow_outcome(named_rulebook_outcome *rbno) {
    if (rbo_from_context(rbno)) return TRUE;
    if (outcome_restrictions_waived()) return TRUE;
    return FALSE;
}
void rb_compile_outcome(OUTPUT_STREAM, named_rulebook_outcome *rbno) {
    rulebook_outcome *rbo = rbo_from_context(rbno);
    if (rbo == NULL) {
        rulebook *rb; int i;
        if (outcome_restrictions_waived() == FALSE)
            internal_error("tried to compile rbno outside its context");
        LOOP_OVER(rb, rulebook)
            for (i=0; i<rb->no_outcomes; i++)
                if (rb->named_outcomes[i].outcome_name == rbno) {
                    rbo = &(rb->named_outcomes[i]);
                    break;
                }
        if (rbo == NULL)
            internal_error("rbno with no rb context");
    }
    switch(rbo->kind_of_outcome) {
        case SUCCESS_OUTCOME:
            WRITE("RulebookSucceeds(true, RBNO_%d); rtrue;\n",
                rbno->allocation_id);
            break;
        case FAILURE_OUTCOME:
            WRITE("RulebookFails(true, RBNO_%d); rtrue;\n",
                rbno->allocation_id);
            break;
        case NO_OUTCOME:
            WRITE("rfalse;\n");
            break;
        default:
            internal_error("bad RBO outcome kind");
    }
}
```

```
void rb_outcomes_index(rulebook *rb, int suppress_outcome) {
    int i;
    for (i=0; i<rb->no_outcomes; i++) {
        named_rulebook_outcome *rbno = rb->named_outcomes[i].outcome_name;
        open_html_paragraph(ifl, 2, "hanging");
        INDEX("<i>outcome</i>  ");
        if (rb->default_named_outcome == i) INDEX("<b>");
        print_raw_text_to_file(rbno->word_ref1, rbno->word_ref2, ifl);
        if (rb->default_named_outcome == i) INDEX("</b> (default)");
        INDEX(" - <i>");
        switch(rb->named_outcomes[i].kind_of_outcome) {
            case SUCCESS_OUTCOME:
                INDEX("a success");
                break;
            case FAILURE_OUTCOME:
                INDEX("a failure");
                break;
            case NO_OUTCOME:
                INDEX("no outcome");
                break;
        }
        INDEX("</i>");
        INDEX("</p>\n");
    }
    if ((rb->default_named_outcome == -1) &&
        (rb->default_rule_outcome != NO_OUTCOME) &&
        (suppress_outcome == FALSE)) {
        open_html_paragraph(ifl, 2, "hanging");
        INDEX("<i>default outcome is</i> ");
        switch(rb->default_rule_outcome) {
            case SUCCESS_OUTCOME:
                INDEX("success");
                break;
            case FAILURE_OUTCOME:
                INDEX("failure");
                break;
        }
        INDEX("</p>");
    }
}
void compile_RulebookOutcomePrintingRule_routine(OUTPUT_STREAM) {
    named_rulebook_outcome *rbno;
    LOOP_OVER(rbno, named_rulebook_outcome) {
        WRITE("Constant RBNO_%d = \"", rbno->allocation_id);
        print_raw_text_to_file(rbno->word_ref1, rbno->word_ref2, OUT);
        WRITE("\";\n");
    }
    WRITE("[ RulebookOutcomePrintingRule rbno;\n"); INDENT;
    WRITE("print (string) rbno; rfalse;\n");
    OUTDENT; WRITE("];\n");
}
```

The function rb_parse_properties is called from 8/mass.
The function rb_compile_default_outcome is called from 12/phrcd.
The function rb_allow_outcome is called from 7/tc.
The function rb_compile_outcome is called from 7/cmsp.
The function compile_RulebookOutcomePrintingRule_routine is invoked by a command in a `.i6t` template file.

§**10. Rules index.** The Rules page of the index is essentially a trawl through the more popular rulebooks, showing their contents in logical order.

```
void index_page_Rules(void) {
    int pass;
    for (pass = 1; pass <= 2; pass++) {
        int segment, f = FALSE;
        extension_file *ef;
        for (segment = 1; segment <= 8; segment++)
            index_rules_segment(segment, pass, FALSE, NULL);
        if (noteworthy_rulebooks(NULL) > 0) {
            f = TRUE;
            index_rules_segment(9, pass, TRUE, NULL);
        }
        LOOP_OVER(ef, extension_file)
            if (ef != standard_rules_extension)
                if (noteworthy_rulebooks(ef) > 0) {
                    f = TRUE;
                    index_rules_segment(10 + ef->allocation_id, pass, TRUE, ef);
                }
        if ((pass == 1) && (f == TRUE))
            INDEX("<p>Rulebooks and activities other than those in the Standard Rules "
                "are indexed according to their origin.</p>");
    }
}
void dlink(int n) {
    index_detail_link("R", n, TRUE);
}
int noteworthy_rulebooks(extension_file *ef) {
    int nb = 0;
    activity *av;
    rulebook *rb;
    LOOP_OVER(rb, rulebook) {
        source_location sl = lw_array[rb->word_ref1].lw_source;
        if (rb->automatically_generated) continue;
        if (((ef == NULL) && (sl.file_of_origin == NULL)) ||
            (sf_get_extension_corresponding(sl.file_of_origin) == ef)) nb++;
    }
    LOOP_OVER(av, activity) {
        source_location sl = lw_array[av->word_ref1].lw_source;
        if (((ef == NULL) && (sl.file_of_origin == NULL)) ||
            (sf_get_extension_corresponding(sl.file_of_origin) == ef)) nb++;
    }
    return nb;
}
```

```
void index_rules_segment(int segment, int pass, int area_based, extension_file *ef) {
    if (pass == 2)
        open_index_file("R.html", "<Rules", segment,
            "A set of rulebooks and activities in detail.");
    if (area_based) {
        INDEX("<p><b>New rulebooks and activities ");
        if (ef) { INDEX("from "); eid_write_to_HTML_file(ifl, ef_get_eid(ef), FALSE); }
        else INDEX("in the source text");
        INDEX("</b>");
        if (pass == 1) dlink(segment); INDEX("</p>");
    } else {
    switch(segment) {
        case 1:
            INDEX("<p><b>Procedural rules</b>"); if (pass == 1) dlink(segment); INDEX("</p>");
            INDEX("<p>Procedural rules are super-powerful, controlling how all other "
                "rulebooks are read. Like any drastic solution, a procedural rule "
                "should only be used when all else fails.</p>");
            break;
        case 2:
            INDEX("<p><b>The top level</b>"); if (pass == 1) dlink(segment); INDEX("</p>");
            INDEX("<p>An Inform story file spends its whole time working through "
                "these three master rulebooks. They can be altered, just as all "
                "rulebooks can, but it's generally better to leave them alone.</p>");
            break;
        case 3:
            INDEX("<p><b>Rules added to the sequence of play</b>"); if (pass == 1) dlink(segment); INDEX("</p>");
            INDEX("<p>These rulebooks are the best places to put rules timed to happen "
                "at the start, at the end, or once each turn. (Each is run through at "
                "a carefully chosen moment in the relevant top-level rulebook.) It is "
                "also possible to have rules take effect at specific times of day "
                "or when certain events happen. Those belong to no rulebook, but are "
                "indexed here for convenience. Rules taking place when scenes begin and "
                "end can be found in the Scenes index.</p>");
            break;
        case 4:
            INDEX("<p><b>How commands are understood</b>"); if (pass == 1) dlink(segment); INDEX("</p>");
            INDEX("<p>'Understanding' here means turning a typed command, like GET FISH, "
                "into one or more actions, like taking the red herring. This is all handled "
                "by a single large rule (the parse command rule), but that rule makes use "
                "of the following activities and rulebooks in its work.</p>");
            break;
        case 5:
            INDEX("<p><b>How actions are processed</b>"); if (pass == 1) dlink(segment); INDEX("</p>");
            INDEX("<p>These form the machinery for dealing with actions, and are "
                "called on at least once every turn. They seldom need to be changed. "
                "To change the effect of actions - say, 'taking or dropping something' - "
                "write an Instead rule, or something similar: see the Actions index for "
                "these. (The rulebooks here are the ones that consult those rules and "
                "deal with the results.)</p>");
            break;
        case 6:
            INDEX("<p><b>How accessibility is judged</b>"); if (pass == 1) dlink(segment); INDEX("</p>");
            INDEX("<p>These rulebooks are used when deciding who can reach what, and "
```

```
            "who can see what.</p>");
        break;
    case 7:
        INDEX("<p><b>Light and darkness</b>"); if (pass == 1) dlink(segment); INDEX("</p>");
        INDEX("<p>These activities control how we describe darkness.</p>");
        break;
    case 8:
        INDEX("<p><b>How things are described</b>"); if (pass == 1) dlink(segment); INDEX("</p>");
        INDEX("<p>These activities control what is printed when naming rooms or "
            "things, and their descriptions.</p>");
        break;
}
}
if (pass == 1) return;
if (area_based) {
    activity *av;
    rulebook *rb;
    LOOP_OVER(rb, rulebook) {
        source_location sl = lw_array[rb->word_ref1].lw_source;
        if (rb->automatically_generated) continue;
        if (((ef == NULL) && (sl.file_of_origin == NULL)) ||
            (sf_get_extension_corresponding(sl.file_of_origin) == ef))
            index_rules_box(NULL, rb->word_ref1, rb->word_ref2, NULL, rb, NULL, NULL, 1);
    }
    LOOP_OVER(av, activity) {
        source_location sl = lw_array[av->word_ref1].lw_source;
        if (((ef == NULL) && (sl.file_of_origin == NULL)) ||
            (sf_get_extension_corresponding(sl.file_of_origin) == ef))
            av_index(av, 1);
    }
} else {
    switch(segment) {
        case 1:
            index_rules_box("Procedural", -1, -1, "rules_proc",
                built_in_rulebooks[PROCEDURAL_RB], NULL, NULL, 1);
            break;
        case 2:
            index_rules_box("Startup rules", -1, -1, NULL,
                built_in_rulebooks[STARTUP_RB], NULL, NULL, 1);
            av_index_by_number(STARTING_VIRTUAL_MACHINE_ACT, 2);
            av_index_by_number(PRINTING_BANNER_TEXT_ACT, 2);
            index_rules_box("Turn sequence rules", -1, -1, NULL,
                built_in_rulebooks[TURN_SEQUENCE_RB], NULL, NULL, 1);
            av_index_by_number(CONSTRUCTING_STATUS_LINE_ACT, 2);
            index_rules_box("Shutdown rules", -1, -1, NULL,
                built_in_rulebooks[SHUTDOWN_RB], NULL, NULL, 1);
            av_index_by_number(AMUSING_A_VICTORIOUS_PLAYER_ACT, 2);
            av_index_by_number(PRINTING_PLAYERS_OBITUARY_ACT, 2);
            av_index_by_number(DEALING_WITH_FINAL_QUESTION_ACT, 2);
            break;
        case 3:
            index_rules_box("When play begins", -1, -1, "rules_wpb",
                built_in_rulebooks[WHEN_PLAY_BEGINS_RB], NULL, NULL, 1);
```

```
    index_rules_box("Every turn", -1, -1, "rules_et",
        built_in_rulebooks[EVERY_TURN_RB], NULL, NULL, 1);
    index_rules_box("When play ends", -1, -1, "rules_wpe",
        built_in_rulebooks[WHEN_PLAY_ENDS_RB], NULL, NULL, 1);
    tiph_index_rules();
    break;
case 4:
    index_rules_box("Does the player mean", -1, -1, "rules_dtpm",
        built_in_rulebooks[DOES_THE_PLAYER_MEAN_RB], NULL, NULL, 1);
    av_index_by_number(READING_A_COMMAND_ACT, 1);
    av_index_by_number(DECIDING_SCOPE_ACT, 1);
    av_index_by_number(DECIDING_CONCEALED_POSSESS_ACT, 1);
    av_index_by_number(DECIDING_WHETHER_ALL_INC_ACT, 1);
    av_index_by_number(CLARIFYING_PARSERS_CHOICE_ACT, 1);
    av_index_by_number(ASKING_WHICH_DO_YOU_MEAN_ACT, 1);
    av_index_by_number(PRINTING_A_PARSER_ERROR_ACT, 1);
    av_index_by_number(SUPPLYING_A_MISSING_NOUN_ACT, 1);
    av_index_by_number(SUPPLYING_A_MISSING_SECOND_ACT, 1);
    av_index_by_number(IMPLICITLY_TAKING_ACT, 1);
    break;
case 5:
    index_rules_box("Action-processing rules", -1, -1, NULL,
        built_in_rulebooks[ACTION_PROCESSING_RB], NULL, NULL, 1);
    index_rules_box("Specific action-processing rules", -1, -1, NULL,
        built_in_rulebooks[SPECIFIC_ACTION_PROCESSING_RB], NULL, NULL, 2);
    index_rules_box("Player's action awareness rules", -1, -1, NULL,
        built_in_rulebooks[PLAYERS_ACTION_AWARENESS_RB], NULL, NULL, 3);
    break;
case 6:
    index_rules_box("Reaching inside", -1, -1, "rules_ri",
        built_in_rulebooks[REACHING_INSIDE_RB], NULL, NULL, 1);
    index_rules_box("Reaching outside", -1, -1, "rules_ri",
        built_in_rulebooks[REACHING_OUTSIDE_RB], NULL, NULL, 1);
    index_rules_box("Visibility", -1, -1, "visibility",
        built_in_rulebooks[VISIBILITY_RB], NULL, NULL, 1);
    break;
case 7:
    av_index_by_number(PRINTING_NAME_OF_DARK_ROOM_ACT, 1);
    av_index_by_number(PRINTING_DESC_OF_DARK_ROOM_ACT, 1);
    av_index_by_number(PRINTING_NEWS_OF_DARKNESS_ACT, 1);
    av_index_by_number(PRINTING_NEWS_OF_LIGHT_ACT, 1);
    av_index_by_number(REFUSAL_TO_ACT_IN_DARK_ACT, 1);
    break;
case 8:
    av_index_by_number(PRINTING_THE_NAME_ACT, 1);
    av_index_by_number(PRINTING_THE_PLURAL_NAME_ACT, 1);
    av_index_by_number(PRINTING_A_NUMBER_OF_ACT, 1);
    av_index_by_number(PRINTING_ROOM_DESC_DETAILS_ACT, 1);
    av_index_by_number(LISTING_CONTENTS_ACT, 1);
    av_index_by_number(GROUPING_TOGETHER_ACT, 1);
    av_index_by_number(WRITING_A_PARAGRAPH_ABOUT_ACT, 1);
    av_index_by_number(LISTING_NONDESCRIPT_ITEMS_ACT, 1);
    av_index_by_number(PRINTING_LOCALE_DESCRIPTION_ACT, 1);
```

```
                av_index_by_number(CHOOSING_NOTABLE_LOCALE_OBJ_ACT, 1);
                av_index_by_number(PRINTING_LOCALE_PARAGRAPH_ACT, 1);
                break;
        }
    }
}
void index_action_rulebooks(void) {
    INDEX("<p><hr><p><b>Rules governing actions</b><p>");
    index_rules_box("Persuasion", -1, -1, "rules_per", built_in_rulebooks[PERSUASION_RB], NULL, NULL,
1);
    index_rules_box("Unsuccessful attempt by", -1, -1, "rules_fail", built_in_rulebooks[UNSUCCESSFUL_ATTEMPT_BY_RB]
NULL, NULL, 1);
    index_rules_box("Before", -1, -1, "rules_before", built_in_rulebooks[BEFORE_RB], NULL, NULL, 1);
    index_rules_box("Instead", -1, -1, "rules_instead", built_in_rulebooks[INSTEAD_RB], NULL, NULL, 1);
    index_rules_box("Check", -1, -1, NULL, NULL, NULL,
        "Check rules are tied to specific actions, and there are too many "
        "to index here. For instance, the check taking rules can only ever "
        "affect the taking action, so they are indexed on the detailed index "
        "page for taking.", 1);
    index_rules_box("Carry out", -1, -1, NULL, NULL, NULL,
        "Carry out rules are tied to specific actions, and there are too many "
        "to index here.", 1);
    index_rules_box("After", -1, -1, "rules_after", built_in_rulebooks[AFTER_RB], NULL, NULL, 1);
    index_rules_box("Report", -1, -1, NULL, NULL, NULL,
        "Report rules are tied to specific actions, and there are too many "
        "to index here.", 1);
}
void index_scene_rulebooks(void) {
    INDEX("<p><b>The scene-changing machinery</b><p>");
    index_rules_box("Scene changing", -1, -1, NULL, built_in_rulebooks[SCENE_CHANGING_RB], NULL, NULL,
1);
}
void index_rules_box(char *name, int w1, int w2, char *doc_link,
    rulebook *rb, activity *av, char *text, int indent) {
    char *col;
    if (av) col = "e8e0c0"; else col = "e0e0e0";
    open_html_paragraph(ifl, indent, "");
    open_coloured_box(ifl, col);
    begin_html_table(ifl, NULL, TRUE, 0, 4, 0, 0, 0);
    first_html_column(ifl, 0);
    char textual_name[600];
    if (name) strcpy(textual_name, name);
    else if (w1 >= 0) print_raw_text_to_string_truncated(w1, w2, textual_name, 500);
    else strcpy(textual_name, "nameless");
    open_html_paragraph(ifl, 1, "tight");
    INDEX("<b>%s</b>", textual_name);
    if (doc_link) index_doc_link(doc_link);
    if (av) INDEX(" activity"); else INDEX(" rulebook");
    INDEX("</p>");
    next_html_column_right_justified(ifl, 0);
    textual_name[0] = tolower(textual_name[0]);
```

```
    open_html_paragraph(ifl, 1, "tight");
    if (av) {
        char skeleton[600];
        sprintf(skeleton, "Before %s:", textual_name);
        write_javascript_paste(ifl, -1, -1, skeleton);
        INDEX(" <i>b</i> ");
        sprintf(skeleton, "Rule for %s:", textual_name);
        write_javascript_paste(ifl, -1, -1, skeleton);
        INDEX(" <i>f</i> ");
        sprintf(skeleton, "After %s:", textual_name);
        write_javascript_paste(ifl, -1, -1, skeleton);
        INDEX(" <i>a</i>");
    } else {
        write_javascript_paste(ifl, -1, -1, textual_name);
        INDEX(" <i>name</i>");
    }
    INDEX("</p>");
    end_html_row(ifl);
    end_html_table(ifl);
    if ((rb) && (rb_is_empty(rb, NULL))) text = "There are no rules in this rulebook.";
    if (text) {
        open_html_paragraph(ifl, 2, "tight");
        INDEX("%s</p>", text);
    } else {
        if (rb) rb_index(rb, "", NULL, NULL);
        if (av) av_index_details(av);
    }
    close_coloured_box(ifl, col);
    INDEX("</p>");
}
```

The function index_page_Rules is invoked by a command in a `.i6t` template file.

The function index_action_rulebooks is called from 11/ina.

The function index_scene_rulebooks is called from 9/tmap.

The function index_rules_box is called from 11/av.

Rule Placement Sentences 12/rps

Purpose

To parse and act upon explicit sentences like “The fire alarm rule is listed after the burglar alarm rule in the House Security rules.”

```
void place_in_rulebook(parse_node *p1, parse_node *p2, int sense) {
    int side = NO_SIDE, new_rule_placement, pw1, pw2, rel1 = -1, rel2 = -1;
    booked_rule *current_rule_booking = NULL,
        *new_rule_booking = NULL,
        *relative_to_which = NULL;
    rulebook *the_rulebook = NULL;
    LOGIF(RULE_ATTACHMENTS, "Placement sentence (%d):\np1=$T\np2=$T\n",
        sense, p1, p2);
    [[pw1, pw2 <-- p2]];
    if [[pw1, pw2 == in ... --> pw1, pw2]] {
        new_rule_placement = MIDDLE_PLACEMENT; side = IN_SIDE;
    }
    else if [[pw1, pw2 == first in ... --> pw1, pw2]] {
        new_rule_placement = FIRST_PLACEMENT; side = IN_SIDE;
    }
    else if [[pw1, pw2 == last in ... --> pw1, pw2]] {
        new_rule_placement = LAST_PLACEMENT; side = IN_SIDE;
    }
    else if [[pw1, pw2 == instead of ... --> pw1, pw2]] {
        new_rule_placement = MIDDLE_PLACEMENT; side = INSTEAD_SIDE;
    }
    else if [[pw1, pw2 == before ... --> pw1, pw2]] {
        new_rule_placement = MIDDLE_PLACEMENT; side = BEFORE_SIDE;
    }
    else if [[pw1, pw2 == after ... --> pw1, pw2]] {
        new_rule_placement = MIDDLE_PLACEMENT; side = AFTER_SIDE;
    }
    if (side == NO_SIDE) {
        quote_source(1, current_sentence);
        handmade_problem(_P_(C12ImproperRulePlacement));
        issue_problem_segment(
            "In %1, you used the special verb 'to be listed' - which specifies "
            "how rules are listed in rulebooks - in a way I didn't recognise. "
            "The usual form is: 'The summer breeze rule is listed in the "
            "meadow noises rulebook'.");
        issue_problem_end();
        return;
    }
    if ((sense == FALSE) &&
        ((new_rule_placement != MIDDLE_PLACEMENT) || (side != IN_SIDE))) {
        quote_source(1, current_sentence);
        handmade_problem(_P_(C12BadRulePlacementNegation));
        issue_problem_segment(
            "In %1, you used the special verb 'to be listed' - which specifies "
```

```
        "how rules are listed in rulebooks - in a way too complicated to "
        "be accompanied by 'not', so that the result was too vague. "
        "The usual form is: 'The summer breeze rule is not listed in the "
        "meadow noises rulebook'.");
    issue_problem_end();
    return;
}
if (side != IN_SIDE) {
    int i;
    for (i=pw2-1; i>pw1; i--)
        if [[word i == in]] {
            rel1 = pw1; rel2 = i-1; pw1 = i+1; break;
        }
    if (rel1 < 0) {
        quote_source(1, current_sentence);
        handmade_problem(_P_(C12UnspecifiedRulebookPlacement));
        issue_problem_segment(
            "In %1, you didn't specify in which rulebook the rule was to "
            "be listed, only which existing rule it should go before or "
            "after.");
        issue_problem_end();
        return;
    }
}
current_rule_booking = br_by_name(p1->word_ref1, p1->word_ref2);
if (current_rule_booking == NULL) {
    quote_source(1, current_sentence);
    quote_words(2, p1->word_ref1, p1->word_ref2);
    handmade_problem(_P_(C12NoSuchRulePlacement));
    issue_problem_segment(
        "In %1, you gave '%2' where a rule was required.");
    issue_problem_end();
    return;
}
if ((sense == FALSE) && [[pw1, pw2 == any rulebook]]) {
    LOOP_OVER(the_rulebook, rulebook)
        rb_detach_rule(the_rulebook, current_rule_booking);
    return;
}
the_rulebook = rb_by_name(pw1, pw2);
if (the_rulebook == NULL) {
    quote_source(1, current_sentence);
    quote_words(2, pw1, pw2);
    handmade_problem(_P_(C12NoSuchRulebookPlacement));
    issue_problem_segment(
        "In %1, you gave '%2' where a rulebook was required.");
    issue_problem_end();
    return;
}
if (sense == FALSE) {
    rb_affected_by_placement(the_rulebook, current_sentence);
    rb_detach_rule(the_rulebook, current_rule_booking);
```

```
        return;
    }
    new_rule_booking = br_new(NULL);
    br_copy(new_rule_booking, current_rule_booking);
    if (rel1 >= 0) {
        relative_to_which = br_by_name(rel1, rel2);
        if (relative_to_which == NULL) {
            quote_source(1, current_sentence);
            quote_words(2, rel1, rel2);
            handmade_problem(_P_(C12NoSuchRulePlacement2));
            issue_problem_segment(
                "In %1, you gave '%2' where a rule was required.");
            issue_problem_end();
            return;
        }
        LOGIF(RULE_ATTACHMENTS, "Relative to which = <$W> = BR%d\n",
            rel1, rel2,
            relative_to_which->allocation_id);
    }
    if ((relative_to_which) &&
        (rule_in_rulebook(relative_to_which, the_rulebook) == FALSE)) {
        quote_source(1, current_sentence);
        quote_words(2, pw1, pw2);
        quote_words(3, rel1, rel2);
        handmade_problem(_P_(C12PlaceWithMissingRule));
        issue_problem_segment(
            "In %1, you talk about the position of the rule '%3' "
            "in the rulebook '%2', but in fact that rule isn't in this "
            "rulebook, so the placing instruction makes no sense.");
        issue_problem_end();
        return;
    }
    if (relative_to_which)
        rb_affected_by_placement(the_rulebook, current_sentence);
    rb_attach_rule(the_rulebook, new_rule_booking, new_rule_placement,
        side, relative_to_which);
}
```

The function place_in_rulebook is called from 12/cs.

13 Grammar

13/igr: *Introduction to Grammar.w* An exposition of the data structures and basic method used to deal with the command-parsing grammar implied by Understand sentences in the source text.

13/tfg: *Traverse for Grammar.w* To create and manipulate grammar, primarily by parsing and acting upon Understand... sentences in the source text.

13/gv: *Grammar Verbs.w* A grammar verb is not literally a verb, as the name is a hangover from the way I6 defines grammar. It might be said to be a necklace onto which we thread the sea-shells of grammar lines. Each grammar verb has its own purpose: to match various possibilities (one for each grammar line) against text aimed at a particular result. For instance, all run time commands beginning with TAKE are parsed with a single grammar verb. If we create a new grammar token, "[polite remark]", for use in other grammar, then that too will have its own "grammar verb". If we define the word "eleventy-one" as meaning the number 111, it will be added to a grammar verb attached to the data type `NUMBER` which parses eccentric names for number values. And so on. Probably a better name for this structure would be simply "grammar", but that might be confusing in other ways, and anyway the ship has sailed.

13/gl: *Grammar Lines.w* A grammar line is a list of tokens to specify a textual pattern. For example, the NI source for a grammar line might be `"take [something] out"`, which is a sequence of three tokens.

13/gty: *Grammar Types.w* Some grammar text specifies one or more values, and we need to keep track of their data type(s). Here we manage the data structure doing this.

13/gtok: *Grammar Tokens.w* To handle grammar at the level of individual tokens. I7 grammar tokens correspond in a 1-to-1 way with I6 tokens: here we determine the I7 type a token represents (if any) and compile it to its I6 grammar token equivalent as needed.

13/nft: *Noun Filter Tokens.w* Filters are used to require nouns to have specific kinds or attributes, or to have specific scoping rules: they correspond to Inform 6's `noun=Routine` and `scope=Routine` tokens. Though these are quite different concepts in I6, their common handling seems natural in I7.

13/gprv: *Tokens Parsing Values.w* In the argot of Inform 6, GPR stands for General Parsing Routine, and I7 makes heavy use of GPR tokens to achieve its ends. This section is where the necessary I6 routines are compiled.

13/gpr: *General Parsing Routines.w* To compile I6 general parsing routines (GPRs) and/or `parse_name` properties as required by the I7 grammar.

13/test: *Test Scripts.w* A rudimentary but useful testing system built in to IF produced by Inform, allowing short sequences of commands to be concisely noted in the source text and tried out in the Inform application using the TEST command.

Introduction to Grammar 13/igr

Purpose

An exposition of the data structures and basic method used to deal with the command-parsing grammar implied by Understand sentences in the source text.

§**1.** This is grammar in the sense of the parsing structures used at run-time, and it occupies a chapter of its own in the source code since it is to some extent detached from the rest of NI: what we create in this chapter is almost an independent compiler in its own right, but of a much simpler language. Although we use many higher-level features of NI in the process, none use this.

Grammar is organised in a three-level hierarchy:

(a) A grammar verb (GV) is a small independent grammar of alternative formulations for some concept: for instance, the possible commands beginning TAKE, or the possible verbal forms of numbers. Each GV is a list of GLs, and an individual GL must belong to exactly one GV. There are five different types of GV, differentiated mostly by the purpose to which the GV is put:
   1. `GV_IS_COMMAND`. An imperative verbal command at run-time.
   2. `GV_IS_TOKEN`. A square-bracketed token in other grammar.
   3. `GV_IS_OBJECT`. A noun phrase at run time: a name for an object.
   4. `GV_IS_VALUE`. A noun phrase at run time: a name for a value.
   5. `GV_IS_CONSULT`. A pattern to match in part of a command (such as "consult").
   6. `GV_IS_PROPERTY_NAME`. A noun phrase at run time: a name for one possibility for an either/or property, say "open" or "fixed in place".

(b) A grammar line (GL) is a single possibility within a GV: for example, the line matching `"take [something]"` in the GV for the TAKE command. Each GL is a list of tokens, and an individual token must belong to exactly one GL.

(c) A grammar token (GTOK) is a single particle of a GL: for example, `'take'` and `something` are tokens.

The picture is not quite so hierarchical as it looks, though, because a GV naming a token can be used as a token inside other GVs. We need to be careful that this does not lead to infinite regress: see below.

Much of what we do with grammar involves recursing down this hierarchy, in some cases allowing results to percolate back upwards. What happens takes place in four chronological phases. (This division into phases is convenient because Inform 6 requires that all general parsing routines and noun filter routines already exist when a `Verb` directive is reached which uses them.)

§**2. Phase I: Slash Grammar.** Slashing is the process of dealing with slashes `/` used in grammar to indicate alternatives.

§**3. Phase II: Determining Grammar.** We check that the grammar is well-founded and find the types of values expressed by it, if any.

Determining well-foundedness means checking that no two grammar tokens each require the use of the other, and that when a grammar token takes several alternative forms, they have compatible results: so, for instance, you can't have one version resulting in a number and another in a thing. (This check is only meaningful for grammar verbs of type `GV_IS_TOKEN`.)

The result of a `GV_IS_TOKEN` is a single type, which is the union of the types resulting from its grammar lines. This is a more sophisticated approach than we really need here, but we may as well use the type-casting machinery for the sake of any later expansion (if kinds of kinds of value are ever allowed, for instance).

Of the determining traverse the following can be said:

(a) either errors are produced, or it is verified that no token's definition depends directly or indirectly on already knowing itself;

(b) also that no grammar line attached to a `GV_IS_COMMAND` produces more than 2 values, and that no grammar line attached to anything else produces more than one; and

(c) also that the grammar lines attached to a `GV_IS_TOKEN` are compatible in that there is a type to which they can all always be cast.

We note of the determining routines that:

(a) `gv_determine` runs at least once for each GV;

(b) `gl_determine` runs exactly once on each GL;

(c) `gtok_determine` runs exactly once on each token.

§**4. Phase III: Sort Grammar.** We must ensure that if grammar line L1 is logically impossible once L2 has been parsed, then L1 must be checked before L2, regardless of the ordering in the source code. Since the data sets are very small and time is not of the essence, we simply insertion-sort the original definition-order list into a second linked list.

§**5. Phase IV: Compile Grammar.** The final run-through, which uses the sorted order and not the original declaration order, actually compiles the necessary I6 `Verb` and `Extend` directives.

Traverse for Grammar 13/tfg

Purpose

To create and manipulate grammar, primarily by parsing and acting upon Understand... sentences in the source text.

13/tfg.§4 Dividing Understand into cases; §5 Understand command verbs; §6 Understand property names; §7 Understand the player's command or snippets

---

Template interpreter commands

```
2   {-callv:traverse_for_grammar}
```

---

§**1.** New grammar arrives in the system in two ways: primarily by means of explicit Understand sentences in the source text, but also secondarily in the form of table entries or other values used to match against snippets. For example:

> Understand "drill [something]" as drilling.
>
> if the player's command matches "room [number]", ...

§**2.** Understand sentences can also revoke existing grammar, in some cases, as we shall see. They are not read in the main assertion traverse, since they depend on too much not known then: they have a traverse of their own, and so do not use the sentence handler system adopted by the main assertion traverse.

```
int base_problem_count;
void traverse_for_grammar(void) {
    int w1, w2, as1, as2, when1, when2, w;
    parse_node *p;
    for (TREE_START(p); p; TREE_NEXT(p))
        if (pn_get_node_type(p) == SENTENCE_NT) {
            if ((p->down)
                && (pn_get_node_type(p->down) == VERB_NT)
                && (pn_int_annotation(p->down, verb_id_ANNOT) == UNDERSTAND_VB)) {
                [[w1, w2 <-- p->down->next]];
                [[as1, as2 <-- p->down->next->next]];
                when1 = -1; when2 = -1;
                if (![[as1, as2 == ... when ... : w --> as1, as2 ... when1, when2]])
                    [[as1, as2 == ... while ... : w --> as1, as2 ... when1, when2]];
                base_problem_count = problem_count;
                understand_sentence(w1+1, w2, as1, as2, when1, when2);
                finish_this_session_of_parsing();
            }
        }
}
```

The function traverse_for_grammar is invoked by a command in a `.i6t` template file.

§**3.** The secondary means of acquiring new grammar is used when compiling type specifications of type `VALUE/UNDERSTANDING` and when compiling the entries of "topic" columns in tables. These will usually be simple constructions of individual grammar lines, but they need to belong to a grammar verb (GV) nevertheless, even if they are the only thing on that GV. Such GVs compile to routines for parsing snippets, and no pointers exist to them in other NI data structures: the result of the routine below, assuming no problems are issued, is simply that the name of a snippet-parsing routine is printed.

```
void compile_understanding(OUTPUT_STREAM, int w1, int w2, int table_entry) {
    if [[w1, w2 == it]] WRITE("0");
    else {
        base_problem_count = problem_count;
        gpr_prepare_consultation_gv();
        understand_block(w1, w2, -1, -1, -1, -1, table_entry);
        gpr_print_consultation_gv_name(OUT);
    }
}
```

The function compile_understanding is called from 7/vasp and 10/tab.

§**4. Dividing Understand into cases.** There are three basic cases, according to what must be understood: the player's command (tacitly assumed), a command verb or a property name.

```
void understand_sentence(int w1, int w2, int as1, int as2, int when1, int when2) {
    LOGIF(GRAMMAR, "Parsing understand <$W> as <$W> when <$W>\n",
        w1, w2, as1, as2, when1, when2);
    if (problem_count > base_problem_count) return;
    if [[w1, w2 == the command/commands ... --> w1, w2]] {
        understand_the_command(w1, w2, as1, as2, when1, when2);
        return;
    }
    if [[w1, w2 == the verb/verbs ...]] {
        sentence_problem(_P_(C13OldVerbUsage),
            "this is an outdated form of words",
            "and Inform now prefers 'Understand the command ...' "
            "rather than 'Understand the verb ...'. (Since this "
            "change was made in beta-testing, quite a few old "
            "source texts still use the old form: the authors "
            "of Inform apologise for any nuisance incurred.)");
        return;
    }
    if (is_list_divided(w1, w2, LOOK_FOR_AND + LOOK_FOR_OR)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        understand_sentence(lw1, lw2, as1, as2, when1, when2);
        understand_sentence(rw1, rw2, as1, as2, when1, when2);
        return;
    }
    if [[w1, w2 == ... property --> w1, w2]] {
        understand_property_block(w1, w2, as1, as2, when1, when2);
        return;
    }
    understand_block(w1, w2, as1, as2, when1, when2, FALSE);
}
```

§**5. Understand command verbs.** These sentences allow us to control the assignment of command verbs such as TAKE or EXAMINE to grammars, which will normally be an automatic process based on grammar lines (see below). We can make one command verb an alias for another, or revoke this by making it "something new".

After some debate, we decided that it ought to be legal to declare "Understand the command "wibble" as something new" even in cases where no "wibble" command existed already: extensions might want this to assure that they have exclusive use of a command, for instance. So the problem message for this case is now commented out.

```
void understand_the_command(int w1, int w2, int as1, int as2, int when1, int when2) {
    grammar_verb *gv;
    if (problem_count > base_problem_count) return;
    if (when1 >= 0) {
        sentence_problem(_P_(C13UnderstandCommandWhen),
            "'understand the command ... as ...' is not allowed to have a "
            "'... when ...' clause",
            "for the moment at any rate.");
        return;
    }
    if (is_list_divided(w1, w2,
        LOOK_FOR_OR+LOOK_FOR_AND)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        understand_the_command(lw1, lw2, as1, as2, when1, when2);
        understand_the_command(rw1, rw2, as1, as2, when1, when2);
        return;
    }
    dequote_word(w2);
    gv = gv_find_command(w2);
    if [[as1, as2 == something new]] {
        if (gv == NULL) return;
        gv_remove_command(gv, w2);
        return;
    }
    if (gv) {
        if (gv_is_empty(gv)) {
            DESTROY(gv, grammar_verb);
            gv = NULL;
        } else {
            sentence_problem(_P_(C13NotNewCommand),
                "'understand the command ... as ...' is only allowed when "
                "the new command has no meaning already",
                "so for instance 'understand \"drop\" as \"throw\"' is not "
                "allowed because \"drop\" already has a meaning.");
            return;
        }
    }
    dequote_word(as1);
    gv = gv_find_command(as1);
    if ((gv == NULL) || (as2 > as1)) {
        sentence_problem(_P_(C13NotOldCommand),
            "'understand the command ... as ...' should end with a command "
```

```
            "already defined",
            "as in 'understand the command \"steal\" as \"take\"'. (This "
            "problem is sometimes seen when the wrong sort of Understand... "
            "sentence has been used: 'Understand the command \"steal\" as "
            "\"take\".' tells me to treat the command STEAL as a "
            "synonym for TAKE when reading the player's commands, whereas "
            "'Understand \"steal [something]\" as taking.' tells me that "
            "here is a specific grammar for what can be said using the "
            "STEAL command.)");
        return;
    }
    gv_add_command(gv, w2);
}
```

§6. Understand property names.

```
void understand_property_block(int w1, int w2, int as1, int as2, int when1, int when2) {
    world_object *wo = NULL;
    property_name *pr;
    int level = 0;
    pr = parse_property_name(w1, w2);
    if (pr == NULL) {
        sentence_problem(_P_(C13UnknownUnderstandProperty),
            "I don't understand what property that refers to",
            "but it doesn't seem to be a property I know. An example of "
            "correct usage is 'understand the transparent property as "
            "describing a container.'");
        return;
    }
    if [[as1, as2 == referring to ... --> as1, as2]] level=1;
    else if [[as1, as2 == describing ... --> as1, as2]] level=2;
    if (level == 0) {
        sentence_problem(_P_(C13BadUnderstandProperty),
            "'understand the ... property as ...' is only allowed if "
            "followed by 'describing ...' or 'referring to ...'",
            "so for instance 'understand the transparent property as "
            "describing a container.'");
        return;
    }
    wo = parse_world_object(as1, as2, FALSE);
    if (wo == NULL) {
        sentence_problem(_P_(C13BadUnderstandPropertyAs),
            "I don't understand what thing or kind that refers to",
            "but it does need to be a thing or kind and not (say) a value. "
            "For instance, 'understand the transparent property as describing "
            "a container.'");
        return;
    }
    if ((prn_is_either_or(pr) == FALSE) &&
        (kov_get_recognition_only_GPR(prn_get_kind_of_value(pr)) == NULL) &&
        (kov_request_I6_GPR(prn_get_kind_of_value(pr)) == FALSE)) {
        sentence_problem(_P_(C13BadReferringProperty),
```

```
            "that property is of a kind which I can't recognise in "
            "typed commands",
            "so that it cannot be understand as describing or referring to "
            "something. I can understand either/or properties, properties "
            "with a limited list of named possible values, numbers, times "
            "of day, or units; but certain built-into-Inform kinds of value "
            "(like snippet or rulebook, for instance) I can't use. In "
            "particular properties which are texts can't be used this way - "
            "you need an indexed text property instead.");
    }
    if (seek_permission(pr, wo, level, when1, when2) == FALSE) {
        sentence_problem(_P_(C13UnknownUnpermittedProperty),
            "That property is not allowed for the thing or kind in question",
            "just as (ordinarily) 'understand the open property as describing a "
            "device' would not be allowed because it makes no sense to call a "
            "device 'open'.");
    }
    return;
}
```

§**7. Understand the player's command or snippets.** If `as_pn` is non-null, we have new grammar for the player's command, and details of the outcome are in this parse node; if it is null, we shall only be parsing a snippet, and we are looking at a question rather than an answer.

```
void understand_block(int w1, int w2, int as1, int as2, int when1, int when2,
    int table_entry) {
    int x1, x2, gv_is = GV_IS_COMMAND,
        reversed = FALSE, file_under_vn = -1, mistake_text_at = 0, pluralised = FALSE;
    kind_of_value *kov = NULL, *lit;
    action_name *an = NULL;
    grammar_line *gl = NULL;
    parse_node *to_pn = NULL;
    world_object *wo = NULL;
    property_name *gv_prn = NULL;
    specification *gl_value = NULL;
    if (problem_count > base_problem_count) return;
    if (is_list_divided(w1, w2, LOOK_FOR_AND + LOOK_FOR_OR)) {
        int lw1 = left_w1, lw2 = left_w2, rw1 = right_w1, rw2 = right_w2;
        understand_block(lw1, lw2, as1, as2, when1, when2, table_entry);
        understand_block(rw1, rw2, as1, as2, when1, when2, table_entry);
        return;
    }
    LOGIF(GRAMMAR, "Understand <$W> (table entry %d) as <$W> when <$W>\n",
        w1, w2, table_entry, as1, as2, when1, when2);
    lit = is_a_literal(w1, w2);
    if ((is_kova(lit, TEXT_TY) == FALSE) && (is_kova(lit, TEXT_ROUTINE_TY) == FALSE)) {
        LOG("Literal value of $W is $u\n", w1, w2, lit);
        if (table_entry)
            sentence_problem(_P_(C13NontextualTable),
                "a table entry in a 'topic' column must be a single double-quoted "
                "text",
                "such as \"eternity\" or \"peruvian skies\".");
```

```
    else
        sentence_problem(_P_(C13NontextualUnderstand),
            "'understand' should be followed by a textual description",
            "as in 'understand \"take [something]\" as taking the noun'.");
    return;
}
if (well_formed_text_routine(lw_array[w1].lw_text) == FALSE) {
    sentence_problem(_P_(C13UnderstandMismatch),
        "'understand' should be followed by text in which brackets "
        "'[' and ']' match",
        "so for instance 'understand \"take [something]\" as taking the noun' "
        "is fine, but 'understand \"take]\" as taking' is not.");
    return;
}
mistake_text_at = 0;
if (as1 < 0) gv_is = GV_IS_CONSULT;
else {
    int mb1, mb2;
    x1 = as1; x2 = as2;
    gv_is = GV_IS_COMMAND;
    an = NULL;
    if [[x1, x2 == a mistake]] mistake_text_at = -1;
    else if [[x1, x2 == a mistake OPENBRACKET ... CLOSEBRACKET --> mb1, mb2]] {
        lit = is_a_literal(mb1, mb2);
        if ((is_kova(lit, TEXT_TY)) || (is_kova(lit, TEXT_ROUTINE_TY)))
            mistake_text_at = mb1;
    }
    if ([[x1, x2 == a mistake ...]] && (mistake_text_at == 0)) {
        sentence_problem(_P_(C13TextlessMistake),
            "when 'understand' results in a mistake it can only be "
            "followed by a textual message in brackets",
            "so for instance 'understand \"take\" as a mistake "
            "(\"In this sort of game, a noun is required there.\").'");
        return;
    }
    if ([[x1, x2 == plural of ... --> x1, x2]] ||
        [[x1, x2 == the plural of ... --> x1, x2]])
        pluralised = TRUE;
    if (mistake_text_at != 0)
        gv_is = GV_IS_COMMAND;
    else if (is_kova(is_a_literal(x1, x2), TEXT_ROUTINE_TY))
        gv_is = GV_IS_TOKEN;
    else {
        if [[x1, x2 == ... OPENBRACKET with nouns reversed CLOSEBRACKET --> x1, x2]]
            reversed = TRUE;
        an = act_by_name(x1, x2);
        if (an == NULL) {
            world_object *target;
            specification *spec = parse_expression(x1, x2, DESCRIPTIVE_TYPE_EXPCON);
            target = spec_object_exactly_described_if_any(spec);
            if (target) {
                wo = target;
```

```
            gv_is = GV_IS_OBJECT;
            if (spec_is_qualified_DESCRIPTION(spec)) {
                LOG("Offending description: $X", spec);
                sentence_problem(_P_(C13UnderstandAsQualified),
                    "I cannot understand text as meaning an object "
                    "qualified by relative clauses or properties",
                    "only a specific thing, a specific value or a kind. "
                    "(But the same effect can usually be achieved with "
                    "a 'when' clause. For instance, although 'Understand "
                    "\"bad luck\" as the broken mirror' is not allowed, "
                    "'Understand \"bad luck\" as the mirror when the "
                    "mirror is broken' produces the desired effect.)");
                return;
            }
        } else {
            RetryValue:
            LOGIF(GRAMMAR_CONSTRUCTION, "Understand as specification: $X", spec);
            if (spec_is_generic_CONSTANT(spec)) goto ImpreciseProblemMessage;
            if (spec_is_phrasal(spec)) goto ImpreciseProblemMessage;
            if (family_is(spec, VALUE_FMY)) {
                kov = spec_get_kind_of_value(spec);
                if (kov_request_I6_GPR(kov)) {
                    gl_value = spec;
                    gv_is = GV_IS_VALUE;
                } else {
                    if (is_kova(kov, ACTIVITY_TY))
                    sentence_problem(_P_(C13UnderstandAsActivity),
                        "this 'understand ... as ...' gives text "
                        "meaning an activity",
                        "rather than an action. Since activities "
                        "happen when Inform decides they need to "
                        "happen, not in response to typed commands, "
                        "this doesn't make sense.");
                    else
                    sentence_problem(_P_(C13UnderstandAsBadValue),
                        "'understand ... as ...' gives text "
                        "meaning a value whose kind is not allowed",
                        "and should be a value such as 100.");
                    return;
                }
            } else if (species_is(spec, DESCRIPTION_SPC)) {
                if ((spec_get_described_object(spec) == NULL) && (spec_get_described_kind(spec)
== NULL)
                    && (number_of_adjectives_applied_to(spec) == 1)
                    && (adjective_used_positively(first_adjective_list_entry(spec)))) {
                    adjectival_phrase *aph =
                        get_adjective_from_list_entry(first_adjective_list_entry(spec));
                    quantity *q = aph_has_ENUMERATIVE_meaning(aph);
                    if (q) {
                        spec = new_QUANTITY_spec(q);
                        goto RetryValue;
                    }
                    property_name *prn = aph_has_EORP_meaning(aph);
```

```
                            if (prn) {
                                gv_is = GV_IS_PROPERTY_NAME;
                                gv_prn = prn;
                                LOGIF(GRAMMAR_CONSTRUCTION, "Grammar confirmed for property $Y\n",
gv_prn);
                            }
                        }
                        if (spec_get_described_kind(spec)) {
                            wo = spec_get_described_kind(spec);
                            gv_is = GV_IS_OBJECT;
                        }
                        if ((spec_is_qualified_DESCRIPTION(spec)) && (gv_prn == NULL)) {
                            LOG("Offending description: $X", spec);
                            sentence_problem(_P_(C13ComplexUnderstand),
                                "I cannot understand text as having so complicated "
                                "a meaning",
                                "only a specific thing, a specific value or a kind. "
                                "So 'Understand \"darce\" as Mr Darcy' is allowed, "
                                "but 'Understand \"bucket\" as an open container' "
                                "is not, because it is too general a description.");
                            return;
                        }
                    } else {
                        ImpreciseProblemMessage:
                        LOG("Offending pseudo-meaning is: $X", spec);
                        sentence_problem(_P_(C13UnderstandVague),
                            "'understand ... as ...' should be followed "
                            "by a meaning",
                            "which might be an action (e.g., "
                            "'understand \"take [something]\" as taking'), a "
                            "thing ('understand \"stove\" as the oven') or more "
                            "generally a value ('understand \"huitante\" as 80'), "
                            "or a named token for use in further grammar "
                            "('understand \"near [something]\" as \"[location "
                            "phrase]\"'). Also, the meaning needs to be precise, "
                            "so 'understand \"x\" as a number' is not "
                            "allowed - it does not say which number.");
                        return;
                    }
                }
            }
        }
    }
    if ((pluralised) && (gv_is != GV_IS_OBJECT)) {
        sentence_problem(_P_(C13UnderstandPluralValue),
            "'understand' as a plural can only apply to things, rooms or kinds "
            "of things or rooms",
            "so 'Understand \"paperwork\" as the plural of a document.' is "
            "fine (assuming a document is a kind of thing), but 'Understand "
            "\"dozens\" as the plural of 12' is not.");
        return;
    }
    int i, skip = FALSE, literal_punct = FALSE; char *p = lw_array[w1].lw_text;
```

```
for (i=0; p[i]; i++) {
    if (p[i] == '[') skip = TRUE;
    if (p[i] == ']') skip = FALSE;
    if (skip) continue;
    if ((p[i] == '.') || (p[i] == ',') ||
        (p[i] == '!') || (p[i] == '?') || (p[i] == ':') || (p[i] == ';'))
        literal_punct = TRUE;
}
if (literal_punct) {
    sentence_problem(_P_(C13LiteralPunctuation),
        "'understand' text cannot contain literal punctuation",
        "or more specifically cannot contain any of these: . , ! ? : ; "
        "since they are already used in various ways by the parser, and "
        "would not correctly match here.");
    return;
}
feed_into_lexer(lw_array[w1].lw_text, TRUE, TRUE);
x1 = lexer_feed_w1; x2 = lexer_feed_w2;
to_pn = new_nounphrase_raw(w1, w2);
gtok_break_into_tokens(to_pn, x1, x2);
if (to_pn->down == NULL) {
    sentence_problem(_P_(C13UnderstandEmptyText),
        "'understand' should be followed by text which contains at least "
        "one word or square-bracketed token",
        "so for instance 'understand \"take [something]\" as taking' "
        "is fine, but 'understand \"\" as the fog' is not.");
    return;
}
if (gv_is == GV_IS_COMMAND) {
    int i;
    LOGIF(GRAMMAR_CONSTRUCTION, "Command grammar: $T\n", to_pn);
    for (i=x1; i<x2; i++) {
        if ((compare_word(i, COMMA_V)) && (compare_word(i+1, COMMA_V))) {
            sentence_problem(_P_(C13UnderstandCommaCommand),
                "'understand' as an action cannot involve a comma",
                "since a command leading to an action never does. "
                "(Although Inform understands commands like 'PETE, LOOK' "
                "only the part after the comma is read as an action command: "
                "the part before the comma is read as the name of someone, "
                "according to the usual rules for parsing a name.) "
                "Because of the way Inform processes text with square "
                "brackets, this problem message is also sometimes seen "
                "if empty square brackets are used, as in 'Understand "
                "\"bless []\" as blessing.'");
            return;
        }
    }
    if (gtok_is_literal(to_pn->down) == FALSE)
        file_under_vn = -1;                                    this will go into the no verb verb
    else file_under_vn = to_pn->down->word_ref1;
}
LOGIF(GRAMMAR, "GV is %d, an is $l, file under is $W\n", gv_is, an,
    file_under_vn, file_under_vn);
```

```
    if (gv_is != GV_IS_COMMAND) gl = gl_new(w1, NULL, to_pn, reversed, pluralised);
    else gl = gl_new(w1, an, to_pn, reversed, pluralised);
    if (mistake_text_at != 0) gl_set_mistake(gl, mistake_text_at);
    if (when1 >= 0) {
        gl_set_understand_when(gl, when1, when2);
        if (gv_is == GV_IS_CONSULT) {
            sentence_problem(_P_(BelievedImpossible),          at present, I7 syntax prevents this anyway
                "'when' cannot be used with this kind of 'Understand'",
                "for the time being at least.");
            return;
        }
    }
    switch(gv_is) {
        case GV_IS_TOKEN:
            feed_into_lexer(lw_array[as1].lw_text, TRUE, TRUE);
            x1 = lexer_feed_w1; x2 = lexer_feed_w2;
            LOGIF(GRAMMAR_CONSTRUCTION, "GV_IS_TOKEN as words: $W\n", x1, x2);
            if (valid_new_token_name(x1, x2) == FALSE) {
                sentence_problem(_P_(C13UnderstandAsCompoundText),
                    "if 'understand ... as ...' gives the meaning as text "
                    "then it must describe a single new token",
                    "so that 'Understand \"group four/five/six\" as "
                    "\"[department]\"' is legal (defining a new token "
                    "\"[department]\", or adding to its definition if it "
                    "already existed) but 'Understand \"take [thing]\" "
                    "as \"drop [thing]\"' is not allowed, and would not "
                    "make sense, because \"drop [thing]\" is a combination "
                    "of two existing tokens - not a single new one.");
            }
            gv_add_line(gv_named_token_new(x1+2, x2-2), gl);
            break;
        case GV_IS_COMMAND:
            gv_add_line(gv_find_or_create_command(file_under_vn), gl);
            break;
        case GV_IS_OBJECT:
            gv_add_line(gv_for_object(wo), gl);
            break;
        case GV_IS_VALUE:
            gl_set_single_type(gl, gl_value);
            gv_add_line(gv_for_kov(kov), gl);
            break;
        case GV_IS_PROPERTY_NAME:
            gv_add_line(gv_for_prn(gv_prn), gl);
            break;
        case GV_IS_CONSULT:
            gl_set_single_type(gl, gl_value);
            gv_add_line(gpr_get_consultation_gv(), gl);
            break;
    }
}

int valid_new_token_name(int w1, int w2) {
    int i, cc=0;
    for (i=w1; i<=w2; i++)
```

```
        if (compare_word(i, COMMA_V)) cc++;
    dequote_word(w1);
    if (*(lw_array[w1].lw_text) != 0) return FALSE;
    dequote_word(w2);
    if (*(lw_array[w2].lw_text) != 0) return FALSE;
    if (cc != 2) return FALSE;
    return TRUE;
}
```

Grammar Verbs 13/gv

Purpose

A grammar verb is not literally a verb, as the name is a hangover from the way I6 defines grammar. It might be said to be a necklace onto which we thread the sea-shells of grammar lines. Each grammar verb has its own purpose: to match various possibilities (one for each grammar line) against text aimed at a particular result. For instance, all run time commands beginning with TAKE are parsed with a single grammar verb. If we create a new grammar token, "[polite remark]", for use in other grammar, then that too will have its own "grammar verb". If we define the word "eleventy-one" as meaning the number 111, it will be added to a grammar verb attached to the data type `NUMBER` which parses eccentric names for number values. And so on. Probably a better name for this structure would be simply "grammar", but that might be confusing in other ways, and anyway the ship has sailed.

13/gv.§2-8 Command words; §9-10 Named grammar tokens; §11 Consultation grammars; §12 Object parsing grammars; §13 Data type parsing grammars; §14 Property name parsing grammars; §15-18 The list of grammar lines; §19 Grammar Preparation; §20 Phase I: Slash Grammar; §21 Phase II: Determining Grammar; §22-26 Phases III and IV: Sort and Compile Grammar

---

Template interpreter commands

```
18  {-callv:compile_grammar_conditions}
19  {-callv:prepare_grammar}
22  {-callv:compile_grammar_verbs}
```

Definitions

¶1. There are five different sorts of grammar verb, then, and only the first of these is associated with a genuine typed-by-the-player command verb:

```
define GV_IS_COMMAND 1                  an imperative verbal command at run-time
define GV_IS_TOKEN   2                  a square-bracketed token in other grammar
define GV_IS_OBJECT  3                  a noun phrase at run time: a name for an object
define GV_IS_VALUE   4                  a noun phrase at run time: a name for a value
define GV_IS_CONSULT 5                  a pattern to match in part of a command (such as "consult")
define GV_IS_PROPERTY_NAME 6            a noun phrase at run time: a name for an either/or pval
```

¶2. The following maxima are imposed by the I6 compiler:

```
define MAX_ALIASED_COMMANDS 32
define MAX_LINES_PER_COMMAND 32

typedef struct grammar_verb {
    int gv_is;                                          one of the five values above
    struct grammar_type gv_type;
    struct grammar_line *first_line;                    linked list in creation order
    struct grammar_line *sorted_first_line;             and in logical applicability order
    int command_wn;                                     GV_IS_COMMAND: word number at which command found
    int aliased_command_wn[MAX_ALIASED_COMMANDS];       ...and other commands synonymous
    int no_aliased_commands;                            ...and how many of them there are
    int word_ref1, word_ref2;                           GV_IS_TOKEN: name of this token
    struct world_object *wo_understood;                 GV_IS_OBJECT: what this provides names for
    struct kind_of_value *kov_understood;               GV_IS_VALUE: for which type it names an instance of
    struct property_name *prn_understood;               GV_IS_PROPERTY_NAME: which prn this names
    struct parse_node *where_gv_created;                for problem message reports
    char gv_I6_identifier[32];                          when a token is delegated to an I6 routine
    MEMORY_MANAGEMENT
} grammar_verb;
int no_verb_verb_defined = FALSE;
```

The structure grammar_verb is private to this section.

¶3. A few imperative verbs are reserved for Inform testing, such as SHOWME. We record those as instances of the following:

```
typedef struct reserved_command_verb {
    char reserved_text[32];
    MEMORY_MANAGEMENT
} reserved_command_verb;
```

The structure reserved_command_verb is private to this section.

---

§**1.** We begin as usual with a constructor and some debug log tracing.

```
grammar_verb *gv_new(int gv_is) {
    grammar_verb *gv;
    gv = CREATE(grammar_verb);
    gv->command_wn = -1;
    gv->first_line = NULL;
    gv->gv_type = gty_new(FALSE);
    gv->gv_is = gv_is;
    gv->word_ref1 = -1; gv->word_ref2 = -1;
    gv->no_aliased_commands = 0;
    gv->sorted_first_line = NULL;
    gv->kov_understood = NULL;
    gv->prn_understood = NULL;
    gv->where_gv_created = current_sentence;
    gv->gv_I6_identifier[0] = 0;
```

```
    return gv;
}
void log_grammar_verb(grammar_verb *gv) {
    LOG("<GV%d:", gv->allocation_id);
    switch(gv->gv_is) {
        case GV_IS_COMMAND:
            if (gv->command_wn == -1) LOG("command=no-verb verb");
            else LOG("command=$W", gv->command_wn, gv->command_wn);
            break;
        case GV_IS_TOKEN: LOG("token=$W", gv->word_ref1, gv->word_ref2); break;
        case GV_IS_OBJECT: LOG("object"); break;
        case GV_IS_VALUE: LOG("value=$u", gv->kov_understood); break;
        case GV_IS_CONSULT: LOG("consult"); break;
        case GV_IS_PROPERTY_NAME: LOG("property-name"); break;
        default: LOG("<unknown>"); break;
    }
    LOG(">");
}
```

The function log_grammar_verb is called from 2/dl.

§2. **Command words.** Some GVs are used to represent the command grammar for imperative verbs used by the player at run-time. Such a GV handles multiple commands, which are considered equivalent at run-time: the first of these is the official command word, and the rest are "aliases". For instance, the Standard Rules create a GV for the command PULL with one alias, DRAG. (This somewhat asymmetric approach is used because it matches the way I6 `Verb` declarations are laid out.)

A complication is that one GV is permitted to be a special case: the so-called "no verb verb", whose main command word is empty and which can have no aliases. This is used to parse verbless commands at run-time: for instance, the I7 designer can specify that a command consisting only of a number followed by the word GO should cause some action, and this is implemented not with a command verb but using I6's hooks for verbless commands. The grammar "[number] go" is attached as a grammar line to the "no verb verb", which is distinguished by having its command word number set to `-1`. Note that the "no verb verb" exists only in runs of NI where it has been needed: the Standard Rules do not use it.

Command GVs other than the "no verb verb" are said to be "genuinely verbal".

```
int gv_get_verb_wn(grammar_verb *gv) {
    return gv->command_wn;
}
int gv_is_genuinely_verbal(grammar_verb *gv) {
    if ((gv->gv_is == GV_IS_COMMAND) && (gv->command_wn != -1))
        return TRUE;
    return FALSE;
}
```

The function gv_get_verb_wn is called from 13/gl.

§**3.** The next routine finds, or if necessary creates, a GV for a given command word encountered without any indication that it should alias another. Note that calling this with word number `-1` finds, or creates, the "no verb verb".

```
grammar_verb *gv_find_or_create_command(int vw) {
    grammar_verb *gv = gv_find_command(vw);
    if (gv) return gv;
    gv = gv_new(GV_IS_COMMAND);
    gv->command_wn = vw;
    if (vw == -1) no_verb_verb_defined = TRUE;
    else {
        LOGIF(GRAMMAR_CONSTRUCTION, "GV%d has verb %d = <%s>\n",
            gv->allocation_id, vw, lw_array[vw].lw_text);
    }
    return gv;
}
```

The function gv_find_or_create_command is called from 13/tfg.

§**4.** By contrast, this routine merely finds a GV, or returns null if none exists with the given command word.

```
grammar_verb *gv_find_command(int vw) {
    grammar_verb *gv;
    LOOP_OVER(gv, grammar_verb)
        if (gv->gv_is == GV_IS_COMMAND) {
            int i;
            if (vw == -1) {
                if (gv->command_wn == -1) return gv;
            } else {
                if (compare_words(vw, gv->command_wn)) return gv;
                for (i=0; i<gv->no_aliased_commands; i++)
                    if (compare_words(vw, gv->aliased_command_wn[i])) return gv;
            }
        }
    return NULL;
}
```

The function gv_find_command is called from 13/tfg.

§**5.** We now have routines to add or remove commands from a given GV. Removing is the tricky case, since detaching the main command word means that one of the aliases must become the new main command; or, in the worst case, that there are no commands left, in which case we need to empty the GV of grammar lines so that it can either be ignored or re-used (in the event that the designer, having cancelled the old meaning of the command, now supplies new ones).

It is not possible to add to, or remove from, the "no verb verb".

```
void gv_add_command(grammar_verb *gv, int wn) {
    if (gv == NULL)
        internal_error("tried to add alias command to null GV");
    if (gv->gv_is != GV_IS_COMMAND)
        internal_error("tried to add alias command to non-command GV");
    if (gv->no_aliased_commands == MAX_ALIASED_COMMANDS) {
        sentence_problem(_P_(C13TooManyAliases),
            "this 'understand the command ... as ...' makes too many aliases "
            "for the same command",
            "exceeding the limit of 32.");
        return;
    }
    gv->aliased_command_wn[gv->no_aliased_commands++] = wn;
    LOGIF(GRAMMAR, "Adding alias '%s' (%d) to G%d '%s' (%d)\n",
        lw_array[wn].lw_text, wn, gv->allocation_id, lw_array[gv->command_wn].lw_text, gv->command_wn);
}
void gv_remove_command(grammar_verb *gv, int wn) {
    if (gv == NULL)
        internal_error("tried to detach alias command from null GV");
    if (gv->gv_is != GV_IS_COMMAND)
        internal_error("tried to detach alias command from non-command GV");
    LOGIF(GRAMMAR, "Detaching verb '%s' from grammar\n", lw_array[wn].lw_text);
    if (gv == NULL) return;
    if (compare_words(gv->command_wn, wn)) {
        LOGIF(GRAMMAR, "Detached verb is the head-verb\n");
        if (gv->no_aliased_commands == 0) {
            gv->first_line = NULL;
            LOGIF(GRAMMAR, "Which had no aliases: clearing grammar to NULL\n");
        } else {
            gv->command_wn = gv->aliased_command_wn[--(gv->no_aliased_commands)];
            LOGIF(GRAMMAR, "Which had aliases: making new head-verb '%s'\n",
                lw_array[gv->command_wn].lw_text);
        }
    } else {
        LOGIF(GRAMMAR, "Detached verb is one of the aliases\n");
        int i, j;
        for (i=0; i<gv->no_aliased_commands; i++) {
            if (compare_words(gv->aliased_command_wn[i], wn)) {
                for (j=i; j<gv->no_aliased_commands-1; j++)
                    gv->aliased_command_wn[j] = gv->aliased_command_wn[j+1];
                gv->no_aliased_commands--;
                break;
            }
        }
    }
}
```

The function gv_add_command is called from 13/tfg.
The function gv_remove_command is called from 13/tfg.

§**6.** Command GVs are destined to be compiled into `Verb` directives, as follows.

```
void gv_compile_Verb_directive_header(OUTPUT_STREAM, grammar_verb *gv) {
    if (gv->gv_is != GV_IS_COMMAND)
        internal_error("tried to compile Verb from non-command GV");
    if (gv->first_line == NULL)
        internal_error("compiling Verb for empty grammar");
    if (gv->command_wn == -1)
        WRITE("Verb 'no.verb'");
    else {
        int i;
        WRITE("Verb ");
        if (command_verb_reserved(lw_array[gv->command_wn].lw_text)) {
            current_sentence = gv->where_gv_created;
            quote_source(1, current_sentence);
            quote_text(2, lw_array[gv->command_wn].lw_text);
            handmade_problem(_P_(C13ReservedVerb));
            issue_problem_segment(
                "You wrote %1, but %2 is a built-in Inform testing verb, which "
                "means it is reserved for Inform's own use and can't be used "
                "for ordinary play purposes. %PThe verbs which are reserved in "
                "this way are all listed in the alphabetical catalogue on the "
                "Actions Index page.");
            issue_problem_end();
        }
        isn_compile_dictionary_word(OUT, lw_array[gv->command_wn].lw_text, FALSE);
        for (i=0; i<gv->no_aliased_commands; i++) {
            WRITE(" ");
            isn_compile_dictionary_word(OUT, lw_array[gv->aliased_command_wn[i]].lw_text, FALSE);
        }
    }
    WRITE("\n");
}
```

§**7.** Reserved verb names are collated as the I6 template files are read:

```
void reserve_command_verb(char *verb_name) {
    reserved_command_verb *rcv = CREATE(reserved_command_verb);
    normalise_cv_to(verb_name, rcv->reserved_text);
    index_test_verb(rcv->reserved_text);
}
int command_verb_reserved(char *verb_tried) {
    reserved_command_verb *rcv;
    char normalised_vt[32];
    normalise_cv_to(verb_tried, normalised_vt);
    LOOP_OVER(rcv, reserved_command_verb)
        if (strcmp(normalised_vt, rcv->reserved_text) == 0)
            return TRUE;
    return FALSE;
}
void normalise_cv_to(char *from, char *to) {
    int i;
    for (i=0; (i<31) && (from[i]); i++) to[i] = tolower(from[i]);
    to[i] = 0;
}
```

The function reserve_command_verb is called from 14/i6t.

§**8.** The "Commands available to the player" portion of the Actions index page is, in effect, an alphabetised merge of the GLs found within the command GVs. GLs for the "no verb verb" appear under the special headword "0" (which is not displayed); otherwise GLs appear under the main command word, and aliases are shown with references like: "drag", same as "pull".

One routine takes a GV and creates suitable entries for the Actions index to process; the other two routines act upon any such entries once they are needed.

```
void gv_make_command_index_entries(grammar_verb *gv) {
    if ((gv->gv_is == GV_IS_COMMAND) && (gv->first_line)) {
        int i;
        if (gv->command_wn == -1)
            vie_new_from("0", gv, NORMAL_COMMAND);
        else
            vie_new_from(lw_array[gv->command_wn].lw_text, gv, NORMAL_COMMAND);
        for (i=0; i<gv->no_aliased_commands; i++)
            vie_new_from(lw_array[gv->aliased_command_wn[i]].lw_text, gv, ALIAS_COMMAND);
    }
}
void gv_index_normal(grammar_verb *gv, char *headword) {
    gl_sorted_list_index_normal(gv->sorted_first_line, headword);
}
void gv_index_alias(grammar_verb *gv, char *headword) {
    if (gv_is_uncensored(gv)) {
        INDEX("&quot;%s&quot;, <i>same as</i> &quot;%s&quot;",
            headword, lw_array[gv->command_wn].lw_text);
        index_below_link(lw_array[gv->command_wn].lw_text);
        INDEX("<br>");
    }
}
```

The function gv_make_command_index_entries is called from 11/ina.
The function gv_index_normal is called from 11/ina.
The function gv_index_alias is called from 11/ina.

§**9. Named grammar tokens.** These are like text substitutions in reverse. For instance, we could define a token "[suitable colour]". These are identified solely by their textual names (e.g., "suitable colour").

```
grammar_verb *gv_named_token_new(int w1, int w2) {
    grammar_verb *gv;
    LOOP_OVER(gv, grammar_verb) {
        if ((gv->gv_is == GV_IS_TOKEN) &&
            (compare_word_range(w1, w2, gv->word_ref1, gv->word_ref2)))
            return gv;
    }
    gv = gv_new(GV_IS_TOKEN);
    gv->word_ref1 = w1; gv->word_ref2 = w2;
    return gv;
}
grammar_verb *gv_named_token_by_name(int w1, int w2) {
    grammar_verb *gv;
    LOOP_OVER(gv, grammar_verb) {
        if ((gv->gv_is == GV_IS_TOKEN) &&
            (compare_word_range(w1, w2, gv->word_ref1, gv->word_ref2)))
            return gv;
    }
    return NULL;
}
```

The function gv_named_token_new is called from 13/tfg.
The function gv_named_token_by_name is called from 13/gtok.

§**10.** A slight variation is provided by those which are defined by I6 routines.

```
void gv_translates(parse_node *pn) {
    parse_node *p1 = pn->down->next;
    parse_node *p2 = pn->down->next->next;
    int w1 = p1->word_ref1, w2 = p1->word_ref2;
    grammar_verb *gv;
    LOOP_OVER(gv, grammar_verb)
        if ((gv->gv_is == GV_IS_TOKEN) &&
            (compare_word_range(w1, w2, gv->word_ref1, gv->word_ref2))) {
                sentence_problem(_P_(C13GrammarTranslatedAlready),
                    "this grammar token has already been translated",
                    "so there must be some duplication somewhere.");
                return;
            }
    gv = gv_named_token_new(w1, w2);
    strcpy(gv->gv_I6_identifier, lw_array[p2->word_ref1].lw_text);
}
void gv_compile_i6_token(OUTPUT_STREAM, grammar_verb *gv) {
    char *n = gv->gv_I6_identifier;
    if (n[0]) WRITE("%s", n);
    else WRITE("GPR_Line_%d", gv->allocation_id);
}
```

The function gv_translates is called from 2/isn.
The function gv_compile_i6_token is called from 13/gtok.

§11. **Consultation grammars.** These are used for grammar included as a column of a table or in a conditional match. The terminology goes back to the early days of I6, when CONSULT was a command capable of parsing arbitrary text, something which a game called Curses made heavy use of.

```
grammar_verb *gv_consultation_new(void) {
    grammar_verb *gv;
    gv = gv_new(GV_IS_CONSULT);
    return gv;
}
```

The function gv_consultation_new is called from 13/gpr.

§12. **Object parsing grammars.** Each world object can optionally have a GV, used to parse unusual forms of its name. The following routine finds or creates this for a given WO.

```
grammar_verb *gv_for_object(world_object *wo) {
    grammar_verb *gv;
    if (wo->understand_as_this_object != NULL)
        return wo->understand_as_this_object;
    gv = gv_new(GV_IS_OBJECT);
    wo->understand_as_this_object = gv;
    gv->wo_understood = wo;
    return gv;
}
void gv_take_out_one_word_grammar(OUTPUT_STREAM, grammar_verb *gv) {
    if (gv->gv_is != GV_IS_OBJECT)
        internal_error("One-word optimisation applies only to objects");
    gv->first_line = gl_list_take_out_one_word_grammar(OUT, gv->first_line);
}
```

The function gv_for_object is called from 13/tfg.
The function gv_take_out_one_word_grammar is called from 9/cot.

§13. **Data type parsing grammars.** Each data type can optionally have a GV, used to parse unusual forms of its literals. The following routine finds or creates this for a given data type.

```
grammar_verb *gv_for_kov(kind_of_value *kov) {
    grammar_verb *gv;
    if (kov_get_parsing_grammar(kov) != NULL)
        return kov_get_parsing_grammar(kov);
    gv = gv_new(GV_IS_VALUE);
    kov_set_parsing_grammar(kov, gv);
    gv->kov_understood = kov;
    return gv;
}
```

The function gv_for_kov is called from 13/tfg.

§**14. Property name parsing grammars.** Only either/or properties can have a GV, used to parse unusual forms of the alternatives as used when properties are describing objects. The following routine finds or creates this for a given property.

```
grammar_verb *gv_for_prn(property_name *prn) {
    grammar_verb *gv;
    if (prn_get_parsing_grammar(prn) != NULL)
        return prn_get_parsing_grammar(prn);
    gv = gv_new(GV_IS_PROPERTY_NAME);
    prn_set_parsing_grammar(prn, gv);
    gv->prn_understood = prn;
    return gv;
}
```

The function gv_for_prn is called from 13/tfg.

§**15. The list of grammar lines.** Every GV has a list of GLs: indeed, this list is really the grammar. Here we test this for emptiness, and provide for adding to it. Removals are not possible.

```
int gv_is_empty(grammar_verb *gv) {
    if ((gv == NULL) || (gv->first_line == NULL)) return TRUE;
    return FALSE;
}
void gv_add_line(grammar_verb *gv, grammar_line *gl) {
    LOGIF(GRAMMAR, "Adding grammar line $g to verb $G\n", gl, gv);
    if ((gv->gv_is == GV_IS_COMMAND) &&
        (gl_list_length(gv->first_line) >= MAX_LINES_PER_COMMAND)) {
        sentence_problem(_P_(C13TooManyGrammarLines),
            "this command verb now has too many Understand possibilities",
            "that is, there are too many 'Understand \"whatever ...\" as ...' "
            "which share the same initial word 'whatever'. The best way to "
            "get around this is to try to consolidate some of those lines "
            "together, perhaps by using slashes to combine alternative "
            "wordings, or by defining new grammar tokens [in square brackets].");
        return;
    }
    gv->first_line = gl_list_add(gv->first_line, gl);
}
```

The function gv_is_empty is called from 13/tfg and 13/gpr.
The function gv_add_line is called from 13/tfg.

§**16.** A censored GV is one which should not appear in the Index because it contains naughty words, in the opinion of the Standard Rules.

```
int gv_is_uncensored(grammar_verb *gv) {
    return gl_line_list_is_uncensored(gv->first_line);
}
```

§**17.** Each GV has the potential to carry a data type made up of the number of values produced, and what their types are. This is only really meaningful for the GVs trying to express a single value: the following routine returns `UNKNOWN` unless that's the case.

```
kind_of_value *gv_get_data_type_as_token(grammar_verb *gv) {
    return gty_get_data_type_as_token(&(gv->gv_type));
}
```

The function gv_get_data_type_as_token is called from 13/gtok.

§**18.** Some tokens require suitable I6 routines to have already been compiled, if they are to work nicely: the following routine goes through the tokens by exploring each GV in turn.

```
void compile_grammar_conditions(OUTPUT_STREAM) {
    grammar_verb *gv;
    LOOP_OVER(gv, grammar_verb) {
        current_sentence = gv->where_gv_created;
        gl_line_list_compile_condition_tokens(OUT, gv->first_line);
    }
}
```

The function compile_grammar_conditions is invoked by a command in a `.i6t` template file.

§**19. Grammar Preparation.** This simply causes Phases I and II of grammar processing to take place, one after the other.

```
void prepare_grammar(void) {
    gv_slash_all();
    gv_determine_all();
}
```

The function prepare_grammar is invoked by a command in a `.i6t` template file.

§**20. Phase I: Slash Grammar.** Slashing is really a grammar-line based activity, so we do no more than pass the buck down to the list of grammar lines.

```
void gv_slash_all(void) {
    grammar_verb *gv;
    log_new_stage_of_NIs_run("Slashing grammar (G1)");
    LOOP_OVER(gv, grammar_verb) {
        LOGIF(GRAMMAR_CONSTRUCTION, "Slashing $G\n", gv);
        gl_line_list_slash(gv->first_line);
    }
}
```

§21. Phase II: Determining Grammar. Again, at this top level we are really only calling downwards.

```
void gv_determine_all(void) {
    grammar_verb *gv;
    log_new_stage_of_NIs_run("Determining grammar (G2)");
    LOOP_OVER(gv, grammar_verb)
        if (gv->first_line) {
            current_sentence = gv->where_gv_created;
            gv_determine(gv, 0);
        }
}

specification *gv_determine(grammar_verb *gv, int depth) {
    specification *spec_union = NULL;
    current_sentence = gv->where_gv_created;
    if (gty_has_return_type(&(gv->gv_type)))
        return gty_get_single_type(&(gv->gv_type));
    if (depth > NUMBER_CREATED(grammar_verb)) {
        sentence_problem(_P_(C13GrammarIllFounded),
            "grammar tokens are not allowed to be defined in terms of "
            "themselves",
            "either directly or indirectly.");
        return NULL;
    }
    LOGIF(GRAMMAR_CONSTRUCTION, "Determining $G\n", gv);
    spec_union = gl_line_list_determine(gv->first_line, depth,
        gv->gv_is, gv, gv_is_genuinely_verbal(gv));
    LOGIF(GRAMMAR_CONSTRUCTION, "Result of verb $G is $S\n", gv, spec_union);
    gty_set_single_type(&(gv->gv_type), spec_union);
    return spec_union;
}
```

The function gv_determine is called from 13/gtok.

§22. Phases III and IV: Sort and Compile Grammar. At this highest level phases III and IV are intermingled, in that Phase III always precedes Phase IV for any given list of grammar lines, but each GV goes through both Phase III and IV before the next begins Phase III. So it would not be appropriate to print banners like "Phase III begins here" in the debugging log.

Finally, though, some substantive work to do: because it is the GV which records the purpose of the grammar in question, we must compile a suitable I6 context for the grammar to appear within.

Four of the five kinds of GV are compiled by the routine below: the fifth kind is compiled in "Tokens Parsing Values", in response to different `.i6t` commands, because the token routines are needed at a different position in the final I6 output.

Sequence is important here: in particular the GPRs must exist before the `Verb` directives, because otherwise I6 will throw not-declared-yet errors.

```
void compile_grammar_verbs(OUTPUT_STREAM) {
    grammar_verb *gv;
    log_new_stage_of_NIs_run("Sorting and compiling non-value grammar (G3, G4)");
    LOOP_OVER(gv, grammar_verb)
        if (gv->gv_is == GV_IS_TOKEN)
            gv_compile(OUT, gv);                    makes GPRs for designed tokens
```

```
    LOOP_OVER(gv, grammar_verb)
        if (gv->gv_is == GV_IS_COMMAND)
            gv_compile(OUT, gv);                                    makes Verb directives
    LOOP_OVER(gv, grammar_verb)
        if (gv->gv_is == GV_IS_OBJECT)
            gv_compile(OUT, gv);                                    makes routines for use in parse_name
    LOOP_OVER(gv, grammar_verb)
        if (gv->gv_is == GV_IS_CONSULT)
            gv_compile(OUT, gv);                                    routines to parse snippets, used as values
    LOOP_OVER(gv, grammar_verb)
        if (gv->gv_is == GV_IS_PROPERTY_NAME)
            gv_compile(OUT, gv);                                    makes routines for use in parse_name
}
```

The function compile_grammar_verbs is invoked by a command in a `.i6t` template file.

§**23.** The following routine unites, so far as possible, the different forms of GV by compiling each of them as a sandwich: top slice, filling, bottom slice. The interesting case is of a GV representing names for an object: the name-behaviour needs to be inherited from the object's kind, and so on up the kinds hierarchy, but this is a case where I7's kind hierarchy does not agree with I6's class hierarchy. I6 has no (nice) way to inherit `parse_name` behaviour from a class to an instance. So we will simply pile up extra fillings into the sandwich. The order of these is important: by getting in first, grammar for the instance takes priority; its immediate kind has next priority, and so on up the hierarchy.

```
void gv_compile(OUTPUT_STREAM, grammar_verb *gv) {
    if (gv->first_line == NULL) return;
    LOGIF(GRAMMAR, "Compiling grammar verb $G\n", gv);
    current_sentence = gv->where_gv_created;
    gl_reset_labels();
    switch(gv->gv_is) {
        case GV_IS_COMMAND:
            gv_compile_Verb_directive_header(OUT, gv);
            break;
        case GV_IS_TOKEN:
            gpr_compile_gpr_head(OUT, gv->allocation_id);
            break;
        case GV_IS_CONSULT:
            gpr_compile_consult_head(OUT, gv->allocation_id);
            break;
        case GV_IS_OBJECT:
            gpr_compile_parse_name_head(OUT, gv->wo_understood, gv);
            break;
        case GV_IS_VALUE:
            break;
        case GV_IS_PROPERTY_NAME:
            gpr_compile_prn_pr_head(OUT, gv->prn_understood);
            break;
    }
    if (gv->gv_is == GV_IS_OBJECT) gv_compile_parse_name_lines(OUT, gv);
    else gv_compile_lines(OUT, gv);
    switch(gv->gv_is) {
        case GV_IS_COMMAND:
```

```
            WRITE(";\n");
            break;
        case GV_IS_TOKEN:
            gpr_compile_gpr_tail(OUT);
            break;
        case GV_IS_CONSULT:
            gpr_compile_consult_tail(OUT);
            break;
        case GV_IS_OBJECT:
            gpr_compile_parse_name_tail(OUT);
            WRITE(";\n");
            break;
        case GV_IS_VALUE:
            break;
        case GV_IS_PROPERTY_NAME:
            gpr_compile_prn_pr_tail(OUT, gv->prn_understood);
            break;
    }
}
```

The function gv_compile is called from 13/gprv.

§**24.** The special thing about `GV_IS_OBJECT` grammars is that each is attached to a world object WO, and when we compile them we recurse up the kind hierarchy: thus if the red ball is of kind ball which is of kind thing, then the `parse_name` for the red ball consists of grammar lines specified for the red ball, then those specified for all balls, and lastly those specified for all things. (This mimics I6 class-to-instance inheritance.)

```
void gv_compile_parse_name_lines(OUTPUT_STREAM, grammar_verb *gv) {
    world_object *wo2 = gv->wo_understood;
    if (wo2->understand_as_this_object != gv)
        internal_error("link between WO and GV broken");
    LOGIF(GRAMMAR, "Parse_name content for $O:\n", wo2);
    gv_compile_lines(OUT, wo2->understand_as_this_object);
    wo2 = wo2->kind;
    while ((wo2) && (wo2 != kind_kind)) {
        if (wo2->understand_as_this_object) {
            LOGIF(GRAMMAR, "And parse_name content inherited from $O:\n", wo2);
            gv_compile_lines(OUT, wo2->understand_as_this_object);
        }
        wo2 = wo2->kind;
    }
}
```

§**25.** All other grammars are compiled just as they are:

```
void gv_compile_lines(OUTPUT_STREAM, grammar_verb *gv) {
    gl_list_assert_ownership(gv->first_line, gv);                    Mark for later indexing
    sort_grammar_verb(gv);                                  Phase III for the GLs in the GV happens here
    gl_sorted_line_list_compile(OUT, gv->sorted_first_line,
        gv->gv_is, gv, gv_is_genuinely_verbal(gv));                  And Phase IV here
}
```

§**26.** All of that was really Phase IV work (compiling), but a very little Phase III business also happens at this top level. Note that some grammars are compiled more than once (if a red ball and a blue ball are both of kind ball, then compiling grammars for them will also involve compiling grammars for the ball in each case: see above), so the following routine may well be called more than once for the same GV. We only want to sort once, though, so:

```
void sort_grammar_verb(grammar_verb *gv) {
    if (gv->sorted_first_line == NULL)
        gv->sorted_first_line = gl_list_sort(gv->first_line);
}
```

Grammar Lines 13/gl

Purpose

A grammar line is a list of tokens to specify a textual pattern. For example, the NI source for a grammar line might be `"take [something] out"`, which is a sequence of three tokens.

Template interpreter commands

```
4   {-routine:MistakeActionSub}
```

Definitions

¶1. A grammar line is in turn a sequence of tokens. If it matches, it will result in 0, 1 or 2 parameters, though only if the grammar verb owning the line is a genuine `GV_IS_COMMAND` command grammar will the case of 2 parameters be possible. (This is for text matching, say, "put X in Y": the objects X and Y are two parameters resulting.) And in that case (alone), there will also be a `resulting_action`.

A small amount of disjunction is allowed in a grammar line: for instance, "look in/inside/into [something]" consists of five tokens, but only three *lexemes*, basic units to be matched. (The first is "look", the second is "one out of in, inside or into", and the third is an object in scope.) In the following structure we cache the lexeme count since it is fiddly to calculate, and useful when sorting grammar lines into applicability order.

The individual tokens are stored simply as parse tree nodes of type `TOKEN_NT`, and are the children of the node `gl->tokens`, which is why (for now, anyway) there is no grammar token structure.

```
define UNCALCULATED_BONUS -1000000

typedef struct grammar_line {
    struct grammar_line *next_line;                    linked list in creation order
    struct grammar_line *sorted_next_line;             and in applicability order
    struct parse_node *where_grammar_specified;        where found in source
    int original_text;                                 the word number of the double-quoted grammar text...
    struct parse_node *tokens;                         ...which is parsed into this list of tokens
    int lexeme_count;                                  number of lexemes, or -1 if not yet counted
    int understand_when_w1, understand_when_w2;        only when this condition holds
    int pluralised;                                    GV_IS_OBJECT: refers in the plural
    struct action_name *resulting_action;              GV_IS_COMMAND: the action
    int reversed;                                      GV_IS_COMMAND: the two arguments are in reverse order
    int mistaken;                                      GV_IS_COMMAND: is this understood as a mistake?
    int mistake_response_wn;                           if so, reply with the text at this word number
    struct grammar_type gl_type;
    int suppress_compilation;                          has been compiled in a single I6 grammar token already?
    struct grammar_line *next_with_action;             used when indexing actions
    struct grammar_verb *belongs_to_gv;                similarly, used only in indexing
    int general_sort_bonus;                            temporary values used in grammar line sorting
    int understanding_sort_bonus;
    MEMORY_MANAGEMENT
} grammar_line;
```

The structure grammar_line is private to this section.

```
grammar_line *gl_new(int wn, action_name *ac,
    parse_node *token_list, int reversed, int pluralised) {
    grammar_line *gl;
    gl = CREATE(grammar_line);
    gl->original_text = wn;
    gl->resulting_action = ac;
    gl->belongs_to_gv = NULL;
    if (ac != NULL) act_add_gl(ac, gl);
    gl->mistaken = FALSE;
    gl->mistake_response_wn = -1;
    gl->next_with_action = NULL;
    gl->next_line = NULL;
    gl->tokens = token_list;
    gl->where_grammar_specified = current_sentence;
    gl->gl_type = gty_new(TRUE);
    gl->lexeme_count = -1;                                    no count made as yet
    gl->reversed = reversed;
    gl->pluralised = pluralised;
    gl->understand_when_w1 = -1;
    gl->understand_when_w2 = -1;
    gl->suppress_compilation = FALSE;
    gl->general_sort_bonus = UNCALCULATED_BONUS;
    gl->understanding_sort_bonus = UNCALCULATED_BONUS;
    return gl;
}
void log_grammar_line(grammar_line *gl) {
    LOG("<GL%d:$W>", gl->allocation_id,
        gl->tokens->word_ref1, gl->tokens->word_ref2);
}
void gl_set_single_type(grammar_line *gl, specification *gl_value) {
    gty_set_single_type(&(gl->gl_type), gl_value);
}
```

The function gl_new is called from 13/tfg.

The function log_grammar_line is called from 2/dl.

The function gl_set_single_type is called from 13/tfg.

§1. **GL lists.** Grammar lines are themselves generally stored in linked lists (belonging, for instance, to a GV). Here we add a GL to the back of a list.

```
int gl_list_length(grammar_line *list_head) {
    int c = 0;
    grammar_line *posn;
    for (posn = list_head; posn; posn = posn->next_line) c++;
    return c;
}
grammar_line *gl_list_add(grammar_line *list_head, grammar_line *new_gl) {
    new_gl->next_line = NULL;
    if (list_head == NULL) list_head = new_gl;
    else {
```

```
        grammar_line *posn = list_head;
        while (posn->next_line) posn = posn->next_line;
        posn->next_line = new_gl;
    }
    return list_head;
}
int gl_line_list_is_uncensored(grammar_line *list_head) {
    grammar_line *gl;
    for (gl = list_head; gl; gl = gl->next_line) {
        action_name *an = gl->resulting_action;
        if ((an) && (act_is_censored(an) == FALSE)) return TRUE;
    }
    return FALSE;
}
```

The function gl_list_length is called from 13/gv.
The function gl_list_add is called from 13/gv.
The function gl_line_list_is_uncensored is called from 13/gv.

§**2. Two special forms of grammar lines.** GLs can have either or both of two orthogonal special forms: they can be mistaken or conditional. (Mistakes only occur in command grammars, but conditional GLs can occur in any grammar.) GLs of this kind need special support, in that I6 general parsing routines need to be compiled for them to use as tokens: here's where that support is provided. The following step needs to take place before the command grammar (I6 `Verb` directives, etc.) is compiled because of I6's requirement that all GPRs be defined as routines prior to the `Verb` directive using them.

```
void gl_line_list_compile_condition_tokens(OUTPUT_STREAM, grammar_line *list_head) {
    grammar_line *gl;
    for (gl = list_head; gl; gl = gl->next_line) {
        gl_compile_condition_token_as_needed(OUT, gl);
        gl_compile_mistake_token_as_needed(OUT, gl);
    }
}
```

The function gl_line_list_compile_condition_tokens is called from 13/gv.

§**3. Conditional lines.** Some grammar lines take effect only when some circumstance holds: any I7 condition is valid to specify this, with the notation "Understand ... as ... when ...". Such GLs have an "understand when" set, as follows. They compile preceded by a match-no-text token which matches correctly if the condition holds and incorrectly if it fails. For instance, for a command grammar, we might have:

```
* Cond_Token_26 'draw' noun -> Draw

void gl_set_understand_when(grammar_line *gl, int w1, int w2) {
    gl->understand_when_w1 = w1;
    gl->understand_when_w2 = w2;
}
void gl_compile_condition_token_as_needed(OUTPUT_STREAM, grammar_line *gl) {
    if (gl->understand_when_w1 >= 0) {
        ph_stack_frame *phsf;
        phsf = phsf_create_nonphrase_stack_frame();
        OUT = begin_compiling_phrase(OUT);
        begin_code_blocks();
```

```
        specification *spec
            = parse_expression(gl->understand_when_w1, gl->understand_when_w2, CONDITION_EXPCON);
        if (validate_when(spec) == FALSE) {
            LOG("Offending SP of condition: ");
            LOG("Error at: $X", spec);
            current_sentence = gl->where_grammar_specified;
            sentence_problem(_P_(C13BadWhen),
                "the condition after 'when' makes no sense to me",
                "although otherwise this worked - it is only "
                "the part after 'when' which I can't follow.");
        } else {
            INDENT;
            WRITE("if ("); spec_compile(OUT, spec); WRITE(") return GPR_PREPOSITION;\n");
            WRITE("return GPR_FAIL;\n");
            OUTDENT; WRITE("];\n");
        }
        OUT = write_routine_header();
        WRITE("[ Cond_Token_%d ", gl->allocation_id);
        copy_compiled_phrase();
        end_code_blocks();
        phsf_remove_nonphrase_stack_frame();
    }
}
void gl_compile_extra_token_for_condition(OUTPUT_STREAM, grammar_line *gl,
    int gv_is, int current_label) {
    if (gl->understand_when_w1 >= 0) {
        if (gv_is == GV_IS_COMMAND)
            WRITE("Cond_Token_%d ", gl->allocation_id);
        else
            WRITE("if (Cond_Token_%d() == GPR_FAIL) jump Fail_%d;\n",
                gl->allocation_id, current_label);
    }
}
```

The function gl_set_understand_when is called from 13/tfg.

§4. **Mistakes.** These are grammar lines used in command GVs for commands which are accepted but only in order to print nicely worded rejections. A number of schemes were tried for this, for instance producing parser errors and setting `pe` to some high value, but the method now used is for a mistaken line to produce a successful parse at the I6 level, resulting in the (I6 only) action `##MistakeAction`. The tricky part is to send information to the I6 action routine `MistakeActionSub` indicating what the mistake was, exactly: we do this by including, in the I6 grammar, a token which matches empty text and returns a "preposition", so that it has no direct result, but which also sets a special global variable as a side-effect. Thus a mistaken line "act [thing]" comes out as something like:

```
* Mistake_Token_12 'act' noun -> MistakeAction
```

Since the I6 parser accepts the first command which matches, and since none of this can be recursive, the value of this variable at the end of I6 parsing is guaranteed to be the one set during the line causing the mistake.

```
void gl_set_mistake(grammar_line *gl, int wn) {
    gl->mistaken = TRUE;
    gl->mistake_response_wn = wn;
```

```
}
void gl_compile_mistake_token_as_needed(OUTPUT_STREAM, grammar_line *gl) {
    if (gl->mistaken) {
        WRITE("[ Mistake_Token_%d;\n", gl->allocation_id); INDENT;
        WRITE("if (actor ~= player) return GPR_FAIL;\n");
        WRITE("understand_as_mistake_number = %d;\n", 100 + gl->allocation_id);
        WRITE("return GPR_PREPOSITION;\n");
        OUTDENT; WRITE("];\n");
    }
}
void gl_compile_extra_token_for_mistake(OUTPUT_STREAM, grammar_line *gl,
    int gv_is, int current_label) {
    if (gl->mistaken) {
        if (gv_is == GV_IS_COMMAND)
            WRITE("Mistake_Token_%d ", gl->allocation_id);
        else
            internal_error("GLs may only be mistaken in command grammar");
    }
}
int gl_compile_result_of_mistake(OUTPUT_STREAM, grammar_line *gl) {
    if (gl->mistaken) { WRITE(" -> MistakeAction\n"); return TRUE; }
    return FALSE;
}
void compile_MistakeActionSub_routine(OUTPUT_STREAM) {
    WRITE("[ MistakeActionSub;\n"); INDENT;
    WRITE("switch(understand_as_mistake_number) {\n"); INDENT;
    grammar_line *gl;
    LOOP_OVER(gl, grammar_line)
        if (gl->mistaken) {
            if (gl->mistake_response_wn >= 0) {
                specification *spec;
                current_sentence = gl->where_grammar_specified;
                spec = parse_expression(
                    gl->mistake_response_wn, gl->mistake_response_wn,
                    VALUE_EXPCON);
                WRITE("%d: ParserError(", 100+gl->allocation_id);
                spec_compile(OUT, spec);
                WRITE(");\n");
            }
        }
    WRITE("default: \"I didn't understand that sentence.\";\n");
    OUTDENT; WRITE("}\n");
    WRITE("say__p = 1;\n");
    OUTDENT; WRITE("];\n");
}
```

The function gl_set_mistake is called from 13/tfg.

The function compile_MistakeActionSub_routine is invoked by a command in a `.i6t` template file.

§**5. Single word optimisation.** The grammars used to parse names of objects are normally compiled into `parse_name` routines. But the I6 parser also uses the `name` property, and it is advantageous to squeeze as much as possible into `name` and as little as possible into `parse_name`. The only possible candidates are grammar lines consisting of single unconditional words, as detected by the following routine:

```
int gl_contains_single_unconditional_word(grammar_line *gl) {
    parse_node *pn = gl->tokens->down;
    if ((pn)
        && (pn->next == NULL)
        && (pn_int_annotation(pn, slash_class_ANNOT) == 0)
        && (pn_int_annotation(pn, grammar_token_literal_ANNOT))
        && (gl->pluralised == FALSE)
        && (gl->understand_when_w1 < 0))
        return pn->word_ref1;
    return -1;
}
```

§**6.** This routine looks through a GL list and marks to suppress all those GLs consisting only of single unconditional words, which means they will not be compiled into a `parse_name` routine (or anywhere else). If the `of` file handle is set, then the words in question are printed as I6-style dictionary words to it. In practice, this is done when compiling the `name` property, so that a single scan achieves both the transfer into `name` and the exclusion from `parse_name` of affected GLs.

```
grammar_line *gl_list_take_out_one_word_grammar(OUTPUT_STREAM,
    grammar_line *list_head) {
    grammar_line *gl, *glp;
    for (gl = list_head, glp = NULL; gl; gl = gl->next_line) {
        int wn = gl_contains_single_unconditional_word(gl);
        if (wn >= 0) {
            if (OUT) {
                isn_compile_dictionary_word(OUT, lw_array[wn].lw_text, FALSE);
                WRITE(" ");
            }
            gl->suppress_compilation = TRUE;
        } else glp = gl;
    }
    return list_head;
}
```

The function gl_list_take_out_one_word_grammar is called from 13/gv.

§**7. Phase I: Slash Grammar.** Slashing is an activity carried out on a per-grammar-line basis, so to slash a list of GLs we simply slash each GL in turn.

```
void gl_line_list_slash(grammar_line *gl_head) {
    grammar_line *gl;
    for (gl = gl_head; gl; gl = gl->next_line) {
        slash_grammar_line(gl);
    }
}
```

The function gl_line_list_slash is called from 13/gv.

§**8.** Now the actual slashing process, which does not descend to tokens. We remove any slashes, and fill in positive numbers in the `qualifier` field corresponding to non-singleton equivalence classes. Thus "take up/in all washing/laundry/linen" begins as 10 tokens, three of them forward slashes, and ends as 7 tokens, with `qualifier` values 0, 1, 1, 0, 2, 2, 2, for four equivalence classes in turn. Each equivalence class is one lexical unit, or "lexeme", so the lexeme count is then 4.

```
void slash_grammar_line(grammar_line *gl) {
    parse_node *pn;
    int alternatives_group = 0;
    current_sentence = gl->where_grammar_specified;                    to report problems
    if (gl->tokens == NULL)
        internal_error("Null tokens on grammar");
    LOGIF(GRAMMAR_CONSTRUCTION, "Preparing grammar line:\n$T", gl->tokens);
    for (pn = gl->tokens->down; pn; pn = pn->next)
        pn_annotate_int(pn, slash_class_ANNOT, 0);
    for (pn = gl->tokens->down; pn; pn = pn->next) {
        if ((pn->next != NULL) && [[pn->next == FORWARDSLASH]]) {          slash follows:
            if (pn_int_annotation(pn, slash_class_ANNOT) == 0)
                alternatives_group++;                                  start new equiv class
            pn_annotate_int(pn, slash_class_ANNOT,
                alternatives_group);                                make two sides of slash equiv
            if (pn->next->next)
                pn_annotate_int(pn->next->next, slash_class_ANNOT, alternatives_group);
            pn->next = pn->next->next;                          excise the slash from the token list
        }
    }
    LOGIF(GRAMMAR_CONSTRUCTION, "Regrouped as:\n$T", gl->tokens);
    for (pn = gl->tokens->down; pn; pn = pn->next)
        if ((pn_int_annotation(pn, slash_class_ANNOT) > 0) &&
            (pn_int_annotation(pn, grammar_token_literal_ANNOT) == FALSE)) {
            sentence_problem(_P_(C13OverAmbitiousSlash),
                "the slash '/' can only be used between single literal words",
                "so 'underneath/under/beneath' is allowed but "
                "'beneath/[florid ways to say under]/under' isn't.");
            break;
        }
    gl->lexeme_count = 0;
```

```
    for (pn = gl->tokens->down; pn; pn = pn->next) {
        int i = pn_int_annotation(pn, slash_class_ANNOT);
        if (i > 0)
            while ((pn->next) && (pn_int_annotation(pn->next, slash_class_ANNOT) == i))
                pn = pn->next;
        gl->lexeme_count++;
    }
    LOGIF(GRAMMAR_CONSTRUCTION, "Slashed as:\n$T", gl->tokens);
}
```

§**9. Phase II: Determining Grammar.** Here there is substantial work to do both at the line list level and on individual lines, and the latter does recurse down to token level too.

The following routine calculates the type of the GL list as the union of the types of the GLs within it, where union means the narrowest type such that every GL in the list casts to it. We return null if there are no GLs in the list, or if the GLs all return null types, or if an error occurs. (Note that actions in command verb grammars are counted as null for this purpose, since a grammar used for parsing the player's commands is not also used to determine a value.)

```
specification *gl_line_list_determine(grammar_line *list_head,
    int depth, int gv_is, grammar_verb *gv, int genuinely_verbal) {
    grammar_line *gl;
    int first_flag = TRUE;
    specification *spec_union = NULL;
    LOGIF(GRAMMAR_CONSTRUCTION, "Determining GL list for $G\n", gv);
    for (gl = list_head; gl; gl = gl->next_line) {
        specification *spec_of_line =
            gl_determine(gl, depth, gv_is, gv, genuinely_verbal);
        if (first_flag) {                              initially no expectations: spec_union is meaningless
            spec_union = spec_of_line;                                          so we set it to the first result
            first_flag = FALSE;
            continue;
        }
        if ((spec_union == NULL) && (spec_of_line == NULL))
            continue;                                  we expected to find no result, and did: so no problem
        if ((spec_union) && (spec_of_line)) {
            if (can_we_match_value_descriptions(spec_union, spec_of_line) == ALWAYS_MATCH) {
                spec_union = spec_of_line;                          here spec_of_line was a wider type
                continue;
            }
            if (can_we_match_value_descriptions(spec_of_line, spec_union) == ALWAYS_MATCH) {
                continue;                                       here spec_union was already wide enough
            }
        }
        current_sentence = gl->where_grammar_specified;
        sentence_problem(_P_(C13MixedOutcome),
            "grammar tokens must have the same outcome whatever the way they are "
            "reached",
            "so writing a line like 'Understand \"within\" or \"next to "
            "[something]\" as \"[my token]\" must be wrong: one way it produces "
            "a thing, the other way it doesn't.");
        spec_union = NULL;
        break;                                 to prevent the problem being repeated for the same grammar
    }
    LOGIF(GRAMMAR_CONSTRUCTION, "Union: $S\n");
    return spec_union;
}
```

The function gl_line_list_determine is called from 13/gv.

§**10.** There are three tasks here: to determine the type of the GL, to issue a problem if this type is impossibly large, and to calculate two numerical quantities used in sorting GLs: the "general sorting bonus" and the "understanding sorting bonus" (see below).

```
specification *gl_determine(grammar_line *gl, int depth,
    int gv_is, grammar_verb *gv, int genuinely_verbal) {
    specification *spec = NULL;
    parse_node *pn, *pn2;
    int nulls_count, i, nrv, line_length;
    current_sentence = gl->where_grammar_specified;

    gl->understanding_sort_bonus = 0;
    gl->general_sort_bonus = 0;

    nulls_count = 0;                                        number of tokens with null results

    pn = gl->tokens->down;                                  start from first token
    if ((genuinely_verbal) && (pn)) pn = pn->next;          unless it's a command verb

    for (pn2=pn, line_length=0; pn2; pn2 = pn2->next) line_length++;

    for (i=0; pn; pn = pn->next, i++) {
        if (pn_get_node_type(pn) != TOKEN_NT)
            internal_error("Bogus node types on grammar");

        spec = gtok_determine(pn, depth);
        LOGIF(GRAMMAR_CONSTRUCTION, "Result of token <$W> is $S\n",
            pn->word_ref1, pn->word_ref2, spec);

        if (spec) {
            if ((spec_is_generic_CONSTANT(spec)) &&
                (is_kova(spec_get_kind_of_value(spec), UNDERSTANDING_TY))) {        "[text]" token
                int usb_contribution = i - 100;
                if (usb_contribution >= 0) usb_contribution = -1;
                usb_contribution = 100*usb_contribution + (line_length-1-i);
                gl->understanding_sort_bonus += usb_contribution;                   reduces!
            }
            gl->general_sort_bonus +=
                gty_add_type(&(gl->gl_type), spec, gtok_is_multiple(pn));
        } else nulls_count++;
    }

    nrv = gty_get_no_resulting_values(&(gl->gl_type));
    if (nrv == 0) gl->general_sort_bonus = 100*nulls_count;
    if (gv_is == GV_IS_COMMAND) spec = NULL;
    else {
        if (nrv < 2) spec = gty_get_single_type(&(gl->gl_type));
        else sentence_problem(_P_(C13TwoValuedToken),
            "there can be at most one varying part in the definition of a "
            "named token",
            "so 'Understand \"button [a number]\" as \"[button indication]\"' "
            "is allowed but 'Understand \"button [a number] on [something]\" "
            "as \"[button indication]\"' is not.");
    }

    LOGIF(GRAMMAR_CONSTRUCTION,
        "Determined $g: lexeme count %d, sorting bonus %d, arguments %d, "
        "fixed initials %d, type $S\n",
        gl, gl->lexeme_count, gl->general_sort_bonus, nrv,
        gl->understanding_sort_bonus, spec);
```

```
    return spec;
}
```

§**11. Phase III: Sort Grammar.** Insertion sort is used to take the linked list of GLs and construct a separate, sorted version. This is not the controversial part.

```
grammar_line *gl_list_sort(grammar_line *list_head) {
    grammar_line *gl, *gl2, *gl3, *sorted_head;
    if (list_head == NULL) return NULL;
    sorted_head = list_head;
    list_head->sorted_next_line = NULL;
    gl = list_head;
    while (gl->next_line) {
        gl = gl->next_line;
        gl2 = sorted_head;
        if (grammar_line_must_precede(gl, gl2)) {
            sorted_head = gl;
            gl->sorted_next_line = gl2;
            continue;
        }
        while (gl2) {
            gl3 = gl2;
            gl2 = gl2->sorted_next_line;
            if (gl2 == NULL) {
                gl3->sorted_next_line = gl;
                break;
            }
            if (grammar_line_must_precede(gl, gl2)) {
                gl3->sorted_next_line = gl;
                gl->sorted_next_line = gl2;
                break;
            }
        }
    }
    return sorted_head;
}
```

The function gl_list_sort is called from 13/gv.

§**12.** This is the controversial part: the routine which decides whether one GL takes precedence (i.e., is parsed earlier than and thus in preference to) another GL. This algorithm has been hacked many times to try to reach a position which pleases all designers: something of a lost cause. The basic motivation is that we need to sort because the various parsers of I7 grammar (`parse_name` routines, general parsing routines, the I6 command parser itself) all work by returning the first match achieved. This means that if grammar line L2 matches a superset of the texts which grammar line L1 matches, then L1 should be tried first: trying them in the order L2, L1 would mean that L1 could never be matched, which is surely contrary to the designer's intention. (Compare the rule-sorting algorithm, which has similar motivation but is entirely distinct, though both use the same primitive methods for comparing types of single values, i.e., at stages 5b1 and 5c1 below.)

Recall that each GL has a numerical USB (understanding sort bonus) and GSB (general sort bonus). The following rules are applied in sequence:

(1) Higher USBs beat lower USBs.

(2a) For sorting GLs in player-command grammar, shorter lines beat longer lines, where length is calculated as the lexeme count.

(2b) For sorting all other GLs, longer lines beat shorter lines.

(3) Mistaken commands beat unmistaken commands.

(4) Higher GSBs beat lower GSBs.

(5a) Fewer resulting values beat more resulting values.

(5b1) A narrower first result type beats a wider first result type, if there is a first result.

(5b2) A multiples-disallowed first result type beats a multiples-allowed first result type, if there is a first result.

(5c1) A narrower second result type beats a wider second result type, if there is a second result.

(5c2) A multiples-disallowed second result type beats a multiples-allowed second result type, if there is a second result.

(6) Conditional lines (with a "when" proviso, that is) beat unconditional lines.

(7) The grammar line defined earlier beats the one defined later.

Rule 1 is intended to resolve awkward ambiguities involved with command grammar which includes "[text]" tokens. Each such token subtracts 10000 from the USB of a line but adds back 100 times the token position (which is at least 0 and which we can safely suppose is less than 99: we truncate just in case so that every `"[text]"` certainly makes a negative contribution of at least −100) and then subtracts off the number of tokens left on the line.

Because a high USB gets priority, and "[text]" tokens make a negative contribution, the effect is to relegate lines containing "[text]" tokens to the bottom of the list – which is good because "[text]" voraciously eats up words, matching more or less anything, so that any remotely specific case ought to be tried first. The effect of the curious addition back in of the token position is that later-placed "[text]" tokens are tried before earlier-placed ones. Thus `"read chapter [text]"` has a USB of −98, and takes precedence over `"read [text]"` with a USB of −99, but both are beaten by just `"read [something]"` with a USB of 0. The effect of the subtraction back of the number of tokens remaining is to ensure that `"read [token] backwards"` takes priority over `"read [token]"`.

The voracity of `"[text]"`, and its tendency to block out all other possibilities unless restrained, has to be addressed by this lexically based numerical calculation because it works in a lexical sort of way: playing with the types system to prefer `DESCRIPTION/UNDERSTANDING` over, say, `VALUE/OBJECT` would not be sufficient.

The most surprising point here is the asymmetry in rule 2, which basically says that when parsing commands typed at the keyboard, shorter beats longer, whereas in all other settings longer beats shorter. This arises because the I6 parser, at run time, traditionally works that way: I6 command grammars are normally stored with short forms first and long forms afterward. The I6 parser can afford to do this because it is matching text of known length: if parsing TAKE FROG FROM AQUARIUM, it will try TAKE FROG first but is able to reject this as not matching the whole text. In other parsing settings, we are trying to make a maximum-length match against a potentially infinite stream of words, and it is therefore important to try to match WATERY CASCADE EFFECT before WATERY CASCADE when looking at text like WATERY CASCADE EFFECT IMPRESSES PEOPLE, given that the simplistic parsers we compile generally return the first match found.

Rule 3, that mistakes beat non-mistakes, was in fact rule 1 during 2006: it seemed logical that since mistakes were exceptional cases, they would be better checked earlier before moving on to general cases. However, an example provided by Eric Eve showed that although this was logically correct, the I6 parser would try to auto-complete lengthy mistakes and thus fail to check subsequent commands. For this reason, `"look behind [something]"` as a mistake needs to be checked after `"look"`, or else the I6 parser will respond to the command LOOK by replying "What do you want to look behind?" – and then saying that you are mistaken.

Rule 4 is intended as a lexeme-based tiebreaker. We only get here if there are the same number of lexemes in the two GLs being compared. Each is given a GSB score as follows: a literal lexeme, which produces no result, such as `"draw"` or `"in/inside/within"`, scores 100; all other lexemes score as follows:

– `"[things inside]"` scores a GSB of 10 as the first parameter, 1 as the second;

– `"[things preferably held]"` similarly scores a GSB of 20 or 2;

– `"[other things]"` similarly scores a GSB of 20 or 2;

– `"[something preferably held]"` similarly scores a GSB of 30 or 3;

– any token giving a logical description of some class of objects, such as `"[open container]"`, similarly scores a GSB of 50 or 5;

– and any remaining token (for instance, one matching a number or some other kind of value) scores a GSB of 0.

Literals score highly because they are structural, and differentiate cases: under the superset rule, `"look up [thing]"` must be parsed before `"look [direction] [thing]"`, and it is only the number of literals which differentiates these cases. If two lines have an equal number of literals, we now look at the first resultant lexeme. Here we find that a lexeme which specifies an object (with a GSB of at least 10/1) beats a lexeme which only specifies a value. Thus the same text will be parsed against objects in preference to values, which is sensible since there are generally few objects available to the player and they are generally likely to be the things being referred to. Among possible object descriptions, the very general catch-all special cases above are given lower GSB scores than more specific ones, to enable the more specific cases to go first.

Rule 5a is unlikely to have much effect: it is likely to be rare for GL lists to contain GLs mixing different numbers of results. But Rule 5b1 is very significant: it causes `"draw [animal]"` to have precedence over `"draw [thing]"`, for instance. Rule 5b2 ensures that `"draw [thing]"` takes precedence over `"draw [things]"`, which may be useful to handle multiple and single objects differently.

The motivation for rule 6 is similar to the case of "when" clauses for rules in rulebooks: it ensures that a match of `"draw [thing]"` when some condition holds beats a match of `"draw [thing]"` at any time, and this is necessary under the strict superset principle.

To get to rule 7 looks difficult, given the number of things about the grammar lines which must match up – same USB, GSB, number of lexemes, number of resulting types, equivalent resulting types, same conditional status – but in fact it isn't all that uncommon. Equivalent pairs produced by the Standard Rules include:

`"get off [something]"` and `"get in/into/on/onto [something]"`

`"turn on [something]"` and `"turn [something] on"`

Only the second of these pairs leads to ambiguity, and even then only if an object has a name like ON VISION ON – perhaps a book about the antique BBC children's television programme *Vision On* – so that the command TURN ON VISION ON would match both of the alternative GLs.

```
int grammar_line_must_precede(grammar_line *L1, grammar_line *L2) {
    int cs, a, b;
    if ((L1 == NULL) || (L2 == NULL))
        internal_error("tried to sort null GLs");
    if ((L1->lexeme_count == -1) || (L2->lexeme_count == -1))
        internal_error("tried to sort unslashed GLs");
    if ((L1->general_sort_bonus == UNCALCULATED_BONUS) ||
        (L2->general_sort_bonus == UNCALCULATED_BONUS))
        internal_error("tried to sort uncalculated GLs");
    if (L1 == L2) return FALSE;
    a = FALSE; if ((L1->resulting_action) || (L1->mistaken)) a = TRUE;
    b = FALSE; if ((L2->resulting_action) || (L2->mistaken)) b = TRUE;
    if (a != b) internal_error("tried to sort on incomparable GLs");
    if (L1->understanding_sort_bonus > L2->understanding_sort_bonus) return TRUE;
    if (L1->understanding_sort_bonus < L2->understanding_sort_bonus) return FALSE;
    if (a) {                                          command grammar: shorter beats longer
        if (L1->lexeme_count < L2->lexeme_count) return TRUE;
        if (L1->lexeme_count > L2->lexeme_count) return FALSE;
    } else {                                          all other grammars: longer beats shorter
        if (L1->lexeme_count < L2->lexeme_count) return FALSE;
        if (L1->lexeme_count > L2->lexeme_count) return TRUE;
    }
```

```
    if ((L1->mistaken) && (L2->mistaken == FALSE)) return TRUE;
    if ((L1->mistaken == FALSE) && (L2->mistaken)) return FALSE;

    if (L1->general_sort_bonus > L2->general_sort_bonus) return TRUE;
    if (L1->general_sort_bonus < L2->general_sort_bonus) return FALSE;

    cs = gtk_must_precede(&(L1->gl_type), &(L2->gl_type));
    if (cs != NOT_APPLICABLE) return cs;

    if ((L1->understand_when_w1 >= 0) && (L2->understand_when_w1 < 0)) return TRUE;
    if ((L1->understand_when_w1 < 0) && (L2->understand_when_w1 >= 0)) return FALSE;

    return FALSE;
}
```

§13. Phase IV: Compile Grammar. At this level we compile the list of GLs in sorted order: this is what the sorting was all for. In certain cases, we skip any GLs marked as "one word": these are cases arising from, e.g., "Understand "frog" as the toad.", where we noticed that the GL was a single word and included it in the `name` property instead. This is faster and more flexible, besides writing tidier code.

The need for this is not immediately obvious. After all, shouldn't we have simply deleted the GL in the first place, rather than leaving it in but marking it? The answer is no, because of the way inheritance works: values of the `name` property accumulate from class to instance in I6, since `name` is additive, but grammar doesn't.

```
void gl_sorted_line_list_compile(OUTPUT_STREAM, grammar_line *list_head,
    int gv_is, grammar_verb *gv, int genuinely_verbal) {
    grammar_line *gl;
    INDENT;
    for (gl = list_head; gl; gl = gl->sorted_next_line)
        if (gl->suppress_compilation == FALSE)
            compile_grammar_line(OUT, gl, gv_is, gv, genuinely_verbal);
    OUTDENT;
}
```

The function gl_sorted_line_list_compile is called from 13/gv.

§14. The following apparently global variables are used to provide a persistent state for the routine below, but are not accessed elsewhere. The label counter is reset at the start of each GV's compilation, though this is a purely cosmetic effect.

```
int current_grammar_block = 0;
int current_label = 1;
void gl_reset_labels(void) {
    current_label = 1;
}
```

The function gl_reset_labels is called from 13/gv.

§15. As fancy as the following routine may look, it contains very little. What complexity there is comes from the fact that command GVs are compiled very differently to all others (most grammars are compiled in "code mode", generating procedural I6 statements, but command GVs are compiled to lines in `Verb` directives) and that GLs resulting in actions (i.e., GLs in command GVs) have not yet been type-checked, whereas all others have.

```
void compile_grammar_line(OUTPUT_STREAM, grammar_line *gl, int gv_is, grammar_verb *gv,
    int genuinely_verbal) {
    parse_node *pn;
    int i;
    int token_values;
    kind_of_value *token_value_kovs[2];
    int lexeme_equivalence_class;
    int code_mode, consult_mode;
    LOGIF(GRAMMAR, "Compiling grammar line: $g\n", gl);
    current_sentence = gl->where_grammar_specified;
    if (gv_is == GV_IS_COMMAND) code_mode = FALSE; else code_mode = TRUE;
    if (gv_is == GV_IS_CONSULT) consult_mode = TRUE; else consult_mode = FALSE;
    switch (gv_is) {
        case GV_IS_COMMAND:
```

```
    case GV_IS_TOKEN:
    case GV_IS_CONSULT:
    case GV_IS_OBJECT:
    case GV_IS_VALUE:
    case GV_IS_PROPERTY_NAME:
        break;
    default: internal_error("tried to compile unknown GV type");
}

current_grammar_block++;
token_values = 0;
for (i=0; i<2; i++) token_value_kovs[i] = NULL;

if (code_mode == FALSE) WRITE("* ");

gl_compile_extra_token_for_condition(OUT, gl, gv_is, current_label);
gl_compile_extra_token_for_mistake(OUT, gl, gv_is, current_label);

pn = gl->tokens->down;
if ((genuinely_verbal) && (pn)) {
    if (pn_int_annotation(pn, slash_class_ANNOT) != 0) {
        sentence_problem(_P_(C13SlashedCommand),
            "at present you're not allowed to use a / between command "
            "words at the start of a line",
            "so 'put/interpose/insert [something]' is out.");
        return;
    }
    pn = pn->next;                                  skip command word: the Verb header contains it already
}

lexeme_equivalence_class = 0;
int alternative_number = 0;
for (; pn; pn = pn->next) {
    char failure_label[32];
    kind_of_value *token_kov;
    int first_token_in_lexeme, last_token_in_lexeme;

    if ((pn_get_grammar_token_relation(pn)) && (gv_is != GV_IS_OBJECT)) {
        sentence_problem(_P_(C13GrammarObjectlessRelation),
            "a grammar token in an 'Understand...' can only be based "
            "on a relation if it is to understand the name of a room or thing",
            "since otherwise there is nothing for the relation to be with.");
        continue;
    }

    if (pn_int_annotation(pn, slash_class_ANNOT) != 0) {                    in a multi-token lexeme
        first_token_in_lexeme = FALSE;
        last_token_in_lexeme = FALSE;
        if ((pn->next == NULL) ||
            (pn_int_annotation(pn->next, slash_class_ANNOT) !=
                pn_int_annotation(pn, slash_class_ANNOT)))
            last_token_in_lexeme = TRUE;
        if (pn_int_annotation(pn, slash_class_ANNOT) != lexeme_equivalence_class)
            first_token_in_lexeme = TRUE;
        lexeme_equivalence_class = pn_int_annotation(pn, slash_class_ANNOT);
        if (first_token_in_lexeme) alternative_number = 1;
        else alternative_number++;
    } else {                                                                in a single-token lexeme
        lexeme_equivalence_class = 0;
```

```
        first_token_in_lexeme = TRUE;
        last_token_in_lexeme = TRUE;
        alternative_number = 1;
    }
    sprintf(failure_label, "Fail_%d", current_label);
    if (lexeme_equivalence_class > 0) {
        if (code_mode) {
            if (first_token_in_lexeme) WRITE("group_wn = wn;\n");
            WRITE(".group_%d_%d_%d; wn = group_wn;\n",
                current_grammar_block, lexeme_equivalence_class, alternative_number);
            if (last_token_in_lexeme == FALSE)
                sprintf(failure_label, "group_%d_%d_%d",
                    current_grammar_block,
                    lexeme_equivalence_class, alternative_number+1);
        }
    }
    token_kov = gtok_compile(OUT, pn, code_mode, failure_label, consult_mode);
    if (token_kov) {
        if (token_values == 2) {
            internal_error(
                "There can be at most two value-producing tokens and this "
                "should have been detected earlier.");
            return;
        }
        token_value_kovs[token_values++] = token_kov;
    }
    if (lexeme_equivalence_class > 0) {
        if (code_mode) {
            if (last_token_in_lexeme) {
                WRITE(".group_%d_%d_end;\n",
                    current_grammar_block, lexeme_equivalence_class);
            } else {
                WRITE("jump group_%d_%d_end;\n",
                    current_grammar_block, lexeme_equivalence_class);
            }
        } else {
            if (last_token_in_lexeme == FALSE) WRITE(" /");
        }
    }
    if (code_mode == FALSE) WRITE(" ");                    Verb tokens divide by spaces
}
switch (gv_is) {
    case GV_IS_COMMAND:
        if (gl_compile_result_of_mistake(OUT, gl)) break;
        WRITE(" -> %s", an_get_I6_representation(gl->resulting_action));
        if (gl->reversed) {
            kind_of_value *swap = token_value_kovs[0];
            token_value_kovs[0] = token_value_kovs[1];
            token_value_kovs[1] = swap;
            WRITE(" reverse");
        }
```

```
            WRITE("\n");
            act_check_types_for_grammar(gl->resulting_action, token_values,
                token_value_kovs);
            break;
        case GV_IS_PROPERTY_NAME:
        case GV_IS_TOKEN:
            WRITE("return rv;\n");
            WRITE(".Fail_%d; rv = GPR_PREPOSITION; wn = original_wn;\n",
                current_label);
            break;
        case GV_IS_CONSULT:
            WRITE("if ((range_words==0) || (wn-range_from==range_words)) return rv;\n");
            WRITE(".Fail_%d; rv = GPR_PREPOSITION; wn = original_wn;\n", current_label);
            break;
        case GV_IS_OBJECT:
            gpr_after_gl_failed(OUT, current_label, gl->pluralised);
            break;
        case GV_IS_VALUE:
            WRITE("parsed_number = ");
            gty_compile_to_string(OUT, &(gl->gl_type)); WRITE(";\n");
            WRITE("return GPR_NUMBER;\n");
            WRITE(".Fail_%d; wn = original_wn;\n", current_label);
            break;
    }
    current_label++;
    return;
}
```

§16. Indexing by grammar. This is the more obvious form of indexing: we show the grammar lines which make up an individual GL. (For instance, this is used in the Actions index to show the grammar for an individual command word, by calling the routine below for that command word's GV.) Such an index list is done in sorted order, so that the order of appearance in the index corresponds to the order of parsing – this is what the reader of the index is interested in.

```
void gl_sorted_list_index_normal(grammar_line *list_head, char *headword) {
    grammar_line *gl;
    for (gl = list_head; gl; gl = gl->sorted_next_line)
        gl_index_normal(gl, headword);
}
void gl_index_normal(grammar_line *gl, char *headword) {
    action_name *an = gl->resulting_action;
    if ((an == NULL) || (act_is_censored(an))) return;
    index_anchor(headword);
    if (act_is_out_of_world(an))
        INDEX("<font color=\"#800000\">");
    INDEX("&quot;");
    index_verb_definition(lw_array[gl->original_text].lw_text, headword);
    INDEX("&quot;");
    index_link(lw_array[gl->original_text].lw_source);
    INDEX(" - <i>");
    print_raw_text_to_file(an->word_ref1, an->word_ref2, ifl);
    index_below_link(an_get_I6_representation(an));
    index_detail_link("A", an->allocation_id, TRUE);
    if (gl->reversed) INDEX(" (reversed)");
    INDEX("</i>");
    if (act_is_out_of_world(an))
        INDEX("</font>");
    INDEX("<br>");
}
```

The function gl_sorted_list_index_normal is called from 13/gv.

§17. Indexing by action. Grammar lines are typically indexed twice: the other time is when all grammar lines belonging to a given action are tabulated. Special linked lists are kept for this purpose, and this is where we unravel them and print to the index. The question of sorted vs unsorted is meaningless here, since the GLs appearing in such a list will typically belong to several different GVs. (As it happens, they appear in order of creation, i.e., in source text order.)

Tiresomely, all of this means that we need to store "uphill" pointers in GLs: back up to the GVs that own them. The following routine does this for a whole list of GLs:

```
void gl_list_assert_ownership(grammar_line *list_head, grammar_verb *gv) {
    grammar_line *gl;
    for (gl = list_head; gl; gl = gl->next_line)
        gl->belongs_to_gv = gv;
}
```

The function gl_list_assert_ownership is called from 13/gv.

§**18.** And this routine accumulates the per-action lists of GLs:

```
void gl_list_with_action_add(grammar_line *list_head, grammar_line *gl) {
    if (list_head == NULL) internal_error("tried to add to null action list");
    while (list_head->next_with_action)
        list_head = list_head->next_with_action;
    list_head->next_with_action = gl;
}
```

The function gl_list_with_action_add is called from 11/act.

§**19.** Finally, here we index an action list of GLs, each getting a line in the HTML index.

```
int gl_index_list_with_action(grammar_line *gl) {
    int said_something = FALSE;
    while (gl != NULL) {
        if (gl->belongs_to_gv) {
            int vwn = gv_get_verb_wn(gl->belongs_to_gv);
            if (vwn != -1) {
                open_html_paragraph(ifl, 2, "hanging");
                INDEX("&quot;");
                index_verb_definition(lw_array[gl->original_text].lw_text,
                    lw_array[vwn].lw_text);
                INDEX("&quot;");
                index_link(lw_array[gl->original_text].lw_source);
                if (gl->reversed) INDEX(" <i>reversed</i>");
                INDEX("</p>");
                said_something = TRUE;
            }
        }
        gl = gl->next_with_action;
    }
    return said_something;
}
```

The function gl_index_list_with_action is called from 11/act.

Grammar Types 13/gty

Purpose

Some grammar text specifies one or more values, and we need to keep track of their data type(s). Here we manage the data structure doing this.

Definitions

```
typedef struct grammar_type {
    int no_resulting_values;                    number of resulting values: 0, 1 or 2; or -1
    struct specification *first_type;                                  and their types
    struct specification *second_type;
    int first_multiplicity;                        lines only: allow a multiple object here?
    int second_multiplicity;                       lines only: allow a multiple object here?
} grammar_type;
```

The structure grammar_type is private to this section.

```
grammar_type gty_new(int supports_return_type) {
    grammar_type gty;
    gty.first_type = NULL;
    gty.second_type = NULL;
    gty.first_multiplicity = FALSE;
    gty.second_multiplicity = FALSE;
    if (supports_return_type)
        gty.no_resulting_values = 0;
    else
        gty.no_resulting_values = -1;
    return gty;
}
```

The function gty_new is called from 13/gv and 13/gl.

§**1.** The multiplication by 10 here is explained in the discussion of GSB tallying during GL sorting in "Grammar Lines". Do not amend it without changing that discussion.

```
int gty_add_type(grammar_type *gty, specification *spec,
    int multiple_flag) {
    switch((gty->no_resulting_values)++) {
        case 0:
            gty->first_type = spec;
            gty->first_multiplicity = multiple_flag;
            return 10*(spec_get_data(spec, GRAMMAR_TOKEN_SCORE_SPDATA));
        case 1:
            gty->second_type = spec;
            gty->second_multiplicity = multiple_flag;
            return spec_get_data(spec, GRAMMAR_TOKEN_SCORE_SPDATA);
        case 2:
            sentence_problem(_P_(C13ThreeValuedLine),
                "there can be at most two varying parts to a line of grammar",
                "so 'put [something] in [a container]' is allowed but 'put "
                "[something] in [something] beside [a door]' is not.");
```

```
    }
    return 0;
}

int gty_has_return_type(grammar_type *gty) {
    if (gty->no_resulting_values == -1) return FALSE;
    return TRUE;
}

int gty_get_no_resulting_values(grammar_type *gty) {
    return gty->no_resulting_values;
}

specification *gty_get_single_type(grammar_type *gty) {
    switch(gty->no_resulting_values) {
        case 0: return NULL;
        case 1: return gty->first_type;
        default: internal_error("gty improperly typed");
    }
    return NULL;
}

void gty_set_single_type(grammar_type *gty, specification *spec) {
    if (spec == NULL) gty->no_resulting_values = 0;
    else {
        gty->no_resulting_values = 1;
        gty->first_type = spec;
    }
}

void gty_compile_to_string(OUTPUT_STREAM, grammar_type *gty) {
    spec_compile(OUT, gty->first_type);
}

kind_of_value *gty_get_data_type_as_token(grammar_type *gty) {
    if (gty->no_resulting_values > 0) {
        if ((species_is(gty->first_type, CONSTANT_SPC)) ||
            (spec_is_generic_CONSTANT(gty->first_type)))
            return spec_get_kind_of_value(gty->first_type);
    }
    return NULL;
}
```

The function gty_add_type is called from 13/gl.

The function gty_has_return_type is called from 13/gv.

The function gty_get_no_resulting_values is called from 13/gl.

The function gty_get_single_type is called from 13/gv and 13/gl.

The function gty_set_single_type is called from 13/gv and 13/gl.

The function gty_compile_to_string is called from 13/gl.

The function gty_get_data_type_as_token is called from 13/gv.

§**2.** The behaviour of this sorting routine is documented in the discussion of GL sorting in "Grammar Lines". Do not amend it without changing that discussion.

```
int gtk_must_precede(grammar_type *gty1, grammar_type *gty2) {
    int cs;
    if ((gty1->no_resulting_values) < (gty2->no_resulting_values)) return TRUE;
    if ((gty1->no_resulting_values) > (gty2->no_resulting_values)) return FALSE;
    if (gty1->no_resulting_values == 0) return NOT_APPLICABLE;
    cs = compare_specificity_of_spec(gty1->first_type, gty2->first_type);
    if (cs == 1) return TRUE;
    if (cs == -1) return FALSE;
    if ((gty1->first_multiplicity) && (gty2->first_multiplicity == FALSE))
        return FALSE;
    if ((gty1->first_multiplicity == FALSE) && (gty2->first_multiplicity))
        return TRUE;
    if (gty1->no_resulting_values == 1) return NOT_APPLICABLE;
    cs = compare_specificity_of_spec(gty1->second_type, gty2->second_type);
    if (cs == 1) return TRUE;
    if (cs == -1) return FALSE;
    if ((gty1->second_multiplicity) && (gty2->second_multiplicity == FALSE))
        return FALSE;
    if ((gty1->second_multiplicity == FALSE) && (gty2->second_multiplicity))
        return TRUE;
    return NOT_APPLICABLE;
}
```

The function gtk_must_precede is called from 13/gl.

Grammar Tokens 13/gtok

Purpose

To handle grammar at the level of individual tokens. I7 grammar tokens correspond in a 1-to-1 way with I6 tokens: here we determine the I7 type a token represents (if any) and compile it to its I6 grammar token equivalent as needed.

13/gtok.§2 Multiple tokens; §3-7 The special tokens; §8 Phase II: Determining Grammar; §9 Phase IV: Compiling Grammar

---

Definitions

¶1. I7 tokens are (at present) stored simply as parse tree nodes of type `TOKEN_NT`, with meaningful information hidden in annotations. At one time I thought this was a simple arrangement, but it now seems obfuscatory, so at some point I plan to create a "grammar token" structure to avoid these arcane annotations of the parse tree.

`grammar_token_nonliteral_ANNOT` is clear for literal words such as `"into"` and set for square-bracketed tokens such as `"[something]"`.

`index` stores the GSB scoring contribution made by the token to the GL sorting algorithm.

The `grammar_token_code_ANNOT` annotation is meaningful only for parse nodes with an evaluation of type `DESCRIPTION`. These are tokens which describe a range of objects. Examples include "[open container]", which compiles to an I6 noun filter, "[any container]", which compiles to an I6 scope filter, or "[things]", one of a small number of special cases compiling to primitive I6 parser tokens. The annotation holds the allocation ID for the noun/scope filter structure built for the occasion in the former cases, and one of the following constants in the latter case. (These must all have negative values in order not to clash with allocation IDs 0, 1, 2, ..., and clearly must all be different, but otherwise the values are not significant and there is no preferred order.)

For tokens with any other evaluation, `general_purpose` is always 0, so that the special values below cannot arise.

define `NOUN_TOKEN_GTC -1` — *I6* `noun`
define `MULTI_TOKEN_GTC -2` — *I6* `multi`
define `MULTIINSIDE_TOKEN_GTC -3` — *I6* `multiinside`
define `MULTIHELD_TOKEN_GTC -4` — *I6* `multiheld`
define `HELD_TOKEN_GTC -5` — *I6* `held`
define `CREATURE_TOKEN_GTC -6` — *I6* `creature`
define `TOPIC_TOKEN_GTC -7` — *I6* `topic`
define `MULTIEXCEPT_TOKEN_GTC -8` — *I6* `multiexcept`

---

§**1.** Tokens are created when text such as "drill [something] with [something]" is parsed, from an Understand sentence or elsewhere. The following routine builds a token list in the parse tree as children of `pn`:

```
void gtok_break_into_tokens(parse_node *pn, int w1, int w2) {
    int i;
    parse_node *newpn;
    if ((i = is_word_intermediate(COMMA_V, w1, w2))>=0) {
        gtok_break_into_tokens(pn, w1, i-1);
        gtok_break_into_tokens(pn, i+1, w2);
        return;
    }
    if (is_kova(is_a_literal(w1, w2), TEXT_TY)) {
```

```
        dequote_word(w1);
        if (*(lw_array[w1].lw_text) == 0) return;
        feed_into_lexer(lw_array[w1].lw_text, FALSE, TRUE);
        w1 = lexer_feed_w1; w2 = lexer_feed_w2;
        for (i=w1; i<=w2; i++) {
            newpn = new_nounphrase_raw(i, i);
            pn_set_node_type(newpn, TOKEN_NT);
            pn_annotate_int(newpn, grammar_token_literal_ANNOT, TRUE);
            graft(newpn, pn);
        }
        return;
    }
    newpn = new_nounphrase_raw(w1, w2);
    pn_set_node_type(newpn, TOKEN_NT);
    pn_annotate_int(newpn, grammar_token_literal_ANNOT, FALSE);
    graft(newpn, pn);
}

int gtok_is_literal(parse_node *pn) {
    return pn_int_annotation(pn, grammar_token_literal_ANNOT);
}
```

The function gtok_break_into_tokens is called from 13/tfg.
The function gtok_is_literal is called from 13/tfg.

§2. Multiple tokens.

A multiple token is one which permits multiple matches in the I6 parser: for instance, permits the use of "all".

```
int gtok_is_multiple(parse_node *pn) {
    switch (pn_int_annotation(pn, grammar_token_code_ANNOT)) {
        case MULTI_TOKEN_GTC:
        case MULTIINSIDE_TOKEN_GTC:
        case MULTIHELD_TOKEN_GTC:
        case MULTIEXCEPT_TOKEN_GTC:
            return TRUE;
    }
    return FALSE;
}
```

The function gtok_is_multiple is called from 13/gl.

§**3. The special tokens.** Do not change any of these GTC numbers without first checking and updating the discussion of GL sorting in Grammar Lines:

```
int gsb_for_special_token(int gtc) {
    switch(gtc) {
        case NOUN_TOKEN_GTC: return 0;
        case MULTI_TOKEN_GTC: return 0;
        case MULTIINSIDE_TOKEN_GTC: return 1;
        case MULTIHELD_TOKEN_GTC: return 2;
        case HELD_TOKEN_GTC: return 3;
        case CREATURE_TOKEN_GTC: return 0;
        case TOPIC_TOKEN_GTC: return -1;
        case MULTIEXCEPT_TOKEN_GTC: return 2;
        default: internal_error("tried to find GSB for invalid GTC");
    }
    return 0;                                                    to prevent a gcc error: never reached
}
```

§**4.** These translate into I6 as follows:

```
char *i6_token_for_special_token(int gtc) {
    switch(gtc) {
        case NOUN_TOKEN_GTC: return "noun";
        case MULTI_TOKEN_GTC: return "multi";
        case MULTIINSIDE_TOKEN_GTC: return "multiinside";
        case MULTIHELD_TOKEN_GTC: return "multiheld";
        case HELD_TOKEN_GTC: return "held";
        case CREATURE_TOKEN_GTC: return "creature";
        case TOPIC_TOKEN_GTC: return "topic";
        case MULTIEXCEPT_TOKEN_GTC: return "multiexcept";
        default: internal_error("tried to find I6 token for invalid GTC");
    }
    return "";                                                   to prevent a gcc error: never reached
}
char *i6_constant_for_special_token(int gtc) {
    switch(gtc) {
        case NOUN_TOKEN_GTC: return "NOUN_TOKEN";
        case MULTI_TOKEN_GTC: return "MULTI_TOKEN";
        case MULTIINSIDE_TOKEN_GTC: return "MULTIINSIDE_TOKEN";
        case MULTIHELD_TOKEN_GTC: return "MULTIHELD_TOKEN";
        case HELD_TOKEN_GTC: return "HELD_TOKEN";
        case CREATURE_TOKEN_GTC: return "CREATURE_TOKEN";
        case TOPIC_TOKEN_GTC: return "TOPIC_TOKEN";
        case MULTIEXCEPT_TOKEN_GTC: return "MULTIEXCEPT_TOKEN";
        default: internal_error("tried to find I6 constant for invalid GTC");
    }
    return "";                                                   to prevent a gcc error: never reached
}
```

§**5.** The special tokens all return a value in I6 which needs a data type to be used in I7: these are defined by the following routine.

```
kind_of_value *data_type_for_special_token(int gtc) {
    if (gtc == TOPIC_TOKEN_GTC) return kova(UNDERSTANDING_TY);
    return kova(OBJECT_TY);
}
```

§**6.** The I7 equivalent names for these special tokens are as follows:

```
int parse_special_token(int w1, int w2) {
    if [[w1, w2 == something]] return NOUN_TOKEN_GTC;
    if [[w1, w2 == anything]] return NOUN_TOKEN_GTC;
    if [[w1, w2 == things]] return MULTI_TOKEN_GTC;
    if [[w1, w2 == things inside]] return MULTIINSIDE_TOKEN_GTC;
    if [[w1, w2 == things preferably held]] return MULTIHELD_TOKEN_GTC;
    if [[w1, w2 == something preferably held]] return HELD_TOKEN_GTC;
    if [[w1, w2 == other things]] return MULTIEXCEPT_TOKEN_GTC;
    if [[w1, w2 == someone]] return CREATURE_TOKEN_GTC;
    if [[w1, w2 == somebody]] return CREATURE_TOKEN_GTC;
    if [[w1, w2 == anybody]] return CREATURE_TOKEN_GTC;
    if [[w1, w2 == anyone]] return CREATURE_TOKEN_GTC;
    if [[w1, w2 == text]] return TOPIC_TOKEN_GTC;
    if [[w1, w2 == something held]] {
        incompatible_change_problem(
            "something held", "something", "something preferably held");
        return HELD_TOKEN_GTC;
    }
    if [[w1, w2 == things held]] {
        incompatible_change_problem(
            "things held", "things", "things preferably held");
        return MULTIHELD_TOKEN_GTC;
    }
    return 0;
}
```

§**7.** Something of an extended mea culpa: but it had the desired effect, in that nobody complained about what might have been a controversial change.

```
void incompatible_change_problem(char *token_tried, char *token_instead,
    char *token_better) {
    quote_source(1, current_sentence);
    quote_text(2, token_tried);
    quote_text(3, token_instead);
    quote_text(4, token_better);
    handmade_problem(_P_(C13ObsoleteHeldTokens));
    issue_problem_segment(
        "In the sentence %1, you used the '[%2]' as a token, which was "
        "allowed in the early Public Beta versions of Inform 7, but became "
        "out of date in August 2006%|. A change was then made so that if an "
        "action needed to apply to something which was carried, this would "
        "now be specified when the action is created - not in the Understand "
        "line for it. For instance, one might say 'Dismantling is an action "
        "which applies to one carried thing', instead of '...which applies "
        "to one thing', and then write grammar such as 'Understand \"dismantle "
        "[something] as dismantling' instead of '...[something held]...'. "
        "So you probably need to change your '[%2]' token to '[%3]', and "
        "change the action's definition (unless it is a built-in action "
        "such as 'dropping'). An alternative, though, for fine-tuning is to "
        "change it to '[%4]', which allows anything to be Understood, but "
        "in cases of ambiguity tends to guess that something held is more "
        "likely to be what the player means than something not held.");
    issue_problem_end();
}
```

§**8. Phase II: Determining Grammar.** Slashing does not recurse down to individual tokens, so the first time we look seriously at tokens is in Phase II.

```
specification *gtok_determine(parse_node *pn, int depth) {
    int w1, w2;
    specification *spec = NULL;
    grammar_verb *gv;
    if (pn_int_annotation(pn, grammar_token_literal_ANNOT)) return NULL;
    [[w1, w2 <-- pn]];
    if ([[w1, w2 == topic]] || [[w1, w2 == a topic]]) {
        quote_source(1, current_sentence);
        quote_words(2, pn->word_ref1, pn->word_ref2);
        handmade_problem(_P_(C13UseTextNotTopic));
        issue_problem_segment(
            "The grammar token '%2' in the sentence %1 would in some "
            "ways be the right logical way to suggest 'any words at "
            "all here', but (partly for historical reasons) Inform in "
            "fact uses the special syntax '[text]' for that.");
        issue_problem_end();
        return NULL;
    }
    if ([[w1, w2 == object]] || [[w1, w2 == an object]]) {
        quote_source(1, current_sentence);
```

```
    quote_words(2, pn->word_ref1, pn->word_ref2);
    handmade_problem(_P_(C13UseThingNotObject));
    issue_problem_segment(
        "The grammar token '%2' in the sentence %1 would in some "
        "ways be the right logical way to suggest 'any object at "
        "all here', but Inform uses the special syntax '[thing]' "
        "for that. (Or '[things]' if multiple objects are allowed.)");
    issue_problem_end();
    return NULL;
}
gv = gv_named_token_by_name(w1, w2);
if (gv) {
    specification *val = grammar_verb_to_UNDERSTANDING_spec(gv);
    spec = gv_determine(gv, depth+1);                          this is where Phase II recurses
    pn_set_evaluation(pn, val);
} else {
    int special_token = 0;
    if [[w1, w2 == any ...]] {
        spec = parse_expression(w1+1, w2, TYPE_EXPCON);
        if (species_is(spec, DESCRIPTION_SPC)) {
            pn_annotate_int(pn, grammar_token_code_ANNOT, nft_new_id(spec, TRUE));
            pn_set_evaluation(pn, spec);
            return spec;
        }
    }
    if [[w1, w2 == something related by ... --> w1, w2]] {
        binary_predicate *bp = NULL;
        int reverse_flag = FALSE;
        if [[w1, w2 == reversed ... --> w1, w2]] reverse_flag = TRUE;
        if (w1 == w2) {
            bp = parse_relation_name(w2);
            if ((reverse_flag) && (bp)) bp = bp_get_reversal(bp);
        }
        if (bp == NULL) {
            quote_source(1, current_sentence);
            quote_words(2, pn->word_ref1, pn->word_ref2);
            handmade_problem(_P_(C13GrammarBadRelation));
            issue_problem_segment(
                "The grammar token '%2' in the sentence %1 "
                "invites me to understand names of related things, "
                "but the relation is not one that I know.");
            issue_problem_end();
            return NULL;
        }
        pn_set_grammar_token_relation(pn, bp);
        return NULL;                                           because no value is returned
    }
    special_token = parse_special_token(w1, w2);
    if (special_token < 0) {
        spec = new_generic_CONSTANT_type(
            data_type_for_special_token(special_token));
        spec->word_ref1 = w1; spec->word_ref2 = w2;
        spec_set_data(spec, GRAMMAR_TOKEN_SCORE_SPDATA, gsb_for_special_token(special_token));
```

```
        pn_set_evaluation(pn, spec);
        pn_annotate_int(pn, grammar_token_code_ANNOT, special_token);
    } else {
        spec = parse_expression(w1, w2, TYPE_EXPCON);
        pn_set_evaluation(pn, spec);
        if (spec_describes_an_object_vaguely_or_exactly(spec)) {
            spec_set_data(spec, GRAMMAR_TOKEN_SCORE_SPDATA, 5);
            pn_annotate_int(pn, grammar_token_code_ANNOT, nft_new_id(spec, FALSE));
        }
    }
}
if (spec == NULL) return spec;                          to recover from an error in recursive Phase II
if (spec_is_generic_CONSTANT(spec)) {
    kind_of_value *kov = spec_get_kind_of_value(spec);
    if ((is_kova(kov, OBJECT_TY) == FALSE) &&
        (is_kova(kov, UNDERSTANDING_TY) == FALSE) &&
        (kov_request_I6_GPR(kov) == FALSE)) {
        quote_source(1, current_sentence);
        quote_words(2, pn->word_ref1, pn->word_ref2);
        handmade_problem(_P_(C13UnparsableKind));
        issue_problem_segment(
            "The grammar token '%2' in the sentence %1 "
            "invites me to understand values typed by the player during "
            "play but for a kind of value which is beyond my ability. "
            "Generally speaking, the allowable kinds of value are "
            "number, time, text and any new kind of value you may "
            "have created - but not, for instance, scene or rule.");
        issue_problem_end();
        return NULL;
    }
} else {
    if (species_is(spec, DESCRIPTION_SPC) == FALSE) {
        if (spec_is_UNKNOWN(spec)) {
            log_subtree(current_sentence, 0);
            quote_source(1, current_sentence);
            quote_words(2, pn->word_ref1, pn->word_ref2);
            handmade_problem(_P_(C13UnknownToken));
            issue_problem_segment(
                "I was unable to understand what you meant by the grammar "
                "token '%2' in the sentence %1.");
            issue_problem_end();
            return NULL;
        } else {
            if (spec_object_exactly_described_if_any(spec) == NULL) {
                log_subtree(current_sentence, 0);
                quote_source(1, current_sentence);
                quote_words(2, pn->word_ref1, pn->word_ref2);
                quote_type_of(3, spec);
                handmade_problem(_P_(C13BizarreToken));
                issue_problem_segment(
                    "The grammar token '%2' in the sentence %1 looked to me as "
                    "if it might be %3, but this isn't something allowed in "
                    "parsing grammar.");
```

```
                    issue_problem_end();
                    return NULL;
                }
            }
        }
    }
    return spec;
}
```

The function gtok_determine is called from 13/gl.

§**9. Phase IV: Compiling Grammar.** Tokens play no direct part in Phase III either, having made their contribution earlier by recording their GSB scores instead. So we next see them at compilation time.

In code mode, we compile a test that the token matches, jumping to the failure label if it doesn't, and setting the I6 local variable `rv` to a suitable GPR return value if it does match and produces an outcome. We are allowed to use the I6 local `w` for temporary storage, but nothing else.

```
int ol_loop_counter = 0;
kind_of_value *gtok_compile(OUTPUT_STREAM, parse_node *pn, int code_mode,
    char *failure_label, int consult_mode) {
    int wn = pn->word_ref1;
    specification *spec;
    binary_predicate *bp;
    grammar_verb *gv;
    if (pn_int_annotation(pn, grammar_token_literal_ANNOT)) {
        if (code_mode) {
            WRITE("if (NextWordStopped() ~= ");
            isn_compile_dictionary_word(OUT, lw_array[wn].lw_text, FALSE);
            WRITE(") jump %s;\n", failure_label);
        } else isn_compile_dictionary_word(OUT, lw_array[wn].lw_text, FALSE);
        return NULL;
    }
    bp = pn_get_grammar_token_relation(pn);
    if (bp) {
        WRITE("ArticleDescriptors(); w = wn;\n");
        if (bp == a_contains_b_predicate)
            WRITE("if (self hasnt container) jump %s;\n", failure_label);
        if (bp == a_supports_b_predicate)
            WRITE("if (self hasnt supporter) jump %s;\n", failure_label);
        if ((bp == a_has_b_predicate) || (bp == a_wears_b_predicate) ||
            (bp == a_carries_b_predicate))
            WRITE("if (self hasnt animate) jump %s;\n", failure_label);
        if ((bp == a_contains_b_predicate) ||
            (bp == a_supports_b_predicate) ||
            (bp == a_has_b_predicate) ||
            (bp == a_wears_b_predicate) ||
            (bp == a_carries_b_predicate)) {
            WRITE("objectloop (rv in self) {\n"); INDENT;
            if (bp == a_carries_b_predicate)
                WRITE("if (rv has worn) continue;\n");
            if (bp == a_wears_b_predicate)
                WRITE("if (rv hasnt worn) continue;\n");
            WRITE("wn = w; wn = w + TryGivenObject(rv, true);\n");
```

```
    WRITE("if (wn > w) jump ol_mm_%d;\n", ol_loop_counter);
    OUTDENT; WRITE("}\n");
    WRITE("rv = 0; jump %s;\n", failure_label);
    WRITE(".ol_mm_%d; rv = 0;\n", ol_loop_counter++);
} else if (bp == a_incorporates_b_predicate) {
    WRITE("for (rv=self.component_child:rv:rv=rv.component_sibling) {\n"); INDENT;
    WRITE("wn = w; wn = w + TryGivenObject(rv, true);\n");
    WRITE("if (wn > w) jump ol_mm_%d;\n", ol_loop_counter);
    OUTDENT; WRITE("}\n");
    WRITE("rv = 0; jump %s;\n", failure_label);
    WRITE(".ol_mm_%d; rv = 0;\n", ol_loop_counter++);
} else if ((bp_get_reversal(bp) == a_contains_b_predicate) ||
    (bp_get_reversal(bp) == a_supports_b_predicate) ||
    (bp_get_reversal(bp) == a_has_b_predicate) ||
    (bp_get_reversal(bp) == a_wears_b_predicate) ||
    (bp_get_reversal(bp) == a_carries_b_predicate)) {
    if (bp_get_reversal(bp) == a_carries_b_predicate)
        WRITE("if (self has worn) continue;\n");
    if (bp_get_reversal(bp) == a_wears_b_predicate)
        WRITE("if (self hasnt worn) continue;\n");
    WRITE("rv = parent(self);\n");
    WRITE("wn = w; wn = w + TryGivenObject(rv, true);\n");
    WRITE("if (wn == w) jump %s;\n", failure_label);
} else if (bp_get_reversal(bp) == a_incorporates_b_predicate) {
    WRITE("rv = self.component_parent;\n");
    WRITE("wn = w; wn = w + TryGivenObject(rv, true);\n");
    WRITE("if (wn == w) jump %s;\n", failure_label);
} else {
    i6_schema *i6s;
    world_object *wo;
    int reverse = FALSE;
    int continue_loop_on_fail = TRUE;

    if (bp == a_is_b_predicate) internal_error("Can't implement == relation");

    i6s = bp_get_test_function(bp);
    LOGIF(GRAMMAR_CONSTRUCTION, "Read I6s $i from $2\n", i6s, bp);
    if ((i6s == NULL) && (bp_get_test_function(bp_get_reversal(bp)))) {
        reverse = TRUE;
        i6s = bp_get_test_function(bp_get_reversal(bp));
        LOGIF(GRAMMAR_CONSTRUCTION, "But read I6s $i from reversal\n", i6s);
    }

    if (i6s) {
        kind_of_value *kov = bp_term_kind_of_value(bp, 1);
        if (kov) {
            quote_source(1, current_sentence);
            quote_source(2, pn);
            handmade_problem(_P_(C13GrammarValueRelation));
            issue_problem_segment(
                "The grammar you give in %1 contains a token "
                "which relates things to values - %2. At present, "
                "this is not allowed: only relations between kinds "
                "of object can be used in 'Understand' tokens.");
            issue_problem_end();
            return kova(OBJECT_TY);
```

```
        } else {
            wo = bp_term_kind(bp, 1);
            LOGIF(GRAMMAR_CONSTRUCTION, "Term 1 of BP is $O\n", wo);
            if (wo == NULL)
                internal_error("null wo as term in bp for grammar token");
            WRITE("objectloop (rv ofclass %s) {\n",
                wo_get_I6_representation(wo)); INDENT;
            WRITE("if ((");
            if (reverse)
                sch_expand_textual(i6s, OUT, "rv", "self");
            else
                sch_expand_textual(i6s, OUT, "self", "rv");
            continue_loop_on_fail = TRUE;
        }
    } else {
        property_name *prn = bp_get_i6_storage_property(bp);
        reverse = FALSE;
        if (bp_is_the_wrong_way_round(bp)) reverse = TRUE;
        if (bp_get_form_of_relation(bp) == Relation_VtoO) {
            if (reverse) reverse = FALSE; else reverse = TRUE;
        }
        if (prn) {
            if (reverse) {
                WRITE("if (self provides %s) {\n", prn_get_i6_identifier(prn));
                INDENT;
                WRITE("rv = self.%s;\n", prn_get_i6_identifier(prn));
                WRITE("if ((rv");
                continue_loop_on_fail = FALSE;
            } else {
                wo = bp_term_kind(bp, 1);
                WRITE("objectloop (rv ofclass %s) {\n",
                    wo_get_I6_representation(wo)); INDENT;
                WRITE("if (((rv provides %s) && (rv.%s == self)",
                    prn_get_i6_identifier(prn),
                    prn_get_i6_identifier(prn));
                continue_loop_on_fail = TRUE;
            }
        } else {
            LOG("Trouble with: $2\n", bp);
            LOG("Whose reversal is: $2\n", bp_get_reversal(bp));
            internal_error("Can't implement this relation");
        }
    }
    if (continue_loop_on_fail == FALSE)
        WRITE(") == false) jump %s;\n", failure_label);
    else
        WRITE(") == false) continue;\n");
    WRITE("wn = w; wn = w + TryGivenObject(rv, true);\n");
    WRITE("if (wn > w) jump ol_mm_%d;\n", ol_loop_counter);
    OUTDENT; WRITE("}\n");
    WRITE("rv = 0; jump %s;\n", failure_label);
    WRITE(".ol_mm_%d; rv = 0;\n", ol_loop_counter++);
}
```

```
        return NULL;
    }
    spec = pn_get_evaluation(pn);
    if (spec == NULL) gtok_determine(pn, 10);
    spec = pn_get_evaluation(pn);
    if (spec == NULL) {
        log_subtree(pn, 1);
        internal_error("NULL result of non-preposition token");
    }
    if ((spec_is_generic_CONSTANT(spec)) &&
        (is_kova(spec_get_kind_of_value(spec), OBJECT_TY) == FALSE) &&
        (is_kova(spec_get_kind_of_value(spec), UNDERSTANDING_TY) == FALSE)) {
        kind_of_value *kov = spec_get_kind_of_value(spec);
        if (kov_offers_I6_GPR(kov)) {
            char *i6_gpr_name = kov_get_explicit_I6_GPR(kov);
            if (code_mode) {
                WRITE("w = ParseTokenStopped(GPR_TT, ");
                if (i6_gpr_name != NULL) WRITE(i6_gpr_name);
                else gprv_compile(OUT, kov);
                WRITE(");\n");
                WRITE("if (w ~= GPR_NUMBER) jump %s; rv = GPR_NUMBER;\n",
                    failure_label);
            } else {
                if (i6_gpr_name != NULL) WRITE(i6_gpr_name);
                else gprv_compile(OUT, kov);
            }
            return spec_get_kind_of_value(spec);
        }
        internal_error("Let an invalid type token through");
    }
    if ((species_is(spec, DESCRIPTION_SPC)) && ((spec_get_proposition(spec)) || (spec_get_described_kov(spec))))
{
        quote_source(1, current_sentence);
        quote_source(2, pn);
        handmade_problem(_P_(C13OverComplexToken));
        issue_problem_segment(
            "The grammar you give in %1 contains a token "
            "which is just too complicated - %2. %PFor instance, a "
            "token using subordinate clauses - such as '[a container "
            "which is open]' will probably not be allowed. It may be "
            "possible to rephrase in a simple adjectives-plus-noun "
            "way, as in '[open container]', but if not, then you may "
            "may wish to investigate writing a new token.");
        issue_problem_end();
        return kova(OBJECT_TY);
    } else {
        int gtc = pn_int_annotation(pn, grammar_token_code_ANNOT);
        kind_of_value *kov = NULL;
        if (gtc < 0) {
            char *i6_token = i6_token_for_special_token(gtc);
            char *i6_token_constant = i6_constant_for_special_token(gtc);
            kov = data_type_for_special_token(gtc);
            if (code_mode) {
```

```
                if (consult_mode) {
                    sentence_problem(_P_(C13TextTokenRestricted),
                        "the '[text]' token is not allowed with 'matches' "
                        "or in table columns",
                        "as it is just too complicated to sort out: a "
                        "'[text]' is supposed to extract a snippet from "
                        "the player's command, but here we already have "
                        "a snippet, and don't want to snip it further.");
                }
                WRITE("w = ParseTokenStopped(ELEMENTARY_TT, %s);\n",
                    i6_token_constant);
                WRITE("if (w == GPR_FAIL) jump %s; rv = w;\n",
                    failure_label);
            } else WRITE("%s", i6_token);
        } else {
            if (species_is(spec, DESCRIPTION_SPC)) {
                if (code_mode) {
                    WRITE("w = ParseTokenStopped(");
                    nft_compile_by_id(OUT, gtc, TRUE);
                    WRITE(");\n");
                    WRITE("if (w == GPR_FAIL) jump %s; rv = w;\n",
                        failure_label);
                } else nft_compile_by_id(OUT, gtc, FALSE);
                kov = kova(OBJECT_TY);
            } else {
                if (species_is(spec, CONSTANT_SPC)) {
                    if (spec_is_CONSTANT_of_kova(spec, UNDERSTANDING_TY)) {
                        gv = UNDERSTANDING_spec_to_grammar_verb(spec);
                        if (code_mode) {
                            WRITE("w = ParseTokenStopped(GPR_TT, ");
                            gv_compile_i6_token(OUT, gv);
                            WRITE(");\n");
                            WRITE("if (w == GPR_FAIL) jump %s;\n", failure_label);
                            WRITE("if (w ~= GPR_PREPOSITION) rv = w;\n");
                        } else gv_compile_i6_token(OUT, gv);
                        kov = gv_get_data_type_as_token(gv);
                    }
                    if (spec_is_CONSTANT_of_kova(spec, OBJECT_TY)) {
                        if (code_mode) {
                            WRITE("w = ParseTokenStopped(");
                            nft_compile_by_id(OUT, gtc, TRUE);
                            WRITE(");\n");
                            WRITE("if (w == GPR_FAIL) jump %s; rv = w;\n",
                                failure_label);
                        } else nft_compile_by_id(OUT, gtc, FALSE);
                        kov = kova(OBJECT_TY);
                    }
                } else kov = kova(OBJECT_TY);
            }
        }
        return kov;
    }
}
```

The function gtok_compile is called from 13/gl.

Noun Filter Tokens 13/nft

Purpose

Filters are used to require nouns to have specific kinds or attributes, or to have specific scoping rules: they correspond to Inform 6's `noun=Routine` and `scope=Routine` tokens. Though these are quite different concepts in I6, their common handling seems natural in I7.

13/nft.§2 Access via ID; §3 Compiling everything

Template interpreter commands

```
3   {-callv:compile_filters}
```

Definitions

```
typedef struct noun_filter_token {
    struct specification *the_filter;
    int global_scope_flag;
    MEMORY_MANAGEMENT
} noun_filter_token;
```

The structure noun_filter_token is private to this section.

---

§**1.** There are only three things we can do with these: create them, compile their names (used as I6 tokens), and compile their routines.

```
noun_filter_token *nft_new(specification *spec, int global_scope) {
    noun_filter_token *nft = CREATE(noun_filter_token);
    nft->the_filter = spec;
    nft->global_scope_flag = global_scope;
    return nft;
}
void nft_compile_routine_name(OUTPUT_STREAM, noun_filter_token *nft) {
    if (nft->global_scope_flag)
        WRITE("Scope_Filter_%d", nft->allocation_id);
    else
        WRITE("Noun_Filter_%d", nft->allocation_id);
}
void nft_compile_routine(OUTPUT_STREAM, noun_filter_token *nft) {
    specification *noun_var = new_QUANTITY_spec(I6_noun_quantity);
    WRITE("[ ");
    nft_compile_routine_name(OUT, nft);
    if (nft->global_scope_flag) {
        WRITE(" obj o2;\n"); INDENT;
        WRITE("switch (scope_stage) {\n"); INDENT;
        WRITE("1: rfalse;\n");
        WRITE("2: obj=noun;\n");
        WRITE("suppress_scope_loops = true;\n");
        WRITE("objectloop(noun ofclass Object && (");
        compile_test_if_var_matches_description(OUT, noun_var, nft->the_filter);
```

```
        WRITE(")) {\n"); INDENT;
        WRITE("o2 = noun; noun = obj; PlaceInScope(o2, true); noun = o2;\n");
        OUTDENT; WRITE("}\n");
        WRITE("noun=obj;\n");
        WRITE("suppress_scope_loops = false;\n");
        WRITE("3: nextbest_etype = NOTINCONTEXT_PE; return -1;\n");
        OUTDENT; WRITE("}\n");
        OUTDENT; WRITE("];\n");
    } else {
        WRITE(" x;\n"); INDENT;
        WRITE("x=noun;\n");
        WRITE("return ");
        if (spec_get_proposition(nft->the_filter)) {
            prop_type_check(spec_get_proposition(nft->the_filter), tc_no_problem_reporting());
            compile_test_of_proposition(
                OUT, noun_var, spec_get_proposition(nft->the_filter));
        } else
            compile_test_if_var_matches_description(
                OUT, noun_var, nft->the_filter);
        WRITE(";\n");
        OUTDENT; WRITE("];\n");
    }
}
```

§**2. Access via ID.** For now, though, these are perhaps strangely accessed by ID number. (Because the `parse_node` structure can't conveniently be annotated with pointers, that's why.)

```
int nft_new_id(specification *spec, int global_scope) {
    return nft_new(spec, global_scope)->allocation_id;
}
void nft_compile_by_id(OUTPUT_STREAM, int id, int code_mode) {
    noun_filter_token *nft;
    LOOP_OVER(nft, noun_filter_token)
        if (nft->allocation_id == id) {
            if (code_mode) {                                  arguments to the I6 routine ParseTokenStopped
                if (nft->global_scope_flag) WRITE("SCOPE_TT, ");
                else WRITE("ROUTINE_FILTER_TT, ");
            } else {                                                  tokens for use in Verb directives
                if (nft->global_scope_flag) WRITE("scope=");
                else WRITE("noun=");
            }
            nft_compile_routine_name(OUT, nft);
        }
}
```

The function nft_new_id is called from 13/gtok.

The function nft_compile_by_id is called from 13/gtok.

§3. Compiling everything. Having referred to these filter routines, we need to compile them.

```
void compile_filters(OUTPUT_STREAM) {
    noun_filter_token *nft;
    LOOP_OVER(nft, noun_filter_token)
        nft_compile_routine(OUT, nft);
}
```

The function compile_filters is invoked by a command in a `.i6t` template file.

Tokens Parsing Values 13/gprv

Purpose

In the argot of Inform 6, GPR stands for General Parsing Routine, and I7 makes heavy use of GPR tokens to achieve its ends. This section is where the necessary I6 routines are compiled.

Template interpreter commands

```
0   {-callv:compile_number_grammar}
0   {-callv:compile_time_grammar}
0   {-callv:compile_truth_state_grammar}
0   {-callv:compile_type_gprs}
```

```
void compile_number_grammar(OUTPUT_STREAM) {
    grammar_verb *gv = kov_get_parsing_grammar(kova(NUMBER_TY));
    if (gv != NULL) {
        gv_compile(OUT, gv);
    }
}
void compile_time_grammar(OUTPUT_STREAM) {
    grammar_verb *gv = kov_get_parsing_grammar(kova(TIME_TY));
    if (gv != NULL) {
        gv_compile(OUT, gv);
    }
}
void compile_truth_state_grammar(OUTPUT_STREAM) {
    grammar_verb *gv = kov_get_parsing_grammar(kova(TRUTH_STATE_TY));
    if (gv != NULL) {
        gv_compile(OUT, gv);
    }
}
void compile_type_gprs(OUTPUT_STREAM) {
    int i, k, next_label = 1, longest;
    grammar_verb *gv;
    LOOP_OVER_DESIGNED_TYPE_IDS(i) {
        quantity *q; literal_pattern *lp;
        kind_of_value *kov = kova(i);
        if (kov_needs_I6_GPR(kov) == FALSE) continue;
        WRITE("[ ");
        gprv_compile(OUT, kov);
        WRITE(" original_wn group_wn v wpos mid_word ");
        WRITE("matched_number cur_word cur_len cur_addr sgn tot f w rv x;\n");
        INDENT;
        WRITE("original_wn = wn;\n");
        LITERAL_FORMS_LOOP(lp, i) {
            gpr_for_lp(OUT, lp);
            WRITE("wn = original_wn;\n");
        }
        gv = kov_get_parsing_grammar(kova(i));
```

```
        if (gv != NULL) {
            gv_compile(OUT, gv);
            WRITE("wn = original_wn;\n");
        }
        longest = 0;
        LOOP_OVER(q, quantity) {
            if (q->word_ref2 - q->word_ref1 > longest)
                longest = q->word_ref2 - q->word_ref1;
        }
        for (; longest >= 0; longest--) {
            LOOP_OVER(q, quantity) {
                if (q->word_ref2 - q->word_ref1 == longest)
                    if ((is_kova(qty_kind_of_value(q), i)) &&
                        (qty_is_a_variable(q) == FALSE)) {
                        WRITE("wn = original_wn;");
                        for (k=q->word_ref1; k<=q->word_ref2; k++) {
                            WRITE("if (NextWordStopped() ~= ");
                            isn_compile_dictionary_word(OUT, lw_array[k].lw_text, FALSE);
                            WRITE(") jump Failed_%d;\n", next_label);
                        }
                        WRITE("parsed_number = ");
                        compile_quantity_name(OUT, q);
                        WRITE("; return GPR_NUMBER;\n");
                        WRITE(".Failed_%d;\n", next_label++);
                    }
            }
        }
        WRITE("return GPR_FAIL;\n");
        OUTDENT; WRITE("];\n");
    }
}

void gprv_compile(OUTPUT_STREAM, kind_of_value *kov) {
    WRITE("Designed_Type_GPR_%d", kov_I6_ID(kov));
}
```

The function compile_number_grammar is invoked by a command in a `.i6t` template file.

The function compile_time_grammar is invoked by a command in a `.i6t` template file.

The function compile_truth_state_grammar is invoked by a command in a `.i6t` template file.

The function compile_type_gprs is invoked by a command in a `.i6t` template file.

The function gprv_compile is called from 13/gtok and 13/gpr.

General Parsing Routines 13/gpr

Purpose

To compile I6 general parsing routines (GPRs) and/or `parse_name` properties as required by the I7 grammar.

§**1.** In this section we compile GPRs, routines to handle `Consult`-like text, and also `parse_name` routines, which as we shall see come in two different forms. These routines share a basic protocol for dealing with the I6 library, which makes things considerably easier. In each case, the routine is compiled as a head and then, subsequently, a tail: the user of the routines is expected to compile the actual grammar in between the two. Every head must be followed by exactly one tail of the same sort; every tail must be preceded by exactly one head of the same sort; but code to parse at `wn` may be placed in between.

The GPRs compiled to parse literal values of given kinds of values (for instance, exotic verbal forms of numbers, or verbal names of the constants for new kinds of value, or literal patterns) are not compiled here: they are in Tokens Parsing Values.

§**2. GPRs used as tokens.** To parse a token, we match as much as possible from word number `wn`.

```
void gpr_compile_gpr_head(OUTPUT_STREAM, int id) {
    WRITE("[ GPR_Line_%d\n", id); INDENT;
    WRITE("original_wn  ! first word of text parsed\n");
    WRITE("group_wn  ! first word matched against A/B/C/... disjunction\n");
    WRITE("w  ! for use by individual grammar lines\n");
    WRITE("rv  ! for use by individual grammar lines\n");
    WRITE(";\n");
    WRITE("original_wn = wn; rv = GPR_PREPOSITION;\n");
}
void gpr_compile_gpr_tail(OUTPUT_STREAM) {
    WRITE("return GPR_FAIL;\n");
    OUTDENT; WRITE("];\n\n");
}
void gpr_compile_prn_pr_head(OUTPUT_STREAM, property_name *prn) {
    WRITE("[ PRN_PN_%d\n", prn->allocation_id); INDENT;
    WRITE("original_wn  ! first word of text parsed\n");
    WRITE("group_wn  ! first word matched against A/B/C/... disjunction\n");
    WRITE("w  ! for use by individual grammar lines\n");
    WRITE("rv  ! for use by individual grammar lines\n");
    WRITE(";\n");
    WRITE("original_wn = wn; rv = GPR_PREPOSITION;\n");
}
void gpr_compile_prn_pr_tail(OUTPUT_STREAM, property_name *prn) {
    WRITE("return GPR_FAIL;\n");
    OUTDENT; WRITE("];\n\n");
}
```

The function gpr_compile_gpr_head is called from 13/gv.

The function gpr_compile_gpr_tail is called from 13/gv.

The function gpr_compile_prn_pr_head is called from 13/gv.

The function gpr_compile_prn_pr_tail is called from 13/gv.

§**3. Consult routines.** These are used to parse an explicit range of words (such as traditionally found in the CONSULT command) at run time, and they are *not* I6 grammar tokens, and do *not* appear in `Verb` declarations: otherwise, such routines are very similar to GPRs.

First, we need to look after a pointer to the GV used to hold the grammar being matched against the snippet of words.

```
grammar_verb *consultation_gv = NULL;                                    used only in routines below
grammar_verb *gpr_get_consultation_gv(void) {
    if (consultation_gv == NULL) consultation_gv = gv_consultation_new();
    return consultation_gv;
}
void gpr_prepare_consultation_gv(void) {
    consultation_gv = NULL;
}
void gpr_print_consultation_gv_name(OUTPUT_STREAM) {
    if (consultation_gv)
        WRITE("Consult_Grammar_%d", consultation_gv->allocation_id);
}
```

The function gpr_get_consultation_gv is called from 13/tfg.
The function gpr_prepare_consultation_gv is called from 13/tfg.
The function gpr_print_consultation_gv_name is called from 13/tfg.

§**4.** We also, at another time, need to compile the routine being named. There are no timing difficulties here: the routine's name is used in the context of an I6 constant rather than in a `Verb` declaration, so no predeclaration is needed.

```
void gpr_compile_consult_head(OUTPUT_STREAM, int id) {
    WRITE("[ Consult_Grammar_%d\n", id); INDENT;
    WRITE("range_from  ! call parameter: word number of snippet start\n");
    WRITE("range_words  ! call parameter: snippet length\n");
    WRITE("original_wn  ! first word of text parsed\n");
    WRITE("group_wn  ! first word matched against A/B/C/... disjunction\n");
    WRITE("w  ! for use by individual grammar lines\n");
    WRITE("rv  ! for use by individual grammar lines\n");
    WRITE(";\n");
    WRITE("wn = range_from; original_wn = wn; rv = GPR_PREPOSITION;\n");
}
void gpr_compile_consult_tail(OUTPUT_STREAM) {
    WRITE("return GPR_FAIL;\n");
    OUTDENT; WRITE("];\n\n");
}
```

The function gpr_compile_consult_head is called from 13/gv.
The function gpr_compile_consult_tail is called from 13/gv.

§**5. Parse name properties.** One of the major services provided by I7, as compared with I6, is that it automatically compiles what would otherwise be laborious `parse_name` routines for its objects. This is messy, because the underlying I6 syntax is messy. The significant complication is that the I6 parser makes two quite different uses of `parse_name`: not just for parsing names, but also for determining whether two objects are visually distinguishable, something it needs to know in order to make plural objects work properly.

If an object has any actual grammar attached, say a collection of grammar lines belonging to GV3, we will compile the `parse_name` as an independent I6 routine with a name like `Parse_Name_GV3`. If not, a `parse_name` may still be needed, because of the distinguishability problem: if so then we will simply compile a `parse_name` routine inline, in the usual I6 way.

```
void gpr_compile_parse_name_property(OUTPUT_STREAM, world_object *wo,
    grammar_verb *gv) {
    if (gv_is_empty(gv) == FALSE)
        WRITE(" with parse_name Parse_Name_GV%d\n", gv->allocation_id);
    else
        if (gpr_compile_parse_name_head(OUT, wo, NULL)) {
            gpr_compile_parse_name_tail(OUT);
            WRITE(",\n");
        }
}
```

The function gpr_compile_parse_name_property is called from 9/cot.

§**6.** The following routine produces one of three outcomes: either (i) the head of an I6 declaration of a free-standing routine to be used as a `parse_name` property, or (ii) the head of an I6 inline declaration of a `parse_name` property as a `with` clause for an `Object` directive, or

(iii) the empty output, in happy cases where neither parsing nor distinguishability need to be investigated. The routine returns a flag indicating if a tail need be compiled (i.e., in cases (i) or (ii) but not (iii)).

In cases (i) and (ii), the head is immediately followed by code which looks at the names of visible properties. Recall that a visible property is one which can be used to describe an object: for instance, if colour is a visible property of a car, then it can be called "green car" if and only if the current value of the colour of the car is "green", and so forth. In all such cases, we need to parse the text to look for the name of the current value.

But if a property can be used as part of the name, then it follows that two objects with the same grammar (and name words) cease to be indistinguishable when their values for this property differ. For instance, given two otherwise identical cars which can only be called "car", we can distinguish them with the names "red car" and "green car" if one is red and the other green. The parser needs to know this. It calls the `parse_name` routine with an I6 global called `parser_action` set to `##TheSame` in such a case, and we can return 0 to make no decision or `-2` to say that they are different.

Note that the parser checks this only if two or more objects share the same `parse_name` routine: which will in I7 happen only if they each inherit it from the I6 class of a common kind. Because `parse_name` is not additive in I6, this can only occur if the objects, and any intervening classes for intervening kinds, define no `parse_name` of their own.

We will test distinguishability only for kinds which have permissions for visible properties: kinds because no other `parse_name` values can ever be duplicated in instance objects, and visible properties because these are the only ways to tell apart instances which have no grammar of their own. (If either had grammar of its own, it would also have its own `parse_name` routine.) For all other kinds, we return a make-no-decision value in response to a `##TheSame` request: this ensures that the I6 parser looks at the `name` properties of the objects instead, and in the absence of either I7-level grammar lines or visible properties, that will be the correct decision.

```
int gpr_compile_parse_name_head(OUTPUT_STREAM, world_object *wo, grammar_verb *gv) {
    int allow_inheritance = FALSE, test_distinguishability = FALSE,
```

```
        sometimes_has_visible_properties = FALSE, use_two_passes = TRUE;
    if (wo == NULL) internal_error("compiling parse_name head for null wo");
    sometimes_has_visible_properties = pr_any_visible_to_wo(wo, TRUE);
    if (gv) {
        top_of_head(OUT, gv->allocation_id, use_two_passes);
        allow_inheritance = TRUE;
        if ((wo->kind_flag) && (sometimes_has_visible_properties == FALSE))
            WRITE("if (parser_action == ##TheSame) return 0;\n");
    } else {
        if (pr_any_visible_to_wo(wo, FALSE) == FALSE) return FALSE;
        top_of_head(OUT, -1, use_two_passes);
    }
    if (wo->kind_flag) test_distinguishability = TRUE;
    consider_visible_properties(OUT, wo, allow_inheritance, test_distinguishability);
    return TRUE;
}
```

The function gpr_compile_parse_name_head is called from 13/gv.

§**7.** Each head is followed by a tail before the next head: there is no recursion. Therefore it is safe to record a state when compiling the head, and to recover it when compiling the tail.

The two-pass form of the routine is used in cases where there are visible properties. The two passes check the possible patterns:

(1) (words in `name` property) (visible property names) (longer grammar)

(2) (visible property names) (longer grammar) (words in `name` property)

The longer match is taken: but note that a match of visible property names alone is rejected unless at least one property has been declared sufficient to identify the object all by itself. Longer grammar means grammar lines containing 2 or more words, since all single-fixed-word grammar lines for GVs destined to be `parse_name`s is stripped out and converted into the `name` property.

There are clearly other possibilities and the above system is something of a pragmatic compromise (in that to check other cases would be slower and more complex). I suspect we will return to this.

```
int this_parse_name_is_two_pass = NOT_APPLICABLE;
void top_of_head(OUTPUT_STREAM, int id, int two_pass) {
    if (id >= 0) WRITE("[ Parse_Name_GV%d\n", id);
    else WRITE("with parse_name [\n");
    INDENT;
    this_parse_name_is_two_pass = two_pass;
    WRITE("original_wn  ! first word of text parsed\n");
    WRITE("group_wn  ! first word matched against A/B/C/... disjunction\n");
    WRITE("try_from_wn  ! position to try matching from\n");
    WRITE("n  ! number of words matched\n");
    WRITE("f  ! flag: sufficiently good match found to justify success\n");
    WRITE("w  ! for use by individual grammar lines\n");
    WRITE("rv  ! for use by individual grammar lines\n");
    WRITE("g  ! temporary: success flag for parsing visibles\n");
    WRITE("ss  ! temporary: saves 'self' in distinguishing visibles\n");
    WRITE("spn  ! temporary: saves 'parsed_number' in parsing visibles\n");
    if (this_parse_name_is_two_pass) {
        WRITE("pass  ! pass counter (1 or 2)\n");
```

```
        WRITE("pass1_n  ! value of n recorded during pass 1\n");
        WRITE(";\n");
        WRITE("if (parser_trace >= 3) print \"Two-pass parse_name called^\";\n");
        WRITE("original_wn = wn;\n");
        WRITE("for (pass = 1: pass <= 2: pass++) {\n"); INDENT;
        WRITE("wn = original_wn;\n");
        WRITE("try_from_wn = wn; f = false; n = 0;\n");
        WRITE("! On pass 1 only, advance wn past name property words\n");
        WRITE("! (but do not do this for ##TheSame, when wn is undefined)\n");
        WRITE("if ((parser_action ~= ##TheSame) && (pass == 1)) {\n"); INDENT;
        WRITE("while (WordInProperty(NextWordStopped(), self, name)) f = true;\n");
        WRITE("wn--; try_from_wn = wn;\n");
        OUTDENT; WRITE("}\n");
    } else {
        WRITE(";\n");
        WRITE("original_wn = wn;\n");
        WRITE("try_from_wn = wn;\n");
    }
}
```

§**8.** The head and tail routines can only be understood by knowing that the following code is used to reset the grammar-line parser after each failure of a GL to parse.

```
void gpr_after_gl_failed(OUTPUT_STREAM, int label, int pluralised) {
    if (pluralised) WRITE("parser_action = ##PluralFound;\n");
    WRITE("try_from_wn = wn; f = true;\n");
    WRITE(".Fail_%d; wn = try_from_wn;\n", label);
}
```

The function gpr_after_gl_failed is called from 13/gl.

§**9.** The interesting point about the tail of the `parse_name` routine is that it ends on the `]` "close routine" character, in mid-source line. This is because the routine may be being used inside an `Object` directive, and would therefore need to be followed by a comma, or in free-standing I6 code, in which case it would need to be followed by a semi-colon.

```
void gpr_compile_parse_name_tail(OUTPUT_STREAM) {
    if (this_parse_name_is_two_pass == NOT_APPLICABLE)
        internal_error("synchrony broken between parse_name head and tail");
    if (this_parse_name_is_two_pass == TRUE) {
        WRITE("! On pass 2 only, match name property words at end\n");
        WRITE("if (pass == 2)\n");
    }
    WRITE("while (WordInProperty(NextWordStopped(), self, name)) n++;\n");
    WRITE("if ((f) || (n>0)) n = n + try_from_wn - original_wn;\n");
    if (this_parse_name_is_two_pass == TRUE) {
        WRITE("if (pass == 1) pass1_n = n;\n");
        OUTDENT; WRITE("} ! End of pass loop\n");
        WRITE("if (parser_trace >= 3)\n"); INDENT;
        WRITE("print \"Pass 1: \", pass1_n, \" Pass 2: \", n, \"^\";\n");
        OUTDENT;
        WRITE("if (pass1_n > n) n = pass1_n;\n");
    }
```

```
    WRITE("wn = original_wn + n;\n");
    WRITE("if (n == 0) return -1;\n");
    WRITE("DetectPluralWord(original_wn, n);\n");
    WRITE("return n;\n");
    OUTDENT; WRITE("]");
    this_parse_name_is_two_pass = NOT_APPLICABLE;
}
```

The function gpr_compile_parse_name_tail is called from 13/gv.

§**10.** We generate code suitable for inclusion in a `parse_name` routine which either tests distinguishability then parses, or else just parses, the visible properties of a given world object (which may be a kind). Sometimes we allow visibility to be inherited from a permission given to a kind, sometimes we require that the permission be given to this specific object.

```
void consider_visible_properties(OUTPUT_STREAM, world_object *wo,
    int allow_inheritance, int test_distinguishability) {
    int phase = 2;
    if (test_distinguishability) phase = 1;
    for (; phase<=2; phase++) {
        property_name *pr;
        start_considering_visible_properties(OUT, phase);
        LOOP_OVER(pr, property_name) {
            property_permission *pp;
            for (pp = prn_permission_list(pr); pp; pp = pp->next)
                if (pp_visible_to_wo(pp, wo, allow_inheritance))
                    consider_visible_property(OUT, wo, pr, pp, phase);
        }
        finish_considering_visible_properties(OUT, phase);
    }
}
```

§**11. Common handling for distinguishing and parsing.** The top-level considering routines parcel up work and hand it over to the distinguishing routines if `phase` is 1, and the parsing routines if `phase` is 2. Note that if there are no sometimes-visible-properties then the correct behaviour is to call none of the routines below this level, and to compile nothing to the file.

```
int visible_properties_code_written = FALSE;                    persistent state used only here
void start_considering_visible_properties(OUTPUT_STREAM, int phase) {
    visible_properties_code_written = FALSE;
}
void consider_visible_property(OUTPUT_STREAM, world_object *wo,
    property_name *pr, property_permission *pp, int phase) {
    int conditional_vis = FALSE;
    specification *spec;
    if (visible_properties_code_written == FALSE) {
        visible_properties_code_written = TRUE;
        if (phase == 1)
            begin_distinguishing_visible_properties(OUT);
        else
            begin_parsing_visible_properties(OUT);
    }
```

```
    spec = pp_get_visibility_condition(pp);
    if (spec) {
        conditional_vis = TRUE;
        TEMPORARY_STREAM;
        LOG("Understand when condition: $S\n", spec);
        spec_compile(TEMP, spec);
        if (phase == 1)
            test_distinguish_visible_property(OUT, TEMP);
        else
            test_parse_visible_property(OUT, TEMP);
        CLOSE_TEMPORARY_STREAM;
    }
    if (phase == 1)
        distinguish_visible_property(OUT, pr);
    else
        parse_visible_property(OUT, wo, pr, pp_visibility_level(pp));
    if (conditional_vis) { OUTDENT; WRITE("}\n"); }
}
void finish_considering_visible_properties(OUTPUT_STREAM, int phase) {
    if (visible_properties_code_written) {
        if (phase == 1)
            finish_distinguishing_visible_properties(OUT);
        else
            finish_parsing_visible_properties(OUT);
    }
}
```

§**12. Distinguishing visible properties.** We distinguish two objects P1 and P2 based on the following criteria:

(i) if any property is currently visible for P1 but not P2 or vice versa, then they are distinguishable; (ii) if any value property is visible but P1 and P2 have different values for it, then they are distinguishable;

(iii) if any either/or property is visible but P1 has it and P2 hasn't, or vice versa, then they are distinguishable; and otherwise we revert to the I6 parser's standard algorithm, which looks at the `name` property.

```
void begin_distinguishing_visible_properties(OUTPUT_STREAM) {
    WRITE("if (parser_action==##TheSame) {\n"); INDENT;
    WRITE("if (parser_trace >= 4) print \"p1, p2 = \", parser_one, \", \", parser_two, \"^\";\n");
    WRITE("ss = self;\n");
}
void test_distinguish_visible_property(OUTPUT_STREAM, STREAM *COND) {
    WRITE("self = parser_one; f = (");
    STREAM_COPY(OUT, COND);
    WRITE(");\n");
    WRITE("self = parser_two; g = (");
    STREAM_COPY(OUT, COND);
    WRITE(");\n");
    WRITE("if (f ~= g) return -2;\n");
    WRITE("if (f) {\n"); INDENT;
}
void distinguish_visible_property(OUTPUT_STREAM, property_name *prn) {
    if (prn_is_either_or(prn)) {
```

```
        WRITE("if ((parser_one "); compile_has_property(OUT, prn);
        WRITE(") && (~~(parser_two "); compile_has_property(OUT, prn);
        WRITE("))) return -2;\n");
        WRITE("if ((parser_two "); compile_has_property(OUT, prn);
        WRITE(") && (~~(parser_one "); compile_has_property(OUT, prn);
        WRITE("))) return -2;\n");
    } else {
        kind_of_value *kov = prn_get_kind_of_value(prn);
        char *distinguisher = kov_get_distinguisher(kov);
        if (distinguisher) {
            WRITE("if (%s(parser_one.%s, parser_two.%s)) return -2;\n",
                distinguisher, prn_get_i6_identifier(prn), prn_get_i6_identifier(prn));
        } else {
            WRITE("if (parser_one.%s ~= parser_two.%s) return -2;\n",
                prn_get_i6_identifier(prn), prn_get_i6_identifier(prn));
        }
    }
}
void finish_distinguishing_visible_properties(OUTPUT_STREAM) {
    WRITE("self = ss; return 0;\n");
    OUTDENT; WRITE("}\n");
}
```

§**13. Parsing visible properties.** Here, unlike in distinguishing visible properties, it is unambiguous that `self` refers to the object being parsed: there is therefore no need to alter the value of `self` to make any visibility condition work correctly.

```
void begin_parsing_visible_properties(OUTPUT_STREAM) {
    WRITE("! Match any number of visible property values\n");
    WRITE("try_from_wn = wn; g = true; while (g) {\n"); INDENT;
    WRITE("g = false;\n");
}
void test_parse_visible_property(OUTPUT_STREAM, STREAM *COND) {
    WRITE("if (");
    STREAM_COPY(OUT, COND);
    WRITE(") {\n"); INDENT;
}
int unique_pvp_counter = 0;
void parse_visible_property(OUTPUT_STREAM,
    world_object *wo, property_name *prn, int visibility_level) {
    if (prn_is_either_or(prn)) {
        property_name *prnbar;
        parse_visible_either_or(OUT, prn, visibility_level, unique_pvp_counter);
        prnbar = prn_either_or_get_negation(prn);
        if (prnbar)
            parse_visible_either_or(OUT, prnbar, visibility_level, unique_pvp_counter);
        WRITE(".pvp_pass_L_%d;\n", unique_pvp_counter++);
    } else {
        kind_of_value *kov = prn_get_kind_of_value(prn);
        char *recog_gpr;
        pvp_test_begins(OUT);
```

```
        WRITE("spn = parsed_number; if ((");
        recog_gpr = kov_get_recognition_only_GPR(kov);
        if (recog_gpr) {
            WRITE("%s(self.%s) == GPR_PREPOSITION))",
                recog_gpr, prn_get_i6_identifier(prn));
        } else {
            if (kov_offers_I6_GPR(kov)) {
                char *i6_gpr_name = kov_get_explicit_I6_GPR(kov);
                if (i6_gpr_name != NULL) WRITE(i6_gpr_name);
                else gprv_compile(OUT, kov);
            }
            else internal_error("Unable to recognise kind of value in parsing");
            WRITE("() == GPR_NUMBER) && (self.%s == parsed_number)) ",
                prn_get_i6_identifier(prn));
        }
        pvp_test_passes(OUT, visibility_level, -1);
        WRITE("parsed_number = spn;\n");
    }
}
void parse_visible_either_or(OUTPUT_STREAM, property_name *prn, int visibility_level,
    int pass_l) {
    grammar_verb *gv = prn_get_parsing_grammar(prn);
    pvp_test_begins(OUT);
    WRITE("if ((self ");
    compile_has_property(OUT, prn);
    WRITE(") ");
    pvp_test_words(OUT, prn->word_ref1, prn->word_ref2);
    pvp_test_passes(OUT, visibility_level, pass_l);
    if (gv) {
        pvp_test_begins(OUT);
        WRITE("if ((self ");
        compile_has_property(OUT, prn);
        WRITE(") && (PRN_PN_%d() == GPR_PREPOSITION)) ", prn->allocation_id);
        pvp_test_passes(OUT, visibility_level, pass_l);
    }
}
void pvp_test_begins(OUTPUT_STREAM) {
    WRITE("wn = try_from_wn; ");
}
void pvp_test_words(OUTPUT_STREAM, int w1, int w2) {
    int i;
    for (i=w1; i<=w2; i++) {
        WRITE(" && ");
        WRITE("(NextWordStopped()==");
        isn_compile_dictionary_word(OUT, lw_array[i].lw_text, FALSE);
        WRITE(")");
    }
    WRITE(") ");
}
void pvp_test_passes(OUTPUT_STREAM, int visibility_level, int pass_l) {
    WRITE("{\n"); INDENT;
    WRITE("try_from_wn = wn;\n");
```

```
    WRITE("g = true;\n");
    if (visibility_level == 2) WRITE("f = true;\n");
    if (pass_l >= 0) WRITE("jump pvp_pass_L_%d;\n", pass_l);
    OUTDENT; WRITE("}\n");
}

void finish_parsing_visible_properties(OUTPUT_STREAM) {
    OUTDENT; WRITE("}\n");
    WRITE("! try_from_wn is now advanced past any visible property values\n");
    WRITE("wn = try_from_wn;\n");
}
```

Test Scripts 13/test

Purpose

A rudimentary but useful testing system built in to IF produced by Inform, allowing short sequences of commands to be concisely noted in the source text and tried out in the Inform application using the TEST command.

Template interpreter commands

```
0   {-callv:write_test_text}
0   {-callv:compile_test_switch}
0   {-callv:compile_test_printout}
1   {-routine:InternalTestCases}
```

Definitions

¶1. Test scenarios are used for the "TEST" command: they consist of a string of commands in text format, with a few stipulations on place and possessions attached.

```
define MAX_LENGTH_OF_SCRIPT 10000                    including length byte, so the max no of chars is one less
define MAX_LENGTH_OF_COMMAND 100                     any single command must be this long or shorter
define MAX_POSSESSIONS_PER_SCENARIO 16

typedef struct test_scenario {
    int name_wn;                                     word number of single word identifying the test
    char text_of_script[MAX_LENGTH_OF_SCRIPT];
    int commands_wn;                                 word number of quoted text of command sequence
    struct world_object *place;                      room we need to be in to perform test
    int no_possessions;                              number of required possessions of player
    struct world_object *possessions[MAX_POSSESSIONS_PER_SCENARIO];   what they are
    struct parse_node *sentence_test_declared_at;
    MEMORY_MANAGEMENT
} test_scenario;
```

The structure test_scenario is private to this section.

¶2.

```
define HEADLINE_INTT 1
define SENTENCE_INTT 2
define DESCRIPTION_INTT 3
define DIMENSIONS_INTT 4
define EVALUATION_INTT 5
define EQUATION_INTT 6
```

```
typedef struct internal_test_case {
    int itc_code;                                   one of the above *_INTT values
    int word_ref1, word_ref2;                       text supplying the case
    struct parse_node *itc_defined_at;
    MEMORY_MANAGEMENT
} internal_test_case;
```

The structure internal_test_case is private to this section.

```
sentence_handler TEST_SH_handler =
    { SENTENCE_NT, TEST_VB, 2, new_test_text };
```

```
void new_test_text(parse_node *PN) {
    int x1, x2, i, j, k;
    test_scenario *test;
    char *p, individual_command[MAX_LENGTH_OF_SCRIPT];
    [[x1, x2 <-- PN->down->next]];
    if [[x1, x2 == test ### OPENBRACKET internal CLOSEBRACKET]] {
        int w = x1+1, it = -1;
        [[x1, x2 <-- PN->down->next->next]];
        if [[word w == headline]] it = HEADLINE_INTT;
        if [[word w == sentence]] it = SENTENCE_INTT;
        if [[word w == description]] it = DESCRIPTION_INTT;
        if [[word w == dimensions]] it = DIMENSIONS_INTT;
        if [[word w == evaluation]] it = EVALUATION_INTT;
        if [[word w == equation]] it = EQUATION_INTT;
        if (it == -1)
            sentence_problem(_P_(C13UnknownInternalTest),
                "that's an internal test case which I don't know",
                "so I am taking no action.");
        else
            new_internal_test_case(it, x1, x2);
        return;
    }
    if (x2 != x1+1) {
        sentence_problem(_P_(C13TestMultiWord),
            "test scenarios must have single-word names",
            "so 'test garden with ...' is allowed but not 'test garden "
            "gate with...'");
        return;
    }
    LOOP_OVER(test, test_scenario) {
        if (compare_words(x1+1, test->name_wn)) {
            quote_source(1, test->sentence_test_declared_at);
            quote_source(2, current_sentence);
```

```
        handmade_problem(_P_(C13TestDuplicate));
        issue_problem_segment(
            "Two test scripts have been set up with the same name: "
            "%1 and %2.");
        issue_problem_end();
    }
}
test = CREATE(test_scenario);
test->name_wn = x1 + 1;
test->sentence_test_declared_at = current_sentence;
[[x1, x2 <-- PN->down->next->next]];
dequote_word(x1);
p = lw_array[x1++].lw_text;
for (i=0, j=0, k=0; p[i]; i++) {
    char c = tolower(p[i]);
    if (c == ' ') {
        int l;
        if (k == 0) continue;
        for (l=i+1; p[l]; l++) if (p[l] != ' ') break;
        if ((p[l] == '/') || (p[l] == 0)) continue;
    }
    if (c == '/') {
        individual_command[k] = 0;
        k = 0;
        check_test_command(individual_command);
    } else individual_command[k++] = c;
    test->text_of_script[j++] = c;
    if (j == MAX_LENGTH_OF_SCRIPT) {
        LOG("Test script: %s\nLength: %d\n",
            lw_array[test->commands_wn].lw_rawtext,
            strlen(lw_array[test->commands_wn].lw_rawtext));
        sentence_problem(_P_(Untestable),                    or at least impracticable: the limit's huge
            "this test script is too long",
            "and exceeds the capacity of a single TEST command. TEST "
            "is only designed for short batches of commands, whose "
            "total text runs to about 250 letters. For really long "
            "test run-throughs, it's probably best to use the Skein, "
            "but another trick would be to divide your script into "
            "several mini-scenarios with their own TEST commands, and "
            "then make a 'test all' command which would run through "
            "all of them.");
        return;
    }
}
if (k > 0) {
    individual_command[k] = 0;
    check_test_command(individual_command);
}
if ((j>0) && (test->text_of_script[j-1] == '/'))
    test->text_of_script[j++] = ' ';
test->text_of_script[j] = 0;
test->place = NULL;
test->no_possessions = 0;
```

```
    while (x2 >= x1) {
        int clause = clause_identifier(x1++);
        int x0;
        x0 = x1;
        world_object *wo = NULL;
        if (clause == 0) {
            DidntRecognise:
            sentence_problem(_P_(C13TestBadRequirements),
                "I didn't recognise the requirements for this test scenario",
                "which should be 'test ... with ... in ...' or '... "
                "holding ...'");
            return;
        }
        while ((x1 < x2) && (clause_identifier(x1)==0)) x1++;
        if (x1<x2) x1--;
        [[x0, x1 == the ... --> x0, x1]];
        if (x0<=x1)
            wo = parse_world_object(x0, x1, FALSE);
        if (wo == NULL) goto DidntRecognise;
        switch(clause) {
            case 1: test->place = wo; break;
            case 2: if (test->no_possessions >=
                    MAX_POSSESSIONS_PER_SCENARIO) goto DidntRecognise;
                test->possessions[test->no_possessions++] = wo;
                break;
        }
        x1++;
    }
}
void check_test_command(char *p) {
    if (strcmp(p, "undo") == 0) {
        sentence_problem(_P_(C13TestContainsUndo),
            "this test script contains an UNDO command",
            "which the story file has no way to automate the running of. "
            "(An UNDO is such a complete reversion to the previous state "
            "that it would necessarily lose where it had got to in the "
            "script, and might even go round in circles indefinitely.)");
        return;
    }
    if (strlen(p) > MAX_LENGTH_OF_COMMAND) {
        sentence_problem(_P_(C13TestCommandTooLong),
            "this test script contains a command which is too long",
            "and cannot be fed into Inform for automatic testing. "
            "(The format for a test script is a sequence of commands, "
            "divided up by slashes '/': maybe you forgot these divisions?)");
        return;
    }
}
int clause_identifier(int wn) {
    if (unexpectedly_upper_case(wn)) return 0;
    if [[word wn == in]] return 1;
    if [[word wn == holding/COMMA/and]] return 2;
    if [[word wn == with]] {
```

```
        sentence_problem(_P_(C13TestDoubleWith),
            "the second 'with' should be 'holding'",
            "as in 'test frogs with \"get frogs\" holding net' rather than "
            "'test frogs with \"get frogs\" with net'.");
        return 2;
    }
    return 0;
}
void write_test_text(OUTPUT_STREAM) {
    int j;
    test_scenario *test;
    LOOP_OVER(test, test_scenario) {
        WRITE("Array TestText_%d ->\n", test->allocation_id); INDENT;
        WRITE("\"");
        isn_compile_string(OUT, test->text_of_script,
            ISN_EXPAND_APOSTROPHES + ISN_RECOGNISE_APOSTROPHE_SUBSTITUTION
            + ISN_FOR_ARRAY);
        WRITE("||||\";\n");
        OUTDENT;
        WRITE("Array TestReq_%d -->\n", test->allocation_id); INDENT;
        if (test->place == NULL) WRITE("0 ");
        else WRITE("%s ", wo_get_I6_representation(test->place));
        for (j=0; j<test->no_possessions; j++) {
            if (test->possessions[j] == NULL) WRITE("0 ");
            else WRITE("%s ", wo_get_I6_representation(test->possessions[j]));
        }
        WRITE("0;\n");
        OUTDENT;
    }
}
void compile_test_switch(OUTPUT_STREAM) {
    test_scenario *test;
    LOOP_OVER(test, test_scenario) {
        int i, l = 0;
        char *p = test->text_of_script;
        for (i=0; p[i]; i++, l++)
            if ((p[i] == '[') && (p[i+1] == '\'') && (p[i+2] == ']'))
                l -= 2;
        WRITE("'%s//': TestStart(TestText_%d, TestReq_%d, %d);\n",
            lw_array[test->name_wn].lw_rawtext,
            test->allocation_id, test->allocation_id, l);
    }
}
void compile_test_printout(OUTPUT_STREAM) {
    test_scenario *test;
    LOOP_OVER(test, test_scenario)
        WRITE("print \"'test %s'^\";\n",
            lw_array[test->name_wn].lw_rawtext);
}
```

The function write_test_text is invoked by a command in a .i6t template file.
The function compile_test_switch is invoked by a command in a .i6t template file.
The function compile_test_printout is invoked by a command in a .i6t template file.

§1.

```
void new_internal_test_case(int code, int x1, int x2) {
    internal_test_case *itc = CREATE(internal_test_case);
    itc->itc_code = code; itc->word_ref1 = x1; itc->word_ref2 = x2;
    itc->itc_defined_at = current_sentence;
}
STREAM *itc_save_dl = NULL, *itc_save_OUT = NULL;
void compile_InternalTestCases_routine(OUTPUT_STREAM) {
    internal_test_case *itc; int n = 0;
    itc_save_OUT = OUT;
    phsf_create_nonphrase_stack_frame();
    WRITE("[ InternalTestCases;\n"); INDENT;
    LOOP_OVER(itc, internal_test_case) {
        n++;
        if (itc->itc_code == HEADLINE_INTT) {
            n = 0;
            WRITE("style bold; print \"^");
            print_raw_text_to_file(itc->word_ref1, itc->word_ref2, OUT);
            WRITE("^\"; style roman;\n");
            continue;
        }
        WRITE("print \"%d. ", n);
        print_raw_text_to_file(itc->word_ref1, itc->word_ref2, OUT);
        WRITE("^\";\n");
        WRITE("print \"");
        current_sentence = itc->itc_defined_at;
        switch (itc->itc_code) {
            case SENTENCE_INTT: {
                int required_S_form = SV_PRODUCTION;
                ⟨Perform an internal test of the sentence converter 2⟩;
                break;
            }
            case DESCRIPTION_INTT: {
                int required_S_form = SN_PRODUCTION;
                ⟨Perform an internal test of the sentence converter 2⟩;
                break;
            }
            case EVALUATION_INTT: {
                specification *spec = parse_expression(itc->word_ref1, itc->word_ref2, VALUE_EXPCON);
                typecheck_without_expectations(spec);
                kind_of_value *kov = spec_evaluates_to(spec);
                WRITE("Kind of value: ");
                ⟨Begin reporting on the internal test case 3⟩;
                log_kind_of_value(kov);
                if (kov_quasinumerical(kov)) LOG(" scaled at k=%d", kov_scale_factor(kov));
                ⟨End reporting on the internal test case 4⟩;
                WRITE("^Prints as: \", (%s) ",
                    kov_get_name_of_printing_rule(kov));
                TEMPORARY_STREAM;
```

```
                spec_compile(TEMP, spec);
                STREAM_COPY(OUT, TEMP);
                CLOSE_TEMPORARY_STREAM;
                WRITE(", \"^");
                break;
            }
            case DIMENSIONS_INTT:
                ⟨Begin reporting on the internal test case 3⟩;
                log_unit_analysis();
                ⟨End reporting on the internal test case 4⟩;
                break;
            case EQUATION_INTT:
                eqn_internal_test(itc->word_ref1, itc->word_ref2);
                break;
        }
        WRITE("^\";\n");
    }
    OUTDENT; WRITE("];\n");
    phsf_remove_nonphrase_stack_frame();
}
void begin_reporting_internal_test(void) {
    ⟨Begin reporting on the internal test case 3⟩;
}
void end_reporting_internal_test(void) {
    ⟨End reporting on the internal test case 4⟩;
}
```

The function compile_InternalTestCases_routine is invoked by a command in a `.i6t` template file.
The function begin_reporting_internal_test is called from 10/eqns.
The function end_reporting_internal_test is called from 10/eqns.

§2.

⟨Perform an internal test of the sentence converter 2⟩ ≡

```
    meaning_list *ml = NULL;
    pcalc_prop *prop = NULL;
    int tc = FALSE;
    if (required_S_form == SV_PRODUCTION)
        ml = SP_condition(itc->word_ref1, itc->word_ref2, FALSE);
    else
        ml = SP_condition(itc->word_ref1, itc->word_ref2, TRUE);
    if ((ml) && (ml_production(ml) == COND_PRODUCTION) &&
        (ml_down(ml)) && (ml_production(ml_down(ml)) == required_S_form)) {
        prop = S_subtree_to_proposition(ml_down(ml), NULL);
        tc = prop_type_check(prop, tc_no_problem_reporting());
    }
    ⟨Begin reporting on the internal test case 3⟩;
    if (ml == NULL) LOG("Failed: not a condition");
    else if (ml_production(ml) != COND_PRODUCTION) LOG("Failed: tree head is $p", ml_production(ml));
    else if (ml_down(ml) == NULL) LOG("Failed: treeless COND");
    else if (ml_production(ml_down(ml)) != required_S_form)
        LOG("Failed: not SV but $p", ml_production(ml_down(ml)));
    else {
```

```
        LOG("$D\n", prop);
        if (tc == FALSE) LOG("Failed: proposition would not type-check\n");
        prop_type_check(prop, tc_problem_logging());
    }
    ⟨End reporting on the internal test case 4⟩;
```

This code is used in §1.

§3.

⟨Begin reporting on the internal test case 3⟩ ≡

```
    itc_save_dl = dl; dl = itc_save_OUT; logging_to_I6_text = TRUE;
```

This code is used in §1,2,1,2,1,2.

§4.

⟨End reporting on the internal test case 4⟩ ≡

```
    dl = itc_save_dl; logging_to_I6_text = FALSE;
```

This code is used in §1,2,1,2,1,2.

14 Program Control

`14/main`: *Main Routine.w* As with all C programs, NI begins execution in the `main` routine, reading command-line arguments to modify its behaviour. This is where we define `main`.

`14/i6t`: *I6 Template Interpreter.w* Inform 6 meta-language is the language used by template files (with extension `.i6t`). It is not itself I6 code, but a list of instructions for making I6 code: most of the content is to be copied over verbatim, but certain escape sequences cause NI to insert more elaborate material, or to do something active. The entire top-level logic of NI is carried out by interpreting the `Main.i6t` file in this way.

Main Routine 14/main

Purpose

As with all C programs, NI begins execution in the `main` routine, reading command-line arguments to modify its behaviour. This is where we define `main`.

Definitions

¶1. From the local environment, we also extract the time at which NI is running. We need this to fill in the correct date of creation in the bibliographic data for a work, for instance.

```
struct tm *the_present = NULL;
```

---

§**1.** As with all C programs, NI begins execution in the `main` routine, which takes command-line arguments with the standard parameters `argc` and `argv`.

In this program, `main` consists only of command-line processing and the very minimum setup necessary to begin an I6 inclusion before handing over to the meta-language interpreter to run through `Main.i6t`.

NI returns only two possible values to the shell, either here or via `exit(1)` in the case of fatal errors: 0 if it completed its run with no errors, 1 if errors were produced.

```
int main(int argc, char *argv[]) {
    int i, a = 1, source_specified = FALSE;
    char *val;
    STREAM_WRITE(STDOUT, "%s build %s has started.\n", NI_VERSION, NI_BUILD);
    STREAM_FLUSH(STDOUT);
    what_day_is_it();
    bundle_name = NULL;
    crash_on_internal_errors = FALSE;
    crash_on_all_errors = FALSE;
    story_filename_extension = NULL;
    make_pathname_of_extensions();
    if (argc == a) return 0;
    while (a < argc) {
        char *option = argv[a];
        if ((option[0] == '-') && (option[1] == '-')) option++;
        if (strcmp("-log", option) == 0) {
            Non-functional for now
            a++; continue;
        }
        if (strlen(option) > 5) {
            if ((option[0] == '-') && (option[1] == 'l') && (option[2] == 'o') &&
                (option[3] == 'g') && (option[4] == '=')) {
                set_dl_from_command_line(option+5);
                a++; continue;
            }
        }
        if (strcmp("-rng", option) == 0) {
            fix_rng_at_start_of_play = TRUE;
```

```
        rng_seed_at_start_of_play = -16339;
        a++; continue;
    }
    if ((val = cli_pair("-extension", option)) != 0) {
        story_filename_extension = val;
        a++; continue;
    }
    if (strcmp("-release", option) == 0) {
        for_release = TRUE;
        a++; continue;
    }
    if (strcmp("-noprogress", option) == 0) {
        show_progress_indicator = FALSE;
        a++; continue;
    }
    if (strcmp("-census", option) == 0) {
        census_mode = TRUE;
        a++; continue;
    }
    if (strcmp("-sigils", option) == 0) {
        echo_problem_message_sigils = TRUE;
        a++; continue;
    }
    if (strcmp("-gdb", option) == 0) {
        crash_on_internal_errors = TRUE;
        a++; continue;
    }
    if (strcmp("-gdball", option) == 0) {
        crash_on_all_errors = TRUE;
        a++; continue;
    }
    if (strcmp("-noindex", option) == 0) {
        do_not_generate_index = TRUE;
        a++; continue;
    }
    if (strcmp("-package", option) == 0) {
        source_specified = TRUE;
        bundle_name = argv[a+1];
        a = a+2; continue;
    }
    if (strcmp("-rules", option) == 0) {
        strcpy(pathname_of_built_in_extensions, argv[a+1]);
        a = a+2; continue;
    }
    if (source_specified) {
        fatal_error("Unknown parameter at the command line");
    } else {
        source_specified = TRUE;
        source_text_file = argv[a]; a++;
    }
}
```

```
    set_VM_identifier(story_filename_extension);
    if ((census_mode == FALSE) && (source_specified == FALSE))
        fatal_error("Except in census mode, source text must be supplied");
    if ((census_mode) && (source_specified))
        fatal_error("In census mode, no source text may be supplied");
    open_log_files();
    LOG("NI called as: ");
    for (i=0; i<argc; i++) LOG("%s ", argv[i]);
    LOG("\n");
    interpret_I6T_file(NULL, NULL, "Main.i6t", -1);
    LOG("Total of %d files written as streams.", total_file_writes);
    close_log_files();
    STREAM_WRITE(STDOUT, "%s has finished.\n", NI_VERSION);
    STREAM_FLUSH(STDOUT);
    if (problem_count > 0) return 1; return 0;
}
void what_day_is_it(void) {
    time_t right_now = time(NULL);
    long long int rni = (long long int) right_now;
    if (rni < 0) right_now = (time_t) 0;
    the_present = localtime(&right_now);
}
char *cli_pair(char *wanted, char *got) {
    int i;
    if (strlen(wanted)+1 >= strlen(got)) return NULL;
    for (i=0; wanted[i]; i++) if (wanted[i] != got[i]) return NULL;
    if (got[i] != '=') return NULL;
    return got+i+1;
}
```

The function main is where execution begins.

I6 Template Interpreter 14/i6t

Purpose

Inform 6 meta-language is the language used by template files (with extension `.i6t`). It is not itself I6 code, but a list of instructions for making I6 code: most of the content is to be copied over verbatim, but certain escape sequences cause NI to insert more elaborate material, or to do something active. The entire top-level logic of NI is carried out by interpreting the `Main.i6t` file in this way.

Template interpreter commands

```
32    {-callv:report_unacted_upon_interventions}
33    {-callv:compile_build_number}
```

Definitions

¶1. The user (or an extension used by the user) is allowed to register gobbets of I6T code to be used before, instead of, or after any whole segment or named part of a segment of the template layer: the following structure holds such a request.

```
typedef struct I6T_intervention {
    int intervention_stage;                                  −1 for before, 0 for instead, 1 for after
    char *segment_name;
    char *part_name;                                         or NULL to mean the entire segment
    char *I6T_matter;                                        to be used at the given position, or NULL
    char *alternative_segment;                               to be used at the given position, or NULL
    int segment_found;                                       did the segment name match one actually read?
    int part_found;                                          did the part name?
    struct parse_node *where_intervention_requested;         at what sentence?
    MEMORY_MANAGEMENT
} I6T_intervention;
```

The structure I6T_intervention is private to this section.

¶2. The following flag is set by the `-noindex` command line option.

```
int do_not_generate_index = FALSE;
```

---

§**1. Syntax of I6T files.** The syntax of these files has been designed so that a valid I6T file is also a valid Inweb section file. This means that no tangling is required to make the I6T files: they can be, and indeed are, simply copied verbatim from Appendix B of the source web.

Formally, an I6T file consists of a preamble followed by one or more parts. The preamble takes the form:

```
B/name: Longer Form of Name.

@Purpose: ...

@-------------------------------------------------------------------------------
```

(for some number of dashes). Each part begins with a heading line in the form

```
@p Title.
```

At some point during the part, a heading line

```
@c
```

introduces the code of the part. When NI interprets an I6T file, it ignores the preamble and the material in every part before the `@c` heading: these are commentary.

It actually doesn't matter if a template file contains lines longer than this, so long as they do not occur inside `{-lines:...}` and `{-endlines}`, and so long as no individual braced command `{-...}` exceeds this length.

```
define MAX_I6T_LINE_LENGTH 1024
```

§**2.** We can regard the whole NI program as basically a filter: it copies its input, the `Main.i6t` template file, directly into its output, but making certain replacements along the way.

The code portions of `.i6t` files are basically written in I6, but with a special escape syntax:

```
{-command:argument}
```

tells NI to act *immediately* on the I6T command given, with the argument supplied. One of these commands is special:

```
{-lines:commandname}
```

tells NI that all subsequent lines in the I6T file, up to the next `{-endlines}`, are to be read as a series of arguments for the `commandname` command. Thus,

```
{-lines:admire}
Jackson Pollock
Paul Klee
Wassily Kandinsky
{-endlines}
```

is a shorthand form for:

```
{-admire:Jackson Pollock}{-admire:Paul Klee}{-admire:Wassily Kandinsky}
```

The following comment syntax is useful mainly for commenting out commands:

```
{-! Something very clever happens next.}
```

The commands all either instruct NI to do something (say, traverse the parse tree and convert its assertions to inferences) but output nothing, or else to compile some I6 code to the output. There are no control structures, no variables: I6T commands do not amount to a programming language.

§**3.** I7 expressions can be included in I6T code exactly as in inline invocation definitions: thus

```
Constant FROG_CLASS = (+ pond-dwelling amphibian +);
```

will expand "pond-dwelling amphibian" into the I6 translation of the kind of object with this name. Because of this syntax, one has to watch out for I6 code like so:

```
if (++counter_of_some_kind > 0) ...
```

which can trigger an unwanted `(+`.

§**4.** It is not quite true that the following routine acts as a filter from input to output, because:

(i) It skips the preamble and the commentary portion of each part in the input.

(ii) It has an `active` mode, outside of which it ignores most commands and copies no output – it begins in active mode and leaves it only when NI issues problem messages, so that subsequent commands almost certainly cannot safely be used. In a successful compilation run, the interpreter remains in active mode throughout. Otherwise, generally speaking, it goes into passive mode as soon as an I6T command has resulted in Problem messages, and then in stays in passive mode until the output file is closed again; then it goes back into active mode to carry out some shutting-down-gracefully steps.

(iii) The output file is not always open. In fact, it starts unopened (and with `of` set to null); two of the I6T commands open and close it. When the file isn't open, no output can be written, but I6T commands telling NI to do something can still take effect: in fact, the `Main.i6t` file begins with dozens of I6T commands before the output file is opened, and concludes with a couple of dozen more after it has been closed. For almost all of the I6T files, however, the output file is open throughout; the other exception being `Types.i6t`, a long run of commands to create NI's data types, which is interpreted before the output file is opened.

(iv) It can abort, cleanly exiting NI when it does so, if a global flag is set as a result of work done by one of its commands. In fact, this is used only to exit NI early after performing an extension census when called with the command line option `-census`, and can never happen on a compilation run, whatever problems or disasters may occur.

§**5.** The I6T interpreter is a single routine which implements the description above:

```
define MAX_I6T_COMMAND_LENGTH 128

void interpret_I6T_file(OUTPUT_STREAM, char *sf, char *segment_name, int N_escape) {
    FILE *Input_File = NULL;
    char default_command[MAX_I6T_COMMAND_LENGTH]; default_command[0] = 0;
    char heading_name[MAX_I6T_LINE_LENGTH+1]; heading_name[0] = 0;
    int active = TRUE, closed = FALSE, indexing = FALSE, skip_part = FALSE, comment = TRUE;
    int no_phases = 0, col = 1, cr, sfp = 0;
    if (segment_name) I6T_file_intervene(OUT, -1, segment_name, NULL);
    if ((segment_name) && (I6T_file_intervene(OUT, 0, segment_name, NULL))) goto OmitFile;
    if (segment_name) {
        ⟨Open the I6 template file 6⟩;
        comment = TRUE;
    } else comment = FALSE;
    do {
        ⟨Read next character from I6T stream 9⟩;
        NewCharacter: if (cr == EOF) break;
        char I6T_buffer[MAX_I6T_LINE_LENGTH+1];                    storage for an I6T command
        char *command = "", *argument = "";
        if (abort_I6T_interpreter) {                               in effect, if NI has thrown an exception
            fclose(Input_File); exit(0);
        }
```

```
        if ((cr == '@') && (col == 1)) {
            ⟨Read the rest of line as an at-heading 10⟩;
            ⟨Act on the at-heading, going in or out of comment mode as appropriate 11⟩;
            continue;
        }
        if (comment == FALSE) {
            if (default_command[0] != 0) {
                if ((cr == 10) || (cr == 13)) continue;                    skip blank lines here
                ⟨Set the command to the default, and read rest of line as argument 12⟩;
                if ((argument[0] == '!') || (argument[0] == 0)) continue;    skip blanks and comments
                if (strcmp(argument, "{-endlines}") == 0) default_command[0] = 0;
                else ⟨Act on I6T command and argument 16⟩;
                continue;
            }
            if (cr == '{') {
                ⟨Read next character from I6T stream 9⟩;
                if (cr == '-') {
                    ⟨Read up to the next close brace as an I6T command and argument 13⟩;
                    if (command[0] == '!') continue;
                    ⟨Act on I6T command and argument 16⟩;
                    continue;
                } else if ((cr == 'N') && (N_escape >= 0)) {
                    ⟨Read next character from I6T stream 9⟩;
                    if (cr == '}') {
                        WRITE("%d", N_escape);
                        continue;
                    }
                    if ((OUT) && (active)) WRITE("{N");
                    goto NewCharacter;
                } else {                                    otherwise the open brace was a literal
                    if ((OUT) && (active)) STREAM_PUT(OUT, '{');
                    goto NewCharacter;
                }
            }
            if (cr == '(') {
                ⟨Read next character from I6T stream 9⟩;
                if (cr == '+') {
                    ⟨Read up to the next plus close-bracket as an I7 expression 14⟩;
                    ⟨Evaluate the I7 expression resulting 15⟩;
                    continue;
                } else {                                  otherwise the open bracket was a literal
                    if ((OUT) && (active)) STREAM_PUT(OUT, '(');
                    goto NewCharacter;
                }
            }
            if ((OUT) && (active)) STREAM_PUT(OUT, cr);
        }
    } while (cr != EOF);
    if (Input_File) { if (dl) STREAM_FLUSH(dl); fclose(Input_File); }
    OmitFile:
    if (segment_name) I6T_file_intervene(OUT, 1, segment_name, NULL);
}
```

The function interpret_I6T_file is called from 10/isin and 14/main.

§**6.** We look for the `.i6t` files first in the `I6T` subfolder of the `Materials` folder for the project, then failing that (as we almost always will) in the `Reserved` part of the built-in extensions area. Note the way we handle a failure to open the file: if this should happen to `Main.i6t` then we must assume that the problems machinery is not in working order, because the commands to initialise this are themselves in `Main.i6t`. If it happens to any other segment then, conversely, `Main.i6t` must have opened safely and processed numerous early commands, so we can safely issue Problems.

⟨Open the I6 template file 6⟩ ≡

```
⟨Look for the I6T file in the Materials folder 7⟩;
if (Input_File == NULL) {
    ⟨Look for the I6T file in the Reserved built-in extensions area 8⟩;
    if (Input_File == NULL) {
        if (strcmp(segment_name, "Main.i6t") == 0)
            fatal_error2("Error: can't open input file", segment_name);
        STREAM_WRITE(STDERR, "inform: Unable to open segment <%s>\n", segment_name);
        unlocated_problem(_P_(BelievedImpossible),                    or anyway not usefully testable
            "I couldn't open a requested I6T segment: see the console "
            "output for details.");
    }
}
```

This code is used in §5.

§**7.** The following is a little wasteful in that it will be tried 20 or so times in a typical NI run, almost always in vain. On a platform where attempts to `fopen` nonexistent directories are slow, we might care about that.

⟨Look for the I6T file in the Materials folder 7⟩ ≡

```
int i;
char the_filename[MAX_FILENAME_LENGTH];
if (bundle_name) strcpy(the_filename, bundle_name);
else the_filename[0] = 0;
i = strlen(the_filename)-1;
while ((i>0) && (the_filename[i] != '.')) i--;
if (i>0) the_filename[i++] = ' ';
sprintf(the_filename+i, "Materials%cI6T%c%s",
    FOLDER_SEPARATOR, FOLDER_SEPARATOR, segment_name);
Input_File = iso_fopen(the_filename, "r");
```

This code is used in §6.

§**8.** Similarly, but more simply:

⟨Look for the I6T file in the Reserved built-in extensions area 8⟩ ≡

```
char the_filename[MAX_FILENAME_LENGTH];
sprintf(the_filename, "%s%cReserved%c%s",
    pathname_of_built_in_extensions,
    FOLDER_SEPARATOR, FOLDER_SEPARATOR, segment_name);
Input_File = iso_fopen(the_filename, "r");
```

This code is used in §6.

§**9.** I6 template files are encoded as ISO Latin-1, not as Unicode UTF-8, so ordinary `fgetc` is used, and no BOM marker is parsed. Lines are assumed to be terminated with either `0x0a` or `0x0d`. (Since blank lines are harmless, we take no trouble over `0a0d` or `0d0a` combinations.) The built-in template files, almost always the only ones used, are line terminated `0x0a` in Unix fashion.

⟨Read next character from I6T stream 9⟩ ≡

```
if (Input_File) cr = fgetc(Input_File);
else if (sf) {
    cr = sf[sfp]; if (cr == 0) cr = EOF; else sfp++;
} else cr = EOF;
col++; if ((cr == 10) || (cr == 13)) col = 0;
```

This code is used in §5,10,12,13,14,5,10,12,13,14,5.

§**10.** Anything following an at-character in the first column is a heading:

⟨Read the rest of line as an at-heading 10⟩ ≡

```
int i = 0;
while (i<MAX_I6T_LINE_LENGTH) {
    ⟨Read next character from I6T stream 9⟩;
    if ((cr == 10) || (cr == 13)) break;
    I6T_buffer[i++] = (char) cr;
}
I6T_buffer[i] = 0;
command = I6T_buffer;
```

This code is used in §5.

§**11.** As can be seen, only a small minority of Inweb syntaxes are allowed: in particular, no `@d` or angle-bracketed macros. This interpreter is not a full-fledged tangler.

⟨Act on the at-heading, going in or out of comment mode as appropriate 11⟩ ≡

```
if ((command[0] == 'p') && (command[1] == ' ')) {
    int i;
    if ((heading_name[0] != 0) && (segment_name))
        I6T_file_intervene(OUT, 1, segment_name, heading_name);
    strcpy(heading_name, command+2);
    i = strlen(heading_name)-1;
    while ((i>=0) &&
        ((heading_name[i] == ' ') || (heading_name[i] == '\t') || (heading_name[i] == '.'))) i--;
    heading_name[i+1] = 0;
    if (heading_name[0] == 0)
        internal_error("Empty heading name in I6 template file");
    comment = TRUE; skip_part = FALSE;
    if (segment_name) {
        I6T_file_intervene(OUT, -1, segment_name, heading_name);
        if (I6T_file_intervene(OUT, 0, segment_name, heading_name)) skip_part = TRUE;
    }
    continue;
}
if (strcmp(command, "c") == 0) {
    if (skip_part == FALSE) comment = FALSE;
    continue;
}
if (command[0] == '-') continue;
```

```
    if (strncmp(command, "Purpose:", 8) == 0) continue;
    LOG("heading: <%s>\n", command);
    internal_error("Unknown heading format in I6 template file");
```

This code is used in §5.

§**12.** Here we are in `{-lines:...}` mode, so that the entire line of the file is to be read as an argument. Note that initial and trailing white space on the line is deleted: this makes it easier to lay out I6T template files tidily.

⟨Set the command to the default, and read rest of line as argument 12⟩ ≡

```
    int i;
    strcpy(I6T_buffer, default_command);
    command = I6T_buffer;
    i = strlen(command)+1;
    argument = command + i;
    I6T_buffer[i++] = (char) cr;
    while (i<MAX_I6T_LINE_LENGTH) {
        ⟨Read next character from I6T stream 9⟩;
        if ((cr == 10) || (cr == 13)) break;
        I6T_buffer[i++] = (char) cr;
    }
    I6T_buffer[i] = 0;
    while ((argument[0] == ' ') || (argument[0] == '\t')) argument++;
    while ((I6T_buffer[i-1] == ' ') || (I6T_buffer[i-1] == '\t'))
        I6T_buffer[--i] = 0;
```

This code is used in §5.

§**13.** And here we read a normal command. The command name must not include `}` or `:`. If there is no `:` then the argument is left unset (so that it will be the empty string: see above). The argument must not include `}`.

⟨Read up to the next close brace as an I6T command and argument 13⟩ ≡

```
    int i=0;
    while (i<MAX_I6T_LINE_LENGTH) {
        ⟨Read next character from I6T stream 9⟩;
        if ((cr == '}') || (cr == EOF)) break;
        I6T_buffer[i++] = (char) cr;
    }
    I6T_buffer[i] = 0;
    command = I6T_buffer;
    for (i=0; I6T_buffer[i]; i++)
        if (I6T_buffer[i] == ':') {
            I6T_buffer[i] = 0;
            argument = I6T_buffer+i+1;
        }
```

This code is used in §5.

§**14.** And similarly, for the `(+ ... +)` notation used to mark I7 material within I6:

⟨Read up to the next plus close-bracket as an I7 expression 14⟩ ≡

```
int i=0;
while (i<MAX_I6T_LINE_LENGTH) {
    ⟨Read next character from I6T stream 9⟩;
    if (cr == EOF) { I6T_buffer[i] = 0; break; }
    if ((cr == ')') && (i>0) && (I6T_buffer[i-1] == '+')) { I6T_buffer[i-1] = 0; break; }
    I6T_buffer[i++] = (char) cr;
}
```

This code is used in §5.

§**15.** We delegate:

⟨Evaluate the I7 expression resulting 15⟩ ≡

```
TEMPORARY_STREAM;
compile_I7_expression_from_I6(TEMP, I6T_buffer);
STREAM_COPY(OUT, TEMP);
CLOSE_TEMPORARY_STREAM;
```

This code is used in §5.

§**16. Acting on I6T commands.** Only a few commands work even in passive mode, but they include file-handling because the close-file command needs to be able to get out of passive mode and back into active (and besides, because the file still needs to be closed).

The `{-type:...}` command hands over the argument to a more specific interpreter, one which constructs data types and is to be found in Chapter 7.

The `{-segment:...}` command recursively calls the I6T interpreter on the supplied I6T filename, which means it acts rather like `#include` in C. Note that because we pass the current output file handle `of` through to this new invocation, it will have the file open if we do, and closed if we do. It will run in active mode, but that's fine, because we're in active mode too. It won't run in indexing mode, so `{-segment:...}` can't be used safely between `{-open-index}` and `{-close-index}`.

⟨Act on I6T command and argument 16⟩ ≡

```
⟨Act on an I6T file-handling command 17⟩;
⟨Act on the I6T lines command 18⟩;
if (active == FALSE) continue;
if (strcmp(command, "segment") == 0) { interpret_I6T_file(OUT, NULL, argument, -1); continue; }
if (strcmp(command, "type") == 0) { despatch_type_command(argument); continue; }
⟨Act on an I6T indexing command 19⟩;
⟨Act on an I6T command which keeps track of progress 20⟩;
⟨Act on the I6T counter command 21⟩;
⟨Act on the I6T value command 22⟩;
⟨Act on the I6T read-assertions command 23⟩;
⟨Act on the I6T callv command 24⟩;
⟨Act on the I6T call command 25⟩;
⟨Act on the I6T array command 26⟩;
⟨Act on the I6T routine command 27⟩;
⟨Act on the I6T test command 28⟩;
LOG("command: <%s> argument: <%s>\n", command, argument);
internal_error("Unknown I6T command in I6 template file");
```

This code is used in §5.

§**17.** This where the primary output from NI, the I6 file it compiles, is opened.

Note that closing this file again sends us back into active move: that is so that, whatever may have happened earlier, we can still follow I6T commands to deallocate memory, issue a Problems report, and so on, as part of the finishing-up process.

⟨Act on an I6T file-handling command 17⟩ ≡

```
if (strcmp(command, "open-file") == 0) {
    if (OUT) internal_error("output file already open");
    if (STREAM_OPEN_TO_FILE(inform6_file, build_filename(I6_OUTPUT_LEAFNAME), ISO_ENC) == FALSE)
        fatal_error("Can't open output file");
    OUT = inform6_file;
    continue;
}
if (strcmp(command, "close-file") == 0) {
    if (OUT) STREAM_CLOSE(OUT);
    OUT = NULL;
    active = TRUE;
    continue;
}
```

This code is used in §16.

§**18.** There is no corresponding code here to act on `{-endlines}` because it is not valid as a free-standing command: it can only occur at the end of a `{-lines:...}` block, and is acted upon above.

⟨Act on the I6T lines command 18⟩ ≡

```
if (strcmp(command, "lines") == 0) {
    truncated_strcpy(default_command, argument, MAX_I6T_COMMAND_LENGTH);
    continue;
}
```

This code is used in §16.

§**19. Indexing commands.** Commands in between `{-open-index}` and `{-close-index}` are skipped when NI has been called with a command-line switch to disable the index. (As is done by `intest`, to save time.) `{-index:name}` opens the index file called `name`.

⟨Act on an I6T indexing command 19⟩ ≡

```
if (strcmp(command, "open-index") == 0) { indexing = TRUE; continue; }
if (strcmp(command, "close-index") == 0) { indexing = FALSE; continue; }
if ((indexing) && (do_not_generate_index)) continue;
if (strcmp(command, "index") == 0) {
    int i;
    char leafname[MAX_FILENAME_LENGTH];
    strcpy(leafname, argument);
    for (i=0; leafname[i]; i++)
        if ((leafname[i] == ' ') || (leafname[i] == '=')) {
            leafname[i] = 0;
            break;
        }
    strcpy(leafname+i, ".png");
    char titling[MAX_FILENAME_LENGTH], explanation[MAX_FILENAME_LENGTH],
        image[MAX_FILENAME_LENGTH], caption[MAX_FILENAME_LENGTH];
    strcpy(image, leafname);
```

```
        strcpy(leafname+i, ".html");
        strcpy(titling, argument); explanation[0] = 0; caption[0] = 0;
        for (i=0; titling[i]; i++)
            if (titling[i] == '=') {
                titling[i] = 0;
                if (explanation[0] == 0) strcpy(explanation, titling+i+1);
                else strcpy(caption, titling+i+1);
            }
        for (i=0; explanation[i]; i++)
            if (explanation[i] == '=') {
                explanation[i] = 0;
            }
        set_thumbnail_image(image, caption);
        open_index_file(leafname, titling, -1, explanation);
        continue;
    }
```

This code is used in §16.

§**20. Annotating progress.** The debugging log has headings at two levels: grand Phases, and mere Stages. These headings are inserted by the `{-log-phase:...}` and `{-log:...}` commands respectively. The `{-progress-stage:...}` command tells NI to print an estimate of the percentage of completion to `stdout`; the Inform interfaces usually catch this output and update graphical progress bars accordingly.

⟨Act on an I6T command which keeps track of progress 20⟩ ≡

```
    if (strcmp(command, "log") == 0) {
        log_new_stage_of_NIs_run(argument); continue;
    }
    if (strcmp(command, "log-phase") == 0) {
        char *phase_names[] = {
            "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX", "X",
            "XI", "XII", "XIII", "XIV", "XV", "XVI", "XVII", "XVIII", "XIX", "XX" };
        log_new_phase_of_NIs_run(phase_names[no_phases], argument);
        if (no_phases < 19) no_phases++;
        continue;
    }
    if (strcmp(command, "progress-stage") == 0) {
        progress_bar(atoi(argument), 0); continue;
    }
```

This code is used in §16.

§**21. Commands accessing NI internals.** These are the real point of the I6T interpreter. We begin with two commands which take numerical values from inside NI and output them as literal decimal numbers: first, the number of labels produced for a given label namespace.

⟨Act on the I6T counter command 21⟩ ≡

```
    if (strcmp(command, "counter") == 0) {
        if (OUT == NULL) continue;
        WRITE("%d", lns_read_counter(argument, FALSE)); continue;
    }
```

This code is used in §16.

§**22.** More generally, `{-value:...}` behaves as if it evaluates ... as a C expression in NI, but of course only works for a very select few identifiers: those "allowed" by the following macro.

```
define ALLOW_VALUE(val)
    if (strcmp(argument, #val)==0) { WRITE("%d", val); continue; }
```

⟨Act on the I6T value command 22⟩ ≡

```
    if (strcmp(command, "value") == 0) {
        if (OUT == NULL) continue;
        ALLOW_VALUE(NUMBER_CREATED(action_name));
        ALLOW_VALUE(NUMBER_CREATED(binary_predicate));
        ALLOW_VALUE(NUMBER_CREATED(blorb_figure));
        ALLOW_VALUE(NUMBER_CREATED(blorb_sound));
        ALLOW_VALUE(NUMBER_CREATED(property_name));
        ALLOW_VALUE(NUMBER_CREATED(rulebook));
        ALLOW_VALUE(NUMBER_CREATED(scene));
        ALLOW_VALUE(NUMBER_CREATED(test_scenario));
        ALLOW_VALUE(NUMBER_CREATED(world_object));
        ALLOW_VALUE(OBJECT_TY);
        ALLOW_VALUE(SNIPPET_TY);
        ALLOW_VALUE(UNDERSTANDING_TY);
        ALLOW_VALUE(extent_of_runtime_quotations_array);
        ALLOW_VALUE(instance_count_for_kind(kind_door));
        ALLOW_VALUE(instance_count_for_kind(kind_room));
        ALLOW_VALUE(max_frame_size_needed);
        ALLOW_VALUE(no_verb_verb_defined);
        ALLOW_VALUE(rng_seed_at_start_of_play);
        LOG("command: <%s> argument: <%s>\n", command, argument);
        internal_error("Unknown {-value:...} value in I6 segment");
    }
```

This code is used in §16.

§**23.** Now three commands which cause the template interpreter to stop and call routines in the rest of NI, which may do something dramatic.

The first command makes `{-read-assertions:1}` and `{-read-assertions:2}` run main traverses 1 and 2 of the parse tree, in which assertion sentences are converted to inferences, among much other useful activity. Should any problem messages be issued, we go into passive mode, thus calling no further NI routines for the time being. This enables all subsequent routines to be written on the assumption that they have correct input as far as can be judged so far, which greatly simplifies error recovery.

⟨Act on the I6T read-assertions command 23⟩ ≡

```
    if (strcmp(command, "read-assertions") == 0) {
        if (strcmp(argument, "1") == 0) traverse_for_assertions(1);
        if (strcmp(argument, "2") == 0) traverse_for_assertions(2);
        if (problem_count > 0) active = FALSE;
        continue;
    }
```

This code is used in §16.

§**24.** The `{-callv:whatever}` command makes a void function call `whatever()`, and is once again allowed only for select NI functions.

Problem messages only throw us into passive mode here if the main output file has not yet been closed. This is done because we might be in the post-compilation phase where release instructions are being written, say, and a problem is found with the release requests made by the user: we mustn't go into passive mode in such a case, because then we wouldn't be able to deallocate memory, complete the problems report with these problem messages in, and so on.

```
define ALLOW_CALLV(routine)
    if (strcmp(argument, #routine)==0) {
        routine();
        if ((problem_count > 0) && (closed == FALSE)) active = FALSE;
        continue;
    }
```

⟨Act on the I6T callv command 24⟩ ≡

```
    if (strcmp(command, "callv") == 0) {
        ALLOW_CALLV(add_rules_to_rulebooks);
        ALLOW_CALLV(allow_no_further_deferrals);
        ALLOW_CALLV(allow_no_further_past_tenses);
        ALLOW_CALLV(allow_no_further_text_subs);
        ALLOW_CALLV(break_source_into_sentences);
        ALLOW_CALLV(break_source_into_sentences);
        ALLOW_CALLV(check_extension_versions);
        ALLOW_CALLV(check_tables_for_kind_clashes);
        ALLOW_CALLV(complete_index);
        ALLOW_CALLV(complete_problems_report);
        ALLOW_CALLV(complete_type_IDs_table);
        ALLOW_CALLV(create_kind_kind);
        ALLOW_CALLV(create_standard_csps);
        ALLOW_CALLV(debug_memory_statistics);
        ALLOW_CALLV(debug_parser_statistics);
        ALLOW_CALLV(debug_verbs);
        ALLOW_CALLV(declare_source_loaded);
        ALLOW_CALLV(define_named_printing_phrases);
        ALLOW_CALLV(empty_all_headings);
        ALLOW_CALLV(free_memory);
        ALLOW_CALLV(handle_census_mode);
        ALLOW_CALLV(include_templates_for_types);
        ALLOW_CALLV(index_equations);
        ALLOW_CALLV(index_extensions);
        ALLOW_CALLV(index_external_files);
        ALLOW_CALLV(index_figures);
        ALLOW_CALLV(index_headings);
        ALLOW_CALLV(index_library_card);
        ALLOW_CALLV(index_page_Actions);
        ALLOW_CALLV(index_page_Kinds);
        ALLOW_CALLV(index_page_Phrasebook);
        ALLOW_CALLV(index_page_Rules);
        ALLOW_CALLV(index_page_Scenes);
        ALLOW_CALLV(index_page_World);
        ALLOW_CALLV(index_quantities);
        ALLOW_CALLV(index_sounds);
        ALLOW_CALLV(index_tables);
```

```
        ALLOW_CALLV(Inform_name_the_actions);
        ALLOW_CALLV(Inform_name_the_objects);
        ALLOW_CALLV(log_brief_picture);
        ALLOW_CALLV(make_built_in_determiners);
        ALLOW_CALLV(make_built_in_relations);
        ALLOW_CALLV(make_built_in_verbs);
        ALLOW_CALLV(make_further_built_in_relations);
        ALLOW_CALLV(make_heading_tree);
        ALLOW_CALLV(make_model_world);
        ALLOW_CALLV(make_reserved_words);
        ALLOW_CALLV(make_type_IDs_table);
        ALLOW_CALLV(parse_rule_parameters);
        ALLOW_CALLV(parse_rule_placements);
        ALLOW_CALLV(plant_parse_tree);
        ALLOW_CALLV(prepare_grammar);
        ALLOW_CALLV(read_primary_source_text);
        ALLOW_CALLV(register_phrase_meanings);
        ALLOW_CALLV(register_recently_lexed_phrases);
        ALLOW_CALLV(report_pairs_allowed);
        ALLOW_CALLV(report_pairs_observed);
        ALLOW_CALLV(report_unacted_upon_interventions);
        ALLOW_CALLV(satisfy_heading_dependencies);
        ALLOW_CALLV(start_lexer);
        ALLOW_CALLV(start_memory);
        ALLOW_CALLV(tidy_up_ofs_and_froms);
        ALLOW_CALLV(traverse_for_adjective_definitions);
        ALLOW_CALLV(traverse_for_extensions);
        ALLOW_CALLV(traverse_for_grammar);
        ALLOW_CALLV(traverse_for_phrase_names);
        ALLOW_CALLV(traverse_for_phrases);
        ALLOW_CALLV(traverse_for_plural_definitions);
        ALLOW_CALLV(traverse_for_verbs);
        ALLOW_CALLV(traverse_to_create_equations);
        ALLOW_CALLV(traverse_to_stock_equations);
        ALLOW_CALLV(traverse_to_create_tables);
        ALLOW_CALLV(traverse_to_stock_tables);
        ALLOW_CALLV(update_census_of_extensions);
        ALLOW_CALLV(validate_definitions);
        ALLOW_CALLV(verify_parse_tree);
        ALLOW_CALLV(write_headings_as_xml);
        ALLOW_CALLV(write_ifiction_and_blurb);
        LOG("command: <%s> argument: <%s>\n", command, argument);
        internal_error("Unknown {-callv:...} function in I6 segment");
    }
```

This code is used in §16.

§**25.** And the `{-call:whatever}` command makes a function call `whatever(OUT)`, supplying the output file handle as an argument, and is once again allowed only for select NI functions.

```
define ALLOW_CALL(routine)
    if (strcmp(argument, #routine)==0) { routine(OUT); continue; }
```

⟨Act on the I6T call command 25⟩ ≡

```
    if (strcmp(command, "call") == 0) {
        if (OUT == NULL) continue;
        ALLOW_CALL(add_showme_details);
        ALLOW_CALL(chronology_extents_i6_escape);
        ALLOW_CALL(compile_action_routines);
        ALLOW_CALL(compile_activity_constants);
        ALLOW_CALL(compile_adjectival_phrases);
        ALLOW_CALL(compile_allocated_counter_storage);
        ALLOW_CALL(compile_attributes);
        ALLOW_CALL(compile_bibliographic_i6_constants);
        ALLOW_CALL(compile_build_number);
        ALLOW_CALL(compile_deferred_propositions);
        ALLOW_CALL(compile_defined_relations);
        ALLOW_CALL(compile_direction_object_constants);
        ALLOW_CALL(compile_equations);
        ALLOW_CALL(compile_filters);
        ALLOW_CALL(compile_grammar_conditions);
        ALLOW_CALL(compile_grammar_verbs);
        ALLOW_CALL(compile_heap_allocator);
        ALLOW_CALL(compile_I6_constants_for_typenames);
        ALLOW_CALL(compile_icl_commands);
        ALLOW_CALL(compile_list_constants);
        ALLOW_CALL(compile_list_together_routines);
        ALLOW_CALL(compile_los_routines);
        ALLOW_CALL(compile_max_score);
        ALLOW_CALL(compile_named_action_patterns);
        ALLOW_CALL(compile_number_grammar);
        ALLOW_CALL(compile_object_tree);
        ALLOW_CALL(compile_phrases);
        ALLOW_CALL(compile_print_table_names);
        ALLOW_CALL(compile_data_type_support_routines);
        ALLOW_CALL(compile_property_numberspace_forcer);
        ALLOW_CALL(compile_resolver_routines);
        ALLOW_CALL(compile_rule_printing_switch);
        ALLOW_CALL(compile_rulebooks);
        ALLOW_CALL(compile_runtime_relation_storage);
        ALLOW_CALL(compile_string_constants);
        ALLOW_CALL(compile_stub_properties);
        ALLOW_CALL(compile_tables);
        ALLOW_CALL(compile_test_printout);
        ALLOW_CALL(compile_test_switch);
        ALLOW_CALL(compile_text_routines);
        ALLOW_CALL(compile_time_grammar);
        ALLOW_CALL(compile_truth_state_grammar);
        ALLOW_CALL(compile_type_gprs);
        ALLOW_CALL(compile_use_options);
        ALLOW_CALL(create_block_constants);
```

```
        ALLOW_CALL(declare_quantities);
        ALLOW_CALL(external_file_arrays);
        ALLOW_CALL(past_actions_i6_routines);
        ALLOW_CALL(past_tenses_i6_escape);
        ALLOW_CALL(resolve_predeclared_booked_rules);
        ALLOW_CALL(write_test_text);
        LOG("command: <%s> argument: <%s>\n", command, argument);
        internal_error("Unknown {-call:...} function in I6 segment");
    }
```

This code is used in §16.

§**26.** The `{-array:...}` command tells NI to compile an I6 array with the given name which must, of course, be one of those in its repertoire.

define `ALLOW_ARRAY(array)`

```
    if (strcmp(argument, #array)==0) { compile_##array##_array(OUT); continue; }
```

⟨Act on the I6T array command 26⟩ ≡

```
    if (strcmp(command, "array") == 0) {
        if (OUT == NULL) continue;
        ALLOW_ARRAY(ActionCoding);
        ALLOW_ARRAY(ActionData);
        ALLOW_ARRAY(ActionHappened);
        ALLOW_ARRAY(Activity_after_rulebooks);
        ALLOW_ARRAY(Activity_atb_rulebooks);
        ALLOW_ARRAY(Activity_before_rulebooks);
        ALLOW_ARRAY(activity_var_creators);
        ALLOW_ARRAY(Activity_for_rulebooks);
        ALLOW_ARRAY(ConstantListPointers);
        ALLOW_ARRAY(Global_Vars);
        ALLOW_ARRAY(TimedEventsTable);
        ALLOW_ARRAY(TimedEventTimesTable);
        ALLOW_ARRAY(InitialSituation);
        ALLOW_ARRAY(property_metadata);
        ALLOW_ARRAY(relation_metadata);
        ALLOW_ARRAY(rulebook_var_creators);
        ALLOW_ARRAY(RulebookNames);
        ALLOW_ARRAY(rulebooks_array);
        ALLOW_ARRAY(scene_recurs);
        ALLOW_ARRAY(tableoffigures);
        ALLOW_ARRAY(tableofsounds);
        ALLOW_ARRAY(UUID_ARRAY);
        LOG("command: <%s> argument: <%s>\n", command, argument);
        internal_error("Unknown {-array:...} function in I6 segment");
    }
```

This code is used in §16.

§**27.** And the `{-routine:...}` command tells NI to compile an I6 routine with the given name. (There are very few of these, because we try to let template files choose the names of routines, for the most part.)

```
define ALLOW_ROUTINE(routine)
    if (strcmp(argument, #routine)==0) { compile_##routine##_routine(OUT); continue; }
```

⟨Act on the I6T routine command 27⟩ ≡

```
    if (strcmp(command, "routine") == 0) {
        if (OUT == NULL) continue;
        ALLOW_ROUTINE(CreatePropertyOffsets);
        ALLOW_ROUTINE(I7_Kind_Name);
        ALLOW_ROUTINE(RulebookOutcomePrintingRule);
        ALLOW_ROUTINE(DetectSceneChange);
        ALLOW_ROUTINE(ShowSceneStatus);
        ALLOW_ROUTINE(PrintSceneName);
        ALLOW_ROUTINE(ShowExtensionVersions);
        ALLOW_ROUTINE(TestUseOption);
        ALLOW_ROUTINE(MistakeActionSub);
        ALLOW_ROUTINE(InternalTestCases);
        LOG("command: <%s> argument: <%s>\n", command, argument);
        internal_error("Unknown {-array:...} function in I6 segment");
    }
```

This code is used in §16.

§**28.** And the `{-test:...}` command tells NI that an I6 verb directive follows for a testing verb. We want to know that in order to police accidental clashes between testing commands built in the template, and genuine commands defined in the source text.

⟨Act on the I6T test command 28⟩ ≡

```
    if (strcmp(command, "testing-command") == 0) {
        if (OUT == NULL) continue;
        reserve_command_verb(argument);
        WRITE("Verb meta '%s'\n", argument);
        continue;
    }
```

This code is used in §16.

§**29. I7 expression evaluation.** This is not quite like regular expression evaluation, because we want "room" and "lighted" to be evaluated as the I6 translation of the relevant class or property, rather than as code to test the predicate "X is a room" or "X is lighted", and similarly for bare names of defined adjectives. So:

```
void compile_I7_expression_from_I6(OUTPUT_STREAM, char *p) {
    feed_into_lexer(p, FALSE, FALSE);
    property_name *prn = parse_property_name(lexer_feed_w1, lexer_feed_w2);
    if (prn) {
        WRITE("%s", prn_get_i6_identifier(prn));
        return;
    }
    world_object *wo = parse_world_object(lexer_feed_w1, lexer_feed_w2, FALSE);
    if (wo) {
        WRITE("%s", wo_get_I6_representation(wo));
        return;
```

```
    }
    int initial_problem_count = problem_count;
    specification *spec = parse_expression(lexer_feed_w1, lexer_feed_w2, VALUE_EXPCON);
    if (initial_problem_count < problem_count) return;
    typecheck_without_expectations(spec);
    if (initial_problem_count < problem_count) return;
    spec_compile(OUT, spec);
}
```

§**30. Intervention.** This is a system allowing the user to hang explicit code before, instead of or after any part of any segment of the I6T files in use.

```
void I6T_intervention_new(int stage, char *segment, char *part, char *i6, char *seg) {
    I6T_intervention *i6ti = NULL;
    if (stage == 0) {
        LOOP_OVER(i6ti, I6T_intervention)
            if ((i6ti->intervention_stage == 0) &&
                (strcmp(i6ti->segment_name, segment) == 0)) {
                if (((part == NULL) || (i6ti->part_name == NULL)) && (part != i6ti->part_name))
                    continue;
                if ((part) && (strcmp(i6ti->part_name, part) != 0)) continue;
                break;
            }
    }
    if (i6ti == NULL) i6ti = CREATE(I6T_intervention);
    i6ti->intervention_stage = stage;
    i6ti->segment_name = segment;
    i6ti->part_name = part;
    i6ti->I6T_matter = i6;
    i6ti->alternative_segment = seg;
    i6ti->segment_found = FALSE;
    i6ti->part_found = FALSE;
    i6ti->where_intervention_requested = current_sentence;
    LOGIF(SEGMENTS, "New stage %d Segment %s Part %s\n", stage, segment, (part)?part:"<none>");
}
```

The function I6T_intervention_new is called from 10/isin.

§**31.** An intervention "instead" (stage 0) replaces any existing one, but at other stages – before and after – they are accumulated.

```
int I6T_file_intervene(OUTPUT_STREAM, int stage, char *segment, char *part) {
    I6T_intervention *i6ti;
    int rv = FALSE;
    if (strcmp(segment, "Main.i6t") == 0) return rv;
    LOGIF(SEGMENTS, "Stage %d Segment %s Part %s\n", stage, segment, (part)?part:"<none>");
    LOOP_OVER(i6ti, I6T_intervention)
        if ((i6ti->intervention_stage == stage) &&
            (strcmp(i6ti->segment_name, segment) == 0)) {
            i6ti->segment_found = TRUE;
            if (((part == NULL) || (i6ti->part_name == NULL)) && (part != i6ti->part_name))
                continue;
            if ((part) && (strcmp(i6ti->part_name, part) != 0)) continue;
            i6ti->part_found = TRUE;
            LOGIF(SEGMENTS, "Intervention at stage %d Segment %s Part %s\n", stage, segment, (part)?part:"<none>")
            if (i6ti->I6T_matter) interpret_I6T_file(OUT, i6ti->I6T_matter, NULL, -1);
            if (i6ti->alternative_segment) interpret_I6T_file(OUT, NULL, i6ti->alternative_segment,
-1);
            if (stage == 0) rv = TRUE;
        }
    return rv;
}
```

§**32.** At the end of the run, we check to see if any of the interventions were never acted on. This generally means the user mistyped the name of a section or part – which would otherwise be an error very difficult to detect.

```
void report_unacted_upon_interventions(void) {
    I6T_intervention *i6ti;
    LOOP_OVER(i6ti, I6T_intervention) {
        if ((i6ti->segment_found == FALSE) && (strcmp(i6ti->segment_name, "Main.i6t") != 0)) {
            current_sentence = i6ti->where_intervention_requested;
            sentence_problem(_P_(C14NoSuchTemplate),
                "no template file of that name was ever read in",
                "so this attempt to intervene had no effect. "
                "The template files have names like 'Output.i6t', 'Parser.i6t' "
                "and so on. (Looking at the typeset form of the template, "
                "available at the Inform website, may help.)");
        } else if ((i6ti->part_found == FALSE) && (i6ti->part_name) &&
            (strcmp(i6ti->segment_name, "Main.i6t") != 0)) {
            current_sentence = i6ti->where_intervention_requested;
            sentence_problem(_P_(C14NoSuchPart),
                "that template file didn't have a part with that name",
                "so this attempt to intervene had no effect. "
                "Each template file is divided internally into a number of "
                "named parts, and you have to quote their names precisely. "
                "(Looking at the typeset form of the template, available at "
                "the Inform website, may help.)");
        }
    }
}
```

The function report_unacted_upon_interventions is invoked by a command in a `.i6t` template file.

§33. The build constant. That was the end of the template interpreter. Now, since this version-numbering constant belongs nowhere else, we provide a single I6T command in this section of NI: the following routine performs `{-call:compile_build_number}`.

```
void compile_build_number(OUTPUT_STREAM) {
    WRITE("Constant NI_BUILD_COUNT \"%s\";\n", NI_BUILD);
}
```

The function compile_build_number is invoked by a command in a `.i6t` template file.

§34. Registration of sentence handlers. The following routine is placed here, right at the end of the NI code, because at this point all of the sentence handlers – with names like `TABLE_SH_handler` – have now been created.

```
void register_sentence_handlers(void) {
    ⟨Add sentence handlers for the top-level node types 35⟩;
    ⟨Add sentence handlers for the SENTENCE/VERB node types 36⟩;
}
```

The function register_sentence_handlers is called from 8/tass.

§35. This is all of the node types still present at the top level of the tree at the end of sentence-breaking.

⟨Add sentence handlers for the top-level node types 35⟩ ≡

```
    REGISTER_SENTENCE_HANDLER(TRACE_SH);
    REGISTER_SENTENCE_HANDLER(BEGINHERE_SH);
    REGISTER_SENTENCE_HANDLER(ENDHERE_SH);
    REGISTER_SENTENCE_HANDLER(BIBLIOGRAPHIC_SH);
    REGISTER_SENTENCE_HANDLER(INFORM6CODE_SH);
    REGISTER_SENTENCE_HANDLER(COMMAND_SH);
    REGISTER_SENTENCE_HANDLER(ROUTINE_SH);
    REGISTER_SENTENCE_HANDLER(TABLE_SH);
    REGISTER_SENTENCE_HANDLER(EQUATION_SH);
    REGISTER_SENTENCE_HANDLER(HEADING_SH);
    REGISTER_SENTENCE_HANDLER(SENTENCE_SH);
```

This code is used in §34.

§**36.** And here are all of the verb types found in `VERB_NT` nodes which are first children of `SENTENCE_NT` nodes.

⟨Add sentence handlers for the SENTENCE/VERB node types 36⟩ ≡

```
REGISTER_SENTENCE_HANDLER(ASSERT_SH);
REGISTER_SENTENCE_HANDLER(HAS_SH);
REGISTER_SENTENCE_HANDLER(CANBE_SH);
REGISTER_SENTENCE_HANDLER(TRANSLATES_SH);
REGISTER_SENTENCE_HANDLER(TRANSLATESU_SH);
REGISTER_SENTENCE_HANDLER(NEW_ACTIVITY_SH);
REGISTER_SENTENCE_HANDLER(NEW_ACTION_SH);
REGISTER_SENTENCE_HANDLER(NEW_RELATION_SH);
REGISTER_SENTENCE_HANDLER(DOC_SH);
REGISTER_SENTENCE_HANDLER(TEST_SH);
REGISTER_SENTENCE_HANDLER(USEMEANS_SH);
REGISTER_SENTENCE_HANDLER(SPECIFIES_SH);
REGISTER_SENTENCE_HANDLER(DEFINED_BY_SH);
REGISTER_SENTENCE_HANDLER(BEGINS_WHEN_SH);
REGISTER_SENTENCE_HANDLER(ENDS_WHEN_SH);
REGISTER_SENTENCE_HANDLER(USE_SH);
REGISTER_SENTENCE_HANDLER(RELEASE_SH);
REGISTER_SENTENCE_HANDLER(EPISODE_SH);
REGISTER_SENTENCE_HANDLER(PLURAL_SH);
REGISTER_SENTENCE_HANDLER(FIGURE_SH);
REGISTER_SENTENCE_HANDLER(SOUND_SH);
REGISTER_SENTENCE_HANDLER(FILE_SH);
```

This code is used in §34.

A The Standard Rules

`A/sr0`: *SR0 - Preamble.w* The titling line and rubric, use options and a few other technicalities before the Standard Rules get properly started.

`A/sr1`: *SR1 - Physical World Model.w* Verbal descriptions of spatial relationships; the hierarchy of kinds of object, and their properties.

`A/sr2`: *SR2 - Variables and Rulebooks.w* The global variables and those built-in rulebooks which do not belong either to specific actions or to specific activities.

`A/sr3`: *SR3 - Activities.w* The built-in activities and their default stock of rules; the locale description mechanism.

`A/sr4`: *SR4 - Actions.w* The standard stock of actions supplied with Inform, along with the rules which define them; and the Understand grammar which corresponds to them.

`A/sr5`: *SR5 - Phrase Definitions.w* The phrases making up the Inform language, and in terms of which all other phrases and rules are defined; and the final sign-off of the Standard Rules extension, including its minimal documentation.

SR0 - Preamble A/sr0

Purpose

The titling line and rubric, use options and a few other technicalities before the Standard Rules get properly started.

A/sr0.§2 Title; §3-7 Starting up

---

§**1.** The Standard Rules are like a boot program for a computer that is starting up: at the beginning, the process is delicate, and the computer needs a fairly exact sequence of things to be done; halfway through, the essential work is done, but the system is still too primitive to be much use, so we begin to create convenient intermediate-level code sitting on top of the basics; so that, by the end, we have a fully flexible machine ready to go in any number of directions. In this commentary, we try to distinguish between what must be done (or else NI will crash, or fail in some other way) and what is done simply as a design decision (to make the Inform language come out the way we want it). Quite interesting hybrid Informs could be built by making different decisions. Still, our design is not entirely free, since it interacts with the I6 template layer (the I7 equivalent of the old I6 library): a really radical alternate Inform would need a different template layer, too.

§**2. Title.** Every Inform 7 extension begins with a standard titling line and a rubric text, and the Standard Rules are no exception:

```
Version 2/090402 of the Standard Rules by Graham Nelson begins here.
"The Standard Rules, included in every project, define the basic framework
of kinds, actions and phrases which make Inform what it is."
```

§**3. Starting up.** The following block of declarations is actually written by `indoc` and modified each time we alter the documentation. It's a dictionary of symbolic names like `HEADINGS` to HTML page leafnames like `doc71`.

...and so on...

§**4.** Some Inform 7 projects are rather heavy-duty by the expectations of the Inform 6 compiler (which it uses as a code-generator): I6 was written fifteen years earlier, when computers were unimaginably smaller and slower. So many of its default memory settings need to be raised to higher maxima.

Note that the Z-machine cannot accommodate more than 255 verbs, so this is the highest `MAX_VERBS` setting we can safely make here.

```
Use MAX_ARRAYS of 1500.
Use MAX_CLASSES of 200.
Use MAX_VERBS of 255.
Use MAX_LABELS of 10000.
Use MAX_ZCODE_SIZE of 50000.
Use MAX_STATIC_DATA of 120000.
Use MAX_PROP_TABLE_SIZE of 200000.
Use MAX_INDIV_PROP_TABLE_SIZE of 20000.
Use MAX_STACK_SIZE of 65536.
Use MAX_SYMBOLS of 20000.
Use MAX_EXPRESSION_NODES of 256.
```

§**5.** These, on the other hand, are settings used by the dynamic memory management code, which runs in I6 as part of the template layer. Each setting translates to an I6 constant declaration, with the value chosen being substituted for `{N}`.

The "dynamic memory allocation" defined here is slightly misleading, in that the memory is only actually consumed in the event that any of the data types needing to use the heap are actually employed in the source text being compiled. (8192 bytes may not sound much these days, but in the tight array space of the Z-machine it's quite a large commitment, and we want to avoid it whenever possible.)

```
Use dynamic memory allocation of at least 8192 translates as
    (- Constant DynamicMemoryAllocation = {N}; -).
Use maximum indexed text length of at least 1024 translates as
    (- Constant IT_MemoryBufferSize = {N}+3; -).

Use dynamic memory allocation of at least 8192.
```

§**6.** This setting is to do with the Inform parser's handling of multiple objects.

```
Use maximum things understood at once of at least 100 translates as
    (- Constant MATCH_LIST_WORDS = {N}; -).

Use maximum things understood at once of at least 100.
```

§**7.** Finally, some definitions of miscellaneous options: none are used by default, but all translate into I6 constant definitions if used. (These are constants whose values are used in the I6 library or in the template layer, which is how they have effect.)

```
Use American dialect translates as (- Constant DIALECT_US; -).
Use the serial comma translates as (- Constant SERIAL_COMMA; -).
Use full-length room descriptions translates as (- Constant I7_LOOKMODE = 2; -).
Use abbreviated room descriptions translates as (- Constant I7_LOOKMODE = 3; -).
Use memory economy translates as (- Constant MEMORY_ECONOMY; -).
Use authorial modesty translates as (- Constant AUTHORIAL_MODESTY; -).
Use no scoring translates as (- Constant NO_SCORING; -).
Use command line echoing translates as (- Constant ECHO_COMMANDS; -).
Use undo prevention translates as (- Constant PREVENT_UNDO; -).
Use predictable randomisation translates as (- Constant FIX_RNG; -).
Use fast route-finding translates as (- Constant FAST_ROUTE_FINDING; -).
Use slow route-finding translates as (- Constant SLOW_ROUTE_FINDING; -).
Use numbered rules translates as (- Constant NUMBERED_RULES; -).
Use telemetry recordings translates as (- Constant TELEMETRY_ON; -).
```

SR1 - Physical World Model A/sr1

Purpose

Verbal descriptions of spatial relationships; the hierarchy of kinds of object, and their properties.

---

§**1.** Although at this point we can freely use the exceptional fixed-form command sentences – like "Use ⟨use-option⟩" – since their wording is built into NI, we are very limited in the general assertion sentences we can make. Only two verbs are built in, and this is because they require rather special handling: *to be*, since it is both fundamental and irregular; and *to have*, since although regular is it used also as an auxiliary coupled with *to be* (as in "if Daphne has been in the garden"). And no prepositions exist at all.

Well, we can't do much in talking about the world model without the fundamental spatial relationships of being inside, on top of and so forth. While the relations themselves are built-in to NI, and so are their one-word names (such as "containment"), the rest is down to us:

```
Part SR1 - The Physical World Model

Section SR1/0 - Language

The verb to provide (he provides, they provide, he provided, it is provided,
he is providing) implies the provision relation.

The verb to be in implies the reversed containment relation.
The verb to be inside implies the reversed containment relation.
The verb to be within implies the reversed containment relation.
The verb to be held in implies the reversed containment relation.
The verb to be held inside implies the reversed containment relation.

The verb to contain (he contains, they contain, he contained, it is contained,
he is containing) implies the containment relation.
The verb to be contained in implies the reversed containment relation.

The verb to be on implies the reversed support relation.
The verb to be on top of implies the reversed support relation.

The verb to support (he supports, they support, he supported, it is supported,
he is supporting) implies the support relation.
The verb to be supported on implies the reversed support relation.

The verb to incorporate (he incorporates, they incorporate, he incorporated,
it is incorporated, he is incorporating) implies the incorporation relation.
The verb to be part of implies the reversed incorporation relation.
The verb to be a part of implies the reversed incorporation relation.
The verb to be parts of implies the reversed incorporation relation.
```

§**2.** The enclosure relation, indirectly defined in terms of the above more fundamental ones, has a verb but no prepositions (though of course "to be enclosed by" is in effect a prepositional expression of this).

```
The verb to enclose (he encloses, they enclose, he enclosed, it is enclosed,
he is enclosing) implies the enclosure relation.
```

§**3.** Those three relations expressed how the inanimate world is arranged, on the small scale: the relationships become a little more complicated once living beings are involved. One living being is special to our language – the protagonist character, that is, the "player" – and so these three verbs all have adjectival forms which imply the player as the missing term. Thus, to say "The felt hat is worn." implies "...by the player": the curious syntax "(adjectival)" below, applied to the past participle, causes this to happen. This is a syntax not documented in the Inform manuals and which may change to be replaced with something more general later on; for the moment, please do not use it.

```
The verb to carry (he carries, they carry, he carried, it is carried
(adjectival), he is carrying) implies the carrying relation.
The verb to hold (he holds, they hold, he held, it is held (adjectival),
he is holding) implies the holding relation.
The verb to wear (he wears, they wear, he wore, it is worn (adjectival),
he is wearing) implies the wearing relation.
```

§**4.** Animate beings also have the ability to see and touch their surroundings, but note that we only model the ability to do these things – we do not attempt to track what they actually do see or touch at any given moment, so there are no built-in verbs *to see* or *to touch*.

```
The verb to be able to see (he is seen) implies the visibility relation.
The verb to be able to touch (he is touched) implies the touchability relation.
```

§**5.** The special status of the player as the sensory focus, so to speak, is again shown in the adjectives defined here:

```
Definition: Something is visible rather than invisible if the player can see it.
Definition: Something is touchable rather than untouchable if the player can touch it.
```

§**6.** While many of the world-modelling assumptions in I7 are carried over from those tried and tested by I6, the idea of concealment is an exception. The old I6 attribute `concealed` simply marked some objects (which we would call "things") as being hidden from view in some way, but was never very satisfactory. What does hidden mean, exactly – to whose eyes, and in what way? Should you be able to take something which is hidden, if you happen to know it's there? And so on. It was the muddiest of all the attributes, and widely disused as a result. In I7, we instead took the view that concealment required an active agent continuously doing the concealing: it applies, for instance, to a dagger which someone intentionally hides beneath a cloak, but not to a key placed at the back of a shelf by somebody long gone.

```
The verb to conceal (he conceals, they conceal, he concealed, it is concealed,
he is concealing) implies the concealment relation.
Definition: Something is concealed rather than unconcealed if the holder of it conceals it.
```

§**7.** A final sort of pseudo-containment: does the entire world contain something, or not? (For things destroyed during play, or not yet created, the answer would be no.) These definitions are rather crudely made not in terms of a defined verb *to include* but instead use the phrase "the world model includes ...", which will be defined later. (It doesn't matter that it hasn't been defined yet, because definitions, phrases and rules can be made in any order and still refer to each other.)

```
Definition: Something is on-stage rather than off-stage if I6 routine "OnStage"
    says so (it is indirectly in one of the rooms).
```

§**8. Verbal descriptions of numerical comparisons.** We might as well declare these now, too, though they're not needed for any of the world-building work. (The verbal usages `<`, `>`, `<=` and `>=` are built into NI; those would be the same in any language, and are unlike other verbs since they have no inflected forms for non-present tenses.)

```
The verb to be greater than implies the numerically-greater-than relation.
The verb to be less than implies the numerically-less-than relation.
The verb to be at least implies the numerically-greater-than-or-equal-to relation.
The verb to be at most implies the numerically-less-than-or-equal-to relation.
```

§**9. Creating the world model.** The 0th kind, "kind", is not created here but by NI itself. The first through to ninth kinds created now follow: they must not be reordered or moved. Note the two alterative plural definitions for the word "person", with "people" being defined earlier to make it the default: "persons" is correct, but "people" is more idiomatically usual.

```
Section SR1/1 - Primitive Kinds

A room is a kind. [1]
A thing is a kind. [2]
A direction is a kind. [3]
A door is a kind of thing. [4]
A container is a kind of thing. [5]
A supporter is a kind of thing. [6]
A backdrop is a kind of thing. [7]
The plural of person is people. The plural of person is persons.
A person is a kind of thing. [8]
A region is a kind. [9]
```

§**10.** At this point, then, the hierarchy looks like so:

```
kind
    room [1]
    thing [2]
        door [4]
        container [5]
        supporter [6]
        backdrop [7]
        person [8]
    direction [3]
    region [9]
```

This framework is the minimum kit needed in order for NI to be able to manage the spatial relationships arising from its basic verbs. Room and thing are needed to distinguish places and objects; door and backdrop because they need to violate the basic rule that an object can only be in one place at once – a door is "in" both of the rooms it faces onto – and this requires special handling by NI; region because it violates the rule that rooms are not themselves subject to being contained in other objects, and again this requires special handling. That leaves "direction", "container", "supporter" and "person", and these are needed to express the concepts inherent in the sentences "A is east of B", "A is in B", "A is on B" and "A is carried by B". (We also need room and person in order to make sense of the words "somewhere" and "someone", for instance.)

Although further kinds will be created later ("vehicle", for instance), those are merely design choices, and NI would not be troubled by their absence.

§**11. Rooms.** We now detail each of the fundamental kinds in turn, in order of their declaration, and thus beginning with rooms.

```
Section SR1/2 - Rooms

The specification of room is "Represents geographical locations, both indoor
and outdoor, which are not necessarily areas in a building. A player in one
room is mostly unable to sense, or interact with, anything in a different room.
Rooms are arranged in a map."
```

§**12.** Rooms have rather few properties built in; this reflects their usual role in IF as ambient environments in which interesting things happen, rather than being direct participants.

```
A room can be privately-named or publically-named. A room is usually publically-named.
A room can be lighted or dark. A room is usually lighted.
A room can be visited or unvisited. A room is usually unvisited.

A room has a text called description.
A room has a text called printed name.
```

§**13.** Note that the "map region" property here is created with the type "object", not "region", even though we think of it as always being a region. This is because of I7's type-checking rule: the type "object" can legally hold 0, meaning "nothing", but more specific object types – in this case "region" – cannot. That would make them illegal to use in a situation where no regions were created, because variables or properties of this kind couldn't be initialised. This is why the Standard Rules almost always declare object properties as "object" rather than anything more specific.

```
A room has an object called map region. The map region of a room is usually nothing.
```

§**14.** Rooms have two specialised spatial relationships of their own, which again we need verbal forms of:

```
The verb to be adjacent to implies the reversed adjacency relation.
Definition: A room is adjacent if it is adjacent to the location.

The verb to be regionally in implies the reversed regional-containment relation.
```

§**15.** There's no detailed writeup of regions, since they have no properties in the usual setup. So let's add this here for the Kinds index:

```
The specification of region is "Represents a broader area than a single
room, and allows rules to apply to a whole geographical territory. Each
region can contain many rooms, and regions can even be inside each other,
though they cannot otherwise overlap. For instance, the room Place d'Italie
might be inside the region 13th Arrondissement, which in turn is inside
the region Paris. Regions are useful mainly when the world is a large one,
and are optional."
```

§**16. Things.** Things are ubiquitous:

```
Section SR1/3 - Things

The specification of thing is "Represents anything interactive in the model
world that is not a room. People, pieces of scenery, furniture, doors and
mislaid umbrellas might all be examples, and so might more surprising things
like the sound of birdsong or a shaft of sunlight."
```

§**17.** The large number of either/or properties things can have reflects the flexibility of the I6 world model, which we largely adopt for I7 too. That is, you can have any combination of lit/unlit, edible/inedible, fixed in place/portable, and so on. We can divide them into three broad categories: first, physical properties. Things come in $2^6 = 64$ physically different varieties, which is rather a lot, but although some combinations are very rare (edible lit pushable between rooms scenery is not met with often) this flexibility is helpful in mitigating the rigidity of the kinds structure, given that we have single inheritance of kinds. Note that, except for "lit", these are all really to do whether and how people can move things around – even edibility, which is the ability to be removed from the world model entirely.

```
A thing can be lit or unlit. A thing is usually unlit.
A thing can be edible or inedible. A thing is usually inedible.
A thing can be fixed in place or portable. A thing is usually portable.
A thing can be scenery.
A thing can be wearable.
A thing can be pushable between rooms.
```

§**18.** Second, status properties, which in effect refer to the past history of an item without our needing to use the past tenses (which can be tricky or inefficient). "Handled" means that the player has at some time carried the thing in question, "initially carried" means that the player started with it. ("Carried", which means the player is carrying it now, is not defined as an either/or property but as an adjectival use of the past participle of *to carry*: see above.)

```
A thing can be handled.
A thing can be initially carried.
```

§**19.** Third, linguistic properties, influencing when and how the thing's name will be included in lists. ("Mentioned" goes here rather than as a status property because it refers only to the current room description, so it carries no long-term historic information. "Marked for listing", similarly, carries only short-term information and is used as workspace by the I6 library and also by some of the I7 template routines.)

```
A thing can be privately-named or publically-named. A thing is usually publically-named.
A thing can be plural-named or singular-named. A thing is usually singular-named.
A thing can be proper-named or improper-named. A thing is usually improper-named.
A thing can be described or undescribed. A thing is usually described.
A thing can be marked for listing or unmarked for listing. A thing is usually
unmarked for listing.
A thing can be mentioned or unmentioned. A thing is usually mentioned.
```

§**20.** We now have a mixed bag of value properties, all descriptive – it's an interesting reflection on how qualitative English text usually is that the world model so seldom needs quantitative properties (sizes, weights, distances, and so on).

```
A thing has a text called an indefinite article.
A thing has a text called a description.
A thing has a text called an initial appearance.
A thing has a text called printed name.
A thing has a text called a printed plural name.
```

§**21.** The kind "thing", like all kinds of object, is compiled by NI into an I6 `Class` definition. The three `component_*` properties form up the "part of" hierarchy: if A and B are parts of C then the `component_parent` of both A and B is C, while A is the `component_child` of C and B is the `component_sibling` of A. (This is all directly analogous to I6's built in object tree for containment and support. Incorporation, being a part of, was a concept which didn't exist in the I6 world model, and we choose not to implement in a way matching the I6 handing of containment and support because we want the extra flexibility of allowing anything to incorporate anything else – we don't want the restrictions inherent in the way, in I6, something can be either a container or a supporter but not both. This is why incorporation is not tied to any I7 kind, or any I6 class – there's no notion of "incorporator" – even though linguistically the incorporation, containment and support relations are so similar to each other. Anyway, the reason something can't be both `container` and `supporter` in I6 is ultimately because they share the same tree structure: to escape from that, we need incorporation to have a tree all its own at the I6 level, and this is it.)

```
Include (-
    with component_parent nothing, component_sibling nothing, component_child nothing,
-) when defining a thing.
```

§**22.** Lastly on things: an implication about scenery. The following sentence looks like an assertion much like others above ("A thing is usually inedible", for instance) – but this is misleading. What is different is that instead of reading $K(x) \Rightarrow Q(x)$, where K is a kind and Q is a property, this has the form $P(x) \Rightarrow Q(x)$: it says that an object having property P also probably has property Q. Such sentences are called implications, and the Standard Rules make only very sparing use of them. They can trip up the user (who may quite reasonably say that it is up to him what properties something has): but they are invaluable if they cause Inform to make deductions which any human reader would always make without thought.

They can of course be overruled by explicit sentences in the source text, just as every sentence qualified by "usually" can.

The handful of implications in the Standard Rules are all commented as such.

```
Scenery is usually fixed in place. [An implication.]
```

§**23. Directions.** The first important point about directions is that they are not things and not rooms. They are not positions in the world, but imaginary arrows pointing in different ways one could go from those positions. In the language of geometry, we could call them tangent vectors which can be taken anywhere in space by parallel transport without altering them: that's to say, the "north" in one place is the same as the "north" anywhere else. (This is how we get away with having just one set of 12 direction objects, not 12 different ones for every location.) Implicit in that assumption is that the model world occupies a "flat" Euclidean space, to use further mathematical jargon: it doesn't wrap around on itself, and there are no bad positions where the directions fail. (Compare the Infocom game *Leather Goddesses of Phobos*, in which the South Pole of Mars is just such a singularity: there are three routes out of this location, all of them "north". This of course required special programming, and so it would in an Inform 7 work, too.) More concisely:

```
Section SR1/4 - Directions

The specification of direction is "Represents a direction of movement, such
as northeast or down. They always occur in opposite, matched pairs: northeast
and southwest, for instance; down and up."
```

§**24.** The only either/or property created for directions is used to allow them to be part of lists of objects:

```
A direction can be privately-named or publically-named. A direction is usually
publically-named.
A direction can be marked for listing or unmarked for listing. A direction is
usually unmarked for listing.
```

§**25.** The following value property expresses that all directions in I7 come in matched, diametrically opposing pairs – north/south, up/down and so on. This is a concept we need to provide so that I7 can apply its assumption that if room X is north of room Y, then probably room Y is also south of room X, and so on. (Geometrically, this is the operation of negation in the tangent bundle.) Note that the kind of value here is "direction", not "object": a value of 0, meaning "there's no opposite", is illegal.

```
A direction has a direction called an opposite.
```

§**26.** I6 historically began with no formal concept of "direction" and has no `direction` attribute marking some of its objects as directions (it looked instead for object-tree chidren of a pseudo-object called `compass`); by the time I6 did want such a formal concept, the use of attributes to encode what amounted to class membership was no longer thought to be good practice. So I6 directions are now expected to belong to a class called `CompassDirection`, and here we assert just that.

Our I7 directions will be created just like any other I7 objects, but we want them to emerge with the traditional names which I6 direction objects had: so, because the I6 object for north was always called `n_obj`, we want to ensure that the I7 direction "north" also comes out as `n_obj` in the compiled code. Special translates-into-I6-as sentences are used to force the I7 object compiler to use a given I6 identifier to represent the object, rather inventing something like `O12_north` as it otherwise would.

```
Include (- class CompassDirection, -) when defining a direction.
```

§**27.** The Standard Rules define only thirteen I7 objects, and here we go with twelve of them: the standard set of directions, which come in six pairs of opposites.

The following set – N/S, NE/SW, E/W, SE/NW, U/D, IN/OUT – is rooted in IF tradition. It seems unlikely that people would make IN/OUT a pair of directions today if starting from a clean slate: this is really a residue of the traditional implementation, in 70s and 80s IF, of commands which moved the player in unorthodox way. Outside the cave mouth, typing IN should take you inside; in the Y2 Rock Room, typing the magic word PLUGH should take you far away. The most convenient way to implement such commands in as few instructions as possible was to regard these as little-used compass directions rather than independent commands (some implementations of the original Adventure regarded XYZZY, PLUGH, PLOVER as all being directions, thus using 15 of the 16 possibilities which could be represented in a 4-bit field). In the 90s this was seen to be a little bogus, but since IN and OUT clearly applied in a variety of settings, they continued to be regarded as bona fide directions. In effect, they allow for one location to surround another: the canonical example would be a small white building in the middle of a field. Anyway, I7 accepts the current orthodoxy, so IN/OUT are allowed, even though they cause headaches for the interpretation of words like "inside" which might refer either to the "horizontal" or "vertical" spatial models as a result.

Of the rest, N/S, NE/SW, E/W, SE/NW and U/D, it's noteworthy that this choice imposes a cubical grid on the world, simply because the compass directions are at 45 and 90 degree angles to each other: a hexagonal tessalation would be more faithful to distances (it would get rid of the awkward point that a NE move is $\sqrt{2}$ times the length of a N move), but in practice the world model doesn't care much about distances, another example of its qualitative nature. A further point is that, in a three-dimensional cubic lattice, we ought to have another eight pairs of directions for "up and northeast", "down and west" and so on – instead of which U/D are the only ways out of the horizontal plane. But natural language doesn't work that way: it overwhelmingly provides words for horizontal travel, because that's the plane in which our eyes normally see, and in which we normally walk. Linguistically, "north" genuinely means north, but "up" allows for any amount of lateral movement into the bargain. It's a doctrine of I7 that linguistic bias is a good guide to what's worth modelling and what is not, so we will now stop worrying about this and declare the actual objects.

The order of definition of the directions affects the way lists come out: the traditional order is N, NE, NW, S, SE, SW, E, W, U, D, IN, OUT.

```
The north is a direction.
The northeast is a direction.
The northwest is a direction.
The south is a direction.
The southeast is a direction.
The southwest is a direction.
The east is a direction.
The west is a direction.
The up is a direction.
The down is a direction.
The inside is a direction.
The outside is a direction.

The north has opposite south. Understand "n" as north.
The northeast has opposite southwest. Understand "ne" as northeast.
The northwest has opposite southeast. Understand "nw" as northwest.
The south has opposite north. Understand "s" as south.
The southeast has opposite northwest. Understand "se" as southeast.
The southwest has opposite northeast. Understand "sw" as southwest.
The east has opposite west. Understand "e" as east.
The west has opposite east. Understand "w" as west.
Up has opposite down. Understand "u" as up.
Down has opposite up. Understand "d" as down.
```

```
Inside has opposite outside. Understand "in" as inside.
Outside has opposite inside. Understand "out" as outside.

The inside object translates into I6 as "in_obj".
The outside object translates into I6 as "out_obj".

The verb to be above implies the mapping-up relation.
The verb to be mapped above implies the mapping-up relation.
The verb to be below implies the mapping-down relation.
The verb to be mapped below implies the mapping-down relation.
```

§**28. Doors.** Doors are, literally, a difficult edge case for the world model of IF, since they occupy the awkward junction between the two different ways of dividing up space: the "vertical" model of objects containing and supporting each other, all within a tree rooted by the room which represents, for the moment, the entire stage-set for the play; and the "horizontal" model of rooms stitched together at compass directions into a map. The difficulty arises because in order for a door to make sense in the horizontal model, it needs to be present in two different rooms at the same time, and then it doesn't make sense in the vertical model any more, because which object tree is it to be in?

```
Section SR1/5 - Doors

The specification of door is "Represents a conduit joining two rooms, most
often a door or gate but sometimes a plank bridge, a slide or a hatchway.
Usually visible and operable from both sides (for instance if you write
'The blue door is east of the Ballroom and west of the Garden.'), but
sometimes only one-way (for instance if you write 'East of the Ballroom is
the long slide. Through the long slide is the cellar.')."
```

§**29.** This is the first kind we have declared to be a kind of something else: a door is a kind of thing. That means a door inherits all of the properties of a thing, but in a way which allows us to change the normal expectations. So here we see the first case of assertions which contradict earlier ones, but in a narrower domain: a thing is usually portable, but a door is usually fixed in place.

Our difficulty with doors being multiply present would be enormously worse if we allowed anybody to move them around during play. So:

```
A door is always fixed in place.
A door is never pushable between rooms.
Include (- has door, -) when defining a door.
```

§**30.** "Every exit is an entrance somewhere else," as Stoppard's play *Rosencrantz and Guildenstern are Dead* puts it: and though not all I7 doors are present on both sides, they do nevertheless have two sides. The representation of this is quite tricky because, as Stoppard implies, it's all a matter of which side you look at it from. What we call the "other side", and whether or not we say that "the Ballroom is through the green door", depends entirely on which side of the green door we stand. The awkward truth is that these expressions are undefined unless the player is in one of the (possibly) two rooms in which the green door is present; and then they are defined relative to him.

The leading-through relation is built in to NI; the other side property, though, is merely a convenient name we give to the property in which the relation data is stored at run-time.

```
A door has an object called other side.
The other side property translates into I6 as "door_to".
Leading-through relates one room (called the other side) to various doors.
The verb to be through implies the leading-through relation.
```

§**31. Containers and supporters.** The carrying capacity property is the exception to the remarks above about the qualitative nature of the world model: here for the first and only time we have a value which can be meaningfully compared.

```
Section SR1/6 - Containers

The specification of container is "Represents something into which portable
things can be put, such as a teachest or a handbag. Something with a really
large immobile interior, such as the Albert Hall, had better be a room
instead."

A container can be enterable.
A container can be opaque or transparent. A container is usually opaque.
A container has a number called carrying capacity.
The carrying capacity of a container is usually 100.

Include (- has container, -) when defining a container.
```

§**32.** The most interesting thing to note here (and we will see it again in the definition of "people") is that "transparent" the I7 property is not a direct match onto `transparent` the I6 attribute. In I7, the term is applicable only to containers (a reform made in January 2008, but clarifying what was already de facto the case). In I6, the `transparent` attribute means that child-objects in the object tree are in scope whenever the parent object is: in the I7 world model that's always true for supporters, so we oblige all supporters to have the attribute `transparent` in their I6 compiled forms. The same will be true for people. That doesn't in practice mean that I7 never has high shelves or people with daggers concealed beneath cloaks – just that we no longer use I6's mechanism for hiding these things, and expect the user to write activity rules instead.

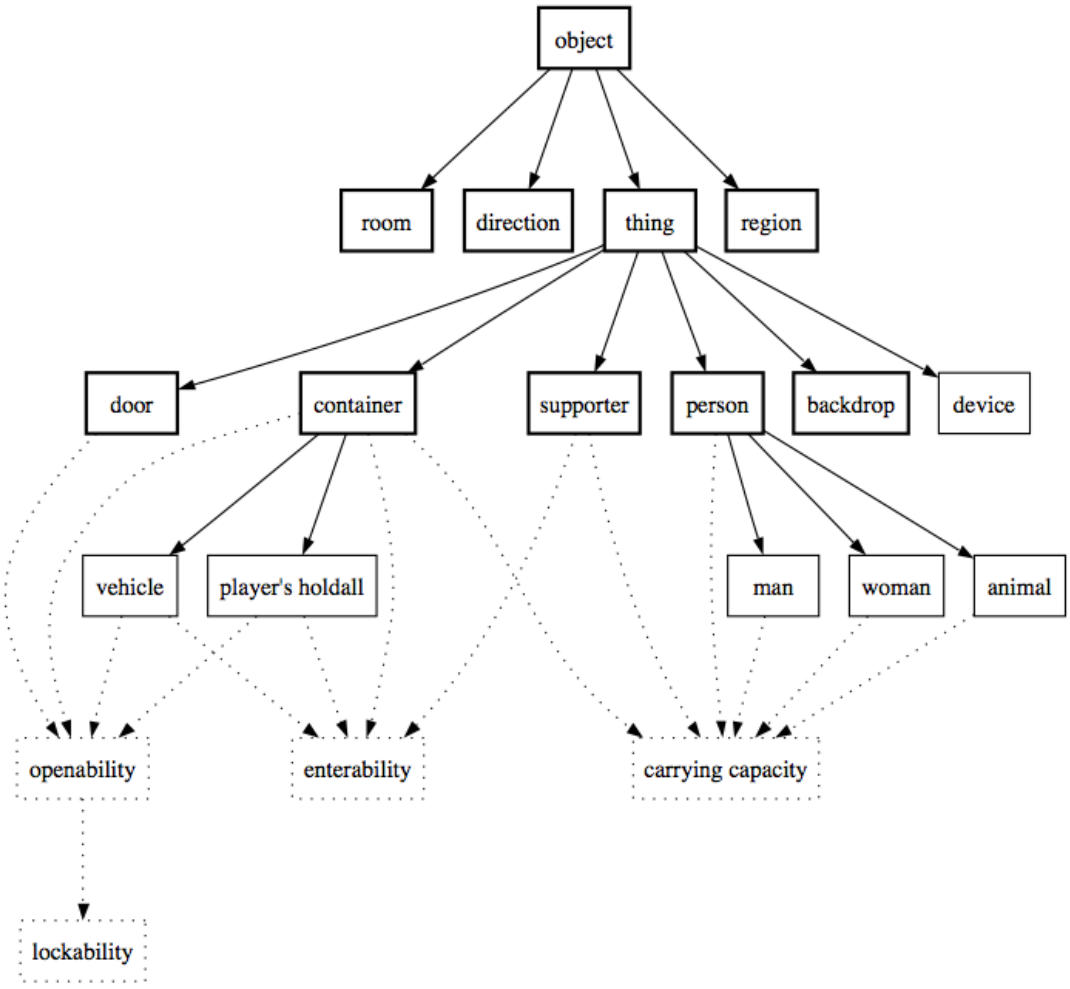
```
Section SR1/7 - Supporters

The specification of supporter is "Represents a surface on which things can be
placed, such as a table."

A supporter can be enterable.
A supporter has a number called carrying capacity.
The carrying capacity of a supporter is usually 100.

A supporter is usually fixed in place.

Include (-
    has transparent supporter
-) when defining a supporter.
```

§**33. Kinds vs patterns.** A problem faced by all object-oriented systems is "fear of the diamond", the problematic diagram of inheritance which results when we have two different subclasses B and C of a class A, which represent quite different ideas, but then we also turn out to want some behaviour D which is shared between some of the Bs and some of the Cs. For instance, we might have one class for people and another for buildings, but want to use the same code when it comes to (say) printing out top ten lists of basketball players (people) and skyscrapers (buildings) in height order: why not? But then again, what does D conceptually represent? Surely we aren't saying there's a natural concept of "basketball player/skyscraper"?

There are various responses, of which the most widely used now is probably that of `C++`'s notion of templates. We would define our top-ten business by writing a function applying to a list of objects of any class T such that T provided a height: there would then be no need for "basketball player/skyscraper" to be a class in its own right. Instead, we would define the behaviour as being available to anything for which it makes sense.

This is broadly what Inform 7 does, too, though not so formally. We use the term "pattern" for this, and have actually seen two patterns already – the way that containers and supporters share the "carrying capacity" limit, and also the notion of transparency – and it's by providing two patterns that we are able to deal with the likeness and also unlikeness of doors and containers. Their unlikeness is obvious; but their likeness is that they both grant or withhold access to some extent of space bordering on the current one. (Doors do this for the "horizontal" spatial model between rooms, whereas containers do it for the "vertical" spatial model of objects enclosing each other.)

The following diagram shows the kinds created by the Standard Rules (bold boxes for fundamental kinds, plain ones for discretionary additions) and the patterns of behaviour they share (dotted boxes).



§**34. The openability pattern.** To satisfy the openability pattern, a thing has to provide both of the either/or properties "open" and "openable". This entitles it to be opened and closed by the opening and closing actions, defined below. Note that I7 has no formal concept of patterns as part of its type-checking: instead, the rules for these actions explicitly check that they are being applied to things matching the pattern, as we shall see.

Doors and containers have, as it happens, exactly opposite conventions about the default values of these properties: but that doesn't mean they don't share the pattern.

```
Section SR1/8 - Openability

A door can be open or closed. A door is usually closed.
A door can be openable or unopenable. A door is usually openable.

A container can be open or closed. A container is usually open.
A container can be openable or unopenable. A container is usually unopenable.
```

§**35. The lockability pattern.** And similarly for lockability, because a principle of the world model is that any spatial barrier can be given a lock if the designer so chooses. To satisfy this pattern, a thing must

(i) satisfy the openability pattern, and
(ii) provide both the either/or properties "lockable" and "locked", and also the value property "matching key".

Both doors and containers make some implications so that the words "lockable" and "locked" carry the implied meanings which human readers expect, but this is not essential to the functioning of lockability: it's only a graceful addition.

```
Section SR1/9 - Lockability

A door can be lockable. A door is usually not lockable.
A door can be locked or unlocked. A door is usually unlocked.
A door has an object called a matching key.
A locked door is usually lockable. [An implication.]
A locked door is usually closed. [An implication.]
A lockable door is usually openable. [An implication.]

A container can be lockable. A container is usually not lockable.
A container can be locked or unlocked. A container is usually unlocked.
A container has an object called a matching key.
A locked container is usually lockable. [An implication.]
A locked container is usually closed. [An implication.]
A lockable container is usually openable. [An implication.]
```

§**36.** Note that the lock-fitting relation has, as its domains, "thing" and "thing". That means that compile-time typechecking will not reject an attempt to apply the relation to (say) two vehicles. At run time, evaluating "if X unlocks P" where P is a peculiar thing with no possibility of a lock will always come out false; but trying to force it with "now X unlocks P" will cause a run-time problem. In short, patterns are defended at run-time, not at compile-time.

```
Lock-fitting relates one thing (called the matching key) to various things.
The verb to unlock (it unlocks, they unlock, it unlocked, it is unlocked)
implies the lock-fitting relation.
```

§**37. Backdrops.** The true subtlety of backdrops is not visible in the brief description here: but they require careful handling both in NI and in the template layer code, because they can be in many rooms at once.

```
Section SR1/10 - Backdrops

The specification of backdrop is "Represents an aspect of the landscape
or architecture which extends across more than one room: for instance,
a stream, the sky or a long carpet."

A backdrop is usually scenery.
A backdrop is always fixed in place.
A backdrop is never pushable between rooms.
```

§**38. People.** From a compilation point of view, people are surprisingly easy to deal with. It may well be argued that this is because the I6 world model is so sketchy in modelling them, but that may actually be a good thing, because it's not at all obvious that any single model will be sensible for what different authors want to do with their characters.

On gender, see also the "man" and "woman" kinds below. Note that we have three gender choices available – male, female and neuter – but these are, for historical reasons to do with how gender is handled by the I6 library, managed using either/or properties rather than a single three-way value property. This doesn't in practice cause trouble. (Specifying something as neuter overrides the male/female choice, if anyone does both for the same object, but in practice nobody does.) When nothing is said about a person's gender, it is assumed male, though this is used only linguistically (for instance, the pronoun HIM can be used in commands about the object, rather than HER or IT). There has to be some convention here, and in a case where we don't know our linguistic ground, opting for the least surprising behaviour seems wisest.

```
Section SR1/11 - People

The specification of person is "Despite the name, not necessarily
a human being, but anything animate enough to envisage having a
conversation with, or bartering with."

A person can be female or male. A person is usually male.
A person can be neuter. A person is usually not neuter.

A person has a number called carrying capacity.
The carrying capacity of a person is usually 100.
```

§**39.** I6 has a concept approximately equivalent to I7's "person" – the I6 library attribute `animate` – but I6 allows only some of those objects to become the protagonist during play (using I6's `ChangePlayer` routine). To be eligible, an object must not only be `animate` but also provide a whole host of writeable properties.

But I7 provides those I6 properties for every "person", for the sake of a clean, uniform design, and accepting the cost that people therefore take more bytes of precious Z-machine array space than they necessarily would in I6. This is all part of the doctrine that in I7, all characters are equal in status: all can be the player, all can carry out actions. Anyway: here are all of those I6 properties, spatchcocked into the `Class` definition which NI will compile for "person" – see §21 of the DM4 for details of why these are needed and what they do.

```
Include (-
    has transparent animate
    with before NULL, ! number 0,
-) when defining a person.
```

§**40.** So to the thirteenth and final object created by the Standard Rules: the enigmatic default protagonist, whose name is not "player" but "yourself". (The I6 library requires this object to be created as `selfobj`, but that's not a name that is ever printed or parsed: it's a constant value used only in I6 source code.)

The `yourself` object has to be proper-named to prevent the I6 library from talking about "the yourself", as it otherwise might. "Undescribed" in this context means that "yourself" is not described as being present in room descriptions: this would be redundant and annoying.

The I6 property `saved_short_name` property is an implementation convenience for use if there is ever a change of player, in which case the printed name of the object will cease to be "yourself" and become "your former self" instead. When this happens, the previous printed name (or `short_name` in I6 terms) is stored in `saved_short_name` so that it can recovered later. (We can't assume it was necessarily "yourself" because the source text might have overridden this with a sentence like "The printed name of the player is "your dreary self".")

```
The yourself is an undescribed person. The yourself is proper-named.

The description of yourself is usually "As good-looking as ever."

The yourself object translates into I6 as "selfobj".
Include (-
    has proper,
    with saved_short_name "yourself",
-) when defining yourself.
```

§**41. Non-fundamental kinds.** We have now finished defining the nine fundamental kinds which NI requires in order for it to function. There are six more to define, but it's worth emphasising that none of these are required or assumed by either NI or its template layer of I6 code. So any of them could be changed drastically or got rid of entirely simply by amending the Standard Rules. (Like the "player-character" kind, born early 2003, died July 2007.)

Equally, we could add others if we wanted. The judgement of what ought to be part of the basic hierarchy of kinds created by the Standard Rules isn't easy. The maximalist position is that users welcome a plethora of kinds to simulate many facets of real life, from canal-boats to candles. The minimalist position is that kinds are necessary only as the domain of relations (so that person is necessary as the domain of P in "P carries X", for instance), and that too many kinds confuses the picture and imposes what may be a constraining structure on the user, who should be free to decide for himself what concepts are most helpful to organise. These arguments are discussed further in the white paper, *Natural Language, Semantic Analysis and Interactive Fiction* (2005), but briefly: we are minimalist but not puritanically so.

§**42. Men, women and animals.** Of these discretionary kinds, so to speak, "man" and "woman" are perhaps the least challengeable. They are not obviously the domains of any natural relation (unless one takes a very old-fashioned idea of gender identity and supposes that, oh, "X keeps Y" implies that X is a wealthy man and Y a mistress). But they are so linguistically natural in story-telling: who would ever write "Jack is a person in the House. Jack is male." in preference to "Jack is a man in the House."

An awkward point here is that, of course, most people would simply say "Jack is in the House." and expect us to infer that Jack is a person from the fact that this is more often a human name than, say, a proprietary brand of microphone plug; and that Jack is male, because relatively few girls called Jacqueline are nicknamed Jack. As it happens the NI compiler doesn't allow for tentative statements about the kinds of objects (only about their property values), but it wouldn't be too hard to add such a system, with a little care. The trouble is that we would then need a large dictionary of boys' and girls' names, valid across American, Canadian, Australian and British English (together with a selection from foreign tongues), and this would always lead to puzzling omissions (why isn't "Glanville" recognised?) or ambiguities (why is "Pat" a man?). And similarly for titles: "Mr", "Mrs" and "Ms" are fairly indicative of gender, except in certain military contexts, but what about (say) "Admiral" or "Reverend", where there is a strong likelihood of masculinity but no more than that? So Inform 7's compromise position is that the user does have to specify gender

explicitly, but that the kinds "man" and "woman" provide conveniently abbreviated ways to do so. (We consider male and female children to qualify in these categories.)

Anyway, we set out the Anglo-Saxon plurals, and then declare these kinds purely in terms of gender: they have no distinguishing behaviour.

```
Section SR1/12 - Animals, men and women

The plural of man is men. The plural of woman is women.

A man is a kind of person.
The specification of man is "Represents a man or boy."
A man is always male. A man is never neuter.

A woman is a kind of person.
The specification of woman is "Represents a woman or girl."
A woman is always female. A woman is never neuter.
```

§**43.** But what about "animal"? Animals turn up often in IF, and of course domestic animals have been part of human society since prehistoric times: but then again, the same can be said for stoves and larders, and we aren't declaring kinds for those.

The reason "animal" exists is mainly because it is almost always peculiar to write "P is a person". Now that we have "man" and "woman" taken care of, the remaining objects we might want to declare will almost always fall into this category: it's intended to be used for "people" who are animate but probably not intelligent, or anyway, not participants in human society. It seems unusual to write "The black Labrador is a person." because that sounds like an insistent assertion of rights and thus a quite different sort of statement. (Don't drown that Labrador! He's a *person*.)

As can be seen from the tiny definition of "animal", though, it's really nothing more than a name for a position in the kinds hierarchy. There is not even any implication for gender.

```
An animal is a kind of person.

The specification of animal is "Represents an animal, or at any rate a
non-human living creature reasonably large and possible to interact with: a
giant Venus fly-trap might qualify, but not a patch of lichen."
```

§**44. Devices.** The justification for providing a "device" kind is much thinner. It's done largely for traditional reasons – such a concept existed in the I6 library, which in turn followed Infocom conventions dating from the early 1980s. The inclusion is defensible as representing a common linguistic category found in everyday situations, where an inanimate object nevertheless does something while under direct or indirect human control: we can also imagine relations for which it could be a domain ("X is able to work D" meaning that person X understands how to use the controls of device D, say). It could equally be attacked as having a rather flimsy world model – it's just an on/off switch – and representing a pretty inchoate mass of concepts, from a mousetrap to a nuclear reactor.

```
Section SR1/13 - Devices

A device is a kind of thing.

A device can be switched on or switched off. A device is usually switched off.
Include (- has switchable, -) when defining a device.

The specification of device is "Represents a machine or contrivance of some
kind which can be switched on or off."
```

§**45. Vehicles.** Here again the justification boils down to tradition. Vehicles were a staple ingredient of the Infocom classics, largely because of code originally written for the inflatable boat in the 1978-79 mainframe version of *Zork*, which was then copied through into later titles. Unlike devices, though, vehicles are genuinely difficult to model, and the implementation provided by the Standard Rules would be quite a lot of work for a user to manage alone. (Consider, for instance, the case when the player is sitting in an open basket when Bill, driving a fork-lift truck, uses his vehicle to push the basket into another room.) There might perhaps be a case for moving all of the vehicles material into an extension, but it would have to be an extension supplied as part of the built-in set, and whenever it was used the result would be that the going action would rely on a pretty complicated interlacing of rules as between this extension and the Standard Rules.

Turning to implementation, I6 – surprisingly, perhaps – doesn't have a `vehicle` attribute: a vehicle is an object which is `enterable` and whose `before` rule for the I6 `##Go` action returns the magic value 1. A troublesome point here is that I6 makes no distinction between vehicles which contain and vehicles which support. But we do, because once we have decided to make "vehicle" a kind, it has to be either a kind of container or a kind of supporter: it can't be both. We get around this by providing for container-vehicles in the Standard Rules, as being the more commonly occurring case, while providing for the other with the extension Rideable Vehicles by Graham Nelson, which is in effect an offshoot of the Standard Rules and is built-in to every installation of Inform 7. This also provides for animals used as vehicles.

The alternative approach here would be to make "vehicle" not a kind but an either/or property of things, so as to provide a pattern of behaviour common to certain animals, containers and supporters. We could then move Rideable Vehicles back into the Standard Rules, but that would add a fair amount of code, and besides, it is unclear that "vehicleness" is something we want to come and go during play, or that it's appropriate as an either/or property of (for instance) a door or a person.

```
Section SR1/14 - Vehicles

A vehicle is a kind of container.

The specification of vehicle is "Represents a container large enough for
a person to enter, and which can then move between rooms at the driver's
instruction. (If a supporter is needed instead, try the extension
Rideable Vehicles by Graham Nelson.)"

A vehicle is always enterable.
```

§**46.** The part about vehicles not usually being portable is simply for realism's sake: generally speaking if something can hold human weight it's pretty large and heavy. (A bicycle is an edge case, and a skateboard is clearly an exception, but that's why the rule is only "usually".)

If all vehicles were wheeled, there would be a case for a rule such as "A vehicle is usually pushable between rooms." But this seems more likely to trip up the designer with a surprise discovery in beta-testing than to help him achieve realism. We don't want to be able to push hot-air balloons, boats or spacecraft between rooms.

```
A vehicle is usually not portable.
```

§**47. Player's holdalls.** This is the final kind created in the Standard Rules, and probably the most doubtful of all. It simply provides a hook to a cute and traditional feature of the I6 library whereby spare possessions are automatically cleared out of the player's way: it derives from the rucksack in the 1993 IF title *Curses*.

```
Section SR1/15 - Player's holdall

A player's holdall is a kind of container.

The specification of player's holdall is "Represents a container which the
player can carry around as a sort of rucksack, into which spare items are
automatically stowed away."

A player's holdall is always portable.
A player's holdall is usually openable.
```

§**48.** To enable the use of player's holdalls, we must declare a constant `RUCKSACK_CLASS` to tell some code in the template layer to use possessions with this I6 class as the rucksack pro tem. This is all a bit of a hack, to retrofit a degree of generality onto the original I6 library feature, and even then it isn't really fully general: only the player has the benefit of a "player's holdall" (hence the name), with other actors oblivious.

```
Include (-
    Constant RUCKSACK_CLASS = (+ player's holdall +);
-) after "Definitions.i6t".
```

§**49. Correspondence between I6 and I7 property and attribute names.** All of the kinds, objects and properties which make up the standard kit provided to every source text are now complete. We conclude Section SR1 by giving the NI compiler a dictionary to tell it how I7's names for properties – some value properties, some either/or – mesh with those in the I6 library.

Ordinarily, a new value property such as "astral significance" would be compiled by NI into an I6 property called something like

```
P73_astral_significance
```

whereas a new either/or property might become either an I6 attribute or an I6 property holding only `true` or `false`, at the compiler's discretion. (It needs to use this discretion because I6 has a hard limit on the number of attributes, whereas there are no limits on the number of properties used in I7.) And if "astral significance" is a concept handled only by I7 source text, that's fine.

But we want our "printed name" property, for instance, to be the text which the I6 library prints out whenever it uses the `short_name` of an object: so we want the NI compiler to use the I6 identifier `short_name` for "printed name", not to invent a new one. NI therefore maintains a dictionary of equivalents, and here it is. (Any I7 property not named is handled purely by I7 source text in the remainder of the Standard Rules.)

§**50.** First, equivalents where I7 either/or properties map directly onto I6 attributes. Note the way "lit" (for things) and "lighted" (for rooms) both map onto the same I6 attribute, `light`. Attributes were in scarce supply in I6 (with a limit of 32 in the early days, later raised to 48) and this sort of reuse seemed sensible in the early 1990s, especially as the meanings were basically similar.

```
Section SR1/16 - Inform 6 equivalents

The wearable property translates into I6 as "clothing".
The undescribed property translates into I6 as "concealed".
The edible property translates into I6 as "edible".
The enterable property translates into I6 as "enterable".
The female property translates into I6 as "female".
The initially carried property translates into I6 as "initially_carried".
The mentioned property translates into I6 as "mentioned".
The lit property translates into I6 as "light".
The lighted property translates into I6 as "light".
The lockable property translates into I6 as "lockable".
The locked property translates into I6 as "locked".
The handled property translates into I6 as "moved".
The neuter property translates into I6 as "neuter".
The switched on property translates into I6 as "on".
The open property translates into I6 as "open".
The openable property translates into I6 as "openable".
The privately-named property translates into I6 as "privately_named".
The plural-named property translates into I6 as "pluralname".
The proper-named property translates into I6 as "proper".
The pushable between rooms property translates into I6 as "pushable".
The scenery property translates into I6 as "scenery".
The fixed in place property translates into I6 as "static".
The transparent property translates into I6 as "transparent".
The visited property translates into I6 as "visited".
The marked for listing property translates into I6 as "workflag".
```

§**51.** Second, the I7 value properties mapping onto I6 properties. Again, `map_region` is a new I6 property of our own, while the rest are I6 staples. And see also "other side", which is translated above for timing reasons.

```
The indefinite article property translates into I6 as "article".
The carrying capacity property translates into I6 as "capacity".
The description property translates into I6 as "description".
The initial appearance property translates into I6 as "initial".
The map region property translates into I6 as "map_region".
The printed plural name property translates into I6 as "plural".
The printed name property translates into I6 as "short_name".
The matching key property translates into I6 as "with_key".
```

SR2 - Variables and Rulebooks A/sr2

Purpose

The global variables and those built-in rulebooks which do not belong either to specific actions or to specific activities.

---

§**1.** With the kinds, objects and properties all now made, we turn to more abstract constructions: rules, rulebooks, activities, global variables and so on. As at the beginning of Part SR1 above, we begin with an entirely clean slate – none of any of these things exists yet – but also with expectations on what we do, since NI and the I7 template layer need certain constructions to be made.

We begin with global variables. The order in which these are defined, and the section subheadings under which they are grouped, determine the way they are indexed in the Contents index of a project – that is, moving them around would reorder the Contents index, and rewording the subheadings SR2/1, SR2/2, ..., below would change the text of the subheadings in the Contents index accordingly.

Some of these variables are special to NI, and it knows which they are not by the order in which they're defined (as with kinds, above) but by their (I7) names. These are marked with the comment `[*]`, and cannot be renamed without altering NI to match. Those marked `[**]` are similarly named in the template layer.

§**2.** It's now the occasion for our first global variable definition, because although users often think of the protagonist object as a fixed object whose name is "player", it's in fact possible to change perspective during play and become somebody else – at which point the "player" variable will point to a different object.

Note that "player" is actually a variable belonging to the I6 library, as indeed most of those in the Standard Rules are. Without the explicit translation supplied below, it would probably be compiled as `Q1_player` or some such name; and might well be implemented by NI as an entry in an array (since the Z-machine supports only a limited number of global variables, in effect using them as a set of 240 registers, and we therefore don't want to create too many ourselves).

```
Part SR2 - Variables and Rulebooks

Section SR2/1 - Situation

The player is a person that varies. [*]
The player variable translates into I6 as "player".
```

§**3.** The I7 variable "location" corresponds to I6's `real_location`, not `location`. Its value is never equal to a pseudo-room representing darkness: it is always an actual room, and I7 has nothing corresponding to I6's `thedark` "room". Similarly, we use an I7 variable "darkness witnessed" for a flag which the I6 library would have stored as the `visited` attribute for the `thedark` object.

The "maximum score" is, rather cheekily, translated to an I6 constant: and this cannot be changed at run-time.

```
The location -- documented at var_location -- is an object that varies. [*]
The score -- documented at var_score -- is a number that varies.
The last notified score is a number that varies.
The maximum score is a number that varies. [*]
The turn count is a number that varies.
The time of day -- documented at var_time -- is a time that varies. [*]
The darkness witnessed is a truth state that varies.

The location variable translates into I6 as "real_location".
The score variable translates into I6 as "score".
The last notified score variable translates into I6 as "last_score".
The maximum score variable translates into I6 as "MAX_SCORE".
The turn count variable translates into I6 as "turns".
The time of day variable translates into I6 as "the_time".
```

§**4.** It is arguable that "noun", "second noun" and "person asked" ought to be rulebook variables belonging to the action-processing rules, so that they essentially did not exist outside of the context of an ongoing action. The reason this isn't done is partly historical (rulebook variables were a fairly late development, implemented in April 2007, though they had long been planned). But it is sometimes useful to get at the nouns in an action after it has finished, and making them global variables also makes them a little faster to look up (a good thing since they are used so much), and causes them to be indexed more prominently.

"Item described" is simply an I7 name for `self`. (In an object-oriented system such as I6, `self` is the natural way to refer to the object currently under discussion, within routines applying to all objects of its class.) In early drafts of I7, it was called "this object", but somehow this raised expectations too high about how often it would be meaningful: it looked like a pronoun running meanings from sentence to sentence.

```
Section SR2/2 - Current action

The noun -- documented at var_noun -- is an object that varies. [*]
The second noun is an object that varies. [*]
The person asked -- documented at var_person_asked -- is an object that varies. [*]
The reason the action failed -- documented at var_reason -- is a rule that varies.
The item described is an object that varies.

The noun variable translates into I6 as "noun".
The second noun variable translates into I6 as "second".
The person asked variable translates into I6 as "actor".
The reason the action failed variable translates into I6 as "reason_the_action_failed".
The item described variable translates into I6 as "self".
```

§**5.** "Person reaching" turns out to have exactly the same meaning as "person asked" – they are both the `actor`, in I6 terms, but are used in different situations.

```
Section SR2/3 - Used when ruling on accessibility

The person reaching -- documented at var_person_reaching -- is an object that varies.
The container in question is an object that varies.
The supporter in question is an object that varies.
The particular possession -- documented at var_particular -- is a thing that varies.

The person reaching variable translates into I6 as "actor".
The container in question variable translates into I6 as "parameter_object".
The supporter in question variable translates into I6 as "parameter_object".
The particular possession variable translates into I6 as "particular_possession".
```

§**6.** Parsing variables follow. The I6 parser tends to put any data read as part of a command into the variable `parsed_number`, but then, I6 is typeless: we can't have a single I7 variable for all these possibilities since it could then have no legal type. We solve this as follows. Whenever a kind of value K is created which can be parsed as part of a command, an I7 variable "the K understood" is also created, as a K that varies. All of these variables are translated into I6's `parsed_number`, so in effect they provide aliases of each possible type for the same underlying memory location. The four exceptional kinds of value not parsed by the systematic approaches in NI for enumerated KOVs and units are "number", "time", "snippet" and "truth state": because of their exceptional status, they don't get "the K understood" variables created automatically for them, so we must construct those by hand. Hence "the number understood", "the time understood", "the topic understood" (for historical reasons this one is not called "the snippet understood"), "the truth state understood" but no others.

```
Section SR2/4 - Used when understanding typed commands

The player's command -- documented at var_command -- is a snippet that varies.
The matched text is a snippet that varies.
The number understood -- documented at var_understood -- is a number that varies. [*]
The time understood is a time that varies. [*]
The topic understood is a snippet that varies. [*]
The truth state understood is a truth state that varies. [*]
The current item from the multiple object list is an object that varies.

The player's command variable translates into I6 as "players_command".
The matched text variable translates into I6 as "matched_text".
The topic understood variable translates into I6 as "parsed_number".
The truth state understood variable translates into I6 as "parsed_number".
The current item from the multiple object list variable translates into I6 as
    "multiple_object_item".
```

§**7.** The following are, unless the user creates global variables of his own, the last in the Contents index...

```
Section SR2/5 - Presentation on screen

The command prompt -- documented at var_prompt -- is a text that varies. [**]
The command prompt is ">".

The left hand status line -- documented at var_sl -- is a text that varies.
The right hand status line is a text that varies.

The left hand status line variable translates into I6 as "left_hand_status_line".
The right hand status line variable translates into I6 as "right_hand_status_line".

The listing group size is a number that varies.
The listing group size variable translates into I6 as "listing_size".
```

§**8.** ...but they are not the last global variables created by the Standard Rules.

These bibliographic data variables are concealed because they are under a heading which ends with the word "unindexed". There are two reasons why these variables are unindexed: first, they appear in a different guise only a little lower in the Contents index as part of the Library Card, and second, we don't want users to think of them as manipulable during play.

Rather sneakily, we also define a Figure here. This is done in order to make it legal to declare variables and properties of the kind of value "figure-name" (because it ensures that such variables can always be initialised – there is always at least one Figure in the world). Of course plenty of Inform projects have no artwork at all: so the cover art figure is unique in that it might refer to nothing. That sounds a little arbitrary but in fact follows a convention used by the Blorb format for binding story files with their resources, which in turn follows Infocom conventions of 1987-89: the cover art always has resource ID number 1, whether it exists or not. NI creates figures and sound effects counting upwards from 1, giving them each unique resource ID numbers, so the first to be created gets ID number 1: by defining "figure of cover" here, we can be sure that we are first, so everything works.

```
Section SR2/6 - Unindexed Standard Rules variables - Unindexed

The story title, the story author, the story headline, the story genre
and the story description are text variables. [*****]
The release number and the story creation year are number variables. [**]
Figure of cover is the file of cover art.

The story title variable translates into I6 as "Story".
```

§**9.** And we finish out the set with some "secret" variables used only by the Standard Rules or by NI, and which are therefore also unindexed. Their names are all hyphenated, to reduce the chance of anyone stumbling into them in a namespace clash.

The first set of three secret variables is used by the predicate calculus machinery in NI. This is the code which handles logical sentences such as "at least six doors are open" or descriptions such as "open doors", by reducing them to a logical notation which sometimes makes use of variables. For instance, "open doors" reduces to something like "all x such that door(x) and open(x)", with x being a variable. When NI works with such logical propositions, it often needs to substitute $x = c$, that is, to replace x with a given constant c. But it can only do this if c is an Inform 7 value. This is a problem if what it wants is to substitute in something which is only meaningful at the I6 level: say, it wants to substitute a value c which will eventually translate to `whatever`, but it can't find any I7 value c which will do this.

We solve this problem by constructing some unusual I7 variables whose only purpose is that NI can use them in substitutions. They should never be referred to in I7 source text anywhere else at all, not even elsewhere in the Standard Rules.

(1) The "substitution-variable" is used when NI needs to compile a proposition with one free variable, such as "door(x)", as a condition: it binds x with the substitution $x =$ *say-parameter* in order to turn this into a well-formed predicate calculus sentence with no free variables, then compiles it. This is a trick used in the definition of some of the "repeat" phrases below, as a result of the `{-bind-variable:...}` invocation command.

(2) The sentence "The i6-nothing-constant is an object that varies." is a rare example of a flat lie in the Standard Rules, as it is the I6 constant `nothing` and never varies at all. Again, it exists as a "variable" so that the substitution $x =$ `nothing` can be made.

(3) Well, once you start telling lies it's so hard to stop, and it's also a lie that the "I6-varying-global" translates to `nothing`. It actually translates to whatever the NI machinery for compiling propositions happens to want at the moment, so it has no permanent meaning at all. (It will always translate to an I6 global variable storing a value whose I7 kind is "object", so the type-checking machinery isn't endangered by this chicanery. It will in fact never translate to `nothing`, but we make the translation sentence below in order to avoid allocating any storage at run-time for what is in the end only a label.)

```
The substitution-variable is an object that varies. [*]
The substitution-variable variable translates into I6 as "subst__v".

The I6-nothing-constant is an object that varies. [*]
The I6-nothing-constant variable translates into I6 as "nothing".

The I6-varying-global is an object that varies. [*]
The I6-varying-global variable translates into I6 as "nothing".
```

§**10.** The remaining secret variables are:

(1) The "item-pushed-between-rooms" is needed to get the identity of an object being pushed by a command like PUSH ARMCHAIR NORTH out of I6 and into the action variable "thing gone with" of the going action.

(2) The "actor-location" is needed temporarily to store the room in which the actor of the current action is standing, and it wants to be an I6 global (rather than, say, a rulebook variable belonging to the action-processing rulebook) so that NI can use common code to handle this alongside `noun`, `second` and `actor` when compiling preambles to rules.

(3) The "parameter-object" is likewise needed in order to compile preambles to rules in object-based rulebooks.

```
The item-pushed-between-rooms is an object that varies.
The item-pushed-between-rooms variable translates into I6 as "move_pushing".

The actor-location is an object that varies. [*]
The actor-location variable translates into I6 as "actor_location".

The parameter-object is an object that varies. [*]
The parameter-object variable translates into I6 as "parameter_object".
```

§**11.** And that completes the run through all the variables created in the Standard Rules.

§**12. Rulebooks.** There are 25 rulebooks which are, so to speak, "primitive" in Inform 7 – which are part of its workings and cannot safely be tampered with. NI requires them to be declared by the Standard Rules as the first 25 rulebooks to be created, and in the exact order below. (In fact, though, this is mostly so that it can prepare the index pages correctly: NI is not the part of I7 which decides what to use these rulebooks for. That is almost always the responsibility of the template I6 layer which, for instance, calls upon the action-processing rulebook when it wants an action processed.)

At the I6 level, a rulebook is referred to by its ID number, which counts upwards from 0 in order of creation. Any reordering of the constants below, therefore, is unsafe unless changes are made elsewhere so that the following three tallies always remain in synchrony:

(a) The sequence of declaration of these rulebooks in the Standard Rules.
(b) The inweb `@d` definitions in the form `TURN_SEQUENCE_RB` in the section Rulebooks of Chapter 12 of the NI source code.
(c) The I6 `Constant` definitions in the form `TURN_SEQUENCE_RB` in the file `Rulebooks.i6t` of the template I6 layer.

Anyway, we will declare the rulebooks and their variables or outcomes first, and come back to stock some of them with rules later. It seems appropriate to give first place to the procedural rules, the super-rulebook capable of controlling all others:

```
Section SR2/7 - The Standard Rulebooks

Procedural rules is a rulebook. [0]
```

§**13.** Every story file created by Inform 6 begins execution in a routine called `Main`, which is analogous to the `main()` function of a C program: it is as if the entire program is a call to this function.

In an I7 story file, the code in `Main` is the only code which is not executed in the context of a rulebook, and by design it does as little as possible. The definition is in part "Main" of "OrderOfPlay.i6t", and this is what it does:

(1) Consider the startup rules.
(2) Repeatedly follow the turn sequence rules until `deadflag` is set, which is an I6 variable used to indicate that the game has ended in one way or another.
(3) Follow the shutdown rules.

The startup rules are only considered, not followed, because we cannot risk a procedural rule being the first thing executed: starting up the virtual machine correctly has to be our first priority. (Except for not using the procedural rules, considering a rulebook is identical to following it.)

Briefly, the startup phase takes us to the end of the room description after the banner is printed. The turn sequence covers a complete turn, and runs through from prompting the player for a command to notifying him of any change in score which occurred. The shutdown rules then go from printing the obituary text, through final score, to the question about quitting or restarting.

```
Startup rules is a rulebook. [1]
Turn sequence rules is a rulebook. [2]
Shutdown rules is a rulebook. [3]
```

§**14.** Now a set of rulebooks to do with the passage of time.

```
Scene changing rules is a rulebook. [4]
When play begins is a rulebook. [5]
When play ends is a rulebook. [6]
Every turn rules is a rulebook. [7]
```

§**15.** The action machinery requires some 16 rulebooks to work, though that is the result of gradual simplification – in 2006 it required 25, for instance. The "action-processing" rulebook, like the turn sequence rulebook, is a master of ceremonies: it belongs to the Standard Rules and is only rarely if at all referred to by users.

As remarked above, it's something of a historical accident that "actor" is a rulebook variable belonging to the action-processing rules (and thus in scope for every rulebook it employs) while "noun" and "second noun" are global variables (and thus in scope everywhere).

The main action-processing rulebook delegates most of its detailed work to a subsidiary, the "specific action-processing" rulebook, at the point where what rulebooks we consult next depends on what the action is (hence "specific") – see below for more on how check/carry out/report rules are filed.

```
Action-processing rules is a rulebook. [8]
The action-processing rulebook has a person called the actor.
Setting action variables is a rulebook. [9]
The specific action-processing rules is a rulebook. [10]
The specific action-processing rulebook has a truth state called action in world.
The specific action-processing rulebook has a truth state called action keeping silent.
The specific action-processing rulebook has a rulebook called specific check rulebook.
The specific action-processing rulebook has a rulebook called specific carry out rulebook.
The specific action-processing rulebook has a rulebook called specific report rulebook.
The specific action-processing rulebook has a truth state called within the player's sight.
The player's action awareness rules is a rulebook. [11]
```

§**16.** The rules on accessibility and visibility, which control whether an action is physically possible, have named outcomes as a taste of syntactic sugar.

```
Accessibility rules is a rulebook. [12]
Reaching inside rules is an object-based rulebook. [13]
Reaching inside rules have outcomes allow access (success) and deny access (failure).
Reaching outside rules is an object-based rulebook. [14]
Reaching outside rules have outcomes allow access (success) and deny access (failure).
Visibility rules is a rulebook. [15]
Visibility rules have outcomes there is sufficient light (failure) and there is
insufficient light (success).
```

§**17.** Two rulebooks govern the processing of asking other people to carry out actions:

```
Persuasion rules is a rulebook. [16]
Persuasion rules have outcomes persuasion succeeds (success) and persuasion fails (failure).
Unsuccessful attempt by is a rulebook. [17]
```

§**18.** Next, the six classic rulebooks best known to users of Inform. It's perhaps an unfortunate point of the design that there are so many as six: that seems rather a lot of stages to go through, and indeed means that there is sometimes some ambiguity about which rulebook to use if one wants to achieve a given effect. There are really two reasons why things are done this way:

(a) To try to encourage a distinction between:

(i) the general implementation of an action, made with carry out, check and report rules – say, a "photographing" action which could be used in any situation and could be copied and pasted into another project; and

(ii) the contingent rules applying in particular situations in play, made with before, instead and after rules, such as that custodians at the Metropolitan Museum of Art forbid flash photography.

(b) To improve the efficiency of action-processing by forcing control to run only through those carry out, check and report rules which can possibly be relevant to the current action. Whereas all before, instead and after rules are all piled up together in their own rulebooks, check, carry out and report rules are divided up into specialised rulebooks tied to the particular action in progress. Thus on a taking action, the six stages followed are before, instead, check taking, carry out taking, after and report taking.

During play, then, the three rulebooks "check", "after" and "report" are completely empty. This is the result of a reform in April 2007 which wasn't altogether popular. Before then, NI rather cleverly filed rules like "Check doing something with the haddock" in the generic "check" rulebook and ran this rulebook as part of the action processing sequence. But this clearly broke principle (i) above, and meant that the six-stage process – already quite complicated enough – was actually a nine-stage process only pretending, by deceitful syntax, to be a six-stage one. Check rules sometimes appeared to be filed in the wrong order, breaking the ordinary precedence conventions, and this was not due to a bug but because they were only pretending all to be in the same rulebook. Still more clever indexing and rule-ordering tricks ameliorated this a little, but in the end it was just a bad design: withdrawing the ability to make check, carry out and report rules apply to multiple actions resulted in much cleaner code, and also a clearer conceptual definition of what these rulebooks were for. (But users still *didn't* like the change: actual functionality was withdrawn.)

So if they are always empty and never used, why are the three rulebooks called simply "check", "after" and "report" created in the first place? The answer is that this is a convenience for parsing rule preambles in NI: it provides a temporary home for such rules before they are divided up into their specific rulebooks, and it also makes it easier for NI to detect and give a helpful Problem message in response to rules like "Check taking or dropping the perch" which can't be filed anywhere in our scheme, and so have to be forbidden.

```
Before rules is a rulebook. [18]
Instead rules is a rulebook. [19]
Check rules is a rulebook. [20]
Carry out rules is a rulebook. [21]
After rules is a rulebook. [22]
Report rules is a rulebook. [23]
```

§**19.** The final rulebook in this roundup is one used for parsing ambiguous commands during play. This looks like the job of an activity rather than a rulebook, but (i) we want information in the form of a nicely-named five point scale of responses, rather than wanting something to be done, and (ii) it doesn't really make any sense to split that question into a before, for and after stage. So this is a named rulebook instead.

```
The does the player mean rules are a rulebook. [24]
The does the player mean rules have outcomes it is very likely, it is likely, it is possible,
it is unlikely and it is very unlikely.
```

§**20.** "Does the player mean" is essentially a front end for the I6 parser's `ChooseObjects` entry point, which relies on numerical scores to assess the likelihood of possible choices. That makes it useful to have an I6 wrapper function which consults the "does the player mean" rulebook, translates the named outcomes into a score from 0 to 4, and returns that. Note that if the rulebook makes no decision and has no outcome, we return the middle-of-the-road value 2. (This wrapper routine looks as if it belongs in the template I6 layer, but having it here allows it to use the `(+` and `+)` escapes.)

```
Include (-
    [ CheckDPMR result sinp1 sinp2 rv;
        sinp1 = inp1; sinp2 = inp2; inp1 = noun; inp2 = second;
        rv = FollowRulebook( (+does the player mean rules+) );
        inp1 = sinp1; inp2 = sinp2;
        if ((rv) && RulebookSucceeded()) {
            result = ResultOfRule();
            if (result == (+ it is very likely outcome +) ) return 4;
            if (result == (+ it is likely outcome +) ) return 3;
            if (result == (+ it is possible outcome +) ) return 2;
            if (result == (+ it is unlikely outcome +) ) return 1;
            if (result == (+ it is very unlikely outcome +) ) return 0;
        }
        return 2;
    ];
-);
```

§**21.** And that's it: all 25 of the named rulebooks now exist. There will, of course, be hundreds more rulebooks soon, created automatically as activities and actions are created – when we create the "dropping" action, for instance, we also create the "check dropping", "carry out dropping" and "report dropping" rulebooks – but there are no more stand-alone rulebooks.

§**22. Rules.** At run-time, the value of a rule is the (packed) address of an I6 routine. In the case of rules created in I7 source text, the body of the rule definition is compiled by NI to a new I6 routine which carries it out. But there are also primitive rules which are implemented in the template I6 layer, and these need no I7 source text definitions: instead we simply tell NI the name of the I6 routine which will be handling this rule, and that it need not bother to create one for itself.

An example of this is provided by the first rule we shall create: the "little-used do nothing rule". This is aptly named on both counts. We never follow it, we never put it into any rulebook. It exists only so that variables and properties with the kind of value "rule" can be initialised automatically to a safely neutral default value. It makes no decision.

```
Section SR2/8 - The Standard Rules

The little-used do nothing rule translates into I6 as "LITTLE_USED_DO_NOTHING_R".

Include (-
[ LITTLE_USED_DO_NOTHING_R; rfalse; ];
-);
```

§**23. Startup.** The startup rulebook is considered but not followed (see above) and so is immunised from the effect of procedural rules.

Every rulebook contains a (possibly empty) run of "first" rules, then a (possibly empty) run of miscellaneous rules, then a (possibly empty) run of "last" rules. It's unusual to have more than one rule anchored to either end as "first" or "last", but entirely legal, and we make use of that ability here.

The "first" rules here are the ones which get the basic machinery working to the point where it is safe to run arbitrary I7 source text. They necessarily do very low-level things, and it is not guaranteed that I7 phrases will behave to specification if executed before these early rules have finished. So it is hazardous to obstruct or alter them.

(a) The "virtual machine startup rule" carries out necessary steps to begin execution on the virtual machine in use: this entails relatively little on the Z-machine versions 5 or 8, but can involve extensive work to get the screen display working on Glulx or Z6. Before anything else happens, however, the "starting the virtual machine" activity (see below) is carried out: again, this is done in a careful way which avoids procedural rules firing.

(b) The "initialise memory rule" starts up the memory allocation heap, if there is one, and sets some essential I6 variables. If there is any rule not to meddle with, this is it.

(c) The "seed random number generator rule" seeds the RNG to a fixed value if NI has requested this (which it does in response to the `-rng` command line switch, which is in turn used by the `intest` testing utility: it's a way to make deterministic tests of programs which use random values).

(d) The "update chronological records rule" is described in further detail below, since it appears both here and also in the turn sequence rulebook. Here it's providing us with a baseline of initial truths from which we can later assess conditions such as "the marble door has been open". A subtle and questionable point of the design is that this rule is placed at a time when the object representing the player is not present in the model world. This is done to avoid regarding the initial situation as one of the turns for purposes of a rule preamble like "... when the player has been in the Dining Room for three turns". It's as if the player teleports into an already-existing world, like some Star Trek crewman, just in time for the first command.

(e) And the "position player in model world rule" completes the initial construction of the spatial model world.

(f) The "start in the correct scenes rule" ensures that we start out in the correct scenes. (This can't wait, because it's just conceivable that somebody has written a rule with a preamble like "When play begins during the Hunting Season...": it's also where the scene Entire Game begins.) That completes the necessary preliminaries before ordinary I7 rules can be run.

```
The start in the correct scenes rule is listed first in the startup rulebook. [6th.]
The position player in model world rule is listed first in the startup rulebook. [5th.]
The update chronological records rule is listed first in the startup rulebook. [4th.]
The seed random number generator rule is listed first in the startup rulebook. [3rd.]
The initialise memory rule is listed first in the startup rulebook. [2nd.]
The virtual machine startup rule is listed first in the startup rulebook. [1st.]

The virtual machine startup rule translates into I6 as "VIRTUAL_MACHINE_STARTUP_R".
The initialise memory rule translates into I6 as "INITIALISE_MEMORY_R".
The seed random number generator rule translates into I6 as "SEED_RANDOM_NUMBER_GENERATOR_R".
The update chronological records rule translates into I6 as "UPDATE_CHRONOLOGICAL_RECORDS_R".
The position player in model world rule translates into I6 as "POSITION_PLAYER_IN_MODEL_R".

This is the start in the correct scenes rule: consider the scene changing rules.
```

§**24.** The remaining rules, though, are fair game for alteration, and as if to prove the point they are all written in standard I7 source text. Note that the baseline score is set only after the when play begins rulebook has finished, because games starting with a non-zero score normally do this by changing the score in a when play begins rule: and we don't want such a change to be notified to the player as if it has happened through some action.

```
The when play begins stage rule is listed in the startup rulebook.
The fix baseline scoring rule is listed in the startup rulebook.
The display banner rule is listed in the startup rulebook.
The initial room description rule is listed in the startup rulebook.

This is the when play begins stage rule: follow the when play begins rulebook.

This is the fix baseline scoring rule: now the last notified score is the score.

This is the display banner rule: say "[banner text]".

This is the initial room description rule: try looking.
```

§**25. The turn sequence.** In each turn, a command is read and parsed from the keyboard, and any action(s) that requested is or are processed. (And may in turn cause other actions, which must also be processed.) There is then a fair amount of business needed to end one turn and get ready for another.

The turn sequences rulebook terminates early if `deadflag` becomes set at any point, so the last turn of play will be incomplete. Besides that consideration, it can also end early if the command for the turn was for an out-of-world action such as saving the game: in such cases, the "generate action rule" stops the rulebook once the action has been fully processed. All the same, play basically consists of running down this list of rules over and over again.

The turn sequence rulebook is the only one protected from being ignored by a procedural rule: ignoring it would lock the story file into an endless loop, so a run-time problem is issued instead if this is tried. (In pre-2008 drafts of Inform 7, ignoring what was then the turn sequence rulebook was a legitimate tactic, so the RTP message explains how to achieve the same effect by new methods.)

§**26.** The "first" rules in the turn sequence cover us up to the end of the events which take place in the model world during this turn's action(s).

(a) The "parse command rule" prints up the prompt, reads a command from the keyboard, parses it into dictionary words, deals with niceties such as UNDO or OOPS, and then runs it through the traditional I6 parser to turn it into a request for an action or list of actions. But see note below.

(b) The "generate action rule" then either sends a single action to the action-processing rules, or else runs through the list, printing the noun up with a colon each time and then sending the action to the action-processing rules. But see note below.

(c) We then run the scene changing rulebook, because people often write every turn rules which are predicated on particular scenes ("Every turn during the Grand Waltz: ..."), and such rules will fail if we haven't kept up with possible scene changes arising from something done in the action(s) just completed.

(d) The "every turn stage rule" follows the every turn rulebook. This earns its place among the "first" rules in order for it to have priority over all the other book-keeping rules at the end of a turn – including any which the user, or an extension included by the user, chooses to add to the turn sequence rules.

An unusual point here is that the "parse command rule" and the "generate action rule" are written such that they *do nothing unless the turn sequence rulebook is being followed at the top level* (by `Main`, that is). This prevents them from being used recursively, which would not work properly, and enables a popular trick from the time before the 2008 reform to keep working: we can simulate six turns going by in which the player does nothing by running "follow the turn sequence rules" six times in a row. Everything happens exactly as it should – the turn count, the time of day, timed events, and so on – except that no commands are read and no consequent actions generated.

```
A first turn sequence rule (this is the every turn stage rule): follow the every turn rules. [4th.]
A first turn sequence rule: consider the scene changing rules. [3rd.]
The generate action rule is listed first in the turn sequence rulebook. [2nd.]
The parse command rule is listed first in the turn sequence rulebook. [1st.]
```

§**27.** Three miscellaneous things then happen, all implemented by primitives in the template I6 layer:

(e) The "timed events rule" is the one which causes other rules, keyed to particular times of day, to fire.

(f) The "advance time rule" then causes the "time of day" global variable to advance by the duration of a single turn, which by default is 1 minute.

(g) The "update chronological records rule" tells the chronology machine that a slice of time has been completed. Inform can only decide past tense conditions like "the black door has been open" by continuously measuring the present to see if the black door is open now, and making a note for future reference if it is. But of course it's impossible for any computer to continuously do anything, and besides, Inform has other calls on it. What in fact happens is the Inform performs these measurements at "chronology points", which are strategic moments during play, and this is one of them.

```
The timed events rule is listed in the turn sequence rulebook.
The advance time rule is listed in the turn sequence rulebook.
The update chronological records rule is listed in the turn sequence rulebook.
```

§**28.** We now come to the rules anchored at the end, using "last". This part of the rulebook is reserved for book-keeping which has to happen positively at the end of the turn.

(h) First, we check for scene changes again. We did this only a short while ago, but scene changes might well have arisen as a result of rules which fired during the every turn rulebook, or from timed events, or in some other way, and it's important to start the next turn in the correct scene – so we check again to make sure.

(i) Then we run the "adjust light rule". Keeping track of light and darkness is quite difficult, and potentially also quite slow: it's not unlike a sort of discretised version of ray-tracing, with many light sources and barriers to think about (some transparent, some opaque). So we do this as little as possible: once per turn we calculate whether the player is in light or not, and act accordingly if so.

(j) The "note object acquisitions rule" does two things:

  (i) Gives the "handled" property to everything carried or worn by the player.

  (ii) Changes the current player's holdall in use, if necessary. (That's to say: if the player has dropped his previous player's holdall, we try to find a new one to use from his remaining possessions.)

(k) The "notify score changes rule" tells the player if the score has changed during the turn, or rather, since the last time either this rule or the startup "fix baseline scoring rule" ran. (If the score were to change in the course of an out-of-world action, it would be notified a turn late, but of course out-of-world actions are not supposed to do that sort of thing.)

```
A last turn sequence rule: consider the scene changing rules. [3rd from last.]
The adjust light rule is listed last in the turn sequence rulebook. [2nd from last.]
The note object acquisitions rule is listed last in the turn sequence rulebook. [Penultimate.]
The notify score changes rule is listed last in the turn sequence rulebook. [Last.]

This is the notify score changes rule:
    if the score is not the last notified score:
        issue score notification message;
        now the last notified score is the score;
```

§**29.** That's it, but we need to map these I7 rule names onto the names of their I6 primitives in the template layer.

```
The adjust light rule translates into I6 as "ADJUST_LIGHT_R".
The advance time rule translates into I6 as "ADVANCE_TIME_R".
The generate action rule translates into I6 as "GENERATE_ACTION_R".
The note object acquisitions rule translates into I6 as "NOTE_OBJECT_ACQUISITIONS_R".
The parse command rule translates into I6 as "PARSE_COMMAND_R".
The timed events rule translates into I6 as "TIMED_EVENTS_R".
```

§**30. Shutdown.** Goodbye is not the hardest word to say, but it does involve a little bit of work. It might not actually be goodbye, for one thing: if this rulebook ends in success, then we go back to repeating the turn sequence rulebook just as if nothing had happened.

(a) The "when play ends stage rule" follows the rulebook of the same name.

(b) The "resurrect player if asked rule" does nothing unless one of the "when play ends" rules ran the "resume the game" phrase, in which case it stops the rulebook with success (see above).

(c) The "print player's obituary rule" carries out the activity of nearly the same name (see below).

(d) The "ask the final question rule" asks the celebrated "Would you like to RESTART, RESTORE a saved game or QUIT?" question, and acts on the consequences. It can also cause an UNDO, and on a victorious ending may carry out the "amusing a victorious player" activity (see below). The rule only actually ends in the event of a QUIT: an UNDO, RESTART or RESTORE is performed at the hardware level by the virtual machine and destroys the current execution context entirely.

```
The when play ends stage rule is listed first in the shutdown rulebook.
The resurrect player if asked rule is listed last in the shutdown rulebook.
The print player's obituary rule is listed last in the shutdown rulebook.
The ask the final question rule is listed last in the shutdown rulebook.

This is the when play ends stage rule: follow the when play ends rulebook.
This is the print player's obituary rule:
    carry out the printing the player's obituary activity.

The resurrect player if asked rule translates into I6 as "RESURRECT_PLAYER_IF_ASKED_R".
The ask the final question rule translates into I6 as "ASK_FINAL_QUESTION_R".
```

§**31. Scene changing.** Scene changing is handled by a routine called `DetectSceneChange` which is compiled directly by NI: this is so primitive that it can't even be handled at the template layer. The rulebook is all a little elaborate given that it contains only one rule, but it's possible to imagine extensions which need to do book-keeping similar to scene changing and which want to make use of this opportunity.

```
The scene change machinery rule is listed last in the scene changing rulebook.

The scene change machinery rule translates into I6 as "DetectSceneChange".
```

§**32.** We couldn't do this earlier (because creating a scene automatically generates two rulebooks, and that would have thrown the rulebook numbering), so let's take this opportunity to define the "Entire Game" scene:

```
Section SR2/9 - The Entire Game scene

The Entire Game is a scene.
The Entire Game begins when the game is in progress.
The Entire Game ends when the game is over.
```

§**33. Action-processing.** Action-processing happens on two levels: an upper level, handled by the main "action-processing" rulebook, and a lower level, "specific action-processing". This division clearly complicates matters, so why do we do it? It turns out be convenient for several reasons:

(a) Out-of-world actions like "saving the game" need to run through the lower level, or they won't do anything at all, but must not run through the upper level, or in-world rules (before or instead rules, for instance) might prevent them from happening.

(b) Requested actions such as generated by a command like "CLARK, BLOW WHISTLE" have the reverse behaviour, being handled at the upper level but not the lower. (If Clark should agree, a definite non-request action "Clark blowing the whistle" is generated afresh: that one does indeed get to the lower level, but the original request action doesn't.)

(c) Specific action-processing has a rather complicated range of outcomes: it must succeed or fail, according to whether the action either reaches the carry out rules or is converted into another action which does, but also ensure that in the event of failure, the exact rule causing the failure is recorded in the "reason the action failed" variable.

(d) The specific action-processing stage is where we have to split consideration into action-specific rulebooks (like "check taking") rather than general ones (like "instead"). To get this right, we want to use some rulebook variables, and these need to be set at exactly the correct moment. It's tricky to arrange for that in the middle of a big rulebook, but easy to do so at the beginning, so we want the start of the SA-P stage to happen at the start of a rulebook.

This does mean that an attempt by the user to move the before stage to just after the check stage (say) will fail – the before and check stages happen in different rulebooks, so no amount of rearranging will do this. (Procedural rules could still achieve this very doubtfully wise effect in any case.)

§**34.** The upper level of action-processing consists of seeing whether the actor's current situation forbids the action from being tried: does anybody or anything intervene to stop it? Are there are any basic reasons of physical realism why nobody could possibly try it? Does it require somebody else to cooperate who in fact chooses not to?

Doctrinally, the before stage must see actions before anything else can happen. (It needs the absolute freedom to start fresh actions and dispose of the one originally intended, sure in the knowledge that no other rules have been there first.) So before-ish material is anchored at the "first" end of the rulebook.

The other book-end on the shelf is provided by the instead stage, which doctrinally happens only when the action is now thought to be physically reasonable. So this is anchored at the "last" end. (In fact it is followed by several more rules also anchored there, but this is essentially just the clockwork machinery showing through: they aren't realism checks.)

Miscellaneous general rules to do with physical realism are placed between the two book-ends, then, and this is where any newly created action-processing rules created by the user or by extensions will go.

```
Section SR2/10 - Action processing

The before stage rule is listed first in the action-processing rules. [3rd.]
The set pronouns from items from multiple object lists rule is listed first in the
    action-processing rules. [2nd.]
The announce items from multiple object lists rule is listed first in the
    action-processing rules. [1st.]
The basic visibility rule is listed in the action-processing rules.
The basic accessibility rule is listed in the action-processing rules.
The carrying requirements rule is listed in the action-processing rules.
The instead stage rule is listed last in the action-processing rules. [4th from last.]
The requested actions require persuasion rule is listed last in the action-processing rules.
The carry out requested actions rule is listed last in the action-processing rules.
The descend to specific action-processing rule is listed last in the action-processing rules.
The end action-processing in success rule is listed last in the action-processing rules. [Last.]
```

§**35.** As we shall see, most of these rules are primitives implemented by the template I6 layer, but five are very straightforward.

```
This is the set pronouns from items from multiple object lists rule:
    if the current item from the multiple object list is not nothing,
        set pronouns from the current item from the multiple object list.
This is the announce items from multiple object lists rule:
    if the current item from the multiple object list is not nothing,
        say "[current item from the multiple object list]: [run paragraph on]".
This is the before stage rule: abide by the before rules.
This is the instead stage rule: abide by the instead rules.
```

§**36.** The final rule in the rulebook always succeeds: this ensures that action-processing always makes a decision. (I7's doctrine is that an action "succeeds" if and only if its carry-out stage is reached, so any action getting right to the end of this rulebook must have succeeded.)

```
This is the end action-processing in success rule: rule succeeds.
```

§**37.** The action-processing rulebook contains six primitives:

(1) The "basic visibility rule" checks the action to see if it requires light, and if so, and if the actor is the player and in darkness, asks the visibility rules (see below) whether the lack of light should stop the action. (It would be cleaner to apply this rule to all actors, but we would need much more extensive light calculations to do this.)

(2) The "basic accessibility rule" checks the action to see if it requires the noun to be touchable, and if so, asks the accessibility rulebook to adjudicate (see below); then repeats the process for the second noun.

(3) The "carrying requirements rule" checks the action to see if it requires the noun to be carried. If so, but the noun is not carried, it generates an implicit taking action (in effect, "try silently taking N"); and if that fails, then the rulebook is halted in failure. The process is then repeated for the second noun.

(4) If the action is one where the player requests somebody else to do something, the "requested actions require persuasion rule" asks the persuasion rulebook for permission.

(5) If the action is one where the player requests somebody else to do something, the "carry out requested actions rule" starts a new action by the person asked, and looks at the result: if it failed, the "unsuccessful attempt by" rulebook is run to tell the player what has (not) happened. (If that does nothing, a bland message is printed.) In all cases, the original action of requesting is ended in success: a success because, whatever happened, the request succeeded in making the actor try to do something.

(6) The "descend to specific action-processing rule" really only runs the specific action-processing rulebook, but it's implemented as a primitive in the template I6 layer because it must also find out which are the specific check, carry out and report rulebooks for the current action (for instance, "check taking" is the specific check rulebook for the "taking" action – which seems obvious from the names: but at run-time, the names aren't so visible).

```
The basic accessibility rule translates into I6 as "BASIC_ACCESSIBILITY_R".
The basic visibility rule translates into I6 as "BASIC_VISIBILITY_R".
The carrying requirements rule translates into I6 as "CARRYING_REQUIREMENTS_R".
The requested actions require persuasion rule translates into I6 as "REQUESTED_ACTIONS_REQUIRE_R".
The carry out requested actions rule translates into I6 as "CARRY_OUT_REQUESTED_ACTIONS_R".
The descend to specific action-processing rule translates into I6 as
"DESCEND_TO_SPECIFIC_ACTION_R".
```

§**38. Specific action-processing.** And now we descend to the lower level, which is much easier to understand.

```
The work out details of specific action rule is listed first in the specific
action-processing rules.

A specific action-processing rule
    (this is the investigate player's awareness before action rule):
    consider the player's action awareness rules;
    if rule succeeded, change within the player's sight to true;
    otherwise change within the player's sight to false.

A specific action-processing rule (this is the check stage rule):
    anonymously abide by the specific check rulebook.

A specific action-processing rule (this is the carry out stage rule):
    consider the specific carry out rulebook.

A specific action-processing rule (this is the after stage rule):
    if action in world is true, abide by the after rules.

A specific action-processing rule
    (this is the investigate player's awareness after action rule):
    if within the player's sight is false:
        consider the player's action awareness rules;
        if rule succeeded, change within the player's sight to true;

A specific action-processing rule (this is the report stage rule):
    if within the player's sight is true and action keeping silent is false,
        consider the specific report rulebook;

The last specific action-processing rule: rule succeeds.
```

§**39.** The unusual use of "anonymously abide by" above is a form of "abide by" which may be worth explaining. Suppose rule X consists of an instruction to abide by rulebook Y, and suppose that Y in fact fails when it reaches Y_k. If the ordinary "abide by" is used then the action will be deemed to have failed at rule X, but if "anonymously abide by" is used then it will be deemed to have failed at Y_k. (Thus X remains anonymous: it can never be the culprit.) We only use this at the check stage, because the carry out, report and after stages are not allowed to fail the action. (The after stage is allowed to end it, but not in failure.)

§**40.** The specific action-processing rulebook is probably more fruitful than the main one if we want to modify what happens. For instance:

```
This is the sixth sense rule: if the player is not the actor, say "You sense that [the actor] is up to
something."

The sixth sense rule is listed before the carry out stage rule in the specific action-processing rules.
```

...produces the message at a time when the action is definitely possible and will succeed, but before anything has been done.

§**41.** The only rule not spelled out was the primitive "work out details of specific action rule", which initialises the rulebook variables so that they record the action's specific check, carry out and report rulebooks, and whether or not it is in world.

```
The work out details of specific action rule translates into I6 as
"WORK_OUT_DETAILS_OF_SPECIFIC_R".
```

§**42. Player's action awareness.** This rulebook decides whether or not an action by somebody should be routinely reported to the player: is he aware of it having taken place? If the rulebook positively succeeds then he is, and otherwise not.

```
A player's action awareness rule
    (this is the player aware of his own actions rule):
    if the player is the actor, rule succeeds.
A player's action awareness rule
    (this is the player aware of actions by visible actors rule):
    if the player is not the actor and the player can see the actor, rule succeeds.
A player's action awareness rule
    (this is the player aware of actions on visible nouns rule):
    if the noun is a thing and the player can see the noun, rule succeeds.
A player's action awareness rule
    (this is the player aware of actions on visible second nouns rule):
    if the second noun is a thing and the player can see the second noun, rule succeeds.
```

§**43. Accessibility.** The "accessibility" rulebook is not very visible to users: it's another behind-the-scenes rulebook for managing the decision as to whether the actor can touch any items which the intended action requires him to be able to reach.

In its default configuration, it contains only the "access through barriers" rule. This in all circumstances either succeeds or fails: in other words, it makes a definite decision, and this is why it is anchored as "last" in the rulebook. If users or extensions want to tweak accessibility at this general level, any new rules they add will get the chance to decide before the "access through barriers" rule settles the matter. But in practice we expect most users to work with one of the two reaching rulebooks instead.

```
Section SR2/11 - Accessibility

The access through barriers rule is listed last in the accessibility rules.

The access through barriers rule translates into I6 as "ACCESS_THROUGH_BARRIERS_R".
```

§**44. Reaching inside.** What the access through barriers rule does is to try to construct a path through the object containment tree from the actor to the item needed, and to apply the reaching inside or reaching outside rulebooks each time this path passes inside or outside of a container (or room). (Supporters never form barriers.)

```
The can't reach inside rooms rule is listed last in the reaching inside rules. [Penultimate.]
The can't reach inside closed containers rule is listed last in the reaching
inside rules. [Last.]

The can't reach inside closed containers rule translates into I6 as "CANT_REACH_INSIDE_CLOSED_R".
The can't reach inside rooms rule translates into I6 as "CANT_REACH_INSIDE_ROOMS_R".
```

§**45. Reaching outside.** And, not quite symmetrically since we don't need to block room-to-room reaching on both the inbound and outbound directions,

```
The can't reach outside closed containers rule is listed last in the reaching outside rules.

The can't reach outside closed containers rule translates into I6 as "CANT_REACH_OUTSIDE_CLOSED_R".
```

§**46. Visibility.** The visibility rulebook actually has the opposite sense to the one the name might suggest. It is applied only when the player (not any other actor) tries to perform an action which requires light to see by; the action is blocked if the rulebook succeeds. (Well, but we could hardly call it the invisibility rulebook. The name is supposed to suggest that the consideration of visibility is being applied: rather the way cricket matches end with a declaration that "light stopped play", meaning of course that darkness did.)

```
The can't act in the dark rule is listed last in the visibility rules.
The last visibility rule (this is the can't act in the dark rule): if in darkness, rule succeeds.
```

§**47. Does the player mean.** This rulebook is akin to a set of preferences about how to interpret the player's commands in cases of ambiguity. No two Inform users are likely to agree about the best way to decide these, so we are fairly hands-off, and make only one rule as standard:

```
Does the player mean taking something which is carried by the player
    (this is the very unlikely to mean taking what's already carried rule):
    it is very unlikely.
```

§**48.** And that completes the creation and stocking of the 25 rulebooks. More than half of them are initially empty, including before, instead and after – at the end of the day, these rulebooks are hooks allowing the user to change the ordinary behaviour of things, but ordinariness is exactly what the Standard Rules is all about.

§**49. Adjectives applied to values.** There is a small stock of built-in adjectives for values.

```
Definition: a number is even rather than odd if the remainder after dividing it by 2 is 0.
Definition: a number is positive if it is greater than zero.
Definition: a number is negative if it is less than zero.
Definition: a text is empty rather than non-empty if it is "".
Definition: an indexed text is empty rather than non-empty if I6 routine
    "INDEXED_TEXT_TY_Empty" says so (it contains no characters).
A scene can be recurring or non-recurring. A scene is usually non-recurring.
Definition: a scene is happening if I6 condition "scene_status-->(*1-1)==1"
    says so (it is currently taking place).
Definition: a table-name is empty rather than non-empty if the number of filled rows in it is 0.
Definition: a table-name is full rather than non-full if the number of blank rows in it is 0.
Definition: a rulebook is empty rather than non-empty if I6 routine "RulebookEmpty" says so (it
    contains no rules, so that following it does nothing and makes no decision).
Definition: an activity is empty rather than non-empty if I6 routine "ActivityEmpty" says so (its
    before, for and after rulebooks are all empty).
Definition: an activity is going on if I6 routine "TestActivity" says so (one
    of its three rulebooks is currently being run).
Definition: a list of values is empty rather than non-empty if I6 routine
    "LIST_OF_TY_Empty" says so (it contains no entries).
```

SR3 - Activities A/sr3

Purpose

The built-in activities and their default stock of rules; the locale description mechanism.

A/sr3.§20 Locale activities; §21 Locale Implementation; §22 Printing the Locale Description; §23 Choosing Notable Locale Objects; §24 Printing a Locale Paragraph

---

§**1.** These must not be created until the basic rulebooks are in place, because creating any activity automatically creates three rulebooks as well.

Once again, there are no activities or assumptions about them built into NI, but the template I6 layer expects the following 26 activities to be created and in this order. (That is, the order here must exactly match that of the `*_ACT` constant definitions made in `Definitions.i6t`.) The activities are fairly completely described in the Inform documentation, so the only notes here concern implementation.

§**2.** We start with activities used in describing things and rooms.

Most of the activities begin with all three rulebooks empty, since by default they do not intervene. This one, however, has a "last" for-rule, because it's required to do something definite: it must actually print a name, and the last for-rule is the default way to do this. (The assumption is that any other for-rule added by the user will halt the rulebook before the default implementation is reached.) We also include a before rule which marks any item whose name is being printed with the "mentioned" property, for reasons to be found below.

```
Part SR3 - Activities

Printing the name of something (documented at act_pn) is an activity. [0]

Before printing the name of a thing (called the item being printed)
    (this is the make named things mentioned rule):
    now the item being printed is mentioned.

The standard name printing rule is listed last in the for printing the name rulebook.
The standard name printing rule translates into I6 as "STANDARD_NAME_PRINTING_R".

Printing the plural name of something (documented at act_ppn) is an activity. [1]

Rule for printing the plural name of something (called the item) (this is the standard
    printing the plural name rule):
    say the printed plural name of the item.
The standard printing the plural name rule is listed last in the for printing
the plural name rulebook.

Printing a number of something (documented at act_pan) is an activity. [2]

Rule for printing a number of something (called the item) (this is the standard
    printing a number of something rule):
    say "[listing group size in words] ";
    carry out the printing the plural name activity with the item.
The standard printing a number of something rule is listed last in the for printing
a number rulebook.
```

§**3.** When they occur in room descriptions, names of things are sometimes supplemented by details:

```
Printing room description details of something (documented at act_details) is an activity. [3]
```

§**4.** Names of things are often formed up into lists, in which they are sometimes grouped together:

```
Listing contents of something (documented at act_lc) is an activity. [4]
The standard contents listing rule is listed last in the for listing contents rulebook.
The standard contents listing rule translates into I6 as "STANDARD_CONTENTS_LISTING_R".
Grouping together something (documented at act_gt) is an activity. [5]
```

§**5.** And such lists of names are formed up in turn into room descriptions. Something which is visible in a room can either have a paragraph of its own or can be relegated to the list of "nondescript" items at the end.

```
Writing a paragraph about something (documented at act_wpa) is an activity. [6]
```

§**6.** When these paragraphs have all gone by, the nondescript items left over are more briefly listed: the following activity gets the chance to change how this is done.

```
Listing nondescript items of something (documented at act_lni) is an activity. [7]
```

§**7.** Darkness behaves, for room description purposes, a little as if it were a room in its own right. Until the 1990s that was almost always how darkness was implemented in IF programs: this persists in I6, but not I7, where the existence of a room-which-is-not-a-room would break type safety.

```
Printing the name of a dark room (documented at act_darkname) is an activity. [8]
Printing the description of a dark room (documented at act_darkdesc) is an activity. [9]
Printing the announcement of darkness (documented at act_nowdark) is an activity. [10]
Printing the announcement of light (documented at act_nowlight) is an activity. [11]
Printing a refusal to act in the dark (documented at act_toodark) is an activity. [12]

The look around once light available rule is listed last in for printing the
announcement of light.

This is the look around once light available rule:
    try looking.
```

§**8.** Two special forms of printing: the status line at the top of the screen, refreshed every turn during play, and the banner which appears at or close to the start of play:

```
Constructing the status line (documented at act_csl) is an activity. [13]
Printing the banner text (documented at act_banner) is an activity. [14]
```

§**9.** Now a brace of activities to intervene in how the Inform parser does its parsing, arranged roughly in chronological order of their typical use during a single turn of a typed command, its parsing, and final conversion into an action.

The unusual notation "(future action)" here allows NI to parse rule preambles for these activities in a way which would refer to the action which might, at some point in the future, be generated – during parsing we don't of course yet know what that action is, but there is always a current guess at what it might be.

```
Reading a command (documented at act_reading) is an activity. [15]
Deciding the scope of something (future action) (documented at act_ds) is an activity. [16]
Deciding the concealed possessions of something (documented at act_con) is an activity. [17]
Deciding whether all includes something (future action) (documented at act_all)
    is an activity. [18]
The for deciding whether all includes rules have outcomes it does not (failure) and
    it does (success).
Clarifying the parser's choice of something (future action) (documented at act_clarify)
    is an activity. [19]
Asking which do you mean (future action) (documented at act_which) is an activity. [20]
Printing a parser error (documented at act_parsererror) is an activity. [21]
Supplying a missing noun (documented at act_smn) is an activity. [22]
Supplying a missing second noun (documented at act_smn) is an activity. [23]
Implicitly taking something (documented at act_implicitly) is an activity. [24]
```

§**10.** The supplying activities are linguistically interesting, for reasons gone into in the paper *Interactive Fiction, Natural Language and Semantic Analysis*: English verbs do not naturally seem to feature optional nouns. Indeed, we say "it rained on Tuesday" where "it" refers to nothing at all, merely because we can't bring ourselves to leave a gap and say just "rained on Tuesday". A better example here would be "it sounded like rain", because we do the same to convey the idea of listening ambiently rather than to any single thing: listening appears to be rare among actions in that it can equally well take a noun as not. Just as English handles this problem by supplying a spurious "it" which appears to mean "the world at large", so Inform handles it by supplying the current location, with the same idea in mind. And the same applies to the sense of smell, which can be similarly defocused.

```
Rule for supplying a missing noun while an actor smelling (this is the ambient odour rule):
    change the noun to the location.
Rule for supplying a missing noun while an actor listening (this is the ambient sound rule):
    change the noun to the location.
```

§**11.** The following rule is something of a dodge to provide a better parser response to commands like GO or BRETT, GO. (Putting the rule here, and giving it a name, allows the user to override it and thus accept the idea of vague going after all.)

```
Rule for supplying a missing noun while an actor going (this is the block vaguely going rule):
    issue library message going action number 7.
```

§**12.** The very first rules: those invoked at the earliest possible moment when the virtual machine is starting up. We need this hook, really, for the Glulx VM, which requires various styles to be created.

```
Starting the virtual machine (documented at act_startvm) is an activity. [25]
The enable Glulx acceleration rule is listed first in for starting the virtual machine.
The enable Glulx acceleration rule translates into I6 as "ENABLE_GLULX_ACCEL_R".
```

§**13.** And the very last rules of all. The obituary is a rare example of a sequence of events in the I6 library having been rolled up into an activity, partly because it's one of the few clear-cut moments where several unconnected things happen in succession.

```
Amusing a victorious player (documented at act_amuse) is an activity. [26]

Printing the player's obituary (documented at act_obit) is an activity. [27]
The print obituary headline rule is listed last in for printing the player's obituary.
The print final score rule is listed last in for printing the player's obituary.
The display final status line rule is listed last in for printing the player's obituary.

The print obituary headline rule translates into I6 as "PRINT_OBITUARY_HEADLINE_R".
The print final score rule translates into I6 as "PRINT_FINAL_SCORE_R".
The display final status line rule translates into I6 as "DISPLAY_FINAL_STATUS_LINE_R".
```

§**14.** There is one last question: the one which usually reads "Would you like to RESTART, RESTORE a saved game, or QUIT?", but which sometimes provides other options too. The "ask the final question rule" handles this, and does so by repeatedly calling the following activity:

```
Handling the final question is an activity. [28]
```

§**15.** It follows that this activity *must* at least sometimes do something dramatic to the execution state: perform a quit, for instance. Four primitive rules are available for the drastic things which the activity might wish to do, but these are not placed in any rulebook: instead they are available for anyone who wants to call them. (In the default implementation below, we put references to them into a table.)

```
The immediately restart the VM rule translates into I6 as "IMMEDIATELY_RESTART_VM_R".
The immediately restore saved game rule translates into I6 as "IMMEDIATELY_RESTORE_SAVED_R".
The immediately quit rule translates into I6 as "IMMEDIATELY_QUIT_R".
The immediately undo rule translates into I6 as "IMMEDIATELY_UNDO_R".
```

§**16.** We structure the activity so that the printing of the question and typing of the answer take place at the "before" stage, and then the parsing and acting upon this answer take place at the "for" stage. Reading the keyboard is the last rule in "before". With the "for" stage, the idea is that any extra rule slipped in by the user can take precedence over the default implementation, so the latter is the last there, too.

```
The print the final question rule is listed in before handling the final question.
The print the final prompt rule is listed in before handling the final question.
The read the final answer rule is listed last in before handling the final question.
The standard respond to final question rule is listed last in for handling the final question.

This is the print the final prompt rule: say "> [run paragraph on]".

The read the final answer rule translates into I6 as "READ_FINAL_ANSWER_R".
```

§**17.** That clears away the underbrush and reduces us to two matching tasks: (i) to print the question, (ii) to parse the answer, given that we want to be able to vary the set of choices available.

We do this by reading the options from the Table of Final Question Options. (See below for its default contents.) Each row is an option, whose wording must be placed in the topic column. The final question wording entry can either be text describing the option – e.g., "perform a RESTART" – or can be left blank, making the option a secret one, omitted from the question but still recognised as an answer. The only if victorious entry can be set to make the option available only after a victorious ending, not after a loss; Infocom's traditional AMUSING option behaved thus. Finally, the table specifies what to do if the option is taken: either it provides a rule, or an activity to carry out. (If it provides only an activity, but that activity is empty, then the option is omitted from the question and not recognised as an answer.)

```
This is the print the final question rule:
    let named options count be 0;
    repeat through the Table of Final Question Options:
        if the only if victorious entry is false or the game ended in victory:
            if there is a final response rule entry
                or the final response activity entry [activity] is not empty:
                if there is a final question wording entry, increase named options count by 1;
    if the named options count is less than 1, abide by the immediately quit rule;
    say "Would you like to ";
    repeat through the Table of Final Question Options:
        if the only if victorious entry is false or the game ended in victory:
            if there is a final response rule entry
                or the final response activity entry [activity] is not empty:
                if there is a final question wording entry:
                    say final question wording entry;
                    decrease named options count by 1;
                    if the named options count is 0:
                        say "?[line break]";
                    otherwise if the named options count is 1:
                        if using the serial comma option, say ",";
                        say " or ";
                    otherwise:
                        say ", ";
```

§**18.** And the matching rule to parse and respond to the answer:

```
This is the standard respond to final question rule:
    repeat through the Table of Final Question Options:
        if the only if victorious entry is false or the game ended in victory:
            if there is a final response rule entry
                or the final response activity entry [activity] is not empty:
                if the player's command matches the topic entry:
                    if there is a final response rule entry, abide by final response rule entry;
                    otherwise carry out the final response activity entry activity;
                    rule succeeds;
    issue miscellaneous library message number 8.
```

§**19.** The table of final options is the only material under the heading "Section SR/Q", to make it easy for users to replace with entirely different tables.

These settings are the traditional ones used by Inform since 1995 or so. The UNDO option has customarily been a "secret", though not much of one, since it somewhat cheapens the announcement of a calamity to be immediately offered the chance to reverse it: death, where is thy sting?

```
Section SR2/12 - Final question options

Table of Final Question Options
final question wording only if victorious topic final response rule final response activity
"RESTART" false "restart" immediately restart the VM rule --
"RESTORE a saved game" false "restore" immediately restore saved game rule --
"see some suggestions for AMUSING things to do" true "amusing" -- amusing a victorious player
"QUIT" false "quit" immediately quit rule --
-- false "undo" immediately undo rule --
```

§**20. Locale activities.** A "locale description" is a segment of the text produced by LOOK: the "locale" is a clutch of objects at a given level in the object tree. Most room descriptions consist of a top line, a description of the place, and then a single (though often, as here, multi-paragraph) locale:

```
Sentier Le Corbusier
A coastal walk along the rocky shore between Nice and Menton.
```
now the locale for the room Sentier Le Corbusier:
```
A translucent jellyfish has been washed up by the waves.

You can also see a bucket and a spade here.
```

A locale typically contains a run of paragraphs specific to interesting items, especially those not yet picked up, followed by a paragraph which lists the "nondescript" items – those not given paragraphs of their own, such as the bucket and spade. (Some items, though – typically scenery, but also for instance the player – are not even nondescript and do not appear at all.) A locale can contain no interesting paragraphs, or no list of nondescript items, or can even contain neither: that is, it can be entirely empty.

When the player is in or on top of something, multiple locales are described:

```
Sentier Le Corbusier (in the golf cart)
A coastal walk along the rocky shore between Nice and Menton.
```
now the locale for the room Sentier Le Corbusier:
```
A translucent jellyfish has been washed up by the waves.

You can also see a bucket and a spade here.
```
now the locale for the golf cart:
```
In the golf cart you can see a map of Villefranche-sur-Mer.
```

To sum up, the text produced by LOOK consists of a header (produced by the carry out looking rules) followed by one or more locale descriptions (produced by the activity below).

§**21. Locale Implementation.** When describing a locale, we keep a Table of interesting objects, each associated with a priority – a number indicating how important, and therefore how near to the top of the description, the object is. A special syntax allows us to create the Table with exactly the same number of rows as there are things in the model world: thus, in the worst case where all things are in a single locale, we still will not run out of table rows. (We do this rather than creating a large but fixed-size table because memory is very short in some Z-machine I7 works, so we want to take only what we might actually need. The table structure is not as wasteful as it might look: an experiment with using a number property of things instead showed that this table was actually more efficient, because of the property numbering overhead in the Z-machine memory representation of objects.)

```
Section SR2/13 - Locale descriptions - Unindexed

Table of Locale Priorities
notable-object (an object) locale description priority (a number)
-- --
with blank rows for each thing.

To describe locale for (O - object):
    carry out the printing the locale description activity with O.

To set the/-- locale priority of (O - an object) to (N - a number):
    if O is a thing:
        if N <= 0, now O is mentioned;
        if there is a notable-object of O in the Table of Locale Priorities:
            choose row with a notable-object of O in the Table of Locale Priorities;
            if N <= 0, blank out the whole row;
            otherwise change the locale description priority entry to N;
        otherwise:
```

```
if N is greater than 0:
    choose a blank row in the Table of Locale Priorities;
    change the notable-object entry to 0;
    change the locale description priority entry to N;
```

§**22. Printing the Locale Description.** This is handled by the "printing the locale description" activity. The before stage works out which objects might be of interest; the for stage actually prints paragraphs; the after stage is initially empty, but can be used to insert all kinds of interesting information into a room description.

We count the paragraphs printed in a global variable, not an activity variable, since it needs to be consulted in sub-activities whose rules are outside what would be its scope; that doesn't matter, though, since locale descriptions are not nested. (If they were, the above table would fail in any case.)

(1) Disaster would ensue if the user tampered with the "initialise locale description rule", but nobody is likely to do this other than intentionally.
(2) The "find notable locale objects rule" in fact only runs a further activity, the "choosing notable locale objects" activity. The task here is to identify the objects which might by virtue of their location appear in the locale, and to assign each of them a priority number.
(3) The "interesting locale paragraphs rule" goes through all of the notable objects chosen at stage (2), in order of priority, and offers each to yet another activity: the "printing a locale paragraph about" activity. This can either print a paragraph related to the item in question, or demote it as being not even nondescript (by changing its priority to 0). The default is to do nothing, in which case the item becomes nondescript.
(4) The "you-can-also-see rule" prints what is, ordinarily, the final paragraph of the locale description, listing the nondescript items. It goes to some trouble to find out whether these all have a common object tree parent, listing them with "list contents of" if they do: this is so that people who have written rules such as "Rule for printing the name of the blur while listing contents..." will take effect, because the "listing contents" activity will be going on. Provided that the notable objects chosen in (2) are all children of the locale domain, this will always happen. If the user should add rules to make quite different objects also notable, then the "you-can-also-see rule" has to resort to listing in a way which doesn't use the "listing contents" activity – since the list is not in fact a list of the contents of anything.

```
Printing the locale description of something (documented at act_pld) is an activity.

The locale paragraph count is a number that varies.

Before printing the locale description (this is the initialise locale description rule):
    now the locale paragraph count is 0;
    repeat with item running through things:
        now the item is not mentioned;
    repeat through the Table of Locale Priorities:
        blank out the whole row.

Before printing the locale description (this is the find notable locale objects rule):
    let the domain be the parameter-object;
    carry out the choosing notable locale objects activity with the domain;
    continue the activity.

For printing the locale description (this is the interesting locale paragraphs rule):
    let the domain be the parameter-object;
    sort the Table of Locale Priorities in locale description priority order;
    repeat through the Table of Locale Priorities:
        [say "[notable-object entry]...";]
        carry out the printing a locale paragraph about activity with the notable-object entry;
    continue the activity.

For printing the locale description (this is the you-can-also-see rule):
```

```
let the domain be the parameter-object;
let the mentionable count be 0;
repeat with item running through things:
    now the item is not marked for listing;
repeat through the Table of Locale Priorities:
    [say "[notable-object entry] - [locale description priority entry].";]
    if the locale description priority entry is greater than 0,
        now the notable-object entry is marked for listing;
    increase the mentionable count by 1;
if the mentionable count is greater than 0:
    repeat with item running through things:
        if the item is mentioned:
            now the item is not marked for listing;
    begin the listing nondescript items activity;
    if the number of marked for listing things is 0:
        abandon the listing nondescript items activity;
    otherwise:
        if handling the listing nondescript items activity:
            if the domain is a room:
                if the domain is the location, say "You ";
                otherwise say "In [the domain] you ";
            otherwise if the domain is a supporter:
                say "On [the domain] you ";
            otherwise if the domain is an animal:
                say "On [the domain] you ";
            otherwise:
                say "In [the domain] you ";
            say "can [if the locale paragraph count is greater than 0]also [end if]see ";
            let the common holder be nothing;
            let contents form of list be true;
            repeat with list item running through marked for listing things:
                if the holder of the list item is not the common holder:
                    if the common holder is nothing,
                        now the common holder is the holder of the list item;
                    otherwise now contents form of list is false;
                if the list item is mentioned, now the list item is not marked for listing;
            filter list recursion to unmentioned things;
            if contents form of list is true and the common holder is not nothing,
                list the contents of the common holder, as a sentence, including contents,
                    giving brief inventory information, tersely, not listing
                    concealed items, listing marked items only;
            otherwise say "[a list of marked for listing things including contents]";
            if the domain is the location, say " here";
            say ".[paragraph break]";
            unfilter list recursion;
        end the listing nondescript items activity;
continue the activity.
```

§**23. Choosing Notable Locale Objects.** By default, the notable objects are exactly the children of the domain, and they all have equal priority (1). Since table sorting is stable, and thus preserves the row order of rows with equal priority, the eventual order of listing is by default the same as the order in which things are added to the table, which in turn is the object-tree traversal order.

```
Choosing notable locale objects of something (documented at act_cnlo) is an activity.

For choosing notable locale objects (this is the standard notable locale objects rule):
    let the domain be the parameter-object;
    let the held item be the first thing held by the domain;
    while the held item is a thing:
        set the locale priority of the held item to 5;
        now the held item is the next thing held after the held item;
    continue the activity.
```

§**24. Printing a Locale Paragraph.** By default there are four kinds of "interesting" locale paragraph, and the following setup is fairly complicated because it implements conventions gradually built up between 1978 and 2008.

To recap, this activity is run on each notable thing in turn, in priority order. (It is only run on notable things for efficiency reasons.)

The basic principle is that, at every stage, we should consider an item only if it is not "mentioned" already. This will happen if it has been named by a previous paragraph, but also if it has been explicitly marked as such to get rid of it. In considering an item, we have three basic options:

(a) Print a paragraph about the item and mark it as mentioned – this is good for interesting items deserving a paragraph of their own.
(b) Print a paragraph, but do not mark it as mentioned – this is only likely to be useful if we want to print information related to the item without mentioning the thing itself. (For instance, if the presence of a mysterious parcel resulted in a ticking noise, we could print a paragraph about the ticking noise without mentioning the parcel, which would then appear later.)
(c) Mark the item as mentioned but print nothing – this gets rid of the item, ensuring that it will not appear in the final "you can also see" sentence, and will not be considered by subsequent rules.
(d) Do nothing at all – the item then becomes "nondescript" and appears in the final "you can also see" sentence, unless somebody else mentions it in the mean time.

Briefly, then, the following is the standard method:

(1) The "don't mention player's supporter in room descriptions rule" excludes anything the player is directly or indirectly standing on or, less frequently, in. The header of the room description has probably already said something like "Boudoir (on the four-poster bed)", so the player can't be unaware of this item.
(2) The "don't mention scenery in room descriptions rule" excludes scenery.
(3) The "don't mention undescribed items in room descriptions rule" excludes the player object. (It's redundant to say "You can also see yourself here.") At present nothing else in I7 is "undescribed" in this sense.
(4) The "set pronouns from items in room descriptions rule" adjusts the meaning of pronouns like IT and HER to pick up items mentioned. Thus if a room description ends "Mme Tourmalet glares at you.", then HER would be adjusted to mean Mme Tourmalet.
(5) The "offer items to writing a paragraph about rule" gives the "printing a paragraph about" activity a chance to intervene. We detect whether it does intervene or not by looking to see if it has printed any text.
(6) The "use initial appearance in room descriptions rule" uses the initial appearance property of an object which has never been handled as a paragraph.
(7) The "describe what's on scenery supporters in room descriptions rule" is a somewhat controversial feature: whereas the rest of Inform's room description conventions are generally consensus, this one is much disliked by some users for its occasional inappropriateness. It prints text such as "On the

mantelpiece is a piece of chalk." for items which, like the mantelpiece, are scenery mentioned (we assume) in the main room description. (It is assumed that scenery supporters make their contents more prominently visible than scenery containers, which we do not announce the contents of.) The ability to modify, replace or abolish this rule was one of the main motivations to break room description up into activities in March 2008.

```
Printing a locale paragraph about something (documented at act_plp) is an activity.

For printing a locale paragraph about a thing (called the item)
    (this is the don't mention player's supporter in room descriptions rule):
    if the item encloses the player, set the locale priority of the item to 0;
    continue the activity.

For printing a locale paragraph about a thing (called the item)
    (this is the don't mention scenery in room descriptions rule):
    if the item is scenery, set the locale priority of the item to 0;
    continue the activity.

For printing a locale paragraph about a thing (called the item)
    (this is the don't mention undescribed items in room descriptions rule):
    if the item is undescribed, set the locale priority of the item to 0;
    continue the activity.

For printing a locale paragraph about a thing (called the item)
    (this is the set pronouns from items in room descriptions rule):
    if the item is not mentioned, set pronouns from the item;
    continue the activity.

For printing a locale paragraph about a thing (called the item)
    (this is the offer items to writing a paragraph about rule):
    if the item is not mentioned:
        if a paragraph break is pending, say "[conditional paragraph break]";
        carry out the writing a paragraph about activity with the item;
        if a paragraph break is pending:
            increase the locale paragraph count by 1;
            now the item is mentioned;
            say "[command clarification break]";
    continue the activity.

For printing a locale paragraph about a thing (called the item)
    (this is the use initial appearance in room descriptions rule):
    if the item is not mentioned:
        if the item provides the property initial appearance and the
            item is not handled:
            increase the locale paragraph count by 1;
            say "[initial appearance of the item]";
            say "[paragraph break]";
            if a locale-supportable thing is on the item:
                repeat with possibility running through things on the item:
                    now the possibility is marked for listing;
                    if the possibility is mentioned:
                        now the possibility is not marked for listing;
                say "On [the item] ";
                list the contents of the item, as a sentence, including contents,
                    giving brief inventory information, tersely, not listing
                    concealed items, prefacing with is/are, listing marked items only;
                say ".[paragraph break]";
            now the item is mentioned;
```

```
    continue the activity.

Definition: a thing (called the item) is locale-supportable if the item is not
scenery and the item is not mentioned and the item is not undescribed.

For printing a locale paragraph about a thing (called the item)
    (this is the describe what's on scenery supporters in room descriptions rule):
    if the item is not undescribed and the item is scenery and
        the item does not enclose the player:
        set pronouns from the item;
        if a locale-supportable thing is on the item:
            repeat with possibility running through things on the item:
                now the possibility is marked for listing;
                if the possibility is mentioned:
                    now the possibility is not marked for listing;
            increase the locale paragraph count by 1;
            say "On [the item] ";
            list the contents of the item, as a sentence, including contents,
                giving brief inventory information, tersely, not listing
                concealed items, prefacing with is/are, listing marked items only;
            say ".[paragraph break]";
    continue the activity.
```

SR4 - Actions A/sr4

Purpose

The standard stock of actions supplied with Inform, along with the rules which define them; and the Understand grammar which corresponds to them.

---

§**1.** NI comes with no actions built in, and only one (perhaps unexpected) assumption: that the 8th action defined is "going".

The order, and the subheadings, here are responsible for the order and subheadings used in the Actions page of the Index.

```
Part SR4 - Actions

Section SR4/1 - Generic action patterns

Section SR4/2 - Standard actions concerning the actor's possessions
```

§2. Taking inventory.

```
Taking inventory is an action with past participle taken, applying to nothing.
The taking inventory action translates into I6 as "Inv".

The specification of the taking inventory action is "Taking an inventory of
one's immediate possessions: the things being carried, either directly or in
any containers being carried. When the player performs this action, either
the inventory listing, or else a special message if nothing is being carried
or worn, is printed during the carry out rules: nothing happens at the report
stage. The opposite happens for other people performing the action: nothing
happens during carry out, but a report such as 'Mr X looks through his
possessions.' is produced (provided Mr X is visible).

The exotic 'use inventory to set pronouns rule' allows the inventory
listing to change the current meanings of IT, HIM, HER, and so on for the
player's future commands. Thus if the player is carrying a certain female
cat, say, taking inventory might print up the name 'Missee Lee', and this
would set HER to refer to Missee Lee. Some IF authors dislike this
convention: it can be abolished by writing 'The use inventory to set
pronouns rule is not listed in any rulebook.'"
```

§**3.** Missee Lee is a black and white cat living in North Oxford, named for a Cambridge-educated pirate queen in the South China seas who is the heroine – or villainess – of the tenth in Arthur Ransome's Swallows and Amazons series of children's books, *Missee Lee* (1941). Since you ask.

§**4.** Carry out.

```
Carry out taking inventory (this is the print empty inventory rule):
    if the first thing held by the player is nothing, stop the action with
        library message taking inventory action number 1.

Carry out taking inventory (this is the print standard inventory rule):
    issue library message taking inventory action number 2;
    say ":[line break]";
    list the contents of the player, with newlines, indented, including contents,
        giving inventory information, with extra indentation.

Carry out taking inventory (this is the use inventory to set pronouns rule):
    set pronouns from possessions of the player.
```

§**5.** Report.

```
Report an actor taking inventory (this is the report other people taking
    inventory rule):
    if the actor is not the player,
        issue actor-based library message taking inventory action number 5 for the actor.
```

§6. Taking.

```
Taking is an action with past participle taken, applying to one thing.
The taking action translates into I6 as "Take".

The specification of the taking action is "The taking action is the only way
an action in the Standard Rules can cause something to be carried by an actor.
It is very simple in operation (the entire carry out stage consists only of
'now the actor carries the noun') but many checks must be performed before it
can be allowed to happen."
```

§7. Check.

```
Check an actor taking (this is the can't take yourself rule):
    if the actor is the noun, stop the action with library message taking
        action number 2 for the noun.

Check an actor taking (this is the can't take other people rule):
    if the noun is a person, stop the action with library message taking
        action number 3 for the noun.

Check an actor taking (this is the can't take component parts rule):
    if the noun is part of something (called the whole), stop the action
        with library message taking action number 7 for the whole.

Check an actor taking (this is the can't take people's possessions rule):
    let the local ceiling be the common ancestor of the actor with the noun;
    let H be the not-counting-parts holder of the noun;
    while H is not nothing and H is not the local ceiling:
        if H is a person, stop the action with library message taking action
            number 6 for H;
        let H be the not-counting-parts holder of H;

Check an actor taking (this is the can't take items out of play rule):
    let H be the noun;
    while H is not nothing and H is not a room:
        let H be the not-counting-parts holder of H;
    if H is nothing, stop the action with library message taking action
        number 8 for the noun.

Check an actor taking (this is the can't take what you're inside rule):
    let the local ceiling be the common ancestor of the actor with the noun;
    if the local ceiling is the noun, stop the action with library message
        taking action number 4 for the noun.

Check an actor taking (this is the can't take what's already taken rule):
    if the actor is carrying the noun, stop the action with library message
        taking action number 5 for the noun;
    if the actor is wearing the noun, stop the action with library message
        taking action number 5 for the noun.

Check an actor taking (this is the can't take scenery rule):
    if the noun is scenery, stop the action with library message taking
        action number 10 for the noun.

Check an actor taking (this is the can only take things rule):
    if the noun is not a thing, stop the action with library message taking
        action number 15 for the noun.

Check an actor taking (this is the can't take what's fixed in place rule):
```

```
    if the noun is fixed in place, stop the action with library message taking
        action number 11 for the noun.

Check an actor taking (this is the use player's holdall to avoid exceeding
    carrying capacity rule):
    if the number of things carried by the actor is at least the
        carrying capacity of the actor:
        if the actor is holding a player's holdall (called the current working sack):
            let the transferred item be nothing;
            repeat with the possible item running through things carried by
                the actor:
                if the possible item is not lit and the possible item is not
                    the current working sack, let the transferred item be the possible item;
            if the transferred item is not nothing:
                issue library message taking action number 13 for the
                    transferred item;
                silently try the actor trying inserting the transferred item
                    into the current working sack;
                if the transferred item is not in the current working sack, stop the action;

Check an actor taking (this is the can't exceed carrying capacity rule):
    if the number of things carried by the actor is at least the
        carrying capacity of the actor, stop the action with library
        message taking action number 12 for the actor.
```

§8. Carry out.

```
Carry out an actor taking (this is the standard taking rule):
    now the actor carries the noun.
```

§9. Report.

```
Report an actor taking (this is the standard report taking rule):
    if the actor is the player, issue library message taking action number 1
        for the noun;
    otherwise issue actor-based library message taking action number 16 for the noun.
```

§10. Removing it from.

```
Removing it from is an action applying to two things.
The removing it from action translates into I6 as "Remove".

The specification of the removing it from action is "Removing is not really
an action in its own right. Whereas there are many ways to put something down
(on the floor, on top of something, inside something else, giving it to
somebody else, and so on), Inform has only one way to take something: the
taking action. Removing exists only to provide some nicely worded replies
to impossible requests, and in all sensible cases is converted into taking.
Because of this, it's usually a bad idea to write rules about removing:
if you write a rule such as 'Instead of removing the key, ...' then it
won't apply if the player simply types TAKE KEY instead. The safe way to
do this is to write a rule about taking, which covers all possibilities."
```

§11. Check.

```
Check an actor removing something from (this is the can't remove what's not inside rule):
    if the holder of the noun is not the second noun, stop the action with
        library message removing it from action number 2 for the noun.

Check an actor removing something from (this is the can't remove from people rule):
    if the holder of the noun is a person (called the owner), stop the action with
        library message taking action number 6 for the owner.

Check an actor removing something from (this is the convert remove to take rule):
    convert to the taking action on the noun.

The can't take component parts rule is listed before the can't remove what's not
inside rule in the check removing it from rules.
```

§12. Dropping.

```
Dropping is an action applying to one thing.
The dropping action translates into I6 as "Drop".

The specification of the dropping action is "Dropping is one of five actions
by which an actor can get rid of something carried: the others are inserting
(into a container), putting (onto a supporter), giving (to someone else) and
eating. Dropping means dropping onto the actor's current floor, which is
usually the floor of a room - but might be the inside of a box if the actor
is also inside that box, and so on.

The can't drop clothes being worn rule silently tries the taking off action
on any clothing being dropped: unlisting this rule removes both this behaviour
and also the requirement that clothes cannot simply be dropped."
```

§13. Check.

```
Check an actor dropping (this is the can't drop yourself rule):
    if the noun is the actor, stop the action with library message putting
        it on action number 4.

Check an actor dropping (this is the can't drop what's already dropped rule):
    if the noun is in the holder of the actor, stop the action with library
        message dropping action number 1 for the noun.

Check an actor dropping (this is the can't drop what's not held rule):
    if the actor is carrying the noun, continue the action;
    if the actor is wearing the noun, continue the action;
    stop the action with library message dropping action number 2 for the noun.

Check an actor dropping (this is the can't drop clothes being worn rule):
    if the actor is wearing the noun:
        issue library message dropping action number 3 for the noun;
        silently try the actor trying taking off the noun;
        if the actor is wearing the noun, stop the action;

Check an actor dropping (this is the can't drop if this exceeds carrying
    capacity rule):
    let H be the holder of the actor;
    if H is a room, continue the action; [room floors have infinite capacity]
    if H provides the property carrying capacity:
        if H is a supporter:
            if the number of things on H is at least the carrying capacity of H:
                if the actor is the player,
                    issue library message dropping action number 5 for H;
                stop the action;
        otherwise if H is a container:
            if the number of things in H is at least the carrying capacity of H:
                if the actor is the player,
                    issue library message dropping action number 6 for H;
                stop the action;
```

§**14.** Carry out.

```
Carry out an actor dropping (this is the standard dropping rule):
    now the noun is in the holder of the actor.
```

§**15.** Report.

```
Report an actor dropping (this is the standard report dropping rule):
    if the actor is the player, issue library message dropping action number 4
        for the noun;
    otherwise issue actor-based library message dropping action number 7 for the noun.
```

§16. Putting it on.

```
Putting it on is an action with past participle put, applying to two things.
The putting it on action translates into I6 as "PutOn".

The specification of the putting it on action is "By this action, an actor puts
something he is holding on top of a supporter: for instance, putting an apple
on a table."
```

§17. Check.

```
Check an actor putting something on (this is the convert put to drop where possible rule):
    if the second noun is down or the actor is on the second noun,
        convert to the dropping action on the noun.

Check an actor putting something on (this is the can't put what's not held rule):
    if the actor is carrying the noun, continue the action;
    if the actor is wearing the noun, continue the action;
    stop the action with library message putting it on action number 1 for the noun.

Check an actor putting something on (this is the can't put something on itself rule):
    let the noun-CPC be the component parts core of the noun;
    let the second-CPC be the component parts core of the second noun;
    let the transfer ceiling be the common ancestor of the noun-CPC with the second-CPC;
    if the transfer ceiling is the noun-CPC,
        stop the action with library message putting it on action number 2 for
            the noun.

Check an actor putting something on (this is the can't put onto what's not a supporter rule):
    if the second noun is not a supporter,
        stop the action with library message putting it on action number 3 for
            the second noun.

Check an actor putting something on (this is the can't put onto something being carried rule):
    if the actor encloses the second noun,
        stop the action with library message putting it on action number 4 for
            the second noun.

Check an actor putting something on (this is the can't put clothes being worn rule):
    if the actor is wearing the noun:
        issue library message putting it on action number 5 for the noun;
        silently try the actor trying taking off the noun;
        if the actor is wearing the noun, stop the action;

Check an actor putting something on (this is the can't put if this exceeds
    carrying capacity rule):
    if the second noun provides the property carrying capacity:
        if the number of things on the second noun is at least the carrying capacity
            of the second noun,
            stop the action with library message putting it on action number 6 for
            the second noun;
```

§18. Carry out.

```
Carry out an actor putting something on (this is the standard putting rule):
    now the noun is on the second noun.
```

§**19.** Report.

```
Report an actor putting something on (this is the concise report putting rule):
    if the actor is the player and the I6 parser is running multiple actions,
        stop the action with library message putting it on action number 7
        for the noun;
    otherwise continue the action.

Report an actor putting something on (this is the standard report putting rule):
    if the actor is the player, issue library message putting it on action
        number 8 for the noun;
    otherwise issue actor-based library message putting it on action
        number 9 for the noun.
```

§20. Inserting it into.

```
Inserting it into is an action applying to two things.
The inserting it into action translates into I6 as "Insert".

The specification of the inserting it into action is "By this action, an actor puts
something he is holding into a container: for instance, putting a coin into a
collection box."
```

§21. Check.

```
Check an actor inserting something into (this is the convert insert to drop where
    possible rule):
    if the second noun is down or the actor is in the second noun,
        convert to the dropping action on the noun.

Check an actor inserting something into (this is the can't insert what's not held rule):
    if the actor is carrying the noun, continue the action;
    if the actor is wearing the noun, continue the action;
    stop the action with library message inserting it into action number 1 for
        the noun.

Check an actor inserting something into (this is the can't insert something into itself rule):
    let the noun-CPC be the component parts core of the noun;
    let the second-CPC be the component parts core of the second noun;
    let the transfer ceiling be the common ancestor of the noun-CPC with the second-CPC;
    if the transfer ceiling is the noun-CPC,
        stop the action with library message inserting it into action number 5 for
            the noun.

Check an actor inserting something into (this is the can't insert into closed containers rule):
    if the second noun is a closed container,
        stop the action with library message inserting it into action number 3 for
            the second noun.

Check an actor inserting something into (this is the can't insert into what's not a
    container rule):
    if the second noun is not a container,
        stop the action with library message inserting it into action number 2 for
            the second noun.

Check an actor inserting something into (this is the can't insert clothes being worn rule):
    if the actor is wearing the noun:
        issue library message inserting it into action number 6 for the noun;
        silently try the actor trying taking off the noun;
        if the actor is wearing the noun, stop the action;

Check an actor inserting something into (this is the can't insert if this exceeds
    carrying capacity rule):
    if the second noun provides the property carrying capacity:
        if the number of things in the second noun is at least the carrying capacity
        of the second noun,
            stop the action with library message inserting it into action number 7 for
                the second noun;
```

§22. Carry out.

```
Carry out an actor inserting something into (this is the standard inserting rule):
    now the noun is in the second noun.
```

§23. Report.

```
Report an actor inserting something into (this is the concise report inserting rule):
    if the actor is the player and the I6 parser is running multiple actions,
        stop the action with library message inserting it into action number 8
        for the noun;
    otherwise continue the action.

Report an actor inserting something into (this is the standard report inserting rule):
    if the actor is the player, issue library message inserting it into action
        number 9 for the noun;
    otherwise issue actor-based library message inserting it into action number 10 for the noun.
```

§24. Eating.

```
Eating is an action with past participle eaten, applying to one carried thing.
The eating action translates into I6 as "Eat".

The specification of the eating action is "Eating is the only one of the
built-in actions which can, in effect, destroy something: the carry out
rule removes what's being eaten from play, and nothing in the Standard
Rules can then get at it again.

Note that, uncontroversially, one can only eat things with the 'edible'
either/or property, and also that, more controversially, one can only
eat things currently being held. This means that a player standing next
to a bush with berries who types EAT BERRIES will force a '(first taking
the berries)' action."
```

§25. Check.

```
Check an actor eating (this is the can't eat unless edible rule):
    if the noun is not a thing or the noun is not edible,
        stop the action with library message eating action number 1 for the noun.

Check an actor eating (this is the can't eat clothing without removing it first rule):
    if the actor is wearing the noun:
        issue library message dropping action number 3 for the noun;
        try the actor trying taking off the noun;
        if the actor is wearing the noun, stop the action;
```

§26. Carry out.

```
Carry out an actor eating (this is the standard eating rule):
    remove the noun from play.
```

§27. Report.

```
Report an actor eating (this is the standard report eating rule):
    if the actor is the player, issue library message eating action number 2
        for the noun;
    otherwise issue actor-based library message eating action number 3 for the noun.
```

§28. Going.

```
Section SR4/3 - Standard actions which move the actor

Going is an action with past participle gone, applying to one visible thing.
The going action translates into I6 as "Go".

The specification of the going action is "This is the action which allows people
to move from one room to another, using whatever map connections and doors are
to hand. The Standard Rules are written so that the noun can be either a
direction or a door in the location of the actor: while the player's commands
only lead to going actions with directions as nouns, going actions can also
happen as a result of entering actions, and then the noun can indeed be
a door."

The going action has a room called the room gone from (matched as "from").
The going action has an object called the room gone to (matched as "to").
The going action has an object called the door gone through (matched as "through").
The going action has an object called the vehicle gone by (matched as "by").
The going action has an object called the thing gone with (matched as "with").

Rule for setting action variables for going (this is the standard set going variables rule):
    now the thing gone with is the item-pushed-between-rooms;
    now the room gone from is the location of the actor;
    if the actor is in an enterable vehicle (called the carriage),
        now the vehicle gone by is the carriage;
    let the target be nothing;
    if the noun is a direction:
        let direction D be the noun;
        let the target be the room-or-door direction D from the room gone from;
    otherwise:
        if the noun is a door, let the target be the noun;
    if the target is a door:
        now the door gone through is the target;
        now the target is the other side of the target from the room gone from;
    now the room gone to is the target.
```

§29. Check.

```
Check an actor going (this is the can't travel in what's not a vehicle rule):
    let H be the holder of the actor;
    if H is the room gone from, continue the action;
    if H is the vehicle gone by, continue the action;
    stop the action with library message going action number 1 for H.

Check an actor going (this is the can't go through undescribed doors rule):
    if the door gone through is not nothing and the door gone through is undescribed,
        stop the action with library message going action number 2 for the room gone from.

Check an actor going (this is the can't go through closed doors rule):
    if the door gone through is not nothing and the door gone through is closed:
        if the noun is up, stop the action with library message going action number 3
            for the door gone through;
        if the noun is down, stop the action with library message going action number 4
            for the door gone through;
        stop the action with library message going action number 5 for the door gone
            through;
```

```
Check an actor going (this is the determine map connection rule):
    let the target be nothing;
    if the noun is a direction:
        let direction D be the noun;
        let the target be the room-or-door direction D from the room gone from;
    otherwise:
        if the noun is a door, let the target be the noun;
    if the target is a door:
        now the target is the other side of the target from the room gone from;
    now the room gone to is the target.
Check an actor going (this is the can't go that way rule):
    if the room gone to is nothing:
        if the door gone through is nothing, stop the action with library
            message going action number 2 for the room gone from;
        stop the action with library message going action number 6 for the door gone through;
```

§30. Carry out.

```
Carry out an actor going (this is the move player and vehicle rule):
    if the vehicle gone by is nothing,
        surreptitiously move the actor to the room gone to during going;
    otherwise
        surreptitiously move the vehicle gone by to the room gone to during going.
Carry out an actor going (this is the move floating objects rule):
    if the actor is the player,
        update backdrop positions.
Carry out an actor going (this is the check light in new location rule):
    if the actor is the player,
        surreptitiously reckon darkness.
```

§31. Report.

```
Report an actor going (this is the describe room gone into rule):
    if the player is the actor:
        produce a room description with going spacing conventions;
    otherwise:
        if the noun is a direction:
            if the location is the room gone from:
                if the location is the room gone to:
                    continue the action;
                otherwise:
                    if the noun is up :
                        issue actor-based library message going action number 8;
                    otherwise if the noun is down:
                        issue actor-based library message going action number 9;
                    otherwise:
                        issue actor-based library message going action number 10 for the noun;
            otherwise:
                let the back way be the opposite of the noun;
                if the location is the room gone to:
                    let the room back the other way be the room back way from the
```

```
                location;
            let the room normally this way be the room noun from the
                room gone from;
            if the room back the other way is the room gone from or
                the room back the other way is the room normally this way:
                if the back way is up:
                    issue actor-based library message going action number 11;
                otherwise if the back way is down:
                    issue actor-based library message going action number 12;
                otherwise:
                    issue actor-based library message going action number 13
                        for the back way;
            otherwise:
                issue actor-based library message going action number 14;
        otherwise:
            if the back way is up :
                issue actor-based library message going action number 15
                    for the room gone to;
            otherwise if the back way is down:
                issue actor-based library message going action number 16
                    for the room gone to;
            otherwise:
                issue actor-based library message going action number 17
                    for the room gone to and the back way;
otherwise if the location is the room gone from:
    issue actor-based library message going action number 18 for the noun;
otherwise:
    issue actor-based library message going action number 19 for the noun;
if the vehicle gone by is not nothing:
    say " ";
    if the vehicle gone by is a supporter, issue actor-based library message
        going action number 20 for the vehicle gone by;
    otherwise issue actor-based library message going action number 21
        for the vehicle gone by;
if the thing gone with is not nothing:
    if the player is within the thing gone with:
        issue actor-based library message going action number 22 for the thing gone with;
    otherwise if the player is within the vehicle gone by:
        issue actor-based library message going action number 23 for the thing gone with;
    otherwise if the location is the room gone from:
        issue actor-based library message going action number 24 for the thing gone with;
    otherwise:
        issue actor-based library message going action number 25 for the thing gone with;
if the player is within the vehicle gone by and the player is not
    within the thing gone with:
    issue actor-based library message going action number 26;
    say ".";
    try looking;
    continue the action;
say ".";
```

§32. Entering.

```
Entering is an action applying to one thing.
The entering action translates into I6 as "Enter".

The specification of the entering action is "Whereas the going action allows
people to move from one location to another in the model world, the entering
action is for movement inside a location: for instance, climbing into a cage
or sitting on a couch. (Entering is not allowed to change location, so any
attempt to enter a door is converted into a going action.) What makes
entering trickier than it looks is that the player may try to enter an
object which is itself inside, or part of, something else, which might in
turn be... and so on. To preserve realism, the implicitly pass through other
barriers rule automatically generates entering and exiting actions needed
to pass between anything which might be in the way: for instance, in a
room with two open cages, an actor in cage A who tries entering cage B first
has to perform an exiting action."
```

§33. Check.

```
Check an actor entering (this is the convert enter door into go rule):
    if the noun is a door, convert to the going action on the noun.

Check an actor entering (this is the convert enter compass direction into go rule):
    if the noun is a direction, convert to the going action on the noun.

Check an actor entering (this is the can't enter what's already entered rule):
    let the local ceiling be the common ancestor of the actor with the noun;
    if the local ceiling is the noun, stop the action with library message
        entering action number 1 for the noun.

Check an actor entering (this is the can't enter what's not enterable rule):
    if the noun is not enterable, stop the action with library message
        entering action number 2 for the noun.

Check an actor entering (this is the can't enter closed containers rule):
    if the noun is a closed container, stop the action with library message
        entering action number 3 for the noun.

Check an actor entering (this is the can't enter something carried rule):
    let the local ceiling be the common ancestor of the actor with the noun;
    if the local ceiling is the actor, stop the action with library message
        entering action number 4 for the noun.

Check an actor entering (this is the implicitly pass through other barriers rule):
    if the holder of the actor is the holder of the noun, continue the action;
    let the local ceiling be the common ancestor of the actor with the noun;
    while the holder of the actor is not the local ceiling:
        let the target be the holder of the actor;
        issue library message entering action number 6 for the target;
        silently try the actor trying exiting;
        if the holder of the actor is the target, stop the action;
    if the holder of the actor is the noun, stop the action;
    if the holder of the actor is the holder of the noun, continue the action;
    let the target be the holder of the noun;
    if the noun is part of the target, let the target be the holder of the target;
    while the target is a thing:
        if the holder of the target is the local ceiling:
```

```
            issue library message entering action number 7 for the target;
            silently try the actor trying entering the target;
            if the holder of the actor is not the target, stop the action;
            convert to the entering action on the noun;
            continue the action;
        let the target be the holder of the target;
```

§**34.** Carry out.

```
Carry out an actor entering (this is the standard entering rule):
    surreptitiously move the actor to the noun.
```

§**35.** Report.

```
Report an actor entering (this is the standard report entering rule):
    if the actor is the player:
        issue library message entering action number 5 for the noun;
    otherwise if the noun is a container:
        issue actor-based library message entering action number 8 for the noun;
    otherwise:
        issue actor-based library message entering action number 9 for the noun;
    continue the action.

Report an actor entering (this is the describe contents entered into rule):
    if the actor is the player, describe locale for the noun.
```

§36. Exiting.

```
Exiting is an action applying to nothing.
The exiting action translates into I6 as "Exit".
The exiting action has an object called the container exited from.

The specification of the exiting action is "Whereas the going action allows
people to move from one location to another in the model world, and the
entering action is for movement deeper inside the objects in a location,
the exiting action is for movement back out towards the main floor area.
Climbing out of a cupboard, for instance, is an exiting action. Exiting
when already in the main floor area of a room with a map connection to
the outside is converted to a going action. Finally, note that whereas
entering works for either containers or supporters, exiting is purely for
getting oneself out of containers: if the actor is on top of a supporter
instead, an exiting action is converted to the getting off action."

Setting action variables for exiting:
    now the container exited from is the holder of the actor.
```

§37. Check.

```
Check an actor exiting (this is the convert exit into go out rule):
    let the local room be the location of the actor;
    if the container exited from is the local room:
        if the room-or-door outside from the local room is not nothing,
            convert to the going action on the outside;

Check an actor exiting (this is the can't exit when not inside anything rule):
    let the local room be the location of the actor;
    if the container exited from is the local room, stop the action with
        library message exiting action number 1 for the actor.

Check an actor exiting (this is the can't exit closed containers rule):
    if the actor is in a closed container (called the cage), stop the action
        with library message exiting action number 2 for the cage.

Check an actor exiting (this is the convert exit into get off rule):
    if the actor is on a supporter (called the platform),
        convert to the getting off action on the platform.
```

§38. Carry out.

```
Carry out an actor exiting (this is the standard exiting rule):
    let the former exterior be the not-counting-parts holder of the container exited from;
    surreptitiously move the actor to the former exterior.
```

§**39.** Report.

```
Report an actor exiting (this is the standard report exiting rule):
    if the actor is the player:
        issue library message exiting action number 3 for the container exited from;
    otherwise:
        issue actor-based library message exiting action number 6 for the container exited from;
    continue the action.

Report an actor exiting (this is the describe room emerged into rule):
    if the actor is the player,
        produce a room description with going spacing conventions.
```

§40. Getting off.

```
Getting off is an action with past participle got, applying to one thing.
The getting off action translates into I6 as "GetOff".

The specification of the getting off action is "The getting off action is for
actors who are currently on top of a supporter: perhaps standing on a platform,
but maybe only sitting on a chair or even lying down in bed. Unlike the similar
exiting action, getting off takes a noun: the platform, chair, bed or what
have you."
```

§41. Check.

```
Check an actor getting off (this is the can't get off things rule):
    if the actor is on the noun, continue the action;
    if the actor is carried by the noun, continue the action;
    stop the action with library message getting off action number 1 for the noun.
```

§42. Carry out.

```
Carry out an actor getting off (this is the standard getting off rule):
    let the former exterior be the not-counting-parts holder of the noun;
    surreptitiously move the actor to the former exterior.
```

§43. Report.

```
Report an actor getting off (this is the standard report getting off rule):
    if the actor is the player:
        issue library message exiting action number 3 for the noun;
    otherwise:
        issue actor-based library message exiting action number 5 for the noun;
    continue the action.

Report an actor getting off (this is the describe room stood up into rule):
    if the actor is the player,
        produce a room description with going spacing conventions.
```

§44. Looking.

```
Section SR4/4 - Standard actions concerning the actor's vision

Looking is an action applying to nothing.
The looking action translates into I6 as "Look".

The specification of the looking action is "The looking action describes the
player's current room and any visible items, but is made more complicated
by the problem of visibility. Inform calculates this by dividing the room
into visibility levels. For an actor on the floor of a room, there is only
one such level: the room itself. But an actor sitting on a chair inside
a packing case which is itself on a gantry would have four visibility levels:
chair, case, gantry, room. The looking rules use a special phrase, 'the
visibility-holder of X', to go up from one level to the next: thus the
visibility-holder of the case is the gantry.

The 'visibility level count' is the number of levels which the player can
actually see, and the 'visibility ceiling' is the uppermost visible level.
For a player standing on the floor of a lighted room, this will be a count
of 1 with the ceiling set to the room. But a player sitting on a chair in
a closed opaque packing case would have visibility level count 2, and
visibility ceiling equal to the case. Moreover, light has to be available
in order to see anything at all: if the player is in darkness, the level
count is 0 and the ceiling is nothing.

Finally, note that several actions other than looking also produce room
descriptions in some cases. The most familiar is going, but exiting a
container or getting off a supporter will also generate a room description.
(The phrase used by the relevant rules is 'produce a room description with
going spacing conventions' and carry out or report rules for newly written
actions are welcome to use this too if they would like to. The spacing
conventions affect paragraph division, and note that the main description
paragraph may be omitted for a place not newly visited, depending on the
VERBOSE settings.) Room descriptions like this are produced by running the
check, carry out and report rules for looking, but are not subject to
before, instead or after rules: so they do not count as a new action. The
looking variable 'room-describing action' holds the action name of the
reason a room description is currently being made: if the player typed
LOOK, this will indeed be set to the looking action, but if we're
describing a room just reached by GO EAST, say, it will be set to the going
action. This can be used to customise carry out looking rules so that
different forms of description are used on going to a room as compared with
looking around while already there."

The looking action has an action-name called the room-describing action.
The looking action has a truth state called abbreviated form allowed.
The looking action has a number called the visibility level count.
The looking action has an object called the visibility ceiling.

Setting action variables for looking (this is the determine visibility ceiling
    rule):
    if the actor is the player, calculate visibility ceiling at low level;
    now the visibility level count is the visibility ceiling count calculated;
    now the visibility ceiling is the visibility ceiling calculated;
    now the room-describing action is the looking action.
```

§**45.** Carry out.

```
Carry out looking (this is the room description heading rule):
    say bold type;
    if the visibility level count is 0:
        begin the printing the name of a dark room activity;
        if handling the printing the name of a dark room activity,
            issue miscellaneous library message number 71;
        end the printing the name of a dark room activity;
    otherwise if the visibility ceiling is the location:
        say "[visibility ceiling]";
    otherwise:
        say "[The visibility ceiling]";
    say roman type;
    let intermediate level be the visibility-holder of the actor;
    repeat with intermediate level count running from 2 to the visibility level count:
        issue library message looking action number 8 for the intermediate level;
        let the intermediate level be the visibility-holder of the intermediate level;
    say line break;
    say run paragraph on with special look spacing.

Carry out looking (this is the room description body text rule):
    if the visibility level count is 0:
        if set to abbreviated room descriptions, continue the action;
        if set to sometimes abbreviated room descriptions and
            abbreviated form allowed is true and
            darkness witnessed is true,
            continue the action;
        begin the printing the description of a dark room activity;
        if handling the printing the description of a dark room activity,
            issue miscellaneous library message number 17;
        end the printing the description of a dark room activity;
    otherwise if the visibility ceiling is the location:
        if set to abbreviated room descriptions, continue the action;
        if set to sometimes abbreviated room descriptions and abbreviated form
            allowed is true and the location is visited, continue the action;
        print the location's description;

Carry out looking (this is the room description paragraphs about objects rule):
    if the visibility level count is greater than 0:
        let the intermediate position be the actor;
        let the IP count be the visibility level count;
        while the IP count is greater than 0:
            now the intermediate position is marked for listing;
            let the intermediate position be the visibility-holder of the
                intermediate position;
            decrease the IP count by 1;
        let the top-down IP count be the visibility level count;
        while the top-down IP count is greater than 0:
            let the intermediate position be the actor;
            let the IP count be 0;
            while the IP count is less than the top-down IP count:
                let the intermediate position be the visibility-holder of the
                    intermediate position;
                increase the IP count by 1;
```

```
            [if we ever support I6-style inside descriptions, here's where]
            describe locale for the intermediate position;
            decrease the top-down IP count by 1;
    continue the action;
Carry out looking (this is the check new arrival rule):
    if in darkness:
        now the darkness witnessed is true;
    otherwise:
        if the location is a room, now the location is visited;
```

§46. Report.

```
Report an actor looking (this is the other people looking rule):
    if the actor is not the player,
        issue actor-based library message looking action number 9.
```

§47. Examining.

```
Examining is an action applying to one visible thing and requiring light.
The examining action translates into I6 as "Examine".

The specification of the examining action is "The act of looking closely at
something. Note that the noun could be either a direction or a thing, which
is why the Standard Rules include the 'examine directions rule' to deal with
directions: it simply says 'You see nothing unexpected in that direction.'
and stops the action. (If you would like to handle directions differently,
list another rule instead of this one in the carry out examining rules.)

For arcane reasons to do with the Inform 6 library underlying what Inform
does, these rules test to see 'if the noun goes undescribed by source
text' (rather than more simply testing whether the description property
of the noun is blank). If so, we search (i.e., look inside) a container,
and show the status (switched on or off) of a device, but otherwise
give up with a bland response. For traditional reasons, we also show
the status of a device as a second paragraph even after any description
property has been printed: this is done by the examine described devices
rule."
```

§48. Carry out.

```
Carry out examining (this is the examine undescribed containers rule):
    if the noun goes undescribed by source text and
        the noun is a container,
        convert to the searching action on the noun.

Carry out examining (this is the examine undescribed devices rule):
    if the noun goes undescribed by source text and
        the noun is a device,
        stop the action with library message examining action number 3
            for the noun.

Carry out examining (this is the examine undescribed things rule):
    if the noun goes undescribed by source text,
        stop the action with library message examining action number 2
            for the noun.

Carry out examining (this is the examine directions rule):
    if the noun is a direction,
        stop the action with library message examining action number 5
            for the noun.

Carry out examining (this is the standard examining rule):
    say "[the description of the noun][line break]".

Carry out examining (this is the examine described devices rule):
    if the noun is a device,
        stop the action with library message examining action number 3
            for the noun.
```

§49. Report.

```
Report an actor examining (this is the report other people examining rule):
    if the actor is not the player,
        issue actor-based library message examining action number 4 for the noun.
```

§50. Looking under.

```
Looking under is an action applying to one visible thing and requiring light.
The looking under action translates into I6 as "LookUnder".

The specification of the looking under action is "The standard Inform world
model does not have a concept of things being under other things, so this
action is only minimally provided by the Standard Rules, but it exists here
for traditional reasons (and because, after all, LOOK UNDER TABLE is the
sort of command which ought to be recognised even if it does nothing useful).
The action ordinarily either tells the player he finds nothing of interest,
or reports that somebody else has looked under something.

The usual way to make this action do something useful is to write a rule
like 'Instead of looking under the cabinet for the first time: now the
player has the silver key; say ...' and so on."
```

§51. Carry out.

```
Carry out an actor looking under (this is the standard looking under rule):
    stop the action with library message looking under action number 2 for
        the noun.
```

§52. Report.

```
Report an actor looking under (this is the report other people looking under rule):
    if the actor is not the player,
        issue actor-based library message looking under action number 3 for the noun.
```

§53. Searching.

```
Searching is an action applying to one thing and requiring light.
The searching action translates into I6 as "Search".

The specification of the searching action is "Searching looks at the contents
of an open or transparent container, or at the items on top of a supporter.
These are often mentioned in room descriptions already, and then the action
is unnecessary, but that wouldn't be true for something like a kitchen
cupboard which is scenery - mentioned in passing in a room description, but
not made a fuss of. Searching such a cupboard would then, by listing its
contents, give the player more information than the ordinary room description
shows.

The usual check rules restrict searching to containers and supporters: so
the Standard Rules do not allow the searching of people, for instance. But
it is easy to add instead rules ('Instead of searching Dr Watson: ...') or
even a new carry out rule ('Check searching someone (called the suspect): ...')
to extend the way searching normally works."
```

§54. Check.

```
Check an actor searching (this is the can't search unless container or supporter rule):
    if the noun is not a container and the noun is not a supporter,
        stop the action with library message searching action number 4 for
            the noun.

Check an actor searching (this is the can't search closed opaque containers rule):
    if the noun is a closed opaque container,
        stop the action with library message searching action number 5 for
            the noun.
```

§55. Report.

```
Report searching a container (this is the standard search containers rule):
    if the noun contains a described thing which is not scenery,
        issue library message searching action number 7 for the noun;
    otherwise
        issue library message searching action number 6 for the noun.

Report searching a supporter (this is the standard search supporters rule):
    if the noun supports a described thing which is not scenery,
        issue library message searching action number 3 for the noun;
    otherwise
        issue library message searching action number 2 for the noun.

Report an actor searching (this is the report other people searching rule):
    if the actor is not the player,
        issue actor-based library message searching action number 8 for the noun.
```

§56. Consulting it about.

```
Consulting it about is an action applying to one thing and one topic.
The consulting it about action translates into I6 as "Consult".

The specification of the consulting it about action is "Consulting is a very
flexible and potentially powerful action, but only because it leaves almost
all of the work to the author to deal with directly. The idea is for it to
respond to commands such as LOOK UP HENRY FITZROY IN HISTORY BOOK, where
the topic would be the snippet of command HENRY FITZROY and the thing would
be the book.

The Standard Rules simply parry such requests by saying that the player finds
nothing of interest. All interesting responses must be provided by the author,
using rules like 'Instead of consulting the history book about...'"
```

§57. Report.

```
Report an actor consulting something about (this is the block consulting rule):
    if the actor is the player,
        issue library message consulting it about action number 1 for the noun;
    otherwise
        issue actor-based library message consulting it about action number 2 for the noun.
```

§58. Locking it with.

```
Section SR4/5 - Standard actions which change the state of things

Locking it with is an action applying to one thing and one carried thing.
The locking it with action translates into I6 as "Lock".

The specification of the locking it with action is "Locking is the act of
using an object such as a key to ensure that something such as a door or
container cannot be opened unless first unlocked. (Only closed things can be
locked.)

Locking can be performed on any kind of thing which provides the either/or
properties lockable, locked, openable and open. The 'can't lock without a lock
rule' tests to see if the noun both provides the lockable property, and if
it is in fact lockable: it is then assumed that the other properties can
safely be checked. In the Standard Rules, the container and door kinds both
satisfy these requirements.

We can create a new kind on which opening, closing, locking and unlocking
will work thus: 'A briefcase is a kind of thing. A briefcase can be openable.
A briefcase can be open. A briefcase can be lockable. A briefcase can be
locked. A briefcase is usually openable, lockable, open and unlocked.'

Inform checks whether the key fits using the 'can't lock without the correct
key rule'. To satisfy this, the actor must be directly holding the second
noun, and it must be the current value of the 'matching key' property for
the noun. (This property is seldom referred to directly because it is
automatically set by assertions like 'The silver key unlocks the wicket
gate.')

The Standard Rules provide locking and unlocking actions at a fairly basic
level: they can be much enhanced using the extension Locksmith by Emily
Short, which is included with all distributions of Inform."
```

§59. Check.

```
Check an actor locking something with (this is the can't lock without a lock rule):
    if the noun provides the property lockable and the noun is lockable,
        continue the action;
    stop the action with library message locking it with action number 1 for the noun.

Check an actor locking something with (this is the can't lock what's already
    locked rule):
    if the noun is locked,
        stop the action with library message locking it with action number 2 for the noun.

Check an actor locking something with (this is the can't lock what's open rule):
    if the noun is open,
        stop the action with library message locking it with action number 3 for the noun.

Check an actor locking something with (this is the can't lock without the correct key rule):
    if the holder of the second noun is not the actor or
        the noun does not provide the property matching key or
        the matching key of the noun is not the second noun,
        stop the action with library message locking it with action number 4 for the second noun.
```

§**60.** Carry out.

```
Carry out an actor locking something with (this is the standard locking rule):
    now the noun is locked.
```

§**61.** Report.

```
Report an actor locking something with (this is the standard report locking rule):
    if the actor is the player:
        issue library message locking it with action number 5 for the noun;
    otherwise:
        if the actor is visible, issue actor-based library message locking it with
            action number 6 for the noun;
```

§62. Unlocking it with.

```
Unlocking it with is an action applying to one thing and one carried thing.
The unlocking it with action translates into I6 as "Unlock".

The specification of the unlocking it with action is "Unlocking undoes the
effect of locking, and renders the noun openable again provided that the
actor is carrying the right key (which must be the second noun).

Unlocking can be performed on any kind of thing which provides the either/or
properties lockable, locked, openable and open. The 'can't unlock without a lock
rule' tests to see if the noun both provides the lockable property, and if
it is in fact lockable: it is then assumed that the other properties can
safely be checked. In the Standard Rules, the container and door kinds both
satisfy these requirements.

We can create a new kind on which opening, closing, locking and unlocking
will work thus: 'A briefcase is a kind of thing. A briefcase can be openable.
A briefcase can be open. A briefcase can be lockable. A briefcase can be
locked. A briefcase is usually openable, lockable, open and unlocked.'

Inform checks whether the key fits using the 'can't unlock without the correct
key rule'. To satisfy this, the actor must be directly holding the second
noun, and it must be the current value of the 'matching key' property for
the noun. (This property is seldom referred to directly because it is
automatically set by assertions like 'The silver key unlocks the wicket
gate.')

The Standard Rules provide locking and unlocking actions at a fairly basic
level: they can be much enhanced using the extension Locksmith by Emily
Short, which is included with all distributions of Inform."
```

§63. Check.

```
Check an actor unlocking something with (this is the can't unlock without a lock rule):
    if the noun provides the property lockable and the noun is lockable,
        continue the action;
    stop the action with library message unlocking it with action number 1 for the noun.

Check an actor unlocking something with (this is the can't unlock what's already unlocked rule):
    if the noun is not locked,
        stop the action with library message unlocking it with action number 2 for the noun.

Check an actor unlocking something with (this is the can't unlock without the correct key rule):
    if the holder of the second noun is not the actor or
        the noun does not provide the property matching key or
        the matching key of the noun is not the second noun,
        stop the action with library message unlocking it with action number 3 for the
            second noun.
```

§64. Carry out.

```
Carry out an actor unlocking something with (this is the standard unlocking rule):
    now the noun is not locked.
```

§**65.** Report.

```
Report an actor unlocking something with (this is the standard report unlocking rule):
    if the actor is the player:
        issue library message unlocking it with action number 4 for the noun;
    otherwise:
        if the actor is visible, issue actor-based library message unlocking it with
            action number 5 for the noun;
```

§66. Switching on.

```
Switching on is an action applying to one thing.
The switching on action translates into I6 as "SwitchOn".

The specification of the switching on action is "The switching on and switching
off actions are for the simplest kind of machinery operation: they are for
objects representing machines (or more likely parts of machines), which are
considered to be either off or on at any given time.

The actions are intended to be used where the noun is a device, but in fact
they could work just as well with any kind which can be 'switched on' or
'switched off'."
```

§67. Check.

```
Check an actor switching on (this is the can't switch on unless switchable rule):
    if the noun provides the property switched on, continue the action;
    stop the action with library message switching on action number 1 for the noun.

Check an actor switching on (this is the can't switch on what's already on rule):
    if the noun is switched on,
        stop the action with library message switching on action number 2 for the noun.
```

§68. Carry out.

```
Carry out an actor switching on (this is the standard switching on rule):
    now the noun is switched on.
```

§69. Report.

```
Report an actor switching on (this is the standard report switching on rule):
    if the actor is the player, issue library message switching on action number 3
        for the noun;
    otherwise issue actor-based library message switching on action number 4 for the noun;
```

§70. Switching off.

```
Switching off is an action applying to one thing.
The switching off action translates into I6 as "SwitchOff".

The specification of the switching off action is "The switching off and switching
on actions are for the simplest kind of machinery operation: they are for
objects representing machines (or more likely parts of machines), which are
considered to be either off or on at any given time.

The actions are intended to be used where the noun is a device, but in fact
they could work just as well with any kind which can be 'switched on' or
'switched off'."
```

§71. Check.

```
Check an actor switching off (this is the can't switch off unless switchable rule):
    if the noun provides the property switched on, continue the action;
    stop the action with library message switching off action number 1 for the noun.

Check an actor switching off (this is the can't switch off what's already off rule):
    if the noun is switched off,
        stop the action with library message switching off action number 2 for the noun.
```

§72. Carry out.

```
Carry out an actor switching off (this is the standard switching off rule):
    now the noun is switched off.
```

§73. Report.

```
Report an actor switching off (this is the standard report switching off rule):
    if the actor is the player, issue library message switching off action number 3
        for the noun;
    otherwise issue actor-based library message switching off action number 4 for the noun;
```

§74. Opening.

```
Opening is an action applying to one thing.
The opening action translates into I6 as "Open".

The specification of the opening action is "Opening makes something no longer
a physical barrier. The action can be performed on any kind of thing which
provides the either/or properties openable and open. The 'can't open unless
openable rule' tests to see if the noun both can be and actually is openable.
(It is assumed that anything which can be openable can also be open.)
In the Standard Rules, the container and door kinds both satisfy these
requirements.

In the event that the thing to be opened is also lockable, we are forbidden
to open it when it is locked. Both containers and doors can be lockable,
but the opening and closing actions would also work fine with kinds which
cannot be.

We can create a new kind on which opening and closing will work thus:
'A case file is a kind of thing. A case file can be openable.
A case file can be open. A case file is usually openable and closed.'

The meaning of open and closed is different for different kinds of thing.
When a container is closed, that means people outside cannot reach in,
and vice versa; when a door is closed, people cannot use the 'going' action
to pass through it. If we were to create a new kind such as 'case file',
we would also need to write rules to make the open and closed properties
interesting for this kind."
```

§75. Check.

```
Check an actor opening (this is the can't open unless openable rule):
    if the noun provides the property openable and the noun is openable,
        continue the action;
    stop the action with library message opening action number 1 for the noun.

Check an actor opening (this is the can't open what's locked rule):
    if the noun provides the property lockable and the noun is locked,
        stop the action with library message opening action number 2 for the noun.

Check an actor opening (this is the can't open what's already open rule):
    if the noun is open,
        stop the action with library message opening action number 3 for the noun.
```

§76. Carry out.

```
Carry out an actor opening (this is the standard opening rule):
    now the noun is open.
```

§**77.** Report.

```
Report an actor opening (this is the reveal any newly visible interior rule):
    if the actor is the player and
        the noun is an opaque container and
        the first thing held by the noun is not nothing and
        the noun does not enclose the actor,
        stop the action with library message opening action number 4 for the noun.
Report an actor opening (this is the standard report opening rule):
    if the actor is the player:
        issue library message opening action number 5 for the noun;
    otherwise if the player can see the actor:
        issue actor-based library message opening action number 6 for the noun;
    otherwise:
        issue actor-based library message opening action number 7 for the noun;
```

§78. Closing.

```
Closing is an action applying to one thing.
The closing action translates into I6 as "Close".

The specification of the closing action is "Closing makes something into
a physical barrier. The action can be performed on any kind of thing which
provides the either/or properties openable and open. The 'can't close unless
openable rule' tests to see if the noun both can be and actually is openable.
(It is assumed that anything which can be openable can also be open, and
hence can also be closed.) In the Standard Rules, the container and door
kinds both satisfy these requirements.

We can create a new kind on which opening and closing will work thus:
'A case file is a kind of thing. A case file can be openable.
A case file can be open. A case file is usually openable and closed.'

The meaning of open and closed is different for different kinds of thing.
When a container is closed, that means people outside cannot reach in,
and vice versa; when a door is closed, people cannot use the 'going' action
to pass through it. If we were to create a new kind such as 'case file',
we would also need to write rules to make the open and closed properties
interesting for this kind."
```

§79. Check.

```
Check an actor closing (this is the can't close unless openable rule):
    if the noun provides the property openable and the noun is openable,
        continue the action;
    stop the action with library message closing action number 1 for the noun.

Check an actor closing (this is the can't close what's already closed rule):
    if the noun is closed,
        stop the action with library message closing action number 2 for the noun.
```

§80. Carry out.

```
Carry out an actor closing (this is the standard closing rule):
    now the noun is closed.
```

§81. Report.

```
Report an actor closing (this is the standard report closing rule):
    if the actor is the player:
        issue library message closing action number 3 for the noun;
    otherwise if the player can see the actor:
        issue actor-based library message closing action number 4 for the noun;
    otherwise:
        issue actor-based library message closing action number 5 for the noun;
```

§82. Wearing.

```
Wearing is an action with past participle worn, applying to one carried thing.
The wearing action translates into I6 as "Wear".

The specification of the wearing action is "The Standard Rules give Inform
only a simple model of clothing. A thing can be worn only if it has the
either/or property of being 'wearable'. (Typing a sentence like 'Mr Jones
wears the Homburg hat.' automatically implies that the hat is wearable,
which is why we only seldom need to use the word 'wearable' directly.)
There is no checking of how much or how little any actor is wearing, or
how incongruous this may appear: nor any distinction between under or
over-clothes.

To put on an article of clothing, the actor must be directly carrying it,
as enforced by the 'can't wear what's not held rule'."
```

§83. Check.

```
Check an actor wearing (this is the can't wear what's not clothing rule):
    if the noun is not a thing or the noun is not wearable,
        stop the action with library message wearing action number 1 for the noun.

Check an actor wearing (this is the can't wear what's not held rule):
    if the holder of the noun is not the actor,
        stop the action with library message wearing action number 2 for the noun.

Check an actor wearing (this is the can't wear what's already worn rule):
    if the actor is wearing the noun,
        stop the action with library message wearing action number 3 for the noun.
```

§84. Carry out.

```
Carry out an actor wearing (this is the standard wearing rule):
    now the actor wears the noun.
```

§85. Report.

```
Report an actor wearing (this is the standard report wearing rule):
    if the actor is the player, issue library message wearing action number 4
        for the noun;
    otherwise issue actor-based library message wearing action number 5
        for the noun.
```

§86. Taking off.

```
Taking off is an action with past participle taken, applying to one carried thing.
The taking off action translates into I6 as "Disrobe".

The specification of the taking off action is "The Standard Rules give Inform
only a simple model of clothing. A thing can be worn only if it has the
either/or property of being 'wearable'. (Typing a sentence like 'Mr Jones
wears the Homburg hat.' automatically implies that the hat is wearable,
which is why we only seldom need to use the word 'wearable' directly.)
There is no checking of how much or how little any actor is wearing, or
how incongruous this may appear: nor any distinction between under or
over-clothes.

When an article of clothing is taken off, it becomes a thing directly
carried by its former wearer, rather than being (say) dropped onto the floor."
```

§87. Check.

```
Check an actor taking off (this is the can't take off what's not worn rule):
    if the actor is not wearing the noun,
        stop the action with library message taking off action number 1 for the noun.
```

§88. Carry out.

```
Carry out an actor taking off (this is the standard taking off rule):
    now the actor carries the noun.
```

§89. Report.

```
Report an actor taking off (this is the standard report taking off rule):
    if the actor is the player, issue library message taking off action number 2
        for the noun;
    otherwise issue actor-based library message taking off action number 3 for the noun.
```

§90. Giving it to.

```
Section SR4/6 - Standard actions concerning other people

Giving it to is an action with past participle given, applying to one carried thing and one thing.
The giving it to action translates into I6 as "Give".

The specification of the giving it to action is "This action is indexed by
Inform under 'Actions concerning other people', but it could just as easily
have gone under 'Actions concerning the actor's possessions' because -
like dropping, putting it on or inserting it into - this is an action
which gets rid of something being carried.

The Standard Rules implement this action fully - if it reaches the carry
out and report rulebooks, then the item is indeed transferred to the
recipient, and this is properly reported. But giving something to
somebody is not like putting something on a shelf: the recipient has
to agree. The final check rule, the 'block giving rule', assumes that
the recipient does not consent - so the gift fails to happen. The way
to make the giving action use its abilities fully is to replace the
block giving rule with a rule which makes a more sophisticated decision
about who will accept what from whom, and only blocks some attempts,
letting others run on into the carry out and report rules."
```

§91. Check.

```
Check an actor giving something to (this is the can't give what you haven't got rule):
    if the actor is not the holder of the noun,
        stop the action with library message giving it to action number 1 for the noun.

Check an actor giving something to (this is the can't give to yourself rule):
    if the actor is the second noun,
        stop the action with library message giving it to action number 2 for the noun.

Check an actor giving something to (this is the can't give to a non-person rule):
    if the second noun is not a person,
        stop the action with library message giving it to action number 4 for the
            second noun.

Check an actor giving something to (this is the block giving rule):
    stop the action with library message giving it to action number 3 for the
        second noun.
```

§92. Carry out.

```
Carry out an actor giving something to (this is the standard giving rule):
    move the noun to the second noun.
```

§93. Report.

```
Report an actor giving something to (this is the standard report giving rule):
    if the actor is the player:
        issue library message giving it to action number 5 for the noun;
    otherwise if the second noun is the player:
        issue actor-based library message giving it to action number 6 for the noun;
    otherwise:
        issue actor-based library message giving it to action number 7 for the noun;
```

§94. Showing it to.

```
Showing it to is an action with past participle shown, applying to one carried thing
    and one visible thing.
The showing it to action translates into I6 as "Show".

The specification of the showing it to action is "Anyone can show anyone
else something which they are carrying, but not some nearby piece of
scenery, say - so this action is suitable for showing the emerald locket
to Katarina, but not showing the Orange River Rock Room to Mr Douglas.

The Standard Rules implement this action in only a minimal way, checking
that it makes sense but then blocking all such attempts with a message
such as 'Katarina is not interested.' - this is the task of the 'block
showing rule'. As a result, there are no carry out or report rules. To
make it into a systematic and interesting action, we would need to
unlist the block showing rule and then to write carry out and report
rules: but usually for IF purposes we only need to make a handful of
special cases of showing work properly, and for those we can simply
write Instead rules to handle them."
```

§95. Check.

```
Check an actor showing something to (this is the can't show what you haven't
    got rule):
    if the actor is not the holder of the noun,
        stop the action with library message showing it to action number 1
            for the noun.

Check an actor showing something to (this is the convert show to yourself to
    examine rule):
    if the actor is the second noun,
        convert to the examining action on the noun.

Check an actor showing something to (this is the block showing rule):
    stop the action with library message showing it to action number 2
        for the second noun.
```

§96. Waking.

```
Waking is an action with past participle woken, applying to one thing.
The waking action translates into I6 as "WakeOther".

The specification of the waking action is "This is the act of jostling
a sleeping person to wake him or her up, and it finds its way into the
Standard Rules only for historical reasons. Inform does not by default
provide any model for people being asleep or awake, so this action does
not do anything in the standard implementation: instead, it is always
stopped by the block waking rule."
```

§97. Check.

```
Check an actor waking (this is the block waking rule):
    stop the action with library message waking action number 1 for the noun.
```

§98. Throwing it at.

```
Throwing it at is an action with past participle thrown, applying to one carried
    thing and one visible thing.
The throwing it at action translates into I6 as "ThrowAt".

The specification of the throwing it at action is "Throwing something at
someone or something is difficult for Inform to model. So many considerations
apply: just because the actor can see the target, does it follow that the
target can accurately hit it? What if the projectile is heavy, like an
anvil, or something not easily aimable, like a feather? What if there
is a barrier in the way, like a cage with bars spaced so that only items
of a certain size get through? And then: what should happen as a result?
Will the projectile break, or do damage, or fall to the floor, or into
a container or onto a supporter? And so on.

Because it seems hopeless to try to model this in any general way,
Inform instead provides the action for the user to attach specific rules to.
The check rules in the Standard Rules simply require that the projectile
is not an item of clothing still worn (this will be relevant for women
attending a Tom Jones concert) but then, in either the 'futile to throw
things at inanimate objects rule' or the 'block throwing at rule', will
refuse to carry out the action with a bland message.

To make throwing do something, then, we must either write Instead rules
for special circumstances, or else unlist these check rules and write
suitable carry out and report rules to pick up the thread."
```

§99. Check.

```
Check an actor throwing something at (this is the implicitly remove thrown clothing rule):
    if the actor is wearing the noun:
        issue library message dropping action number 3 for the noun;
        silently try the actor trying taking off the noun;
        if the actor is wearing the noun, stop the action;

Check an actor throwing something at (this is the futile to throw things at inanimate
    objects rule):
    if the second noun is not a person,
        stop the action with library message throwing it at action number 1
            for the second noun.

Check an actor throwing something at (this is the block throwing at rule):
    stop the action with library message throwing it at action number 2
        for the noun.
```

§100. Attacking.

```
Attacking is an action applying to one thing.
The attacking action translates into I6 as "Attack".

The specification of the attacking action is "Violence is seldom the answer,
and attempts to attack another person are normally blocked as being unrealistic
or not seriously meant. (I might find a shop assistant annoying, but IF is
not Grand Theft Auto, and responding by killing him is not really one of
my options.) So the Standard Rules simply block attempts to fight people,
but the action exists for rules to make exceptions."
```

§101. Check.

```
Check an actor attacking (this is the block attacking rule):
    stop the action with library message attacking action number 1 for the noun.
```

§102. Kissing.

```
Kissing is an action applying to one thing.
The kissing action translates into I6 as "Kiss".

The specification of the kissing action is "Possibly because Inform was
originally written by an Englishman, attempts at kissing another person are
normally blocked as being unrealistic or not seriously meant. So the
Standard Rules simply block attempts to kiss people, but the action exists
for rules to make exceptions."
```

§103. Check.

```
Check an actor kissing (this is the kissing yourself rule):
    if the noun is the actor,
        stop the action with library message touching action number 3 for the noun.

Check an actor kissing (this is the block kissing rule):
    stop the action with library message kissing action number 1 for the noun.
```

§104. Answering it that.

```
Answering it that is an action applying to one thing and one topic.
The answering it that action translates into I6 as "Answer".

The specification of the answering it that action is "The Standard Rules do
not include any systematic way to handle conversation: instead, Inform is
set up so that it is as easy as we can make it to write specific rules
handling speech in particular games, and so that if no such rules are
written then all attempts to communicate are gracefully if not very
interestingly rejected.

The topic here can be any double-quoted text, which can itself contain
tokens in square brackets: see the documentation on Understanding.

Answering is an action existing so that the player can say something free-form
to somebody else. A convention of IF is that a command such as DAPHNE, TAKE
MASK is a request to Daphne to perform an action: if the persuasion rules in
force mean that she consents, the action 'Daphne taking the mask' does
indeed then result. But if the player types DAPHNE, 12375 or DAPHNE, GREAT
HEAVENS - or anything else not making sense as a command - the action
'answering Daphne that ...' will be generated.

The name of the action arises because it is also caused by typing, say,
ANSWER 12375 when Daphne (say) has asked a question."
```

§105. Report.

```
Report an actor answering something that (this is the block answering rule):
    stop the action with library message answering it that action number 1
        for the noun.
```

§106. Telling it about.

```
Telling it about is an action with past participle told, applying to one thing and one topic.
The telling it about action translates into I6 as "Tell".

The specification of the telling it about action is "The Standard Rules do
not include any systematic way to handle conversation: instead, Inform is
set up so that it is as easy as we can make it to write specific rules
handling speech in particular games, and so that if no such rules are
written then all attempts to communicate are gracefully if not very
interestingly rejected.

The topic here can be any double-quoted text, which can itself contain
tokens in square brackets: see the documentation on Understanding.

Telling is an action existing only to catch commands like TELL ALEX ABOUT
GUITAR. Customarily in IF, such a command is shorthand which the player
accepts as a conventional form: it means 'tell Alex what I now know about
the guitar' and would make sense if the player had himself recently
discovered something significant about the guitar which might interest
Alex."
```

§107. Check.

```
Check an actor telling something about (this is the telling yourself rule):
    if the actor is the noun,
        stop the action with library message telling it about action number 1
            for the noun.
```

§108. Report.

```
Report an actor telling something about (this is the block telling rule):
    stop the action with library message telling it about action number 2
        for the noun.
```

§109. Asking it about.

```
Asking it about is an action applying to one thing and one topic.
The asking it about action translates into I6 as "Ask".

The specification of the asking it about action is "The Standard Rules do
not include any systematic way to handle conversation: instead, Inform is
set up so that it is as easy as we can make it to write specific rules
handling speech in particular games, and so that if no such rules are
written then all attempts to communicate are gracefully if not very
interestingly rejected.

The topic here can be any double-quoted text, which can itself contain
tokens in square brackets: see the documentation on Understanding.

Asking is an action existing only to catch commands like ASK STEPHEN ABOUT
PENELOPE. Customarily in IF, such a command is shorthand which the player
accepts as a conventional form: it means 'engage Mary in conversation and
try to find out what she might know about'. It's understood as a convention
of the genre that Mary should not be expected to respond in cases where
there is no reason to suppose that she has anything relevant to pass on -
ASK JANE ABOUT RICE PUDDING, for instance, need not conjure up a recipe
even if Jane is a 19th-century servant and therefore almost certainly
knows one."
```

§110. Report.

```
Report an actor asking something about (this is the block asking rule):
    stop the action with library message asking it about action number 1
        for the noun.
```

§111. Asking it for.

```
Asking it for is an action applying to two things.
The asking it for action translates into I6 as "AskFor".

The specification of the asking it for action is "The Standard Rules do
not include any systematic way to handle conversation, but this is
action is not quite conversation: it doesn't involve any spoken text as
such. It exists to catch commands like ASK SALLY FOR THE EGG WHISK,
where the whisk is something which Sally has and the player can see.

Slightly oddly, but for historical reasons, an actor asking himself for
something is treated to an inventory listing instead. All other cases
are converted to the giving action: that is, ASK SALLY FOR THE EGG WHISK
is treated as if it were SALLY, GIVE ME THE EGG WHISK - an action for
Sally to perform and which then follows rules for giving.

To ask for information or something intangible, see the asking it about
action."
```

§112. Check.

```
Check an actor asking something for (this is the asking yourself for something rule):
    if the actor is the noun and the actor is the player,
        try taking inventory instead.

Check an actor asking something for (this is the translate asking for to giving rule):
    convert to request of the noun to perform giving it to action with the
        second noun and the actor.
```

§113. Waiting.

```
Section SR4/7 - Standard actions which are checked but then do nothing unless rules intervene

Waiting is an action applying to nothing.
The waiting action translates into I6 as "Wait".

The specification of the waiting action is "The inaction action: where would
we be without waiting? Waiting does not cause time to pass by - that happens
anyway - but represents a positive choice by the actor not to fill that time.
It is an action so that rules can be attached to it: for instance, we could
imagine that a player who consciously decides to sit and wait might notice
something which a busy player does not, and we could write a rule accordingly.

Note the absence of check or carry out rules - anyone can wait, at any time,
and it makes nothing happen."
```

§114. Report.

```
Report an actor waiting (this is the standard report waiting rule):
    if the actor is the player, stop the action with library message waiting
        action number 1 for the actor;
    issue actor-based library message waiting action number 2.
```

§115. Touching.

```
Touching is an action applying to one thing.
The touching action translates into I6 as "Touch".

The specification of the touching action is "Touching is just that, touching
something without applying pressure: a touch-sensitive screen or a living
creature might react, but a standard push-button or lever will probably not.

In the Standard Rules there are no check touching rules, since touchability
is already a requirement of the noun for the action anyway, and no carry out
rules because nothing in the standard Inform world model reacts to
a mere touch - though report rules do mean that attempts to touch other
people provoke a special reply."
```

§116. Report.

```
Report an actor touching (this is the report touching yourself rule):
    if the noun is the actor:
        if the actor is the player, issue library message touching action number 3
            for the noun;
        otherwise issue actor-based library message touching action number 4;
        stop the action;
    continue the action.

Report an actor touching (this is the report touching other people rule):
    if the noun is a person:
        if the actor is the player:
            issue library message touching action number 1 for the noun;
        otherwise if the noun is the player:
            issue actor-based library message touching action number 5;
        otherwise:
            issue actor-based library message touching action number 6 for the noun;
        stop the action;
    continue the action.

Report an actor touching (this is the report touching things rule):
    if the actor is the player, issue library message touching action number 2
        for the noun;
    otherwise issue actor-based library message touching action number 6 for the noun.
```

§117. Waving.

```
Waving is an action applying to one thing.
The waving action translates into I6 as "Wave".

The specification of the waving action is "Waving in this sense is like
waving a sceptre: the item to be waved must be directly held (or worn)
by the actor.

In the Standard Rules there are no carry out rules for this action because
nothing in the standard Inform world model which reacts to it. The action
is provided for authors to hang more interesting behaviour onto for special
cases: say, waving a particular rusty iron rod with a star on the end."
```

§118. Check.

```
Check an actor waving (this is the can't wave what's not held rule):
    if the actor is not the holder of the noun,
        stop the action with library message waving action number 1 for the noun.
```

§119. Report.

```
Report an actor waving (this is the report waving things rule):
    if the actor is the player, issue library message waving action number 2
        for the noun;
    otherwise issue actor-based library message waving action number 3 for the noun.
```

§120. Pulling.

```
Pulling is an action applying to one thing.
The Pulling action translates into I6 as "Pull".

The specification of the pulling action is "Pulling is the act of pulling
something not grossly larger than the actor by an amount which would not
substantially move it.

In the Standard Rules there are no carry out rules for this action because
nothing in the standard Inform world model which reacts to it. The action
is provided for authors to hang more interesting behaviour onto for special
cases: say, pulling a lever. ('The big red lever is a fixed in place device.
Instead of pulling the big red lever, try switching on the lever. Instead
of pushing the big red lever, try switching off the lever.')"
```

§121. Check.

```
Check an actor pulling (this is the can't pull what's fixed in place rule):
    if the noun is fixed in place,
        stop the action with library message pulling action number 1 for the noun.

Check an actor pulling (this is the can't pull scenery rule):
    if the noun is scenery,
        stop the action with library message pulling action number 2 for the noun.

Check an actor pulling (this is the can't pull people rule):
    if the noun is a person,
        stop the action with library message pulling action number 4 for the noun.
```

§122. Report.

```
Report an actor pulling (this is the report pulling rule):
    if the actor is the player, issue library message pulling action number 3
        for the noun;
    otherwise issue actor-based library message pulling action number 5 for the noun.
```

§123. Pushing.

```
Pushing is an action applying to one thing.
The Pushing action translates into I6 as "Push".

The specification of the pushing action is "Pushing is the act of pushing
something not grossly larger than the actor by an amount which would not
substantially move it. (See also the pushing it to action, which involves
a longer-distance push between rooms.)

In the Standard Rules there are no carry out rules for this action because
nothing in the standard Inform world model which reacts to it. The action
is provided for authors to hang more interesting behaviour onto for special
cases: say, pulling a lever. ('The big red lever is a fixed in place device.
Instead of pulling the big red lever, try switching on the lever. Instead
of pushing the big red lever, try switching off the lever.')"
```

§124. Check.

```
Check an actor pushing something (this is the can't push what's fixed in place rule):
    if the noun is fixed in place,
        stop the action with library message pushing action number 1 for the noun.

Check an actor pushing something (this is the can't push scenery rule):
    if the noun is scenery,
        stop the action with library message pushing action number 2 for the noun.

Check an actor pushing something (this is the can't push people rule):
    if the noun is a person,
        stop the action with library message pushing action number 4 for the noun.
```

§125. Report.

```
Report an actor pushing something (this is the report pushing rule):
    if the actor is the player, issue library message pushing action number 3
        for the noun;
    otherwise issue actor-based library message pushing action number 6 for the noun.
```

§126. Turning.

```
Turning is an action applying to one thing.
The Turning action translates into I6 as "Turn".

The specification of the turning action is "Turning is the act of rotating
something - say, a dial.

In the Standard Rules there are no carry out rules for this action because
nothing in the standard Inform world model which reacts to it. The action
is provided for authors to hang more interesting behaviour onto for special
cases: say, turning a capstan."
```

§127. Check.

```
Check an actor turning (this is the can't turn what's fixed in place rule):
    if the noun is fixed in place,
        stop the action with library message turning action number 1 for the noun.
Check an actor turning (this is the can't turn scenery rule):
    if the noun is scenery,
        stop the action with library message turning action number 2 for the noun.
Check an actor turning (this is the can't turn people rule):
    if the noun is a person,
        stop the action with library message turning action number 4 for the noun.
```

§128. Report.

```
Report an actor turning (this is the report turning rule):
    if the actor is the player, issue library message turning action number 3
        for the noun;
    otherwise issue actor-based library message turning action number 7 for the noun.
```

§129. Pushing it to.

```
Pushing it to is an action applying to one thing and one visible thing.
The Pushing it to action translates into I6 as "PushDir".

The specification of the pushing it to action is "This action covers pushing
a large object, not being carried, so that the actor pushes it from one room
to another: for instance, pushing a bale of hay to the east.

This is rapidly converted into a special form of the going action. If the
noun object has the either/or property 'pushable between rooms', then the
action is converted to going by the 'standard pushing in directions rule'.
If that going action succeeds, then the original pushing it to action
stops; it's only if that fails that we run on into the 'block pushing in
directions rule', which then puts an end to the matter."
```

§130. Check.

```
Check an actor pushing something to (this is the can't push unpushable things rule):
    if the noun is not pushable between rooms,
        stop the action with library message pushing it to action number 1 for
            the noun.

Check an actor pushing something to (this is the can't push to non-directions rule):
    if the second noun is not a direction,
        stop the action with library message pushing it to action number 2 for
            the noun.

Check an actor pushing something to (this is the can't push vertically rule):
    if the second noun is up or the second noun is down,
        stop the action with library message pushing it to action number 3 for
            the noun.

Check an actor pushing something to (this is the standard pushing in directions rule):
    convert to special going-with-push action.

Check an actor pushing something to (this is the block pushing in directions rule):
    stop the action with library message pushing it to action number 1 for
        the noun.
```

§131. Squeezing.

```
Squeezing is an action applying to one thing.
The Squeezing action translates into I6 as "Squeeze".

The specification of the squeezing action is "Squeezing is an action which
can conveniently vary from squeezing something hand-held, like a washing-up
liquid bottle, right up to squeezing a pillar in a bear hug.

In the Standard Rules there are no carry out rules for this action because
nothing in the standard Inform world model which reacts to it. The action
is provided for authors to hang more interesting behaviour onto for special
cases. A mildly fruity message is produced to players who attempt to
squeeze people, which is blocked by a check squeezing rule."
```

§132. Check.

```
Check an actor squeezing (this is the innuendo about squeezing people rule):
    if the noun is a person,
        stop the action with library message squeezing action number 1 for
            the noun.
```

§133. Report.

```
Report an actor squeezing (this is the report squeezing rule):
    if the actor is the player, issue library message squeezing action number 2
        for the noun;
    otherwise issue actor-based library message squeezing action number 3 for the noun.
```

§134. Saying yes.

```
Section SR4/8 - Standard actions which always do nothing unless rules intervene

Saying yes is an action with past participle said and applying to nothing.
The Saying yes action translates into I6 as "Yes".

The specification of the saying yes action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§135. Check.

```
Check an actor saying yes (this is the block saying yes rule):
    stop the action with library message saying yes action number 1.
```

§136. Saying no.

```
Saying no is an action with past participle said and applying to nothing.
The Saying no action translates into I6 as "No".

The specification of the saying no action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§137. Check.

```
Check an actor saying no (this is the block saying no rule):
    stop the action with library message saying no action number 1.
```

§138. Burning.

```
Burning is an action applying to one thing.
The Burning action translates into I6 as "Burn".

The specification of the burning action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§139. Check.

```
Check an actor burning (this is the block burning rule):
    stop the action with library message burning action number 1.
```

§140. Waking up.

```
Waking up is an action with past participle woken and applying to nothing.
The Waking up action translates into I6 as "Wake".

The specification of the waking up action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§141. Check.

```
Check an actor waking up (this is the block waking up rule):
    stop the action with library message waking up action number 1.
```

§142. Thinking.

```
Thinking is an action past participle thought and applying to nothing.
The Thinking action translates into I6 as "Think".

The specification of the thinking action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§143. Check.

```
Check an actor thinking (this is the block thinking rule):
    stop the action with library message thinking action number 1.
```

§144. Smelling.

```
Smelling is an action applying to one thing.
The Smelling action translates into I6 as "Smell".

The specification of the smelling action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§145. Check.

```
Check an actor smelling (this is the block smelling rule):
    stop the action with library message smelling action number 1 for the noun.
```

§146. Listening to.

```
Listening to is an action applying to one thing.
The Listening to action translates into I6 as "Listen".

The specification of the listening to action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§147. Check.

```
Check an actor listening (this is the block listening rule):
    stop the action with library message listening to action number 1 for the noun.
```

§148. Tasting.

```
Tasting is an action applying to one thing.
The Tasting action translates into I6 as "Taste".

The specification of the tasting action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§149. Check.

```
Check an actor tasting (this is the block tasting rule):
    stop the action with library message tasting action number 1 for the noun.
```

§150. Cutting.

```
Cutting is an action with past participle cut and applying to one thing.
The Cutting action translates into I6 as "Cut".

The specification of the cutting action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§151. Check.

```
Check an actor cutting (this is the block cutting rule):
    stop the action with library message cutting action number 1 for the noun.
```

§152. Jumping.

```
Jumping is an action applying to nothing.
The Jumping action translates into I6 as "Jump".

The specification of the jumping action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§153. Check.

```
Check an actor jumping (this is the block jumping rule):
    stop the action with library message jumping action number 1.
```

§154. Tying it to.

```
Tying it to is an action with past participle tied, and applying to two things.
The Tying it to action translates into I6 as "Tie".

The specification of the tying it to action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§155. Check.

```
Check an actor tying something to (this is the block tying rule):
    stop the action with library message tying it to action number 1 for the noun.
```

§156. Drinking.

```
Drinking is an action with past participle drunk, and applying to one thing.
The Drinking action translates into I6 as "Drink".

The specification of the drinking action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§157. Check.

```
Check an actor drinking (this is the block drinking rule):
    stop the action with library message drinking action number 1 for the noun.
```

§158. Saying sorry.

```
Saying sorry is an action with past participle said and applying to nothing.
The Saying sorry action translates into I6 as "Sorry".

The specification of the saying sorry action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§159. Check.

```
Check an actor saying sorry (this is the block saying sorry rule):
    stop the action with library message saying sorry action number 1.
```

§160. Swearing obscenely.

```
Swearing obscenely is an action censored, with past participle sworn, and applying to nothing.
The Swearing obscenely action translates into I6 as "Strong".

The specification of the swearing obscenely action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§161. Check.

```
Check an actor swearing obscenely (this is the block swearing obscenely rule):
    stop the action with library message swearing obscenely action number 1.
```

§162. Swearing mildly.

```
Swearing mildly is an action censored, with past participle sworn, and applying to nothing.
The Swearing mildly action translates into I6 as "Mild".

The specification of the swearing mildly action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§163. Check.

```
Check an actor swearing mildly (this is the block swearing mildly rule):
    stop the action with library message swearing mildly action number 1.
```

§164. Swinging.

```
Swinging is an action past participle swung and applying to one thing.
The Swinging action translates into I6 as "Swing".

The specification of the swinging action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§165. Check.

```
Check an actor swinging (this is the block swinging rule):
    stop the action with library message swinging action number 1 for the noun.
```

§166. Rubbing.

```
Rubbing is an action applying to one thing.
The Rubbing action translates into I6 as "Rub".

The specification of the rubbing action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§167. Check.

```
Check an actor rubbing (this is the block rubbing rule):
    stop the action with library message rubbing action number 1 for the noun.
```

§168. Setting it to.

```
Setting it to is an action with past participle set, applying to one thing and one topic.
The Setting it to action translates into I6 as "SetTo".

The specification of the setting it to action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§169. Check.

```
Check an actor setting something to (this is the block setting it to rule):
    stop the action with library message setting it to action number 1 for the noun.
```

§170. Waving hands.

```
Waving hands is an action applying to nothing.
The Waving hands action translates into I6 as "WaveHands".

The specification of the waving hands action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§171. Check.

```
Check an actor waving hands (this is the block waving hands rule):
    stop the action with library message waving hands action number 1.
```

§172. Buying.

```
Buying is an action with past participle bought, and applying to one thing.
The Buying action translates into I6 as "Buy".

The specification of the buying action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§173. Check.

```
Check an actor buying (this is the block buying rule):
    stop the action with library message buying action number 1 for the noun.
```

§174. Singing.

```
Singing is an action with past participle sung and applying to nothing.
The Singing action translates into I6 as "Sing".

The specification of the singing action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§175. Check.

```
Check an actor singing (this is the block singing rule):
    stop the action with library message singing action number 1.
```

§176. Climbing.

```
Climbing is an action applying to one thing.
The Climbing action translates into I6 as "Climb".

The specification of the climbing action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§177. Check.

```
Check an actor climbing (this is the block climbing rule):
    stop the action with library message climbing action number 1 for the noun.
```

§178. Sleeping.

```
Sleeping is an action past participle slept and applying to nothing.
The Sleeping action translates into I6 as "Sleep".

The specification of the sleeping action is
"The Standard Rules define this action in only a minimal way, blocking it
with a check rule which stops it in all cases. It exists so that before
or instead rules can be written to make it do interesting things in special
cases. (Or to reconstruct the action as something more substantial, unlist
the block rule and supply carry out and report rules, together perhaps
with some further check rules.)"
```

§179. Check.

```
Check an actor sleeping (this is the block sleeping rule):
    stop the action with library message sleeping action number 1.
```

§**180. Out of world actions.** We start with a brace of actions which control the (virtual) hardware of the virtual machine: restore, save, quit, restart, verify, and transcript on and off. All of these are implemented at the I6 level where, in fact, they are delegated quickly to assembly language instructions for whichever is the current VM: so these are close to the metal, as they say.

```
Section SR4/9 - Standard actions which happen out of world

Quitting the game is an action out of world and applying to nothing.
The quitting the game action translates into I6 as "Quit".

The quit the game rule is listed in the carry out quitting the game rulebook.
The quit the game rule translates into I6 as "QUIT_THE_GAME_R".

Saving the game is an action out of world and applying to nothing.
The saving the game action translates into I6 as "Save".

The save the game rule is listed in the carry out saving the game rulebook.
The save the game rule translates into I6 as "SAVE_THE_GAME_R".

Restoring the game is an action out of world and applying to nothing.
The restoring the game action translates into I6 as "Restore".

The restore the game rule is listed in the carry out restoring the game rulebook.
The restore the game rule translates into I6 as "RESTORE_THE_GAME_R".

Restarting the game is an action out of world and applying to nothing.
The restarting the game action translates into I6 as "Restart".

The restart the game rule is listed in the carry out restarting the game rulebook.
The restart the game rule translates into I6 as "RESTART_THE_GAME_R".

Verifying the story file is an action out of world, with past participle verified
    and applying to nothing.
The verifying the story file action translates into I6 as "Verify".

The verify the story file rule is listed in the carry out verifying the story file rulebook.
The verify the story file rule translates into I6 as "VERIFY_THE_STORY_FILE_R".

Switching the story transcript on is an action out of world and applying to nothing.
The switching the story transcript on action translates into I6 as "ScriptOn".

The switch the story transcript on rule is listed in the carry out switching the story
    transcript on rulebook.
The switch the story transcript on rule translates into I6 as "SWITCH_TRANSCRIPT_ON_R".

Switching the story transcript off is an action out of world and applying to nothing.
The switching the story transcript off action translates into I6 as "ScriptOff".

The switch the story transcript off rule is listed in the carry out switching the story
    transcript off rulebook.
The switch the story transcript off rule translates into I6 as "SWITCH_TRANSCRIPT_OFF_R".
```

§**181.** The VERSION command is not quite so close to the metal – it is implemented in I6, at the end of the day – but it does involve reading the bytes of the story file header, so it needs to take quite different forms for the different formats being compiled to.

```
Requesting the story file version is an action out of world and applying to nothing.
The requesting the story file version action translates into I6 as "Version".

The announce the story file version rule is listed in the carry out requesting the story
    file version rulebook.
The announce the story file version rule translates into I6 as "ANNOUNCE_STORY_FILE_VERSION_R".
```

§**182.** There's really no very good reason why we provide the out-of-world command SCORE but not (say) TIME, or any one of dozens of other traditional what's-my-status commands: DIAGNOSE, say, or PLACES. But we are conservative on this; it's easy for users or extensions to provide these verbs if they want them, and they are not always appropriate for every project. Even SCORE is questionable, but its removal would be a gesture too far.

```
Requesting the score is an action out of world and applying to nothing.
The requesting the score action translates into I6 as "Score".

The announce the score rule is listed in the carry out requesting the score rulebook.
The announce the score rule translates into I6 as "ANNOUNCE_SCORE_R".
```

§**183.** It's perhaps clumsy to have three actions for switching the style of room description, but this accords with I6 custom (and Infocom's, for that matter), and does no harm.

```
Preferring abbreviated room descriptions is an action out of world and applying to nothing.
The preferring abbreviated room descriptions action translates into I6 as "LMode3".

The prefer abbreviated room descriptions rule is listed in the carry out preferring
    abbreviated room descriptions rulebook.
The prefer abbreviated room descriptions rule translates into I6 as "PREFER_ABBREVIATED_R".

The standard report preferring abbreviated room descriptions rule is listed in the
    report preferring abbreviated room descriptions rulebook.
The standard report preferring abbreviated room descriptions rule translates into
    I6 as "REP_PREFER_ABBREVIATED_R".

Preferring unabbreviated room descriptions is an action out of world and applying to nothing.
The preferring unabbreviated room descriptions action translates into I6 as "LMode2".

The prefer unabbreviated room descriptions rule is listed in the carry out preferring
    unabbreviated room descriptions rulebook.
The prefer unabbreviated room descriptions rule translates into I6 as "PREFER_UNABBREVIATED_R".

The standard report preferring unabbreviated room descriptions rule is listed in the
    report preferring unabbreviated room descriptions rulebook.
The standard report preferring unabbreviated room descriptions rule translates into
    I6 as "REP_PREFER_UNABBREVIATED_R".

Preferring sometimes abbreviated room descriptions is an action out of world and
    applying to nothing.
The preferring sometimes abbreviated room descriptions action translates into I6 as "LMode1".

The prefer sometimes abbreviated room descriptions rule is listed in the carry out
    preferring sometimes abbreviated room descriptions rulebook.
The prefer sometimes abbreviated room descriptions rule translates into I6 as
    "PREFER_SOMETIMES_ABBREVIATED_R".

The standard report preferring sometimes abbreviated room descriptions rule is listed
    in the report preferring sometimes abbreviated room descriptions rulebook.
The standard report preferring sometimes abbreviated room descriptions rule translates
    into I6 as "REP_PREFER_SOMETIMES_ABBR_R".
```

§**184.** Similarly, two different actions handle "notify" and "notify off".

```
Switching score notification on is an action out of world and applying to nothing.
The switching score notification on action translates into I6 as "NotifyOn".

The switch score notification on rule is listed in the carry out switching score
    notification on rulebook.
The switch score notification on rule translates into I6 as "SWITCH_SCORE_NOTIFY_ON_R".

The standard report switching score notification on rule is listed in the report
    switching score notification on rulebook.
The standard report switching score notification on rule translates into
    I6 as "REP_SWITCH_NOTIFY_ON_R".

Switching score notification off is an action out of world and applying to nothing.
The switching score notification off action translates into I6 as "NotifyOff".

The switch score notification off rule is listed in the carry out switching score
    notification off rulebook.
The switch score notification off rule translates into I6 as "SWITCH_SCORE_NOTIFY_OFF_R".

The standard report switching score notification off rule is listed in the report
    switching score notification off rulebook.
The standard report switching score notification off rule translates into
    I6 as "REP_SWITCH_NOTIFY_OFF_R".
```

§**185.** Lastly, the "pronouns" verb, which is perhaps more often used by people debugging the I6 parser than by actual players.

```
Requesting the pronoun meanings is an action out of world and applying to nothing.
The requesting the pronoun meanings action translates into I6 as "Pronouns".

The announce the pronoun meanings rule is listed in the carry out requesting the
    pronoun meanings rulebook.
The announce the pronoun meanings rule translates into I6 as "ANNOUNCE_PRONOUN_MEANINGS_R".
```

§**186. Miscellaneous Grammar Tokens.** There's only one of these, at present.

```
The understand token a time period translates into I6 as "RELATIVE_TIME_TOKEN".
```

§187. Grammar.

```
Section SR4/10 - Grammar

Understand "take [things]" as taking.
Understand "take off [something]" as taking off.
Understand "take [things inside] from [something]" as removing it from.
Understand "take [things inside] off [something]" as removing it from.
Understand "take inventory" as taking inventory.
Understand the commands "carry" and "hold" as "take".

Understand "get out/off/up" as exiting.
Understand "get [things]" as taking.
Understand "get in/into/on/onto [something]" as entering.
Understand "get off [something]" as getting off.
Understand "get [things inside] from [something]" as removing it from.

Understand "pick up [things]" or "pick [things] up" as taking.

Understand "stand" or "stand up" as exiting.
Understand "stand on [something]" as entering.

Understand "remove [something preferably held]" as taking off.
Understand "remove [things inside] from [something]" as removing it from.

Understand "shed [something preferably held]" as taking off.
Understand the commands "doff" and "disrobe" as "shed".

Understand "wear [something preferably held]" as wearing.
Understand the command "don" as "wear".

Understand "put [other things] in/inside/into [something]" as inserting it into.
Understand "put [other things] on/onto [something]" as putting it on.
Understand "put on [something preferably held]" as wearing.
Understand "put down [things preferably held]" or "put [things preferably held] down" as dropping.

Understand "insert [other things] in/into [something]" as inserting it into.

Understand "drop [things preferably held]" as dropping.
Understand "drop [other things] in/into/down [something]" as inserting it into.
Understand "drop [other things] on/onto [something]" as putting it on.
Understand "drop [something preferably held] at/against/on/onto [something]" as throwing it at.
Understand the commands "throw" and "discard" as "drop".

Understand "give [something preferably held] to [someone]" as giving it to.
Understand "give [someone] [something preferably held]" as giving it to (with nouns reversed).
Understand the commands "pay" and "offer" and "feed" as "give".

Understand "show [someone] [something preferably held]" as showing it to (with nouns reversed).
Understand "show [something preferably held] to [someone]" as showing it to.
Understand the commands "present" and "display" as "show".

Understand "go" as going.
Understand "go [direction]" as going.
Understand "go [something]" as entering.
Understand "go into/in/inside/through [something]" as entering.
Understand the commands "walk" and "run" as "go".

Understand "inventory" as taking inventory.
Understand the commands "i" and "inv" as "inventory".

Understand "look" as looking.
Understand "look at [something]" as examining.
Understand "look inside/in/into/through [something]" as searching.
Understand "look under [something]" as looking under.
```

```
Understand "look up [text] in [something]" as consulting it about (with nouns reversed).
Understand the command "l" as "look".

Understand "consult [something] on/about [text]" as consulting it about.

Understand "open [something]" as opening.
Understand "open [something] with [something preferably held]" as unlocking it with.
Understand the commands "unwrap", "uncover" as "open".

Understand "close [something]" as closing.
Understand "close up [something]" as closing.
Understand "close off [something]" as switching off.
Understand the commands "shut" and "cover" as "close".

Understand "enter [something]" as entering.
Understand the command "cross" as "enter".

Understand "sit on top of [something]" as entering.
Understand "sit on/in/inside [something]" as entering.

Understand "exit" as exiting.
Understand the commands "leave" and "out" as "exit".

Understand "examine [something]" as examining.
Understand the commands "x", "watch", "describe" and "check" as "examine".

Understand "read [something]" as examining.
Understand "read about [text] in [something]" as consulting it about (with nouns reversed).
Understand "read [text] in [something]" as consulting it about (with nouns reversed).

Understand "yes" as saying yes.
Understand the command "y" as "yes".

Understand "no" as saying no.

Understand "sorry" as saying sorry.

Understand "bother" as swearing mildly.
```

...and so on...

```
Understand "search [something]" as searching.

Understand "wave" as waving hands.

Understand "wave [something]" as waving.

Understand "set [something] to [text]" as setting it to.
Understand the command "adjust" as "set".

Understand "pull [something]" as pulling.
Understand the command "drag" as "pull".

Understand "push [something]" as pushing.
Understand "push [something] [direction]" or "push [something] to [direction]" as pushing it to.
Understand the commands "move", "shift", "clear" and "press" as "push".

Understand "turn [something]" as turning.
Understand "turn [something] on" or "turn on [something]" as switching on.
Understand "turn [something] off" or "turn off [something]" as switching off.
Understand the commands "rotate", "twist", "unscrew" and "screw" as "turn".

Understand "switch [something]" or "switch on [something]" or "switch [something] on" as
    switching on.
Understand "switch [something] off" or "switch off [something]" as switching off.

Understand "lock [something] with [something preferably held]" as locking it with.

Understand "unlock [something] with [something preferably held]" as unlocking it with.

Understand "attack [something]" as attacking.
```

```
Understand the commands "break", "smash", "hit", "fight", "torture", "wreck", "crack", "destroy",
    "murder", "kill", "punch" and "thump" as "attack".

Understand "wait" as waiting.
Understand the command "z" as "wait".

Understand "answer [text] to [someone]" as answering it that (with nouns reversed).
Understand the commands "say", "shout" and "speak" as "answer".

Understand "tell [someone] about [text]" as telling it about.

Understand "ask [someone] about [text]" as asking it about.
Understand "ask [someone] for [something]" as asking it for.

Understand "eat [something preferably held]" as eating.

Understand "sleep" as sleeping.
Understand the command "nap" as "sleep".

Understand "sing" as singing.

Understand "climb [something]" or "climb up/over [something]" as climbing.
Understand the command "scale" as "climb".

Understand "buy [something]" as buying.
Understand the command "purchase" as "buy".

Understand "squeeze [something]" as squeezing.
Understand the command "squash" as "squeeze".

Understand "swing [something]" or "swing on [something]" as swinging.

Understand "wake" or "wake up" as waking up.
Understand "wake [someone]" or "wake [someone] up" or "wake up [someone]" as waking.
Understand the commands "awake" and "awaken" as "wake".

Understand "kiss [someone]" as kissing.
Understand the commands "embrace" and "hug" as "kiss".

Understand "think" as thinking.

Understand "smell" as smelling.
Understand "smell [something]" as smelling.
Understand the command "sniff" as "smell".

Understand "listen" as listening.
Understand "hear [something]" as listening.
Understand "listen to [something]" as listening.

Understand "taste [something]" as tasting.

Understand "touch [something]" as touching.
Understand the command "feel" as "touch".

Understand "rub [something]" as rubbing.
Understand the commands "shine", "polish", "sweep", "clean", "dust", "wipe" and "scrub" as "rub".

Understand "tie [something] to [something]" as tying it to.
Understand the commands "attach", "fix" and "fasten" as "tie".

Understand "burn [something]" as burning.
Understand the command "light" as "burn".

Understand "drink [something]" as drinking.
Understand the commands "swallow" and "sip" as "drink".

Understand "cut [something]" as cutting.
Understand the commands "slice", "prune" and "chop" as "cut".

Understand "jump" as jumping.
Understand the commands "skip" and "hop" as "jump".
```

```
Understand "score" as requesting the score.
Understand "quit" or "q" as quitting the game.
Understand "save" as saving the game.
Understand "restart" as restarting the game.
Understand "restore" as restoring the game.
Understand "verify" as verifying the story file.
Understand "version" as requesting the story file version.
Understand "script" or "script on" or "transcript" or "transcript on" as switching the story
    transcript on.
Understand "script off" or "transcript off" as switching the story transcript off.
Understand "superbrief" or "short" as preferring abbreviated room descriptions.
Understand "verbose" or "long" as preferring unabbreviated room descriptions.
Understand "brief" or "normal" as preferring sometimes abbreviated room descriptions.
Understand "nouns" or "pronouns" as requesting the pronoun meanings.
Understand "notify" or "notify on" as switching score notification on.
Understand "notify off" as switching score notification off.
```

SR5 - Phrase Definitions A/sr5

Purpose

The phrases making up the Inform language, and in terms of which all other phrases and rules are defined; and the final sign-off of the Standard Rules extension, including its minimal documentation.

---

§**1.** Our last task is to create the phrases: more or less all of them, but that does need a little qualification. NI has no phrase definitions built in, but it does contain assumptions about how "say ...", "repeat ...", "let ...", "otherwise ..." and "end ..." will behave when defined: we would not be allowed to call these something else, or redefine them in fundamentally different ways. Apart from that, we are more or less free.

Most of these phrases are defined in terms of I6 code, using the `(-` and `-)` notation – it would be too cumbersome to use the "... translates into I6 as ..." verb for this, too. The fact that phrases are not so much translated as transliterated was one source of early criticism of Inform 7. Phrases appeared to have very simplistic definitions, with the natural language simply being a verbose description of obviously equivalent I6 code. However, the simplicity is misleading, because the definitions below tend to conceal where the complexity of the translation process suddenly increases. If the preamble includes "(c - condition)", and the definition includes the expansion `{c}`, then the text forming c is translated in a way much more profound than any simple substitution process could describe. Type-checking also complicates the code produced below, since NI automatically generates the code needed to perform run-time type checking at any point where doubt remains as to the phrase definition which must be used.

§**2. Say phrases.** We begin with saying phrases: the very first phrase to exist is the one printing a single piece of static text, which seems only appropriate for an IF system. But in fact this is only one of a family of similar phrases. For every kind of value K which can be printed out, we have to define "say (something - K)": this first phrase is that definition for the case where K is "text". (Well: strictly speaking, it's only true that we define this for each atomic K and also for each KOV constructor. Thus "list of K_1" and "list of K_2" are each said by the same phrase, "say (X - list of values)", which is viable since lists store their types at run-time. This ensures that the number of "say (something - K)" doesn't skyrocket with all of the possibilities for lists, lists of lists and so on. And similarly we only make one "say (X - object)", not a whole pile of definitions for "say (X - room)", "say (X - direction)" and so on for each other kind.)

We have this plethora of alternative "say (something - K)" definitions in order to give the type-checking machinery options as to which to use on any given value. But the fact that all these alternative definitions exist is not altogether obvious, because they are hidden in no fewer than three ways:

(1) Very few of them are defined below, because most are defined instead by the data type interpreter, either using definitions tabulated in the `Types.i6t` template file (for built-in KOVS like "table" or "rule"), or when new KOVS are created (when the user creates a unit or a new enumerated KOV).

(2) The Phrasebook page of the index covers all of them with the generic entry: say "[*a value of some sort*]".

(3) Invocation lists built during parsing, which normally hold all possible interpretations of text as a phrase, are optimised a little so that in clear-cut cases (which is almost all cases, in practice) they will only ever contain one "say (something - K)" possibility. For instance, if "say 23" is parsed, the invocation list will not bother to store "say (something - text)" as one of the possibilities, because manifestly 23 is not a piece of text.

Anyway, here are three of the four cases of "say (something - K)" to be created in the Standard Rules rather than elsewhere.

```
Part SR5 - Phrasebook

Section SR5/1/1 - Saying - Values

To say (something - text)
    (documented at ph_say):
    (- print (PrintText) {something}; -).
To say (something - number):
    (- print (say__n={something}); -).
```

§**3.** Note that emitting a Unicode character requires different code on the Z-machine to Glulx; we have to handle this within I6 conditional compilation blocks because neither syntax will compile when I6 is compiling for the other VM. It would be tidier to abstract this with a function call, but it would cost a function call.

```
To say (ch - unicode-character) -- running on:
    (- #ifdef TARGET_ZCODE; @print_unicode {ch};
    #ifnot; if (unicode_gestalt_ok) glk_put_char_uni({ch}); else print "?"; #endif; -).
```

§**4.** Three little grace-notes for printing values: we can have numbers or times of day "in words", rather than given as digits, and we can produce an optional "s" where a number not equal to 1 has recently been printed. This is how "You see [X] balloon[s]." is handled: the printing of the value X sets the I6 variable `say__n` as a side-effect (see definition above) and the routine handling "[s]" looks at this variable to see whether to print an "s" or not.

```
To say (something - number) in words
    (documented at ph_sayn):
    (- print (number) say__n=({something}); -).
To say (something - time) in words:
    (- print (PrintTimeOfDayEnglish) {something}; -).
To say s
    (documented at ph_sayn):
    (- STextSubstitution(); -).
```

§**5.** Now we come to the fourth and last of the "say (something - *K*)" definitions in the SR, followed by six close variations. Note that say phrases are case sensitive on the first word, so that "to say a something" and "to say A something" are different.

A curiosity of the original I6 design, arising I think mostly from the need to save property memory in *Curses* (1993), the work of IF for which Inform 1 had been created, is that it lacks the `print (A) ...` syntax to match the other forms. The omission is made good by using a routine in the I6 library instead.

```
Section SR5/1/2 - Saying - Names with articles

To say (something - object):
    (- print (name) {something}; -).
To say a (something - object)
    (documented at ph_saya):
    (- print (a) {something}; -).
To say an (something - object)
    (documented at ph_saya):
    (- print (a) {something}; -).
To say A (something - object):
    (- CIndefArt({something}); -).
To say An (something - object):
    (- CIndefArt({something}); -).
To say the (something - object):
    (- print (the) {something}; -).
To say The (something - object):
    (- print (The) {something}; -).
```

§**6.** Now for "[if ...]", which expands into a rather assembly-language-like usage of `jump` statements, I6's form of goto. For instance, the text "[if the score is 10]It's ten![otherwise]It's not ten, alas." compiles thus:

```
if (~~(score == 10)) jump L_Say3;
    ...
jump L_SayX2; .L_Say3;
    ...
.L_Say4; .L_SayX2;
```

Though labels actually have local namespaces in I6 routines, we use globally unique labels throughout the whole program: compiling the same phrase again would involve say labels 5 and 6 and "say exit" label 3. This example text demonstrates the reason we `jump` about, rather than making use of `if... else...` and bracing groups of statements: it is legal in I7 either to conclude with or to omit the "[end if]". (If statements in I6 compile to jump instructions in any event, and on our virtual machines there is no speed penalty for branches.) We also need the same definitions to accommodate what amounts to a switch statement. The trickier text "[if the score is 10]It's ten![otherwise if the score is 8]It's eight?[otherwise]It's not ten, alas." comes out as:

```
if (~~(score == 10)) jump L_Say5;
    ...
jump L_SayX3; .L_Say5; if (~~(score == 8)) jump L_Say6;
    ...
jump L_SayX3; .L_Say6;
    ...
.L_Say7; .L_SayX3;
```

In either form of the construct, control passes into at most one of the pieces of text. The terminal labels (the two on the final line) are automatically generated; often – when there is a simple "otherwise" or "end if" to conclude the construct – they are not needed, but labels are quick to process in I6, are soon discarded from I6's memory when not needed any more, and compile no code.

We assume in each case that the next say label number to be free is always the start of the next block, and that the next say exit label number is always the one at the end of the current construct. This is true because NI does not allow "say if" to be nested.

(The use of `{-erase}` below only tidies up the white space to set out the compiled I6 code neatly, and has no effect on the eventual story file.)

```
Section SR5/1/3 - Saying - Say if and otherwise

To say if (c - condition)
    (documented at ph_sayif): (- {-erase}
    if (~~({c})) jump {-next-label:Say};
        -).
To say unless (c - condition): (- {-erase}
    if ({c}) jump {-next-label:Say};
        -).
To say end if: (- {-erase}
    .{-label:Say}; .{-label:SayX};
        -).
To say end unless: (- {-erase}
    .{-label:Say}; .{-label:SayX};
        -).
To say otherwise/else if (c - condition): (- {-erase}
    jump {-next-label:SayX}; .{-label:Say}; if (~~({c})) jump {-next-label:Say};
        -).
To say otherwise/else unless (c - condition): (- {-erase}
    jump {-next-label:SayX}; .{-label:Say}; if ({c}) jump {-next-label:Say};
        -).
To say otherwise: (- {-erase}
    jump {-next-label:SayX}; .{-label:Say};
        -).
To say else: (- {-erase}
    jump {-next-label:SayX}; .{-label:Say};
        -).
```

§**7.** The other control structure: the random variations form of saying. This part of the Standard Rules was in effect contributed by the community: it reimplements a form of Jon Ingold's former extension Text Variations, which itself built on code going back to the days of I6.

The head phrase here has one of the most complicated definitions in the SR, but is actually documented fairly explicitly in the *Extensions* chapter of *Writing with Inform*, so we won't repeat all that here. Essentially it uses its own allocated cell of storage in an array to remember a state between uses, and compiles as a switch statement based on the current state.

```
Section SR5/1/4 - Saying - Say one of

To say one of -- beginning say_one_of (documented at ph_sayoneof):
    (- {-allocate-storage:say_one_of}I7_ST_say_one_of-->{-counter:say_one_of} =
    {-final-segment-marker}(I7_ST_say_one_of-->{-counter:say_one_of}, {-segment-count});
    switch((I7_ST_say_one_of-->{-advance-counter:say_one_of})%({-segment-count}+1)-1) {-open-brace}
        0: -).
To say or -- continuing say_one_of:
    (- @nop; {-segment-count}: -).
To say purely at random -- ending say_one_of with marker I7_SOO_PAR:
    (- {-close-brace} -).
To say at random -- ending say_one_of with marker I7_SOO_RAN:
```

```
    (- {-close-brace} -).
To say sticky random -- ending say_one_of with marker I7_SOO_STI:
    (- {-close-brace} -).
To say as decreasingly likely outcomes -- ending say_one_of with marker I7_SOO_TAP:
    (- {-close-brace} -).
To say in random order -- ending say_one_of with marker I7_SOO_SHU:
    (- {-close-brace} -).
To say cycling -- ending say_one_of with marker I7_SOO_CYC:
    (- {-close-brace} -).
To say stopping -- ending say_one_of with marker I7_SOO_STOP:
    (- {-close-brace} -).
```

§**8.** For an explanation of the paragraph breaking algorithm, see the template file "Printing.i6t".

```
Section SR5/1/5 - Saying - Paragraph control

To say line break -- running on
    (documented at ph_lbreak):
    (- new_line; -).
To say no line break -- running on: do nothing.
To say conditional paragraph break -- running on:
    (- DivideParagraphPoint(); -).
To say command clarification break -- running on:
    (- CommandClarificationBreak(); -).
To say paragraph break -- running on:
    (- DivideParagraphPoint(); new_line; -).
To say run paragraph on -- running on:
    (- RunParagraphOn(); -).
To say run paragraph on with special look spacing -- running on:
    (- SpecialLookSpacingBreak(); -).
To decide if a paragraph break is pending:
    (- (say__p) -).
```

§**9.** Now some text substitutions which are the equivalent of escape characters. (In double-quoted I6 text, the notation for a literal quotation mark is a tilde ~.)

```
Section SR5/1/6 - Saying - Special characters

To say bracket -- running on:
    (- print "["; -).
To say close bracket -- running on:
    (- print "]"; -).
To say apostrophe/' -- running on:
    (- print "'"; -).
To say quotation mark -- running on:
    (- print "~"; -).
```

§**10.** Now some visual effects, which may or may not be rendered the way the user hopes: that's partly up to the virtual machine, unfortunately.

```
Section SR5/1/7 - Saying - Fonts and visual effects

To say bold type -- running on
    (documented at ph_types):
    (- style bold; -).
To say italic type -- running on:
    (- style underline; -).
To say roman type -- running on:
    (- style roman; -).
To say fixed letter spacing -- running on:
    (- font off; -).
To say variable letter spacing -- running on:
    (- font on; -).
To display the boxed quotation (Q - boxed-quotation)
    (documented at ph_boxed):
    (- DisplayBoxedQuotation({Q}); -).
```

§**11.** And now some oddball special texts which must sometimes be said.

```
Section SR5/1/8 - Saying - Some built-in texts

To say the/-- banner text
    (documented at act_banner):
    (- Banner(); -).
To say the/-- list of extension credits
    (documented at ph_extcr):
    (- ShowExtensionVersions(); -).
To say the/-- complete list of extension credits:
    (- ShowFullExtensionVersions(); -).
To say the/-- player's surroundings
    (documented at ph_surrounds):
    (- SL_Location(); -).
```

§**12. Using the list-writer.** The I7 list-writer resembles the old I6 library one, but has been reimplemented in a more general way: see the template file "ListWriter.i6t". The following is the main routine for listing:

```
Section SR5/1/9 - Saying - Saying lists of things

To list the contents of (O - an object),
    with newlines,
    indented,
    giving inventory information,
    as a sentence,
    including contents,
    including all contents,
    tersely,
    giving brief inventory information,
    using the definite article,
    listing marked items only,
    prefacing with is/are,
```

```
    not listing concealed items,
    suppressing all articles
    and/or with extra indentation
    (documented at ph_list):
    (- WriteListFrom(child({O}), {phrase options}); -).
To say contents of (O - an object):
    list the contents of O, as a sentence.
To say the contents of (O - an object):
    list the contents of O, as a sentence, using the definite article.
```

§**13. Text substitutions using the list-writer.** These all look (and are) repetitive. We want to avoid passing a description value to some routine, because that's tricky if the description needs to refer to a value local to the current stack frame. (There are ways round that, but it minimises nuisance to avoid the need.) So we mark out the set of objects matching by giving them, and only them, the `workflag2` attribute.

```
To say a list of (OS - description):
    (- @push subst__v;
        objectloop (subst__v ofclass Object) if ({-bind-variable:OS})
        give subst__v workflag2; else give subst__v ~workflag2;
        WriteListOfMarkedObjects(ENGLISH_BIT);
        @pull subst__v; -).
To say A list of (OS - description):
    (- @push subst__v;
        objectloop (subst__v ofclass Object) if ({-bind-variable:OS})
        give subst__v workflag2; else give subst__v ~workflag2;
        WriteListOfMarkedObjects(ENGLISH_BIT+CFIRSTART_BIT);
        @pull subst__v; -).
To say list of (OS - description):
    (- @push subst__v;
        objectloop (subst__v ofclass Object) if ({-bind-variable:OS})
        give subst__v workflag2; else give subst__v ~workflag2;
        WriteListOfMarkedObjects(ENGLISH_BIT+NOARTICLE_BIT);
        @pull subst__v; -).
To say the list of (OS - description):
    (- @push subst__v;
        objectloop (subst__v ofclass Object) if ({-bind-variable:OS})
        give subst__v workflag2; else give subst__v ~workflag2;
        WriteListOfMarkedObjects(ENGLISH_BIT+DEFART_BIT);
        @pull subst__v; -).
To say The list of (OS - description):
    (- @push subst__v;
        objectloop (subst__v ofclass Object) if ({-bind-variable:OS})
        give subst__v workflag2; else give subst__v ~workflag2;
        WriteListOfMarkedObjects(ENGLISH_BIT+DEFART_BIT+CFIRSTART_BIT);
        @pull subst__v; -).
To say is-are a list of (OS - description):
    (- @push subst__v;
        objectloop (subst__v ofclass Object) if ({-bind-variable:OS})
        give subst__v workflag2; else give subst__v ~workflag2;
        WriteListOfMarkedObjects(ENGLISH_BIT+ISARE_BIT);
        @pull subst__v; -).
To say is-are list of (OS - description):
```

```
    (- @push subst__v;
        objectloop (subst__v ofclass Object) if ({-bind-variable:OS})
        give subst__v workflag2; else give subst__v ~workflag2;
        WriteListOfMarkedObjects(ENGLISH_BIT+ISARE_BIT+NOARTICLE_BIT);
        @pull subst__v; -).
To say is-are the list of (OS - description):
    (- @push subst__v;
        objectloop (subst__v ofclass Object) if ({-bind-variable:OS})
        give subst__v workflag2; else give subst__v ~workflag2;
        WriteListOfMarkedObjects(ENGLISH_BIT+DEFART_BIT+ISARE_BIT);
        @pull subst__v; -).
To say a list of (OS - description) including contents:
    (- @push subst__v;
        objectloop (subst__v ofclass Object) if ({-bind-variable:OS})
        give subst__v workflag2; else give subst__v ~workflag2;
        WriteListOfMarkedObjects(ENGLISH_BIT+RECURSE_BIT+PARTINV_BIT+
            TERSE_BIT+CONCEAL_BIT);
        @pull subst__v; -).
```

§**14. Grouping in the list-writer.** See the specifications of `list_together` and `c_style` in the DM4, which are still broadly accurate.

```
Section SR5/1/10 - Saying - Group in and omit from lists

To group (OS - description) together
    (documented at ph_group):
    (- @push subst__v;
        objectloop (subst__v provides list_together) if ({-bind-variable:OS})
        subst__v.list_together = {-list-together};
        @pull subst__v; -).
To group (OS - description) together giving articles:
    (- @push subst__v;
        objectloop (subst__v provides list_together) if ({-bind-variable:OS})
        subst__v.list_together = {-articled-list-together};
        @pull subst__v; -).
To group (OS - description) together as (T - text):
    (- @push subst__v;
        objectloop (subst__v provides list_together) if ({-bind-variable:OS})
        subst__v.list_together = {T};
        @pull subst__v; -).
To omit contents in listing
    (documented at ph_omit):
    (- c_style = c_style &~ (RECURSE_BIT+FULLINV_BIT+PARTINV_BIT); -).
```

§**15. Lists written but not by the list-writer.** These are lists in the sense of the "list of" kind of value constructor, and they might contain any values, not just objects. They're printed by code in "Lists.i6t".

```
Section SR5/1/11 - Saying - Lists of values

To say (L - a list of values) in brace notation
    (documented at ph_saylv):
    (- LIST_OF_TY_Say({-pointer-to:L}, 1); -).
To say (L - a list of objects) with definite articles:
    (- LIST_OF_TY_Say({-pointer-to:L}, 2); -).
To say (L - a list of objects) with indefinite articles:
    (- LIST_OF_TY_Say({-pointer-to:L}, 3); -).
```

§**16. Filtering in the list-writer.** Something of a last resort, which is intentionally not documented. It's needed by the Standard Rules to tidy up an implementation and avoid I6, but is not an ideal trick and may be dropped in later builds. Recursion occurs when the list-writer descends to the contents of, or items supported by, something it lists. Here we can restrict to just those contents, or supportees, matching a description `D`.

```
Section SR5/1/12 - Saying - Filtering contents - Unindexed

To filter list recursion to (D - description):
    (- list_filter_routine = {D}; -).
To unfilter list recursion:
    (- list_filter_routine = 0; -).
```

§**17. Values and data structures.** To begin with, counting the number of items matching a description.

```
Section SR5/2/1 - Values and data structures - Counting

To decide which number is number of (S - description)
    (documented at ph_numof):
    (- {-number-of:S} -).
To decide which number is number of (S - domain-description)
    (documented at ph_numof):
    (- {-number-of:S} -).
```

§**18.** There are eight arithmetic operations, internally numbered 0 to 7, and given verbal forms below. These are handled unusually in the type-checker because they need to be more polymorphic than most phrases: Inform uses dimension-checking to determine the kind of value resulting. (Thus a height times a number is another height, and so on.)

The totalling code (10) is not structly to do with arithmetic in the same way, but it's needed to flag the phrase for the Inform typechecker's special attention.

```
Section SR5/2/2 - Values and data structures - Arithmetic

To decide which number is (X - number) + (Y - number)
    (arithmetic operation 0)
    (documented at ph_plus): (- ({X}+{Y}) -).
To decide which number is (X - number) - (Y - number)
    (arithmetic operation 1):
    (- ({X}-{Y}) -).
To decide which number is (X - number) * (Y - number)
    (arithmetic operation 2):
    (- ({X}*{Y}{-rescale-times}) -).
To decide which number is (X - number) / (Y - number)
    (arithmetic operation 3):
    (- (IntegerDivide({X}{-rescale-divide},{Y})) -).
To decide which number is (X - number) plus (Y - number)
    (arithmetic operation 0):
    (- ({X}+{Y}) -).
To decide which number is (X - number) minus (Y - number)
    (arithmetic operation 1):
    (- ({X}-{Y}) -).
To decide which number is (X - number) times (Y - number)
    (arithmetic operation 2):
    (- ({X}*{Y}{-rescale-times}) -).
To decide which number is (X - number) multiplied by (Y - number)
    (arithmetic operation 2):
    (- ({X}*{Y}{-rescale-times}) -).
To decide which number is (X - number) divided by (Y - number)
    (arithmetic operation 3):
    (- (IntegerDivide({X}{-rescale-divide},{Y})) -).
To decide which number is remainder after dividing (X - number)
    by (Y - number)
    (arithmetic operation 4):
    (- (IntegerRemainder({X},{Y})) -).
To decide which number is (X - number) to the nearest (Y - number)
    (arithmetic operation 5):
    (- (RoundOffTime({X},{Y})) -).
```

```
To decide which number is the square root of (X - number)
    (arithmetic operation 6):
    (- (SquareRoot({X}{-rescale-root})) -).
To decide which number is the cube root of (X - number)
    (arithmetic operation 7):
    (- (CubeRoot({X}{-rescale-cuberoot})) -).
To decide which number is total (p - property) of (S - description)
    (arithmetic operation 11)
    (documented at ph_total):
    (- {-total-of:S} -).
```

§**19.** The following exists only to convert a condition to a value, and is needed because I7 does not silently cast from one to the other in the way that C would.

```
Section SR5/2/3 - Values and data structures - Truth states

To decide what truth state is whether or not (C - condition)
    (documented at ph_wornot):
    (- ({C}) -).
```

§**20.** Random numbers and random items chosen from sets of objects matching a given description ("a random closed door").

```
Section SR5/2/4 - Values and data structures - Randomness

To decide which object is a/-- random (S - description)
    (documented at ph_randobj):
    (- {-random-of:S} -).
To decide which number is a random number from (N - number) to (M - number)
    (documented at ph_random):
    (- (GenerateRandomNumber({N}, {M})) -).
To decide which number is a random number between (N - number) and (M - number):
    (- (GenerateRandomNumber({N}, {M})) -).
To decide whether a random chance of (N - number) in (M - number) succeeds:
    (- (GenerateRandomNumber(1, {M}) <= {N}) -).
To seed the random-number generator with (N - number):
    (- VM_Seed_RNG({N}); -).
To decide which value is a/-- random (S - domain-description)
    (amended kind of value 1):
    (- {-random-of:S} -).
```

§**21.** The "now" phrase is somewhat unclassifiable, and though it doesn't quite belong here, it doesn't quite belong anywhere else either:

```
Section SR5/2/5 - Values and data structures - Changing stored values

To now (cn - now-condition)
    (documented at ph_now):
    (- {cn} -).
```

§**22.** Assignment is probably the most difficult thing the type-checker has to cope with, since "let" has to work when applied to both unknown names (it creates a new variable) and existing ones (kind of value permitting). Moreover, the assignment itself works differently depending on whether the values are pointers to blocks of data on the heap, needing a deep destruction and a deep copy, or are simple values, so that one can simply overwrite the other. All of this makes the "To let" section here only slightly shorter than John Galsworthy's Forsyte novel of the same name:

```
To let (t - nonexisting variable) be (u - word value)
    (assignment operation)
    (documented at ph_let):
    (- {t} = {u}; -).
To let (t - nonexisting variable) be (u - pointer value)
    (assignment operation):
    (- {-pointer-to:t}={-allocate-storage-for:u};BlkValueCopy({-pointer-to:t},{-pointer-to:u}); -).
To let (t - nonexisting variable) be (u - kind of word value)
    (assignment operation):
    (- {t} = {-default-value-for:u}; -).
To let (t - nonexisting variable) be (u - kind of pointer value)
    (assignment operation):
    (- {-pointer-to:t} = {-allocate-storage-for:u}; -).
To let (t - existing variable) be (u - assignable-value)
    (assignment operation):
    (- {-assignment}; -).
```

§**23.** These are not strictly speaking assignments; the value placed in `t` is found by solving the equation `Q`. This does require special typechecking, but of a different kind to that requested by "(assignment operation)".

```
To let (t - existing variable) be given by (Q - equation-name):
    (- {-solve-equation}; -).
To let (t - nonexisting variable) be given by (Q - equation-name):
    (- {-solve-equation}; -).
```

§**24.** There are five sorts of storage in I7 at present: local variables (i.e., "let" and "called" variables, together with parameters in "To..." preambles), global variables, table entries, property values and list entries. Objects are not a form of storage in this sense. The reason we handle "change O to X" specially where O is an object is that it's legal to "change O to open" if O is a container or door, or to "change O to 20kg" if O has a weight, for instance. The meaning then is to do with the transference of a property, not an assignment. This requires careful disambiguation, since O may be, e.g., a global variable whose kind of value is "object".

```
To change (p - property-value) to (w - assignable-value)
    (assignment operation)
    (documented at ph_changev):
    (- {-assignment}; -).
To change (gv - global variable) to (w - assignable-value)
    (assignment operation)
    (documented at ph_change):
    (- {-assignment}; -).
To change (tr - table-reference) to (w - assignable-value)
    (assignment operation):
    (- {-assignment}; -).
To change (le - list-entry) to (w - assignable-value)
```

```
    (assignment operation):
    (- {-assignment}; -).
To change (o - object) to (p - property)
    (assignment operation)
    (documented at ph_changep):
    (- SetEitherOrProperty({o}, {p}, false, {-adjective-definition:p}); -).
To change (o - object) to (w - value)
    (assignment operation):
    (- WriteValueProperty({o},{-convert-adjectival-constants:w},{w}); -).
To change (lv - existing variable) to (w - assignable-value)
    (assignment operation):
    (- {-assignment}; -).
```

§**25. Incrementing and decrementing.** It is not explicit in the following definitions that Inform should typecheck that the values held by these storage objects can be incremented or decremented (as a room, say, cannot, but a number or a height can): Inform nevertheless contains code which does this. Special cases in the type-checker are gradually moving into instructions in phrase preambles, but these haven't done so yet.

```
Section SR5/2/6 - Values and data structures - Increase and decrease

To increase/increment (p - property-value) by (w - value)
    (documented at ph_increase):
    (- Write{p}{-delete},{p}+{w}); -).
To increase/increment (gv - global variable) by (w - value):
    (- {gv} = {gv} + {w}; -).
To increase/increment (lv - existing variable) by (w - value):
    (- {lv} = {lv} + {w}; -).
To increase/increment (tr - table-reference) by (w - value):
    (- {tr}{-delete},2,{w}); -).
To decrease/decrement (p - property-value) by (w - value):
    (- Write{p}{-delete},{p}-{w}); -).
To decrease/decrement (gv - global variable) by (w - value):
    (- {gv} = {gv} - {w}; -).
To decrease/decrement (lv - existing variable) by (w - value):
    (- {lv} = {lv} - {w}; -).
To decrease/decrement (tr - table-reference) by (w - value):
    (- {tr}{-delete},3,{w}); -).
```

§**26. Tables.** And off we go into the world of tables, the code for which is all in "Tables.i6t". Note that changing a table entry is not here: it's above, with the phrases for changing other storage objects.

```
Section SR5/2/8 - Values and data structures - Tables

To decide which number is number of rows in/from (T - table-name)
    (documented at ph_numrows):
    (- TableRows({T}) -).
To decide which number is number of blank rows in/from (T - table-name)
    (documented at ph_numblank):
    (- TableBlankRows({T}) -).
To decide which number is number of filled rows in/from (T - table-name):
    (- TableFilledRows({T}) -).
To decide if there is (TR - table-reference)
    (documented at ph_thereis):
    (- (Exists{-do-not-dereference:TR}) -).
To decide if there is no (TR - table-reference):
    (- (Exists{-do-not-dereference:TR} == false) -).
To delete (tr - table-reference)
    (documented at ph_blankout):
    (- {tr}{-delete},4); -).
To blank out the whole row
    (documented at ph_blankout):
    (- {-require-ctvs}TableBlankOutRow(ct_0, ct_1); -).
To choose a/the/-- row (N - number) in/from (T - table-name)
    (documented at ph_chooserow):
    (- {-require-ctvs}ct_0 = {T}; ct_1 = {N}; -).
To choose a/the/-- row with (TC - table-column) of (w - value) in/from (T - table-name):
    (- {-require-ctvs}ct_0 = {T}; ct_1 = TableRowCorr(ct_0, {TC}, {w}); -).
To choose a/the/-- blank row in/from (T - table-name):
    (- {-require-ctvs}ct_0 = {T}; ct_1 = TableBlankRow(ct_0); -).
To choose a/the/-- random row in/from (T - table-name):
    (- {-require-ctvs}ct_0 = {T}; ct_1 = TableRandomRow(ct_0); -).
```

§**27.** Sorting.

```
Section SR5/2/9 - Values and data structures - Sorting tables

To sort (T - table-name) in random order
    (documented at ph_sort):
    (- TableShuffle({T}); -).
To sort (T - table-name) in (TC - table-column) order:
    (- TableSort({T}, {TC}, 1); -).
To sort (T - table-name) in reverse (TC - table-column) order:
    (- TableSort({T}, {TC}, -1); -).
```

§**28. Indexed text.** As repetitive as the following is, it's much simpler and less prone to possible namespace trouble if we don't define kinds of value for the different structural levels of text (character, word, punctuated word, etc.).

```
Section SR5/2/10 - Values and data structures - Indexed text

To decide what number is the number of characters in (txb - indexed text)
    (documented at ph_numofc):
    (- IT_BlobAccess({-pointer-to:txb}, CHR_BLOB) -).
To decide what indexed text is character number (N - a number) in (txb - indexed text):
    (- IT_GetBlob({-pointer-to-new:INDEXED_TEXT_TY}, {-pointer-to:txb}, {N}, CHR_BLOB) -).
To replace character number (N - a number) in (txb - indexed text)
    with (rtxb - indexed text):
    (- IT_ReplaceBlob(CHR_BLOB, {-pointer-to:txb}, {N}, {-pointer-to:rtxb}); -).

To decide what number is the number of words in (txb - indexed text):
    (- IT_BlobAccess({-pointer-to:txb}, WORD_BLOB) -).
To decide what indexed text is word number (N - a number) in (txb - indexed text):
    (- IT_GetBlob({-pointer-to-new:INDEXED_TEXT_TY}, {-pointer-to:txb}, {N}, WORD_BLOB) -).
To replace word number (N - a number) in (txb - indexed text)
    with (rtxb - indexed text):
    (- IT_ReplaceBlob(WORD_BLOB, {-pointer-to:txb}, {N}, {-pointer-to:rtxb}); -).

To decide what number is the number of punctuated words in (txb - indexed text):
    (- IT_BlobAccess({-pointer-to:txb}, PWORD_BLOB) -).
To decide what indexed text is punctuated word number (N - a number) in (txb - indexed text):
    (- IT_GetBlob({-pointer-to-new:INDEXED_TEXT_TY}, {-pointer-to:txb}, {N}, PWORD_BLOB) -).
To replace punctuated word number (N - a number) in (txb - indexed text)
    with (rtxb - indexed text):
    (- IT_ReplaceBlob(PWORD_BLOB, {-pointer-to:txb}, {N}, {-pointer-to:rtxb}); -).

To decide what number is the number of unpunctuated words in (txb - indexed text):
    (- IT_BlobAccess({-pointer-to:txb}, UWORD_BLOB) -).
To decide what indexed text is unpunctuated word number (N - a number) in (txb - indexed text):
    (- IT_GetBlob({-pointer-to-new:INDEXED_TEXT_TY}, {-pointer-to:txb}, {N}, UWORD_BLOB) -).
To replace unpunctuated word number (N - a number) in (txb - indexed text)
    with (rtxb - indexed text):
    (- IT_ReplaceBlob(UWORD_BLOB, {-pointer-to:txb}, {N}, {-pointer-to:rtxb}); -).

To decide what number is the number of lines in (txb - indexed text):
    (- IT_BlobAccess({-pointer-to:txb}, LINE_BLOB) -).
To decide what indexed text is line number (N - a number) in (txb - indexed text):
    (- IT_GetBlob({-pointer-to-new:INDEXED_TEXT_TY}, {-pointer-to:txb}, {N}, LINE_BLOB) -).
To replace line number (N - a number) in (txb - indexed text) with (rtxb - indexed text):
    (- IT_ReplaceBlob(LINE_BLOB, {-pointer-to:txb}, {N}, {-pointer-to:rtxb}); -).

To decide what number is the number of paragraphs in (txb - indexed text):
    (- IT_BlobAccess({-pointer-to:txb}, PARA_BLOB) -).
To decide what indexed text is paragraph number (N - a number) in (txb - indexed text):
    (- IT_GetBlob({-pointer-to-new:INDEXED_TEXT_TY}, {-pointer-to:txb}, {N}, PARA_BLOB) -).
To replace paragraph number (N - a number) in (txb - indexed text) with (rtxb - indexed text):
    (- IT_ReplaceBlob(PARA_BLOB, {-pointer-to:txb}, {N}, {-pointer-to:rtxb}); -).
```

§**29. Matching text.** A common matching engine is used for matching plain text...

```
Section SR5/2/11 - Values and data structures - Matching text
To decide if (txb - indexed text) exactly matches the text (ftxb - indexed text),
    case insensitively
    (documented at ph_exactm):
    (- IT_Replace_RE(CHR_BLOB,{-pointer-to:txb},{-pointer-to:ftxb},0,{phrase options},1) -).
To decide if (txb - indexed text) matches the text (ftxb - indexed text),
    case insensitively:
    (- IT_Replace_RE(CHR_BLOB,{-pointer-to:txb},{-pointer-to:ftxb},0,{phrase options}) -).
To decide what number is number of times (txb - indexed text) matches the text
    (ftxb - indexed text), case insensitively:
    (- IT_Replace_RE(CHR_BLOB,{-pointer-to:txb},{-pointer-to:ftxb},1,{phrase options}) -).
```

§**30.** ...and for regular expressions, though here we also have access to the exact text which matched (not interesting in the plain text case since it's the same as the search text, up to case at least), and the values of matched subexpressions (which the plain text case doesn't have).

```
To decide if (txb - indexed text) exactly matches the regular expression (ftxb - indexed text),
    case insensitively
    (documented at ph_regexp):
    (- IT_Replace_RE(REGEXP_BLOB,{-pointer-to:txb},{-pointer-to:ftxb},0,{phrase options},1) -).
To decide if (txb - indexed text) matches the regular expression (ftxb - indexed text),
    case insensitively:
    (- IT_Replace_RE(REGEXP_BLOB,{-pointer-to:txb},{-pointer-to:ftxb},0,{phrase options}) -).
To decide what number is number of times (txb - indexed text) matches the regular expression
    (ftxb - indexed text),case insensitively:
    (- IT_Replace_RE(REGEXP_BLOB,{-pointer-to:txb},{-pointer-to:ftxb},1,{phrase options}) -).
To decide what indexed text is text matching regular expression:
    (- IT_RE_GetMatchVar({-pointer-to-new:INDEXED_TEXT_TY}, 0) -).
To decide what indexed text is text matching subexpression (N - a number):
    (- IT_RE_GetMatchVar({-pointer-to-new:INDEXED_TEXT_TY}, {N}) -).
```

§**31. Replacing text.** The same engine, in "RegExp.i6t", handles replacement.

```
Section SR5/2/12 - Values and data structures - Replacing text
To replace the text (ftxb - indexed text) in (txb - indexed text) with (rtxb - indexed text),
    case insensitively
    (documented at ph_replace):
    (- IT_Replace_RE(REGEXP_BLOB, {-pointer-to:txb}, {-pointer-to:ftxb},
        {-pointer-to:rtxb}, {phrase options}); -).
To replace the regular expression (ftxb - indexed text) in (txb - indexed text) with
    (rtxb - indexed text), case insensitively:
    (- IT_Replace_RE(REGEXP_BLOB, {-pointer-to:txb}, {-pointer-to:ftxb},
        {-pointer-to:rtxb}, {phrase options}); -).
To replace the word (ftxb - indexed text) in (txb - indexed text) with
    (rtxb - indexed text):
    (- IT_ReplaceText(WORD_BLOB, {-pointer-to:txb}, {-pointer-to:ftxb}, {-pointer-to:rtxb}); -).
To replace the punctuated word (ftxb - indexed text) in (txb - indexed text)
    with (rtxb - indexed text):
    (- IT_ReplaceText(PWORD_BLOB, {-pointer-to:txb}, {-pointer-to:ftxb}, {-pointer-to:rtxb}); -).
```

§32. Casing of text.

```
Section SR5/2/13 - Values and data structures - Casing of text

To decide what indexed text is (txb - indexed text) in lower case
    (documented at ph_casing):
    (- IT_CharactersToCase({-pointer-to-new:INDEXED_TEXT_TY}, {-pointer-to:txb}, 0) -).
To decide what indexed text is (txb - indexed text) in upper case:
    (- IT_CharactersToCase({-pointer-to-new:INDEXED_TEXT_TY}, {-pointer-to:txb}, 1) -).
To decide what indexed text is (txb - indexed text) in title case:
    (- IT_CharactersToCase({-pointer-to-new:INDEXED_TEXT_TY}, {-pointer-to:txb}, 2) -).
To decide what indexed text is (txb - indexed text) in sentence case:
    (- IT_CharactersToCase({-pointer-to-new:INDEXED_TEXT_TY}, {-pointer-to:txb}, 3) -).
To decide if (txb - indexed text) is in lower case:
    (- IT_CharactersOfCase({-pointer-to:txb}, 0) -).
To decide if (txb - indexed text) is in upper case:
    (- IT_CharactersOfCase({-pointer-to:txb}, 1) -).
```

§**33. Lists.** The following are all for adding and removing values to dynamic lists:

```
Section SR5/2/14 - Values and data structures - Lists

To add (N - value = kov reference 1) to (L - list of values = list of kov marker 1),
    if absent
    (documented at ph_addlist):
    (- LIST_OF_TY_InsertItem({-pointer-to:L}, {N}, 0, 0, {phrase options}); -).
To add (N - value = kov reference 1) at entry (E - number) in
    (L - list of values = list of kov marker 1), if absent:
    (- LIST_OF_TY_InsertItem({-pointer-to:L}, {N}, 1, {E}, {phrase options}); -).
To add (LX - list of values = list of kov reference 1) to
    (L - list of values = list of kov marker 1), if absent:
    (- LIST_OF_TY_AppendList({-pointer-to:L}, {-pointer-to:LX}, 0, 0, {phrase options}); -).
To add (LX - list of values = list of kov reference 1) at entry (E - number)
    in (L - list of values = list of kov marker 1):
    (- LIST_OF_TY_AppendList({-pointer-to:L}, {-pointer-to:LX}, 1, {E}, 0); -).
To remove (N - value = kov reference 1) from (L - list of values = list of kov marker 1),
    if present
    (documented at ph_remlist):
    (- LIST_OF_TY_RemoveValue({-pointer-to:L}, {N}, {phrase options}); -).
To remove (N - list of values = list of kov reference 1) from
    (L - list of values = list of kov marker 1), if present:
    (- LIST_OF_TY_Remove_List({-pointer-to:L}, {-pointer-to:N}, {phrase options}); -).
To remove entry (N - number) from (L - list of values), if present:
    (- LIST_OF_TY_RemoveItemRange({-pointer-to:L}, {N}, {N}, {phrase options}); -).
To remove entries (N - number) to (N2 - number) from (L - list of values), if present:
    (- LIST_OF_TY_RemoveItemRange({-pointer-to:L}, {N}, {N2}, {phrase options}); -).
```

§**34.** Searching a list is implemented in a somewhat crude way at present, and the following syntax may later be replaced with a suitable verb "to be listed in", so that there's no need to imitate.

```
To decide if (N - value = kov reference 1) is listed in
    (L - list of values = list of kov marker 1)
    (documented at ph_islistedin):
    (- (LIST_OF_TY_FindItem({-pointer-to:L}, {N})) -).
To decide if (N - value = kov reference 1) is not listed in
    (L - list of values = list of kov marker 1):
    (- (LIST_OF_TY_FindItem({-pointer-to:L}, {N}) == false) -).
```

§**35.** The following are casts to and from other data types or objects. Lists are not the only I7 way to hold list-like data, because sometimes the memory requirements for dynamic lists are beyond what the virtual machine can sustain.

(a) A description is a representation of a set of objects by means of a predicate (e.g., "open unlocked doors"), and it converts into a list of current members (in creation order), but there's no reverse process.

(b) The multiple object list is a data structure used in the parser when processing commands like TAKE ALL.

```
To decide what list of objects is the list of (D - description)
    (documented at ph_listd):
    (- LIST_OF_TY_Desc({-pointer-to-new:LIST_OF_TY}, {D}) -).
To decide what list of values is the list of (D - domain-description)
    (amended kind of value 0):
    (- LIST_OF_TY_Desc({-pointer-to-new:LIST_OF_TY}, {D}, {-domain-kov:D}) -).
To decide what list of objects is the multiple object list
    (documented at ph_mobjl):
    (- LIST_OF_TY_Mol({-pointer-to-new:LIST_OF_TY}) -).
To alter the multiple object list to (L - list of objects):
    (- LIST_OF_TY_Set_Mol({-pointer-to:L}); -).
```

§**36.** Determining and setting the length:

```
Section SR5/2/15 - Values and data structures - Length of lists

To decide what number is the number of entries in/of (L - a list of values)
    (documented at ph_numel):
    (- LIST_OF_TY_GetLength({-pointer-to:L}) -).
To truncate (L - a list of values) to (N - a number) entries
    (documented at ph_truncate):
    (- LIST_OF_TY_SetLength({-pointer-to:L}, {N}, -1, 1); -).
To truncate (L - a list of values) to the first (N - a number) entries:
    (- LIST_OF_TY_SetLength({-pointer-to:L}, {N}, -1, 1); -).
To truncate (L - a list of values) to the last (N - a number) entries:
    (- LIST_OF_TY_SetLength({-pointer-to:L}, {N}, -1, -1); -).
To extend (L - a list of values) to (N - a number) entries:
    (- LIST_OF_TY_SetLength({-pointer-to:L}, {N}, 1); -).
To change (L - a list of values) to have (N - a number) entries:
    (- LIST_OF_TY_SetLength({-pointer-to:L}, {N}, 0); -).
```

§**37.** Easy but useful list operations:

```
Section SR5/2/16 - Values and data structures - Reversing and rotating lists

To reverse (L - a list of values)
    (documented at ph_revl):
    (- LIST_OF_TY_Reverse({-pointer-to:L}); -).
To rotate (L - a list of values)
    (documented at ph_rotl):
    (- LIST_OF_TY_Rotate({-pointer-to:L}, 0); -).
To rotate (L - a list of values) backwards:
    (- LIST_OF_TY_Rotate({-pointer-to:L}, 1); -).
```

§**38.** Sorting ultimately uses a common sorting mechanism, in "Sort.i6t", which handles both lists and tables.

```
Section SR5/2/17 - Values and data structures - Sorting lists

To sort (L - a list of values)
    (documented at ph_sortl):
    (- LIST_OF_TY_Sort({-pointer-to:L}, 1); -).
To sort (L - a list of values) in reverse order:
    (- LIST_OF_TY_Sort({-pointer-to:L}, -1); -).
To sort (L - a list of values) in random order:
    (- LIST_OF_TY_Sort({-pointer-to:L}, 2); -).
To sort (L - a list of objects) in (P - property) order:
    (- LIST_OF_TY_Sort({-pointer-to:L}, 1, {P}); -).
To sort (L - a list of objects) in reverse (P - property) order:
    (- LIST_OF_TY_Sort({-pointer-to:L}, -1, {P}); -).
```

§**39.** To call use options a "data structure" is a stretch. Still:

```
Section SR5/2/18 - Values and data structures - Use options

To decide whether using the/-- (UO - use-option)
    (documented at ph_testuo):
    (- (TestUseOption({UO})) -).
```

§**40.** Relations are the final data structure given here. In some ways they are the most fundamental of all, but they're not either set or tested by procedural phrases – they lie in the linguistic structure of conditions. So all we have here are the route-finding phrases:

```
Section SR5/2/19 - Values and data structures - Relations

To decide which object is next step via (R - abstract-relation)
    from (O1 - object) to (O2 - object)
    (documented at ph_nextstep):
    (- RelationRouteTo({R},{O1},{O2},false) -).
To decide which number is number of steps via (R - abstract-relation)
    from (O1 - object) to (O2 - object):
    (- RelationRouteTo({R},{O1},{O2},true) -).
```

§**41. Loops and conditionals.** While "unless" is supposed to be exactly like "if" but with the reversed sense of the condition, that isn't quite true. For example, there is no "unless ... then ...": logical it might be, English it is not.

```
Section SR5/3/1 - Loops and conditionals - If and unless

To if (c - condition) then (ph - phrase)
    (documented at ph_if):
    (- if {c} {ph} -).
To if (c - condition) , (ph - phrase):
    (- if {c} {ph} -).
To unless (c - condition) , (ph - phrase):
    (- if (~~{c}) {ph} -).
```

§**42.** It looks as if the definitions below could be abbreviated a little by making more use of the slash ("To else/otherwise if...", say), but in fact the slash isn't compatible with these built-in control structure definitions – or at any rate using it causes a handful of problem messages from the type-checker to be worded badly in some cases.

```
To otherwise if (c - condition):
    (- } else if {c} { -).
To else if (c - condition):
    (- } else if {c} { -).
To otherwise unless (c - condition):
    (- } else if (~~{c}) { -).
To else unless (c - condition):
    (- } else if (~~{c}) { -).
To otherwise (ph - phrase)
    (documented at ph_otherwise):
    (- else {ph} -).
To else (ph - phrase)
    (documented at ph_otherwise):
    (- else {ph} -).
To if (c - condition) begin -- end:
    (- if {c}  -).
To unless (c - condition) begin -- end:
    (- if (~~{c})  -).
```

§**43.** The switch form of "if" is subtly different, and here again "unless" is not allowed in its place.

The begin and end markers here are in a sense bogus, in that the end user isn't supposed to type them: they are inserted automatically in the sentence subtree maker, which converts indentation into block structure.

```
To if (V - word value) is begin -- end:
    (- switch({V})  -).
```

§**44.** After all that, the while loop is simplicity itself. Perhaps the presence of "unless" for "if" argues for a similarly negated form, "until" for "while", but users haven't yet petitioned for this.

```
Section SR5/3/2 - Loops and conditionals - While

To while (c - condition) repeatedly (ph - phrase)
    (documented at ph_while):
    (- while {c} {ph} -).
To while (c - condition) , (ph - phrase):
    (- while {c} {ph} -).
To while (c - condition) begin -- end:
    (- while {c}  -).
```

§**45.** The repeat loop looks like a single construction, but isn't, because the range can be given in four fundamentally different ways (and the loop variable then has a different kind of value accordingly). First, the equivalents of BASIC's `for` loop and of Inform 6's `objectloop`, respectively:

```
Section SR5/3/3 - Loops and conditionals - Repeat

To repeat with (loopvar - nonexisting number variable)
    running from (v - number) to (w - number) begin -- end
    (documented at ph_repeat):
        (- for ({loopvar}={v}: {loopvar}<={w}: {loopvar}++)  -).
To repeat with (loopvar - nonexisting object variable)
    running through (OS - description) begin -- end
    (documented at ph_runthrough):
        (- {-loop-over:OS}  -).
To repeat with (loopvar - nonexisting object variable)
    running through (OS - domain-description) begin -- end
    (documented at ph_runthrough):
        (- {-loop-over-domain:OS}  -).
```

§**46.** The following are all repeats where the range is the set of rows of a table, taken in some order, and the repeat variable – though it does exist – is never specified since the relevant row is instead the one selected during each iteration of the loop.

```
Section SR5/3/4 - Loops and conditionals - Repeat through tables

To repeat through (T - table-name) begin -- end
    (documented at ph_tabrepeat):
    (- {-push-ctvs}
        for ({-ct-v0}={T},{-ct-v1}=1,ct_0={-ct-v0},ct_1={-ct-v1}:
            {-ct-v1}<=TableRows({-ct-v0}):{-ct-v1}++,ct_0={-ct-v0},ct_1={-ct-v1})
            if (TableRowIsBlank(ct_0,ct_1)==false)  -).
To repeat through (T - table-name) in reverse order begin -- end:
    (- {-push-ctvs}
        for ({-ct-v0}={T},{-ct-v1}=TableRows({-ct-v0}),ct_0={-ct-v0},ct_1={-ct-v1}:
            {-ct-v1}>=1:{-ct-v1}--,ct_0={-ct-v0},ct_1={-ct-v1})
            if (TableRowIsBlank(ct_0,ct_1)==false)  -).
To repeat through (T - table-name) in (TC - table-column) order begin -- end:
    (- {-push-ctvs}
        for ({-ct-v0}={T},{-ct-v1}=TableNextRow({-ct-v0},{TC},0,1),ct_0={-ct-v0},ct_1={-ct-v1}:
            {-ct-v1}~=0:
            {-ct-v1}=TableNextRow({-ct-v0},{TC},{-ct-v1},1),ct_0={-ct-v0},ct_1={-ct-v1})  -).
```

```
To repeat through (T - table-name) in reverse (TC - table-column) order begin -- end:
    (- {-push-ctvs}
        for ({-ct-v0}={T},{-ct-v1}=TableNextRow({-ct-v0},{TC},0,-1),ct_0={-ct-v0},ct_1={-ct-v1}:
            {-ct-v1}~=0:
            {-ct-v1}=TableNextRow({-ct-v0},{TC},{-ct-v1},-1),ct_0={-ct-v0},ct_1={-ct-v1})  -).
```

§**47.** Finally, we can repeat through the other dynamic data structure: the list.

```
Section SR5/3/5 - Loops and conditionals - Repeat through lists

To repeat with (loopvar - nonexisting object variable)
    running through (L - list of values) begin -- end:
    (- {-loop-over-list:L}  -).
```

§**48.** The equivalent of `break` or `continue` in C or I6, or of `last` or `next` in Perl. Here "in loop" means "in any of the forms of while or repeat".

```
Section SR5/3/6 - Loops and conditionals - Changing the flow of loops

To break -- in loop (documented at ph_break):
    (- break; -).
To next -- in loop (documented at ph_next):
    (- continue; -).
```

§**49.** The following certainly aren't loops and aren't quite conditionals either, but they reflect the use of rules within rulebooks as being a form of control structure, and they have to be indexed somewhere.

The antique forms "yes" and "no" are now somewhat to be regretted, with "decide yes" and "decide no" being clearer ways to write the same thing. But we seem to be stuck with them.

```
Section SR5/3/7 - Loops and conditionals - Deciding outcomes

To yes
    (documented at ph_yes):
    (- rtrue; -) - in to decide if only.
To decide yes:
    (- rtrue; -) - in to decide if only.
To no:
    (- rfalse; -) - in to decide if only.
To decide no:
    (- rfalse; -) - in to decide if only.
```

§**50.** Note that returning a value has to invoke the type-checker to ensure that the return value matches the kind of value expected. This certainly rejects the phrase if it's used in a definition which isn't meant to be deciding a value at all, so an "in... only" clause is not needed.

```
To decide on (something - value)
    (documented at ph_result):
    (- return {-check-return-type:something}; -).
```

§**51.** "Do nothing" is useful mainly when other syntax has backed us into something clumsy, but it can't be dispensed with. (In the examples, it used to be used when conditions were awkward to negate – if condition, do nothing, otherwise blah blah blah – but the creation of "unless" made it possible to remove most of the "do nothing"s.)

```
Section SR5/3/8 - Loops and conditionals - Stop or go

To do nothing (documented at ph_nothing):
    (- ; -).
To stop (documented at ph_stop):
    (- return; -) - in to only.
```

§**52. Actions, activities and rules.** We begin with the firing off of new actions. The current action runs silently if the I6 global variable `keep_silent` is set, so the result of the definitions below is that one can go into silence mode, using "try silently", but not climb out of it again. This is done because many actions try other actions as part of their normal workings: if we want action X to be tried silently, then any action X itself tries should also be tried silently.

```
Section SR5/4/1 - Actions, activities and rules - Trying actions

To try (doing something - action)
    (documented at ph_try):
    (- {doing something}; -).
To silently try (doing something - action):
    (- @push keep_silent; keep_silent=1; {doing something}; @pull keep_silent; -).
To try silently (doing something - action):
    (- @push keep_silent; keep_silent=1; {doing something}; @pull keep_silent; -).
```

§**53.** The requirements of the current action can be tested. The following may be reimplemented using a verb *to require* at some future point.

```
Section SR5/4/2 - Actions, activities and rules - Action requirements

To decide whether the action requires a touchable noun
    (documented at ph_requires):
    (- (NeedToTouchNoun()) -).
To decide whether the action requires a touchable second noun:
    (- (NeedToTouchSecondNoun()) -).
To decide whether the action requires a carried noun:
    (- (NeedToCarryNoun()) -).
To decide whether the action requires a carried second noun:
    (- (NeedToCarrySecondNoun()) -).
To decide whether the action requires light:
    (- (NeedLightForAction()) -).
```

§**54.** Within the rulebooks to do with an action, returning `true` from a rule is sufficient to stop the rulebook early: there is no need to specify success or failure because that is determined by the rulebook itself. (For instance, if the check taking rules stop for any reason, the action failed; if the after rules stop, it succeeded.) In some rulebooks, notably "instead" and "after", the default is to stop, so that execution reaching the end of the I6 routine for a rule will run into an `rtrue`. "Continue the action" prevents this.

```
Section SR5/4/3 - Actions, activities and rules - Stop or continue

To stop the action
    (documented at ph_stopac):
    (- rtrue; -) - in to only.
To continue the action:
    (- rfalse; -) - in to only.
```

§**55.** Note that we define "try", "silently try" and "try silently" all over again here, but for stored actions rather than actions: NI's type-checking means that it will automatically use whichever definition is appropriate.

```
Section SR5/4/4 - Actions, activities and rules - Stored actions

To decide what stored action is the current action
    (documented at ph_curract):
    (- STORED_ACTION_TY_Current({-pointer-to-new:STORED_ACTION_TY}) -).
To decide what stored action is the action of (A - action):
    (- {A}{-delete}{-delete}, STORED_ACTION_TY_Current({-pointer-to-new:STORED_ACTION_TY})) -).
To try (S - stored action):
    (- STORED_ACTION_TY_Try({S}); -).
To silently try (S - stored action):
    (- STORED_ACTION_TY_Try({S}, true); -).
To try silently (S - stored action):
    (- STORED_ACTION_TY_Try({S}, true); -).
To decide if (act - a stored action) involves (X - an object)
    (documented at ph_involves):
    (- (STORED_ACTION_TY_Involves({-pointer-to:act}, {X})) -).
To decide what action-name is the action-name part of (act - a stored action):
    (- (STORED_ACTION_TY_Part({-pointer-to:act}, 0)) -).
To decide what object is the noun part of (act - a stored action):
    (- (STORED_ACTION_TY_Part({-pointer-to:act}, 1)) -).
To decide what object is the second noun part of (act - a stored action):
    (- (STORED_ACTION_TY_Part({-pointer-to:act}, 2)) -).
To decide what object is the actor part of (act - a stored action):
    (- (STORED_ACTION_TY_Part({-pointer-to:act}, 3)) -).
```

§**56.** Firing off activities:

```
Section SR5/4/5 - Actions, activities and rules - Carrying out activities

To carry out the (A - activity) activity
    (documented at ph_carryout):
    (- CarryOutActivity({A}); -).
To carry out the (A - activity) activity with (O - object):
    (- CarryOutActivity({A}, {O}); -).
```

§**57.** The following circumlocutions are needed because, at present, we can't define adjectives for kinds of value other than "object".

```
[To decide whether (A - activity) activity is going on
    (documented at ph_goingon):
    (- (TestActivity({A})) -).
To decide whether (A - activity) activity is not going on:
    (- (~~(TestActivity({A}))) -).
To decide whether (A - activity) activity is empty:
    (- (ActivityEmpty({A})) -).
To decide whether (A - activity) activity is not empty:
    (- (~~(ActivityEmpty({A}))) -).]
```

§**58.** This is analogous to "continue the action":

```
To continue the activity:
    (- rfalse; -) - in to only.
```

§**59.** Advanced activity phrases: for setting up one's own activities structured around I7 source text. People tend not to use this much, and perhaps that's a good thing, but it does open up possibilities, and it's good for retro-fitting onto extensions to make the more customisable.

```
Section SR5/4/6 - Actions, activities and rules - Advanced activities

To begin the (A - activity) activity
    (documented at ph_beginact):
    (- BeginActivity({A}); -).
To begin the (A - activity) activity with (O - object):
    (- BeginActivity({A}, {O}); -).
To decide whether handling (A - activity) activity:
    (- (~~(ForActivity({A}))) -).
To decide whether handling (A - activity) activity with (O - object):
    (- (~~(ForActivity({A}, {O}))) -).
To end the (A - activity) activity:
    (- EndActivity({A}); -).
To end the (A - activity) activity with (O - object):
    (- EndActivity({A}, {O}); -).
To abandon the (A - activity) activity:
    (- AbandonActivity({A}); -).
To abandon the (A - activity) activity with (O - object):
    (- AbandonActivity({A}, {O}); -).
```

§**60. Rules.** Four different ways to invoke a rule or rulebook:

```
Section SR5/4/7 - Actions, activities and rules - Following rules

To follow (RL - a rule)
    (documented at ph_follow):
    (- FollowRulebook({RL}); -).
To follow (RL - a rule) for (O - object):
    (- FollowRulebook({RL}, {O}, true); -).
To consider (RL - a rule)
    (documented at ph_consider):
    (- ProcessRulebook({RL}); -).
To consider (RL - a rule) for (O - object):
    (- ProcessRulebook({RL}, {O}, true); -).
To abide by (RL - a rule)
    (documented at ph_abide):
    (- if (ProcessRulebook({RL})) rtrue; -) - in to only.
To abide by (RL - a rule) for (O - object):
    (- if (ProcessRulebook({RL}, {O}, true)) rtrue; -) - in to only.
To anonymously abide by (RL - a rule):
    (- if (temporary_value = ProcessRulebook({RL})) {
        if (RulebookSucceeded()) ActRulebookSucceeds(temporary_value);
        else ActRulebookFails(temporary_value);
        rtrue;
    } -) - in to only.
To anonymously abide by (RL - a rule) for (O - object):
    (- if (temporary_value = ProcessRulebook({RL}, {O}, true)) {
        if (RulebookSucceeded()) ActRulebookSucceeds(temporary_value);
        else ActRulebookFails(temporary_value);
        rtrue;
    } -) - in to only.
```

§**61.** Rules return `true` to indicate a decision, which could be either a success or a failure, and optionally may also return a value. If they return `false`, there's no decision.

```
Section SR5/4/8 - Actions, activities and rules - Success and failure

To make no decision: (- rfalse; -) - in to only.
To rule succeeds
    (documented at ph_succeeds):
    (- RulebookSucceeds(); rtrue; -) - in to only.
To rule fails:
    (- RulebookFails(); rtrue; -) - in to only.
To rule succeeds with result (O - a miscellaneous-value):
    (- RulebookSucceeds(true,{O}); rtrue; -) - in to only.
To rule fails with result (O - a miscellaneous-value):
    (- RulebookFails(true,{O}); rtrue; -) - in to only.
To decide if rule succeeded:
    (- (RulebookSucceeded()) -).
To decide if rule succeeded with result (O - miscellaneous-value):
    (- ((RulebookSucceeded()) && (ResultOfRule() == {O})) -).
To decide if rule failed:
    (- (RulebookFailed()) -).
To decide if rule failed with result (O - miscellaneous-value):
```

```
    (- ((RulebookFailed()) && (ResultOfRule() == {0})) -).
To decide which miscellaneous-value is the result of the rule
    (documented at ph_resultr):
    (- (ResultOfRule()) -).
To decide which rulebook-outcome is the outcome of the rulebook
    (documented at ph_outcomer):
    (- (ResultOfRule()) -).
```

§**62.** And lastly the suite of procedural rule tricks, though happily these have been less and less needed as Inform has grown in flexibility. (They incur an appreciable speed penalty at run-time.)

```
Section SR5/4/9 - Actions, activities and rules - Procedural manipulation

To ignore (RL - a rule)
    (documented at ph_ignore):
    (- SuppressRule({RL}); -).
To reinstate (RL - a rule):
    (- ReinstateRule({RL}); -).
To reject the result of (RL - a rule):
    (- DonotuseRule({RL}); -).
To accept the result of (RL - a rule):
    (- DonotuseRule({RL}); -).
To substitute (RL1 - a rule) for (RL2 - a rule):
    (- SubstituteRule({RL1},{RL2}); -).
To restore the original (RL1 - a rule):
    (- SubstituteRule({RL1},{RL1}); -).
To move (RL1 - a rule) to before (RL2 - a rule):
    (- MoveRuleBefore({RL1},{RL2}); -).
To move (RL1 - a rule) to after (RL2 - a rule):
    (- MoveRuleAfter({RL1},{RL2}); -).
```

§**63. The model world.** The score and outcome of play phrases only interface with two I6 library variables, `score` and `deadflag`. The latter takes values 0 (in progress), 1 (lost), 2 (won), or the packed address of a string or routine to print a string which explains in what way the world ended ("You have sailed off into the sunset").

```
Section SR5/5/1 - Model world - Score

To award (some - number) point/points
    (documented at ph_awardp):
    (- score=score+{some}; -).
```

§**64.** Phrase definitions with wordings like "the game is in progress" are a necessary evil. The "is" here is parsed literally, not as the present tense of *to be*, so inflected forms like "the game had been in progress" are not available: nor is there any value "the game" for the subject noun phrase to hold, nor can the relation "in"... and so on. Ideally, we would word all conditional phrases so as to avoid the verbs, but natural language just doesn't work that way.

```
Section SR5/5/2 - Model world - Outcome of play

To end the game in death (documented at ph_end): (- deadflag=1; -).
To end the game in victory: (- deadflag=2; -).
To end the game saying (finale - text): (- deadflag={finale}; -).
To resume the game: (- resurrect_please = true; -).
To decide whether the game is in progress: (- (deadflag==0) -).
To decide whether the game is over: (- (deadflag~=0) -).
To decide whether the game ended in death: (- (deadflag==1) -).
To decide whether the game ended in victory: (- (deadflag==2) -).
```

§**65.** Times of day.

```
Section SR5/5/3 - Model world - Times of day

To decide which number is the minutes part of (t - time)
    (documented at ph_minspart):
    (- ({t}%ONE_HOUR) -).
To decide which number is the hours part of (t - time):
    (- ({t}/ONE_HOUR) -).
```

§**66.** Comparing times of day is inherently odd, because the day is circular. Every 2 PM comes after a 1 PM, but it also comes before another 1 PM. Where do we draw the meridian on this circle? The legal day divides at midnight but for other purposes (daylight savings time, for instance) society often chooses 2 AM as the boundary. Inform uses 4 AM instead as the least probable time through which play continues. (Modulo a 24-hour clock, adding 20 hours is equivalent to subtracting 4 AM from the current time: hence the use of `20*ONE_HOUR` below.) Thus 3:59 AM is after 4:00 AM, the former being at the very end of a day, the latter at the very beginning.

```
To decide if (t - time) is before (t2 - time)
    (documented at ph_timeshift):
    (- ((({t}+20*ONE_HOUR)%(TWENTY_FOUR_HOURS))<(({t2}+20*ONE_HOUR)%(TWENTY_FOUR_HOURS))) -).
To decide if (t - time) is after (t2 - time):
    (- ((({t}+20*ONE_HOUR)%(TWENTY_FOUR_HOURS))>(({t2}+20*ONE_HOUR)%(TWENTY_FOUR_HOURS))) -).
To decide if it is before (t2 - time):
    (- (((the_time+20*ONE_HOUR)%(TWENTY_FOUR_HOURS))<(({t2}+20*ONE_HOUR)%(TWENTY_FOUR_HOURS))) -).
To decide if it is after (t2 - time):
    (- (((the_time+20*ONE_HOUR)%(TWENTY_FOUR_HOURS))>(({t2}+20*ONE_HOUR)%(TWENTY_FOUR_HOURS))) -).
To decide which time is (t - time) before (t2 - time)
    (documented at ph_timeshift):
    (- (({t2}-{t}+TWENTY_FOUR_HOURS)%(TWENTY_FOUR_HOURS)) -).
To decide which time is (t - time) after (t2 - time):
    (- (({t2}+{t}+TWENTY_FOUR_HOURS)%(TWENTY_FOUR_HOURS)) -).
```

§**67.** Durations are in effect casts from “number” to “time”.

```
Section SR5/5/4 - Model world - Durations

To decide which time is (n - number) minutes
    (documented at ph_durations):
    (- (({n})%(TWENTY_FOUR_HOURS)) -).
To decide which time is (n - number) hours:
    (- (({n}*ONE_HOUR)%(TWENTY_FOUR_HOURS)) -).
```

§**68.** Timed events.

```
Section SR5/5/5 - Model world - Timed events

To (R - rule) in (t - number) turn from now
    (documented at ph_future):
    (- SetTimedEvent({R}, {t}+1, 0); -).
To (R - rule) in (t - number) turns from now:
    (- SetTimedEvent({R}, {t}+1, 0); -).
To (R - rule) at (t - time):
    (- SetTimedEvent({R}, {t}, 1); -).
To (R - rule) in (t - time) from now:
    (- SetTimedEvent({R}, (the_time+{t})%(TWENTY_FOUR_HOURS), 1); -).
```

§**69.** Scenes.

```
Section SR5/5/6 - Model world - Scenes

To decide if (sc - scene) has happened:
    (- (scene_endings-->({sc}-1)) -).
To decide if (sc - scene) has not happened:
    (- (scene_endings-->({sc}-1) == 0) -).
To decide if (sc - scene) has ended:
    (- (scene_endings-->({sc}-1) > 1) -).
To decide if (sc - scene) has not ended:
    (- (scene_endings-->({sc}-1) <= 1) -).
```

§**70.** Timing of scenes.

```
Section SR5/5/7 - Model world - Timing of scenes

To decide which time is the time since (sc - scene) began
    (documented at ph_tsince):
    (- (SceneUtility({sc}, 1)) -).
To decide which time is the time when (sc - scene) began:
    (- (SceneUtility({sc}, 2)) -).
To decide which time is the time since (sc - scene) ended:
    (- (SceneUtility({sc}, 3)) -).
To decide which time is the time when (sc - scene) ended:
    (- (SceneUtility({sc}, 4)) -).
```

§**71.** Player's identity and location.

```
Section SR5/5/8 - Model world - Player's identity and location

To change the/-- player to (O - an object)
    (documented at ph_changep):
    (- ChangePlayer({O}); -).
To decide whether in (somewhere - an object):
    (- (WhetherIn({somewhere})) -).
To decide whether in darkness
    (documented at ph_indarkness):
    (- (location==thedark) -).
```

§**72.** Moving and removing things.

```
Section SR5/5/9 - Model world - Moving and removing things

To move (something - object) to (something else - object),
    without printing a room description
    or printing an abbreviated room description
    (documented at ph_move):
    (- MoveObject({something}, {something else}, {phrase options}, false); -).
To remove (something - object) from play
    (documented at ph_remove):
    (- RemoveFromPlay({something}); -).
To move (O - object) backdrop to all (D - description)
    (documented at ph_movebd):
    (- MoveBackdrop({O}, {D}); -).
To update backdrop positions
    (documented at ph_updatebp):
    (- MoveFloatingObjects(); -).
```

§**73.** The map.

```
Section SR5/5/10 - Model world - The map

To decide which room is location of (O - object)
    (documented at ph_locof):
    (- LocationOf({O}) -).
To decide which room is room (D - direction) from/of (R1 - room)
    (documented at ph_roomdirof):
    (- MapConnection({R1},{D}) -).
To decide which door is door (D - direction) from/of (R1 - room)
    (documented at ph_roomdirof):
    (- DoorFrom({R1},{D}) -).
To decide which object is the other side of (D - door) from (R1 - room):
    (- OtherSideOfDoor({D},{R1}) -).
To decide which object is the direction of (D - door) from (R1 - room):
    (- DirectionDoorLeadsIn({D},{R1}) -).
To decide which object is room-or-door (D - direction) from/of (R1 - room):
    (- RoomOrDoorFrom({R1},{D}) -).
To change (D - direction) exit of (R1 - room) to (R2 - room)
    (documented at ph_changex):
    (- AssertMapConnection({R1},{D},{R2}); -).
To change (D - direction) exit of (R1 - room) to nothing/nowhere:
    (- AssertMapConnection({R1},{D},nothing); -).
To decide which room is the front side of (D - object)
    (documented at ph_frontside):
    (- FrontSideOfDoor({D}) -).
To decide which room is the back side of (D - object):
    (- BackSideOfDoor({D}) -).
```

§**74.** Route-finding.

```
Section SR5/5/11 - Model world - Route-finding

To decide which object is best route from (R1 - object) to (R2 - object),
    using doors or using even locked doors
    (documented at ph_bestroute):
    (- MapRouteTo({R1},{R2},0,{phrase options}) -).
To decide which number is number of moves from (R1 - object) to (R2 - object),
    using doors or using even locked doors:
    (- MapRouteTo({R1},{R2},0,{phrase options},true) -).
To decide which object is best route from (R1 - object) to (R2 - object) through
    (RS - description),
    using doors or using even locked doors:
    (- MapRouteTo({R1},{R2},{RS},{phrase options}) -).
To decide which number is number of moves from (R1 - object) to (R2 - object) through
    (RS - description),
    using doors or using even locked doors:
    (- MapRouteTo({R1},{R2},{RS},{phrase options},true) -).
```

§**75.** The object tree.

```
Section SR5/5/12 - Model world - The object tree

To decide which object is holder of (something - object)
    (documented at ph_holder):
    (- (HolderOf({something})) -).
To decide which object is next thing held after (something - object):
    (- (sibling({something})) -).
To decide which object is first thing held by (something - object):
    (- (child({something})) -).
```

§**76. Understanding.** First, asking yes/no questions.

```
Section SR5/6/1 - Understanding - Asking yes/no questions

To decide whether player consents
    (documented at ph_consents):
        (- YesOrNo() -).
```

§**77.** Support for snippets, which are substrings of the player's command.

```
Section SR5/6/2 - Understanding - The player's command

To decide if (S - a snippet) matches (T - a topic)
    (documented at act_reading):
    (- (SnippetMatches({S}, {T})) -).
To decide if (S - a snippet) does not match (T - a topic)
    (documented at act_reading):
    (- (SnippetMatches({S}, {T}) == false) -).
To decide if (S - a snippet) includes (T - a topic):
    (- (matched_text=SnippetIncludes({T},{S})) -).
To decide if (S - a snippet) does not include (T - a topic):
    (- (SnippetIncludes({T},{S})==0) -).
```

§**78.** Changing the player's command.

```
Section SR5/6/3 - Understanding - Changing the player's command

To change the text of the player's command to (txb - indexed text)
    (documented at act_reading):
    (- SetPlayersCommand({-pointer-to:txb}); -).
To replace (S - a snippet) with (T - text):
    (- SpliceSnippet({S}, {T}); -).
To cut (S - a snippet):
    (- SpliceSnippet({S}, 0); -).
To reject the player's command:
    (- RulebookFails(); rtrue; -) - in to only.
```

§**79.** Scope and pronouns.

```
Section SR5/6/4 - Understanding - Scope and pronouns

To place (O - an object) in scope, but not its contents
    (documented at ph_scope):
    (- PlaceInScope({O}, {phrase options}); -).
To place the/-- contents of (O - an object) in scope
    (documented at ph_scope):
    (- ScopeWithin({O}); -).
To set pronouns from possessions of the player:
    (- PronounNoticeHeldObjects(); -).
To set pronouns from (O - an object):
    (- PronounNotice({O}); -).
```

§**80.** The I6 parser errors cry out to be an enumerated kind of value, and probably will be in a later build. The following will then be more concise.

```
Section SR5/6/5 - Understanding - I6 parser errors

To decide whether parser error is didn't understand
    (documented at act_parsererror):
    (- (etype == STUCK_PE) -).
To decide whether parser error is only understood as far as:
    (- (etype == UPTO_PE) -).
To decide whether parser error is didn't understand that number:
    (- (etype == NUMBER_PE) -).
To decide whether parser error is can't see any such thing:
    (- (etype == CANTSEE_PE) -).
To decide whether parser error is said too little:
    (- (etype == TOOLIT_PE) -).
To decide whether parser error is aren't holding that:
    (- (etype == NOTHELD_PE) -).
To decide whether parser error is can't use multiple objects:
    (- (etype == MULTI_PE) -).
To decide whether parser error is can only use multiple objects:
    (- (etype == MMULTI_PE) -).
To decide whether parser error is not sure what it refers to:
    (- (etype == VAGUE_PE) -).
To decide whether parser error is excepted something not included:
    (- (etype == EXCEPT_PE) -).
To decide whether parser error is can only do that to something animate:
    (- (etype == ANIMA_PE) -).
To decide whether parser error is not a verb I recognise:
    (- (etype == VERB_PE) -).
To decide whether parser error is not something you need to refer to:
    (- (etype == SCENERY_PE) -).
To decide whether parser error is can't see it at the moment:
    (- (etype == ITGONE_PE) -).
To decide whether parser error is didn't understand the way that finished:
    (- (etype == JUNKAFTER_PE) -).
To decide whether parser error is not enough of those available:
    (- (etype == TOOFEW_PE) -).
To decide whether parser error is nothing to do:
    (- (etype == NOTHING_PE) -).
To decide whether parser error is I beg your pardon:
    (- (etype == BLANKLINE_PE) -).
To decide whether parser error is noun did not make sense in that context:
    (- (etype == NOTINCONTEXT_PE) -).
```

§**81. Using external resources.** The following all refer to "FileIO.i6t" and work only on Glulx.

```
Section SR5/7/1 - Using external resources - Files

To read (filename - external-file) into (T - table-name)
    (documented at ph_filetables):
    (- FileIO_GetTable({filename}, {T}); -).
To write (filename - external-file) from (T - table-name):
    (- FileIO_PutTable({filename}, {T}); -).
To decide if (filename - external-file) exists:
    (- (FileIO_Exists({filename}, false)) -).
To decide if ready to read (filename - external-file)
    (documented at ph_readytr):
    (- (FileIO_Ready({filename}, false)) -).
To mark (filename - external-file) as ready to read:
    (- FileIO_MarkReady({filename}, true); -).
To mark (filename - external-file) as not ready to read:
    (- FileIO_MarkReady({filename}, false); -).
To write (T - text) to (FN - external-file)
    (documented at ph_writef):
    (- FileIO_PutContents({FN}, {-allow-stack-frame-access:T}, false); -).
To append (T - text) to (FN - external-file):
    (- FileIO_PutContents({FN}, {-allow-stack-frame-access:T}, true); -).
To say text of (FN - external-file):
    (- FileIO_PrintContents({FN}); say__p = 1; -).
```

§**82.** Figures and sound effects. Ditto, but for "Figures.i6t".

```
Section SR5/7/2 - Using external resources - Figures and sound effects

To display (F - figure-name), one time only
    (documented at ph_displayf):
    (- DisplayFigure({F}, {phrase options}); -).
To play (SFX - sound-name), one time only
    (documented at ph_playsf):
    (- PlaySound({SFX}, {phrase options}); -).
```

§**83. Message support.** "Unindexed" here is a euphemism for "undocumented". This is where experimental or intermediate phrases go: things we don't want people to use because we will probably revise them heavily in later builds of Inform. For now, the Standard Rules do make use of these phrases, but nobody else should. They will change without comment in the change log.

```
Section SR5/8/1 - Message support - Issuance - Unindexed

To stop the action with library message (AN - an action-name) number
    (N - a number) for (H - an object):
    (- return GL__M({AN},{N},{H}); -) - in to only.
To stop the action with library message (AN - an action-name) number
    (N - a number):
    (- return GL__M({AN},{N},noun); -) - in to only.
To issue miscellaneous library message number (N - a number):
    (- GL__M(##Miscellany,{N}); -).
To issue library message (AN - an action-name) number
    (N - a number) for (H - an object):
    (- GL__M({AN},{N},{H}); -).
To issue library message (AN - an action-name) number (N - a number):
    (- GL__M({AN},{N},noun); -).
To issue actor-based library message (AN - an action-name) number
    (N - a number) for (H - an object) and (H2 - an object):
    (- AGL__M({AN},{N},{H},{H2}); -).
To issue actor-based library message (AN - an action-name) number
    (N - a number) for (H - an object):
    (- AGL__M({AN},{N},{H}); -).
To issue actor-based library message (AN - an action-name) number (N - a number):
    (- AGL__M({AN},{N},noun); -).

To issue score notification message:
    (- NotifyTheScore(); -).
To say pronoun dictionary word:
    (- print (address) pronoun_word; -).
To say recap of command:
    (- PrintCommand(); -).
The pronoun reference object is an object that varies.
The pronoun reference object variable translates into I6 as "pronoun_obj".
The library message action is an action-name that varies.
The library message action variable translates into I6 as "lm_act".
The library message number is a number that varies.
The library message number variable translates into I6 as "lm_n".
The library message amount is a number that varies.
The library message amount variable translates into I6 as "lm_o".
The library message object is an object that varies.
The library message object variable translates into I6 as "lm_o".
The library message actor is an object that varies.
The library message actor variable translates into I6 as "actor".
The second library message object is an object that varies.
The second library message object variable translates into I6 as "lm_o2".
```

§**84.** Intervention. These are hooks for the new library messages system coming in a later build; they won't last.

```
Section SR5/8/2 - Message support - Intervention - Unindexed

To decide if intervened in miscellaneous message:
    decide on false;

To decide if intervened in miscellaneous list message:
    decide on false;

To decide if intervened in action message:
    decide on false;
```

§**85. Miscellaneous other phrases.** Again, *these are not part of Inform's public specification.*

```
Section SR5/9/1 - Miscellaneous other phrases - Unindexed
```

§**86.** These are actually sensible concepts in the world model, and could even be opened to public use, but they're quite complicated to explain.

```
To decide which object is the component parts core of (X - an object):
    (- CoreOf({X}) -).
To decide which object is the common ancestor of (O - an object) with
    (P - an object):
    (- (CommonAncestor({O}, {P})) -).
To decide which object is the not-counting-parts holder of (O - an object):
    (- (CoreOfParentOfCoreOf({O})) -).
To decide which object is the visibility-holder of (O - object):
    (- VisibilityParent({O}) -).
To calculate visibility ceiling at low level:
    (- FindVisibilityLevels(); -).
```

§**87.** These are in effect global variables, but aren't defined as such, to prevent people using them. Their contents are only very briefly meaningful, and they would be dangerous friends to know.

```
To decide which number is the visibility ceiling count calculated:
    (- visibility_levels -).
To decide which object is the visibility ceiling calculated:
    (- visibility_ceiling -).
```

§**88.** This is a unique quasi-action, using the secondary action processing stage only. A convenience, but also an anomaly, and let's not encourage its further use.

```
To produce a room description with going spacing conventions:
    (- LookAfterGoing(); -).
```

§**89.** Two ugly little tricks needed because of the mismatch between I6 and I7 property implementation, and because of legacy code from the old I6 library. Please don't touch.

```
To print the location's description:
    (- PrintOrRun(location, description); -).
To decide if (O - object) goes undescribed by source text:
    (- ({O}.description == 0) -).
```

§**90.** This is a bit trickier than it looks, because it isn't always set when one thinks it is.

```
To decide whether the I6 parser is running multiple actions:
    (- (multiflag==1) -).
```

§**91.** Again, the following cries out for an enumerated kind of value.

```
To decide if set to sometimes abbreviated room descriptions:
    (- (lookmode == 1) -).
To decide if set to unabbreviated room descriptions:
    (- (lookmode == 2) -).
To decide if set to abbreviated room descriptions:
    (- (lookmode == 3) -).
```

§**92.** Action conversion is a trick used in the Standard Rules to simplify the implementation of actions: it allows one action to become another one mid-way, without causing spurious action failures. (There are better ways to make user-defined actions convert, and some of the examples show this.)

```
To convert to (AN - an action-name) on (O - an object):
    (- return GVS_Convert({AN},{O},0); -) - in to only.
To convert to request of (X - object) to perform (AN - action-name) with
    (Y - object) and (Z - object):
    (- TryAction(true, {X}, {AN}, {Y}, {Z}); rtrue; -).
To convert to special going-with-push action:
    (- ConvertToGoingWithPush(); rtrue; -).
```

§**93.** The "surreptitiously" phrases shouldn't be used except in the Standard Rules because they temporarily violate invariants for the object tree and the light variables; the SR uses them carefully in situations where it's known to work out all right.

```
To surreptitiously move (something - object) to (something else - object):
    (- move {something} to {something else}; -).
To surreptitiously move (something - object) to (something else - object) during going:
    (- MoveDuringGoing({something}, {something else}); -).
To surreptitiously reckon darkness:
    (- SilentlyConsiderLight(); -).
```

§**94.** And so, at last...

```
The Standard Rules end here.
```

§**95.** ...except that this is not quite true, because like most extensions they then quote some documentation for Inform to weave into index pages: though here it's more of a polite refusal than a manual, since the entire system documentation is really the description of what was defined in this extension.

```
---- DOCUMENTATION ----
Unlike other extensions, the Standard Rules are compulsorily included
with every project. They define the phrases, kinds and relations which
are basic to Inform, and which are described throughout the documentation.
```

B The Template Layer

`B/introt`: *Introduction.i6t* A short introduction to the template and its organisation.

`B/maint`: *Main.i6t* The top-level logic of NI: the sequence of operations followed by NI up to the point where the output file is opened, resuming after it is closed again.

`B/typet`: *Types.i6t* To set up the atomic kinds of value (the base data types, in other words) within NI.

`B/outt`: *Output.i6t* This is the superstructure of the file of I6 code output by NI: from ICL commands at the top down to the signing-off comments at the bottom.

`B/defnt`: *Definitions.i6t* Miscellaneous constant definitions, usually providing symbolic names for otherwise inscrutable numbers, which are used throughout the template layer.

`B/ordt`: *OrderOfPlay.i6t* The sequence of events in play: the Main routine which runs the startup rulebook, the turn sequence rulebook and the shutdown rulebook; and most of the I6 definitions of primitive rules in those rulebooks.

`B/actt`: *Actions.i6t* To try actions by people in the model world, processing the necessary rulebooks.

`B/acvt`: *Activities.i6t* To run the necessary rulebooks to carry out an activity.

`B/rbt`: *Rulebooks.i6t* To work through the rules in a rulebook until a decision is made.

`B/parst`: *Parser.i6t* The parser for turning the text of the typed command into a proposed action by the player.

`B/lwt`: *ListWriter.i6t* A flexible object-lister taking care of plurals, inventory information, various formats and so on.

`B/oowt`: *OutOfWorld.i6t* To implement some of the out of world actions.

`B/wmt`: *WorldModel.i6t* Testing and changing the fundamental spatial relations.

`B/light`: *Light.i6t* The determination of light, visibility and physical access.

`B/testt`: *Tests.i6t* The command grammar and I6 implementation for testing commands such as TEST, ACTIONS and PURLOIN.

`B/langt`: *Language.i6t* The fundamental definitions needed by the parser and the verb library in order to specify the language of play – that is, the language used for communications between the story file and the player.

`B/stackt`: *MStack.i6t* To allocate space on the memory stack for frames of variables to be used by rulebooks, activities and actions.

`B/chrt`: *Chronology.i6t* To record information now which will be needed later, when a condition phrased in the perfect tense is tested.

`B/print`: *Printing.i6t* To manage the line skips which space paragraphs out, and to handle the printing of names of objects, pieces of text and numbers.

`B/rtpt`: *RTP.i6t* To issue run-time problem messages, and to perform some run-time type checking which may issue such messages.

`B/utilt`: *Utilities.i6t* Miscellaneous utility routines for some fundamental I6 needs.

`B/numt`: *Number.i6t* Support for parsing integers.

`B/timet`: *Time.i6t* Support for parsing and printing times of day.

`B/tabt`: *Tables.i6t* To read, write, search and allocate rows in the Table data structure.

`B/sortt`: *Sort.i6t* To sort arrays.

`B/relt`: *Relations.i6t* To manage run-time storage for relations between objects, and to find routes through relations and the map.

`B/figst`: *Figures.i6t* To display figures and play sound effects.

`B/inxt`: *IndexedText.i6t* Code to support the indexed text kind of value.

`B/regxt`: *RegExp.i6t* Code to match and replace on regular expressions against indexed text strings.

`B/chart`: *Char.i6t* To decide whether letters are upper or lower case, and convert between the two.

`B/unict`: *UnicodeData.i6t* To tabulate casings in the character set.

`B/stact`: *StoredAction.i6t* Code to support the stored action kind of value.

`B/listt`: *Lists.i6t* Code to support the list of... kind of value constructor.

`B/blkvt`: *BlockValues.i6t* Routines for copying, comparing, creating and destroying block values, and for reading and writing them as if they were arrays.

`B/flext`: *Flex.i6t* To allocate flexible-sized blocks of memory as needed to hold arbitrary-length strings of text, stored actions or other block values.

`B/zmt`: *ZMachine.i6t* To provide routines handling low-level Z-machine facilities.

`B/glut`: *Glulx.i6t* To start up the Glk interface for the Glulx virtual machine, and provide Glulx-specific printing functions.

`B/iot`: *FileIO.i6t* Reading and writing external files, in the Glulx virtual machine only.

Introduction Template B/introt

Purpose

A short introduction to the template and its organisation.

§**1. The Software Stack.** The term "software stack" is sometimes used to refer to the total body of code running in a computer. As the word "stack" implies, there tends to be a fairly clear division between components, and an architecture in which one component is supported by another. The ground on which the stack rests is the hardware. Then come device drivers, for communicating with hardware, and a kernel of very low-level code to regulate who talks to the hardware and when. On top of that are usually several layers of operating-system facilities and utility programs, and on top of all of that are Firefox, iTunes and other consumer programs which the user has made a conscious decision to run. These top-level programs are visible, while the lower layers are concealed from the user and are generally very reliable, so it is easy to forget that they are there at all.

Similarly, when writing and testing a work with I7, it's easy to think that the only code running is the code directly generated by the source text. In fact, though, it runs inside a simulated computer called the virtual machine (or VM), and the VM has a software stack just as other computers do. Here the distinction between "program" and "operating system" is more blurred, because it can afford to be – the VM only ever has a single program running, and is not connected to any valuable hardware or data, so there is no need to protect the system's integrity from a malicious program, or to protect one program from the failings of another running at the same time. But there is still a recognisable software stack. From top to bottom, the VM at run-time contains:

(a) The rules, text and tables written by the author of this specific work of IF;
(b) Material contributed by Extensions which the author chose to borrow from;
(c) Rules, phrases and other constructions made by the Standard Rules extension, whose use is compulsory;
(d1) Infrastructure for rulebooks and code for maintaining a world model common to all works of IF;
(d2) The code needed to manage complex data structures such as relations, tables, indexed text, and so on;
(e) The interface to the VM, which is abstracted so that higher levels do not need to know whether Glulx or the Z-machine is being used.

Here (d1) and (d2) are independent blocks, so the picture is roughly so:

the player typing at a keyboard

(a) MATERIAL WRITTEN BY THE AUTHOR

(b) EXTENSIONS CHOSEN BY THE AUTHOR

(c) THE STANDARD RULES

(d1) MECHANICS OF PLAY (d2) DATA STRUCTURE SUPPORT

(e) VIRTUAL MACHINE INTERFACE

virtual machine "hardware"

NI is not like a compiler for a conventional program, because it has to conjure up the entire software stack whenever it compiles anything. (Until 2008, NI compiled code which ran on top of the traditional I6 library: nowadays it compiles stand-alone code which never uses the `#Include` directive. While it's true that the I6 compiler does add a very thin extra layer of code itself – the "veneer" – in all other respects NI specifies every single line of I6 code which makes up the eventual story file.)

NI compiles layers (a), (b) and (c) from the I7 source text found in the project file, in the Extensions installed by the user, and in the Standard Rules extension which comes pre-installed. But parts (d1), (d2) and (e) are almost exactly the same I6 code whatever may be sitting above them: they are simply copied, with minor variations, from a large body of standing code installed in the I7 application which is called the "template".

§**2. Segments and Paragraphs.** The template is divided into about 38 individual "segments", each stored in its own file and with the `.i6t` file extensions. This stands for "I6 template", because the code generated by NI is Inform 6 (I6) code. What makes the file a template rather than being raw code is that it is divided into named paragraphs which contain both commentary and also I6 code: when NI turns a template into the output code, the commentary (and each paragraph heading) is stripped out.

See the Inform documentation for how to modify the template in use for any particular project: I6 code can be added before, instead of or after any named segment or paragraph, and in addition, it's possible to replace entire segment files with your own versions stored in the Materials folder for a project.

§**3. Architecture.** To recap, then, the template contributes the following parts of the software stack:

(d1) MECHANICS OF PLAY (d2) DATA STRUCTURE SUPPORT

(e) VIRTUAL MACHINE INTERFACE

A more detailed version of this diagram organises its 38 segments follows:

NI Control. `Main.i6t`, `Types.i6t`, `Output.i6t`, `Definitions.i6t`.

Mechanics of Play I: Rules. `OrderOfPlay.i6t`, `Actions.i6t`, `Activities.i6t`, `Rulebooks.i6t`.

II: Infrastructure. `Parser.i6t`, `ListWriter.i6t`, `OutOfWorld.i6t`, `WorldModel.i6t`, `Light.i6t`, `Tests.i6t`.

III: Basic Services. `Language.i6t`, `MStack.i6t`, `Chronology.i6t`, `Printing.i6t`, `RTP.i6t`, `Utilities.i6t`.

Data Structure Support I: Basics. `Number.i6t`, `Time.i6t`, `Tables.i6t`, `Sort.i6t`, `Relations.i6t`, `Figures.i6t`.

II: Block Values. `IndexedText.i6t`, `RegExp.i6t`, `Char.i6t`, `UnicodeData.i6t`, `StoredAction.i6t`, `Lists.i6t`, `BlockValues.i6t`, `Flex.i6t`.

Virtual Machine Interface. `ZMachine.i6t` *or* `Glulx.i6t` and `FileIO.i6t`.

To elaborate:

NI Control. As well as containing commentary and I6 code, template files can also contain commands to tell NI to do something more interesting than simply copying over material verbatim into its output. Four of the segments use this ability a great deal: the remaining segments hardly use it at all. These four segments therefore don't fit anywhere in the diagram of the software stack above. "Main.i6t" controls the top-level logic of NI and gives the sequence of operations; "Types.i6t" specifies the basic kinds of value known to NI – numbers, times, texts, rulebooks and so on; "Output.i6t" is essentially the arrangement used to join all the many fragments of I6 from the template and the I7 source text into a single end-to-end I6 program which will become the story file; "Definitions.i6t" defines many named constants which are used across the template.

Mechanics of Play I: Rules. "OrderOfPlay.i6t" is the highest-level description of what happens when a story file runs: it contains the I6 `Main` routine, and definitions of the primitive rules in the most important rulebooks.

"Actions.i6t" and "Activities.i6t" contain code which runs actions and activities, respectively: these sit on top of "Rulebooks.i6t", which performs general rulebook-running.

Mechanics of Play II: Infrastructure. "Parser.i6t" breaks down a command typed by the player into a slate of variables which can be formed into an action. "ListWriter.i6t" is a general-purpose service for printing lists of objects satisfying various descriptions, and formatted in different ways. "OutOfWorld.i6t" provides code to handle out-of-world actions not needing direct access to VM internals: PRONOUNS and SUPERBRIEF, for instance. "Tests.i6t" provides code for the testing commands – TEST, ACTIONS, SHOWME and so forth.

"WorldModel.i6t" contains code for performing object movements and the like in such a way that the world model rules are preserved: it handles component parts, decides touchability and so on. "Light.i6t" determines the level of light and decides on visibility.

Mechanics of Play III: Basic Services. "Language.i6t" provides English-language messages issued by template routines and the Standard Rules during play: replacing this segment is the way to translate I7 story files into other languages of play. (It's the equivalent of the old I6 "language definition files", and retains most of the same structure.) "MStack.i6t" provides a small general-purpose memory stack. "Chronology.i6t"

performs the continuous monitoring required to ensure that past-tense conditions work: for instance we can only determine whether or not "the Black Door has been open" if we have spent the whole time so far checking on whether it is open or not, and this is where the checking is done. "Printing.i6t" handles paragraph-breaking, the printing of object names with suitable articles attached, and miscellaneous other printing needs. "RTP.i6t" issues run-time problem messages as needed: these should appear only if the story file does something clearly illegal, and point to bugs not yet removed from the author's code. "Utilities.i6t" provides miscellaneous utility functions at a very low level, such as for unsigned comparison of two numbers.

Data Structure Support I: Basics. Many kinds of value need no maintenance, and little or no support. "Number.i6t" and "Time.i6t" contain code to parse numbers and times of day from text typed by the player, together with a few basic operations: for instance, the rounding off of times of day. "Tables.i6t" provides access to table entries, specified in a variety of direct and indirect ways; this makes use of "Sort.i6t", which abstracts a choice of sorting algorithms for use on tables and other I6 data. "Relations.i6t" provides code for testing and asserting relations: the information about what is related to what else is stored in a variety of different ways, depending on the relation, to make best use of memory. "Figures.i6t" displays figures and plays back sound effects, or rather, passes instructions to do so down to the VM.

Data Structure Support II: Block Values. Kinds of value can be divided into the ordinary ones – number, object, time and so on – together with "block values" such as indexed text, stored action and list. Block values have to be stored in dynamically allocated memory from a heap. It would be wasteful to include all of this code, and to spare memory for the heap, if no block values were actually needed, so the following segments are only compiled if the source text makes specific reference to at least one block value. "IndexedText.i6t" manages character-indexed strings of text, which can shrink or stretch to arbitrary lengths: it makes use of "RegExp.i6t" for regular expression matching and search-and-replace; and also of "Char.i6t" for code to deal with lower and upper casing of letters, and "UnicodeData.i6t" to provide character set details, mechanically converted from the Unicode 4.0 standard. "StoredAction.i6t" manages stored actions: it can convert the current action into a stored one, and also try a long-stored action so that it now takes place. "Lists.i6t" manages the flexibly sized lists produced by values whose kind is list of numbers, list of texts, list of lists of lists of stored actions, and so forth: it can merge, insert, delete, resize, rotate, and reverse lists, and makes use once again of "Sort.i6t" (q.v.) to sort them.

"BlockValues.i6t" provides the basic support for kinds of value which are stored as blocks of memory on the heap.

At the lowest level, "Flex.i6t" manages flexible memory allocation as required by "BlockValues.i6t": it organises the heap of unclaimed memory, for instance, and maks allocations and deallocations when needed.

Virtual Machine Interface. Depending on the Settings used for the I7 project being compiled, we either use "ZMachine.i6t" or "Glulx.i6t": if Glulx, we also add "FileIO.i6t", which provides support for the limited file-handling abilities offered by Glulx.

Properly speaking, there is one further template file: "Introduction.i6t". It provides the commentary you are now reading, but has no other function, contains no code, and is finished now anyway.

Main Template B/maint

Purpose

The top-level logic of NI: the sequence of operations followed by NI up to the point where the output file is opened, resuming after it is closed again.

---

§**1. Startup.** This segment of the template is not like any other. It contains almost the complete logical sequence of operations followed by NI – indeed, NI essentially works by parsing some command-line arguments to set switches and then asking the template interpreter to run through "Main.i6t", the only template file which must always exist. (The other template files are only ever involved when called on by the `{-segment:...}` command from a template file already running.) Whereas most template segments contain I6 code to pass through into NI's output, this one runs both before and after NI's output file is being written, and contains only commands.

The Startup paragraph is not in fact modifiable in any easy way, because of course "Include..." sentences – used to tell the template interpreter to override the template files – will not be understood until long after this paragraph has been fully dealt with. But perhaps that's no bad thing.

```
{-callv:start_memory}
{-callv:start_lexer}
{-callv:make_reserved_words}
{-callv:handle_census_mode}
{-segment:Types.i6t}
{-callv:make_built_in_relations}
{-callv:make_built_in_verbs}
{-callv:make_built_in_determiners}
{-progress-stage:0}
{-log-phase:Lexical analysis}
{-callv:read_primary_source_text}
{-callv:create_kind_kind}
{-callv:create_standard_csps}
```

§**2. Semantic Analysis.** Similarly: Include... sentences are not read until this paragraph is long forgotten.

```
{-progress-stage:1}
{-log-phase:Semantic analysis I}
{-callv:plant_parse_tree} ! Initialise the parse tree
{-callv:break_source_into_sentences} ! Build first tranche of sentences
{-callv:traverse_for_extensions} ! Expand extension inclusions and build sentences
{-callv:satisfy_heading_dependencies}
{-callv:traverse_for_verbs} ! Find verbs in the assertion sentences
{-callv:traverse_for_plural_definitions} ! Build irregular plurals dictionary
{-callv:tidy_up_ofs_and_froms} ! More "of" wrangling
{-callv:register_recently_lexed_phrases} ! To make commands children of their routines
{-callv:declare_source_loaded}
{-callv:include_templates_for_types}
{-log-phase:Semantic analysis II}
{-callv:verify_parse_tree} ! Purely to check that NI is running correctly
{-callv:check_extension_versions} ! Do the extension version numbers meet needs?
{-callv:make_heading_tree} ! Stratify headings and subheadings
{-callv:write_headings_as_xml} ! Dump them to a file for the GUI to use
{-log-phase:Semantic analysis III}
{-callv:traverse_for_adjective_definitions}
{-callv:traverse_to_create_equations}
{-callv:traverse_to_create_tables}
{-callv:traverse_for_phrase_names}
```

§**3. Assertion Reading.** Since Include... sentences are read during pass 2, they might just be able to meddle by adding instructions to take place after this paragraph, but would be too late to work before or instead of it.

```
{-progress-stage:2}
{-log-phase:First pass through assertions}
{-read-assertions:1}
{-log-phase:All objects and properties now created}
{-callv:log_brief_picture}
{-log-phase:Second pass through assertions}
{-read-assertions:2}
```

§**4. Model Construction.**

```
{-log-phase:Making the model world}
{-callv:Inform_name_the_actions}
{-callv:Inform_name_the_objects}
{-callv:make_model_world}
{-callv:validate_definitions}
{-callv:make_further_built_in_relations}
```

§5. Tables and Grammar.

```
{-log-phase:Tables and grammar}
{-callv:check_tables_for_kind_clashes}
{-callv:traverse_to_stock_tables}
{-callv:traverse_to_stock_equations}
{-callv:traverse_for_grammar}
```

§6. Phrases and Rules.

```
{-progress-stage:3}
{-log-phase:Phrases and rules}
{-callv:define_named_printing_phrases}
{-log-phase:Phrases and rules A}
{-callv:traverse_for_phrases}
{-log-phase:Phrases and rules B}
{-callv:register_phrase_meanings}
{-log-phase:Phrases and rules C}
{-callv:parse_rule_parameters}
{-log-phase:Phrases and rules D}
{-callv:add_rules_to_rulebooks}
{-log-phase:Phrases and rules E}
{-callv:parse_rule_placements}
{-callv:empty_all_headings}
```

§**7. Code Generation.** This is where we hand over to regular template files – containing code passed through as I6 source, as well as a few further commands – starting with "Output.i6t".

```
{-progress-stage:4}
{-log-phase:Code generation}
{-open-file}{-segment:Output.i6t}{-close-file}
{-log-phase:Compilation now complete}
```

§8. Metadata.

```
{-callv:write_ifiction_and_blurb}
```

§**9. Indexing.** This paragraph can be skipped without harming any of the rest of NI's work: the only effect is to suppress the generation of the index. (Indeed, if NI is called with the `-noindex` command-line switch, then the template interpreter ignores all commands in between `{-open-index}` and `{-close-index}`, thus skipping exactly this paragraph.)

```
{-open-index}
{-index:Phrasebook Index=A guide to the phrases allowed; [LEXICON]a lexicon of words; a [RELTABLE]table
of relations, and [VERBTABLE]of verbs.|What are phrases?<PHRASES>; And descriptions?<DESCRIPTIONS>; Relations?<RELA
Verbs?<VERBS>=John Florio's Italian-English dictionary of 1598-1611}
    {-callv:index_page_Phrasebook}
{-index:Kinds Index=Table of all the kinds; [ARITHMETIC]how arithmetic affects them; and [KDETAILS]details
of each kind in turn.|What are kinds?<KINDS>; More about kinds of object<NEWKINDS>; And kinds of value<KINDSVALUE>=
medieval bestiary, showing every kind of animal}
    {-callv:index_page_Kinds}
{-index:World Index=A map of the geographical layout; [MDETAILS]contents of each room; and [LEXICON]an
A to Z of rooms and things.|What's the map?<MAP>; Using regions<REGIONS>; If the map looks wrong...<MAPHINTS>;
Exporting to EPS<EPSMAP>=William Smith's first geological map of England, 1815}
    {-callv:index_page_World}
{-index:Actions Index=The actions (click magnifiers for details); [NAP]kinds of action; [COMMANDS]commands
A to Z; [LEXICON]actions A to Z; [ARULES]the applicable rules.|What are actions?<ACTIONS>; Creating new
actions<NEWACTIONS>; Out of world actions (in red)<OUTOFWORLD>=Standard English men-at-work icon, used
on road signs}
    {-callv:index_page_Actions}
{-index:Rules Index=Rules not directly tied to actions (see the Actions Index) or scenes (the Scenes
Index).|What are rulebooks?<RULEBOOKS>; What are activities?<ACTIVITIES>; Moving or abolishing rules<RLISTING>=Baby
tablet of the Laws of Hammurabi, c.1760 BC}
    {-callv:index_page_Rules}
{-index:Contents Index=Headings and [EXTLIST]extensions; [NAMES]named values, Tables and so on; the [LCARD]Library
Card; the [STORYFILE]story file.|How do headings work?<HEADINGS>; What's the Library Card?<LCARDS>=The
first Gutenberg printing press in action, c.1455}
    {-callv:index_headings}
    {-callv:index_extensions}
    {-callv:update_census_of_extensions}
    {-callv:index_quantities}
    {-callv:index_tables}
    {-callv:index_equations}
    {-callv:index_figures}
    {-callv:index_sounds}
    {-callv:index_external_files}
    {-callv:index_library_card}
{-index:Scenes Index=A map of how the scenes begin and end; and [SDETAILS]details of each scene in turn.|What
are scenes?<SCENESINTRO>; How they link together<LINKINGSCENES>=Strolling beside a riverside theatre
in 18th century London}
    {-callv:index_page_Scenes}
{-callv:complete_index}
{-close-index}
```

§**10. Shutdown.** Closing the problems report, making any final reports to the debugging log, and freeing all allocated memory: and that's then it.

```
{-callv:report_unacted_upon_interventions}
{-callv:complete_problems_report}
{! -callv:report_pairs_observed}
{! -callv:report_pairs_allowed}
{! -callv:debug_memory_statistics}
{! -callv:debug_parser_statistics}
{! -callv:debug_verbs}
{-callv:free_memory}
```

Types Template B/typet

Purpose

To set up the atomic kinds of value (the base data types, in other words) within NI.

---

§**1. Header.** The following mass of type definitions is not heavily subdivided into paragraphs since it's read so early in NI's run that "Include..." sentences haven't had time to be read yet – so they aren't easy to customise. (Except by putting a whole customised version of the file into the relevant `Materials/I6T` folder.)

The `{-lines:type}` command causes all subsequent lines to be sent as parameters to the `{-type:...}` command, until the next `{-endlines}` command.

```
{-callv:make_type_IDs_table}
{-lines:type}
#DEFAULTS:
defined-in-source-text:no
is-incompletely-defined:no
is-a-unit:no
is-an-enumeration:no
uses-signed-comparisons:no
can-coincide-with-property:no
named-values-created-with-assertions:no
used-in-let-without-casts:no
can-be-type-of-global-variable:no
has-i6-GPR:no
i6-printing-routine:DecimalNumber
i6-printing-routine-actions:DA_Name
block-data:no
multiple-block:no
constant-compilation-method:none
index-priority:4
```

§**2. Macros.** New kinds of value cause some I7 source to be generated, following various combinations of the macros below.

```
*UNDERSTOOD-VARIABLE:
<type> understood is a <type> which varies.
*END

*PRINTING-ROUTINE:
To say (value - <type>):
    (- print (<printing-routine>) {value}; -).
*END

*PRINTING-ROUTINE-BLOCK:
To say (value - <type>):
    (- print (<printing-routine>) {-pointer-to:value}; -).
*END

*RANDOM-CHOOSERS:
To decide which <type> is a random <lower-case-type> between (first value - <type>) and (second value
- <type>) - <type-number>:
    (- R_<printing-routine>({first value}, {second value}) -).
To decide which <type> is a random <lower-case-type> from (first value - <type>) to (second value - <type>)
- <type-number>:
    (- R_<printing-routine>({first value}, {second value}) -).
*END

*ENUMERATION-CHOOSERS:
To decide which <type> is <lower-case-type> after (X - <type>) - <type-number>:
    (- A_<printing-routine>({X}) -).
To decide which <type> is <lower-case-type> before (X - <type>) - <type-number>:
    (- B_<printing-routine>({X}) -).
To decide which <type> is a random <lower-case-type> - <type-number>:
    (- R_<printing-routine>() -).
*END
```

§**3. Group 1.** These are data types used in type-checking but not representing specific kinds of value at run-time.

The + notation indicates a type name which NI requires us to construct, and with this specific name.

```
+ANY_VALUE_TY:
description:a value
singular:value
plural:values
constant-compilation-method:none
cast:<all>
! Matches any value

+ASSIGNABLE_VALUE_TY:
description:a value
singular:assignable-value
plural:assignable-values
constant-compilation-method:none
cast:<assignable>
! Matches any value which can be the rvalue of an assignment

ANY_POINTER_VALUE_TY:
```

```
description:a value
singular:pointer value
constant-compilation-method:none
cast:<pointer>
! Matches any value where the I6 representation is a pointer to a block

ANY_WORD_VALUE_TY:
description:a value
singular:word value
constant-compilation-method:none
cast:<word>
! Matches any value where the I6 representation is a single word

+KIND_OF_POINTER_VALUE_TY:
description:the name of a kind of complex value
singular:kind of pointer value
constant-compilation-method:none
! Matches name of kind of value pointing to a block, compiles to NI type number

+KIND_OF_WORD_VALUE_TY:
description:the name of a kind of simple value
singular:kind of word value
constant-compilation-method:none
! Matches name of kind of value in a single word, compiles to NI type number

CLASSIFIED_TY:
description:some text or the name of a room, thing or rule
singular:miscellaneous-value
plural:miscellaneous-values
used-in-let-with-cast-priority:50
constant-compilation-method:none
cast:OBJECT_TY
cast:TEXT_TY
cast:TEXT_ROUTINE_TY
cast:RULE_TY
! Deprecated: a union of the four I6-metaclass-distinguishable data types

+VALUE_DESCRIPTION_TY:
description:a description of values
singular:domain-description
plural:domain-descriptions
constant-compilation-method:special
! Used only in type-checking to distinguish value from object descriptions

TEMPLATE_VAR1_TY:
is-template-variable:yes
template-variable-number:1
description:template variable 1
singular:kov marker 1
plural:kov marker 1s
! Used in template matches

TEMPLATE_VAR2_TY:
is-template-variable:yes
template-variable-number:-1
description:template variable 1 reference
singular:kov reference 1
plural:kov reference 1s
! Used in template matches
```

```
TEMPLATE_VAR3_TY:
is-template-variable:yes
template-variable-number:2
description:template variable 2
singular:kov marker 2
plural:kov marker 2s
! Used in template matches
TEMPLATE_VAR4_TY:
is-template-variable:yes
template-variable-number:-2
description:template variable 2 reference
singular:kov reference 2
plural:kov reference 2s
! Used in template matches
+NO_ENTRIES_TY:
description:no entries
constant-compilation-method:none
! Used for the type of the empty list { }
```

§**4. Group 2.** Data types for basic word-sized values.

```
+NUMBER_TY:
description:a number
documentation-reference:kind_number
singular:number
plural:numbers
quasinumerical:yes
has-i6-GPR:yes
explicit-i6-GPR:DECIMAL_TOKEN
i6-printing-routine-actions:DA_Number
used-in-let-with-cast-priority:10
can-be-type-of-global-variable:yes
uses-signed-comparisons:yes
can-exchange:yes
default-value:0
index-default-value:0
specification-text:Whole number in the range -32768, -32767, ..., -2, -1, 0, 1, 2, 3, ..., 32767: small
numbers can be written textually as 'one', 'two', 'three', ..., 'ten', 'eleven', 'twelve'. (A much larger
number range is allowed if we compile the source to Glulx rather than the Z-machine: see the Settings
panel.)
index-priority:1
constant-compilation-method:literal
+TIME_TY:
description:a time
documentation-reference:kind_time
singular:time
plural:times
quasinumerical:yes
used-in-let-with-cast-priority:20
can-be-type-of-global-variable:yes
uses-signed-comparisons:yes
can-exchange:yes
```

```
default-value:540
loop-domain-schema:for (*1=0: *1<TWENTY_FOUR_HOURS: *1++)
has-i6-GPR:yes
explicit-i6-GPR:TIME_TOKEN
i6-printing-routine:PrintTimeOfDay
i6-printing-routine-actions:PrintTimeOfDay
specification-text:A time of day, written in the form '2:34 AM' or '12:51 PM', or a length of time such
as '10 minutes' or '3 hours 31 minutes', which must be between 0 minutes and 23 hours 59 minutes inclusive.
index-default-value:9:00 AM
index-priority:1
constant-compilation-method:literal
apply-template:*PRINTING-ROUTINE

+TEXT_TY:
description:some text
singular:text
plural:texts
used-in-let-with-cast-priority:30
can-be-type-of-global-variable:yes
default-value:EMPTY_TEXT_VALUE
i6-printing-routine:PrintText
index-default-value:""
specification-text:Some text in double quotation marks, perhaps with substitutions written in square
brackets.
index-priority:3
cast:TEXT_ROUTINE_TY
sometimes-cast:CLASSIFIED_TY
constant-compilation-method:special
! Because of casting, this can be text either with or without substitutions

+OBJECT_TY:
description:an object
singular:object
plural:objects
used-in-let-with-cast-priority:40
can-be-type-of-global-variable:yes
default-value:nothing
index-default-value:nothing
index-priority:2
i6-printing-routine:PrintShortName
i6-printing-routine-actions:DA_Name
sometimes-cast:CLASSIFIED_TY
constant-compilation-method:special
specification-text:Objects are values intended to simulate physical things: places, people, things, and
so on. They come in many kinds. The special value 'nothing' is also allowed, and can be used to mean
'no object at all'.
! For any single specific object (direction, room, thing, etc.), or "nothing"

+TEXT_ROUTINE_TY:
description:some text with substitutions
singular:text-routine
plural:text-routine
sometimes-cast:CLASSIFIED_TY
constant-compilation-method:special
! Quoted text which certainly does contain substitutions
```

```
+QUOT_TY:
description:some formatted text without substitutions
singular:boxed-quotation
plural:boxed-quotations
constant-compilation-method:special
! Quoted text used only for I6 box quotation statements, without substitutions
+SNIPPET_TY:
description:a snippet
documentation-reference:kind_snippet
singular:snippet
plural:snippets
can-be-type-of-global-variable:yes
default-value:101
index-default-value:<i>word 1 of command</i>
specification-text:A fragment of the player's most recent typed command, taking in a run of consecutive
words.
index-priority:5
i6-printing-routine:PrintSnippet
constant-compilation-method:none
apply-template:*PRINTING-ROUTINE
! Numerical reference to a word range (indicated by word counts) in the command
+UNICODECHAR_TY:
description:a Unicode character
singular:unicode-character
plural:unicode-characters
constant-compilation-method:literal
! For constants like "unicode 65" or "unicode black knight chess piece" only
+OBJECT_DESCRIPTION_TY:
description:a description
documentation-reference:kind_description
singular:description
plural:descriptions
used-in-let-without-casts:yes
can-be-type-of-global-variable:yes
default-value:Prop_Falsity
index-default-value:<i>matching nothing</i>
index-priority:5
constant-compilation-method:special
specification-text:A description of objects - say, 'animals which are in closed containers' - may not
look like a value, but it is. Writing 'the number of visible women', for instance, actually applies the
phrase 'number of...' to the value 'visible women', this being a description. (We very rarely need to
know this, but occasionally it's useful to store descriptions or do other value-related things with them.)
! Routine determining whether its argument is an object matching a description
+USEOPTION_TY:
description:the name of a use option
singular:use-option
plural:use-options
constant-compilation-method:special
! Constants only: the name of a use option, stored as its internal number
+PROPERTY_TY:
description:a property of something
singular:property
```

```
plural:properties
constant-compilation-method:special
unassignable:yes
! Used for names of properties such as "closed" or "carrying capacity"
+RULE_TY:
description:the name of a rule
documentation-reference:kind_rule
singular:rule
plural:rules
used-in-let-with-cast-priority:60
can-be-type-of-global-variable:yes
default-value:LITTLE_USED_DO_NOTHING_R
i6-printing-routine:RulePrintingRule
index-default-value:the little-used do nothing rule
sometimes-cast:CLASSIFIED_TY
cast:RULEBOOK_TY
constant-compilation-method:special
apply-template:*PRINTING-ROUTINE
specification-text:One of many, many rules which determine what happens during play. Rules can be triggered
by scenes beginning or ending, by certain actions, at certain times, or in the course of carrying out
certain activities.
indexed-grey-if-empty:yes
! Routine to test if a rule applies and carry it out if so
RULE_OUTCOME_TY:
description:a rule outcome
singular:outcome
plural:outcomes
constant-compilation-method:none
apply-template:*PRINTING-ROUTINE
! Used fleetingly to handle return values from rules (success, failure, none)
+RULEBOOK_TY:
modifying-adjective:action-based
modifying-adjective:object-based
description:the name of a rulebook
documentation-reference:kind_rulebook
singular:rulebook
plural:rulebooks
can-be-type-of-global-variable:yes
default-value: 0
loop-domain-schema:for (*1=0: *1<NUMBER_RULEBOOKS_CREATED: *1++)
index-default-value:the action-processing rules
constant-compilation-method:special
i6-printing-routine:RulePrintingRule
specification-text:A list of rules to follow, in sequence, to get something done. A rulebook is like
a ring-binder, with the individual rules as sheets of paper. Inform normally sorts these into their 'natural'
order, with the most specific rules first, but it's easy to shuffle the pages if you need to. When some
task is carried out during play, Inform is normally working through a rulebook, turning the pages one
by one.
indexed-grey-if-empty:yes
! A list of rules to be followed in sequence
+RULEBOOK_OUTCOME_TY:
description:the name of a rulebook outcome
```

```
singular:rulebook-outcome
plural:rulebook-outcomes
i6-printing-routine:RulebookOutcomePrintingRule
constant-compilation-method:special
apply-template:*PRINTING-ROUTINE
! Used to handle return values from rulebooks, which can be named and varied

+ACTION_NAME_TY:
description:an action name
documentation-reference:kind_actionname
singular:action-name
plural:action-names
can-be-type-of-global-variable:yes
default-value:##Wait
index-default-value:waiting action
constant-compilation-method:special
i6-printing-routine:SayActionName
apply-template:*PRINTING-ROUTINE
specification-text:An action is what happens when one of the people in the simulated world decides to
do something. A full action would be something like 'dropping the box', but an action name is just the
choice of which sort of thing is being done: here, it's 'the dropping action'. (Action names are always
written with the word 'action' at the end, to make sure they aren't mistaken for full actions.)
! To refer to an action in abstract, without any of the nouns it applies to

+ACTIVITY_TY:
description:an activity
documentation-reference:kind_activity
singular:activity
plural:activities
constant-compilation-method:special
can-be-type-of-global-variable:yes
used-in-let-with-cast-priority:80
default-value:PRINTING_THE_NAME_ACT
index-default-value:printing the name
specification-text:An activity is something which Inform does as part of the mechanics of play - for
instance, printing the name of an object, which Inform often has to do. An activity can happen by itself
('printing the banner text', for instance) or can be applied to an object ('printing the name of something',
say).
! The name of an activity, stored as its internal number

+SCENE_TY:
description:a scene
documentation-reference:kind_scene
singular:scene
plural:scenes
named-values-created-with-assertions:yes
can-be-type-of-global-variable:yes
used-in-let-with-cast-priority:80
default-value: 0
loop-domain-schema:for (*1=1: *1<=NUMBER_SCENES_CREATED: *1++)
index-default-value:the Entire Game
index-priority:2
i6-printing-routine:PrintSceneName
constant-compilation-method:quantitative
apply-template:*PRINTING-ROUTINE
```

```
specification-text:Like a scene in a play: a period of time which is usually tied to events in the plot. Scenes are created by sentences like 'Midnight Arrival is a scene.'
indexed-grey-if-empty:yes
! Scenes, as numbered in sequence in order of creation (from 0 not 1)

+TABLE_TY:
description:the name of a table
documentation-reference:kind_tablename
singular:table-name
plural:table-names
used-in-let-with-cast-priority:70
can-be-type-of-global-variable:yes
default-value:TheEmptyTable
loop-domain-schema:for (*2=0, *1=TableOfTables-->*2: *1: *2++, *1=TableOfTables-->*2)
index-default-value:<i>a table with no rows or columns</i>
i6-printing-routine:PrintTableName
constant-compilation-method:special
apply-template:*PRINTING-ROUTINE
specification-text:Like tables of information in a book or newspaper, tables in Inform hold values which have been organised into rows and columns. A table-name is just a single value, identifying which table is meant - say, 'Table of US Presidents' might be a table-name value.
indexed-grey-if-empty:yes
! Tables, stored as their array addresses inside the virtual machine

+TABLE_COLUMN_TY:
description:the name of a column
singular:table-column
plural:table-columns
constant-compilation-method:special
! Constants only: a table column name, by which its kind of value can be known

+EQUATION_TY:
description:the name of an equation
singular:equation-name
plural:equation-names
used-in-let-with-cast-priority:70
can-be-type-of-global-variable:yes
default-value:0
index-default-value:<i>an equation doing nothing</i>
i6-printing-routine:DA_Number
apply-template:*PRINTING-ROUTINE
constant-compilation-method:special
specification-text:Like formulae in a textbook or a scientific paper, equations in Inform are written out in displayed form and given names.
indexed-grey-if-empty:yes
! Equations, not stored at run-time

+RELATION_TY:
description:a relation
singular:abstract-relation
plural:abstract-relations
constant-compilation-method:special
! Constants only: a relation name, stored as its index in a run-time table

+UNDERSTANDING_TY:
description:a topic
singular:topic
```

```
plural:topics
i6-printing-routine-actions:DA_Topic
cast:TEXT_TY
cast:TEXT_ROUTINE_TY
constant-compilation-method:special
! I6 general parsing routine to match some specified text
+FIGURENAME_TY:
description:the name of a figure
documentation-reference:kind_figurename
singular:figure-name
plural:figure-names
can-be-type-of-global-variable:yes
used-in-let-with-cast-priority:80
default-value:1
loop-domain-schema:for (*2=0, *1=TableOfFigures-->*2: *1: *2++, *1=TableOfFigures-->*2)
index-default-value:figure of cover
constant-compilation-method:literal
i6-printing-routine:PrintFigureName
apply-template:*PRINTING-ROUTINE
specification-text:When made with the Glulx setting, an Inform project can include images as well as
words, and these are called figures. A figure-name is just the name of one of the figures in the current
project.
indexed-grey-if-empty:yes
! Blorb resource ID number for a picture
+SOUNDNAME_TY:
description:the name of a sound effect
documentation-reference:kind_soundname
singular:sound-name
plural:sound-names
can-be-type-of-global-variable:yes
used-in-let-with-cast-priority:80
constant-compilation-method:literal
default-value:0
loop-domain-schema:for (*2=0, *1=TableOfSounds-->*2: *1: *2++, *1=TableOfSounds-->*2)
index-default-value:<i>a silent non-sound</i>
i6-printing-routine:PrintSoundName
apply-template:*PRINTING-ROUTINE
specification-text:When made with the Glulx setting, an Inform project can include sound effects or pieces
of music. A sound-name is just the name of one of these sounds in the current project.
indexed-grey-if-empty:yes
! Blorb resource ID number for a sound effect
+EXTERNALFILE_TY:
description:the name of a file
documentation-reference:kind_externalfile
singular:external-file
plural:external-files
can-be-type-of-global-variable:yes
used-in-let-with-cast-priority:80
default-value:0
loop-domain-schema:for (*2=0, *1=TableOfExternalFiles-->*2: *1: *2++, *1=TableOfExternalFiles-->*2)
index-default-value:<i>a non-file</i>
constant-compilation-method:special
i6-printing-routine:PrintExternalFileName
```

```
apply-template:*PRINTING-ROUTINE
specification-text:When made with the Glulx setting, an Inform project can make limited use of files
stored on the computer which is operating the story at run-time. An external-file is just the name of
one of these files (not the filename in the usual sense, but a name given to it in the Inform source
text).
indexed-grey-if-empty:yes
! Address of an array holding a file I/O structure managed at run-time

+TRUTH_STATE_TY:
description:something which is either true or false
documentation-reference:kind_truthstate
singular:truth state
plural:truth states
can-be-type-of-global-variable:yes
default-value:false
index-default-value:false
loop-domain-schema:for (*1=0: *1<=1: *1++)
has-i6-GPR:yes
explicit-i6-GPR:TRUTH_STATE_TOKEN
i6-printing-routine:DA_TruthState
i6-printing-routine-actions:DA_TruthState
used-in-let-with-cast-priority:50
can-exchange:yes
specification-text:The state of whether something is 'true' or 'false'. (In other computing languages,
this might be called 'boolean', after the 19th-century logician George Boole, who first realised this
was a kind of value.)
index-priority:2
constant-compilation-method:literal
apply-template:*PRINTING-ROUTINE-BLOCK

+DESCRIPTION_OF_ACTION_TY:
description:a stored action
constant-compilation-method:special
! To represent an action test implicitly cast as a conditional rvalue

+INTERMEDIATE_TY:
description:the intermediate result of some complex calculation
quasinumerical:yes
i6-printing-routine:DA_Number
i6-printing-routine-actions:DA_Number
! Used when typechecking arithmetic in dimensional formulae
```

§**5. Group 3.** Data types for block-sized values; those where the I6 representation at run-time is a pointer to a block of memory, usually on the heap.

```
+INDEXED_TEXT_TY:
description:an indexed text
documentation-reference:kind_indexedtext
used-in-let-without-casts:yes
can-be-type-of-global-variable:yes
singular:indexed text
plural:indexed texts
block-data:yes
has-native-constants:no
multiple-block:yes
heap-size-estimate:256
cast:TEXT_TY
cast:TEXT_ROUTINE_TY
cast:SNIPPET_TY
constant-compilation-method:none
recognition-only-GPR:INDEXED_TEXT_TY_ROGPR
distinguisher:INDEXED_TEXT_TY_Distinguish
i6-printing-routine:INDEXED_TEXT_TY_Say
apply-template:*PRINTING-ROUTINE-BLOCK
can-exchange:yes
index-default-value:""
specification-text:A flexible-length form of text which can be internally altered and searched. Inform
automatically changes text to this format when necessary.
index-priority:3
! A flexible-length, character-accessible string of text

+STORED_ACTION_TY:
description:a stored action
documentation-reference:kind_storedaction
used-in-let-without-casts:yes
can-be-type-of-global-variable:yes
comparison-schema:DESCRIPTION_OF_ACTION_TY>>>*=-((STORED_ACTION_TY_Adopt(*1), SAT_Tmp-->0=(*2), STORED_ACTION_TY_Un
singular:stored action
plural:stored actions
block-data:yes
has-native-constants:yes
multiple-block:no
heap-size-estimate:16
constant-compilation-method:special
distinguisher:STORED_ACTION_TY_Distinguish
i6-printing-routine:STORED_ACTION_TY_Say
apply-template:*PRINTING-ROUTINE-BLOCK
can-exchange:yes
index-default-value:waiting
specification-text:A stored action, which can later be tried.
index-priority:3
! A stored action, which can later be tried

+LIST_OF_TY:
is-constructor:yes
description:a list of
documentation-reference:kind_listof
```

```
used-in-let-without-casts:yes
can-be-type-of-global-variable:yes
singular:list of
plural:lists of
block-data:yes
has-native-constants:no
multiple-block:yes
heap-size-estimate:256
constant-compilation-method:special
distinguisher:LIST_OF_TY_Distinguish
i6-printing-routine:LIST_OF_TY_Say
apply-template:*PRINTING-ROUTINE-BLOCK
can-exchange:yes
index-default-value:{ }
specification-text:A flexible-length list of values, where all of the items have to have the same kind
of value as each other - for instance, a list of rooms, or a list of lists of numbers. The empty list,
with no items yet, is written { }, and a list with items in is written with commas dividing them - say
{2, 5, 9}.
index-priority:3
! A list of numbers, for now
```

§6. Groups 4 and 5. Enumerations and units.

```
! New kinds of value are initially given these settings:
#NEW:
is-incompletely-defined:yes
named-values-created-with-assertions:yes
is-a-unit:no
is-an-enumeration:no
can-coincide-with-property:yes
defined-in-source-text:yes
can-be-type-of-global-variable:yes
description:a designed type
! When the source text specifies either a named constant value, or a literal
! pattern, it decides whether the new type is to be an enumeration or a unit,
! at which point one of the following macros is applied to the type:
#ENUMERATION:
is-incompletely-defined:no
is-an-enumeration:yes
named-values-created-with-assertions:yes
used-in-let-without-casts:yes
can-be-type-of-global-variable:yes
default-value:1
index-default-value:<first-constant>
index-priority:2
has-i6-GPR:yes
uses-signed-comparisons:yes
can-exchange:yes
description:a designed type
constant-compilation-method:quantitative
apply-template:*PRINTING-ROUTINE
apply-template:*UNDERSTOOD-VARIABLE
```

```
apply-template:*RANDOM-CHOOSERS
apply-template:*ENUMERATION-CHOOSERS
#UNIT:
quasinumerical:yes
is-incompletely-defined:no
is-a-unit:yes
uses-signed-comparisons:yes
can-exchange:yes
defined-in-source-text:yes
named-values-created-with-assertions:no
used-in-let-without-casts:yes
can-be-type-of-global-variable:yes
default-value:0
index-default-value:<0-in-literal-pattern>
index-priority:1
has-i6-GPR:yes
constant-compilation-method:literal
apply-template:*PRINTING-ROUTINE
apply-template:*UNDERSTOOD-VARIABLE
apply-template:*RANDOM-CHOOSERS
```

§7. Tail.

```
{-endlines}
{-callv:complete_type_IDs_table}
```

Output Template B/outt

Purpose

This is the superstructure of the file of I6 code output by NI: from ICL commands at the top down to the signing-off comments at the bottom.

---

§**1. ICL Commands.** The Inform Control Language is a mini-language for controlling the I6 compiler, able to set command-line switches, memory settings and so on. I6 ordinarily discards lines beginning with exclamation marks as comments, but at the very top of the file, lines beginning `!%` are read as ICL commands: as soon as any line (including a blank line) doesn't have this signature, I6 exits ICL mode. So, basically, we'd better do this here:

```
{-call:compile_icl_commands}
```

§**2. Other Configuration.** Setting the `Grammar__Version` constant is a rather creaky old way to tell the I6 compiler to use more detailed grammar tables: GV1 has not in fact been used since about 1994.

```
Constant Grammar__Version 2;
```

§**3. Identification.** Both of the compiler and template layer, and of the story file to be produced.

The "library" identifying texts are now a little anachronistic, since the template layer is not strictly speaking an I6 library in the same way as the original: but it is surely the spiritual successor to 6/11N, so we may as well mark it 6/12N.

```
! This file was compiled by Inform 7: the build number and version of the
! I6 template layer used are as follows.
{-call:compile_build_number}
Constant LibSerial = "080126";
Constant LibRelease = "6/12N";
Constant LIBRARY_VERSION = 612;
{-array:UUID_ARRAY}
{-call:compile_bibliographic_i6_constants}
Default Story 0;
Default Headline 0;
{-routine:ShowExtensionVersions}
```

§**4. Use options.** Use options are translated into I6 constant declarations, and NI puts them here:

```
{-call:compile_use_options}
```

§**5. Constants.**

```
{-segment:Definitions.i6t}
```

§**6. Global Variables.** These are not the only global variables defined in the template layer: those needed locally only by single sections (and not used in definitions of phrases in the Standard Rules, or referred to by NI directly) are defined within those sections – they can be regarded as unimportant implementation details, subject to change at whim. The variables here, on the other hand, are more important to understand.

(1) The first three variables to be defined are special in that they are significant to very early-style Z-machine interpreters, where they are used to produce the status line display (hence `sline1` and `sline2`). The first variable must always equal a valid object number, which is why we – pretty weirdly – set it equal to the placeholder object `InformLibrary`, which takes no part in play, and is not a valid I7 object. This is not typesafe in I7 terms, but that doesn't matter because initialisation will correct it to a typesafe value before any I7 source text can execute. (`sline1` and `sline2` are entirely unused on when we target Glulx.) Once these variables are defined, the sequence of definition of the rest is not significant.

(2) The `say__*` are used for the finite state machine used in printing text, which keeps track of automatic paragraph breaking and the like. For details see the "Printing.i6t" section. For `subst__v`, see the explanation of "substitution-variable" in the Standard Rules. The variables `ct_0` and `ct_1` hold the current choice of table and table row, respectively, but in some ways these global variables are misleading: in very many routines, `ct_0` and `ct_1` are defined as local variables, and it's the local definitions which take precedence. (The global variables allow table-selecting code to compile in routines where it isn't possible to define these locals because no stack frame exists.)

(3) `standard_interpreter` is used only for the Z-machine VM, and is always 0 for Glulx. For Z, a non-zero value here is the version number of the *Z-Machine Standards Document* which the interpreter claims to support, in the form (upper byte).(lower). `undo_flag`, similarly, behaves slightly differently on the two platforms according to whether they support multiple consecutive UNDOs. UNDO basically works by taking memory snapshots of the whole VM ("saving UNDO") to revert to at a later point ("performing UNDO"), so it is expensive on memory, and traditional VMs can only store a single memory snapshot – making two UNDOs in a row, going back two steps, impossible. Given this, `undo_flag` has three possible states: 0 means UNDO is not available at all, 1 means it is not available now because there is no further saved state to go back to from here, and 2 means it is available.

(4) `deadflag` in normally 0, or false, meaning play continues; 1 means the game ended in death; 2 for ended in victory; higher numbers represent exotic endings. `deadflag` switching state normally triggers an end to rulebook processing, so is the single most important global variable to the running of a story file.

(5) At present, `time_rate` is not made use of in I7: if positive, it is the number of minutes which pass each turn; if negative, the number of turns which pass each minute. This is quite a neat way to approximate a wide range of time steps with an integer such that fractions are exact and we can approximate any duration to a fair accuracy (the worst case being 3/4 minute, where we have to choose between 1 minute or 1/2 minute).

(6) Note that `notify_mode` is irrelevant if the use option "Use no scoring" is in force: it isn't looked at, and can't be changed, and shouldn't have an effect anyway since `score` will never be altered.

(7) `player` is a variable, not a constant, since the focus of play can change. `SACK_OBJECT` is likewise an unexpected variable: in the I6 library, there could only be one player's holdall, a single rucksack-like possession which had to be the value of the constant `SACK_OBJECT`. Here we define `SACK_OBJECT` as a global variable instead, the value of which is the player's holdall currently in use. `visibility_ceiling` is the highest object in the tree visible from the player's point of view: usually the room, but sometimes nothing (in darkness), and sometimes a closed non-transparent container.

(8) See "OrderOfPlay.i6t" for the meaning of action variables.

(9) This is a slate of global variables used by the parser to give some context to the general parsing routines (GPRs) which it calls; in the I6 design, any object can provide its own GPR, in the form of a `parse_name` property. GPRs are in effect parser plug-ins, and I7 makes extensive use of them.

(10) Similarly, variables for the parser to give context to another sort of plug-in routine: a scope filter. I7 uses these too.

(11) The `move_*` variables are specific to the `##Going` action.

(12) These variables hold current settings for listing objects and, more elaborately, performing room descriptions.

(13) The current colour scheme is stored in variables in order that it can be saved in the save game state, and changed correctly on an UNDO: if it were a transient state inside the VM interpreter's screen model, then a RESTORE or UNDO will upset what the original author may have intended the appearance of text in particular scenes to be. (Cf. Adam Cadre's I6 patch `L61007`.)

(14) These pixel dimensions are used both for the Glulx and v6 Z-machines, but not for the more commonly used versions 5 or 8, whose screen model is based on character cells.

(15) Parameters used by `LibraryMessages`: this may be replaced later.

(16) For debugging. `debug_flag` is traditionally called so, but is actually now a bitmap of flags for tracing actions, calls to object routines, and so on.

```
! [1]
Global location = InformLibrary; ! does not = I7 "location": see below
Global sline1; Global sline2;
! [2]
Global say__p = 1; Global say__pc = 0; Global say__n;
Global ct_0 = 0; Global ct_1 = 0;
Global los_rv = false;
Global subst__v; ! = I7 "substitution-variable"
Global parameter_object; ! = I7 "parameter-object" = I7 "container in question"
Array deferred_calling_list --> 16;
Global property_to_be_totalled; ! used to implement "total P of..."
Global property_loop_sign; ! $+1$ for increasing order, $-1$ for decreasing
Global suppress_scope_loops;
Global temporary_value; ! can be used anywhere side-effects can't occur
Global enable_rte = true; ! reporting of run-time problems is enabled
Constant BLOCKV_STACK_SIZE = 224;
Global blockv_sp = 0;
Array blockv_stack --> BLOCKV_STACK_SIZE;
Global IT_RE_Err = 0;
Array LocalParking --> 16;
! [3]
Global standard_interpreter = 0;
Global undo_flag;
! [4]
Global deadflag = 0;
Global resurrect_please = false;
! [5]
Global not_yet_in_play = true; ! set false when first command received
Global turns = 1; ! = I7 "turn count"
Global the_time = NULL; ! = I7 "time of day"
Global time_rate = 1;
Constant NUMBER_SCENES_CREATED = {-value:NUMBER_CREATED(scene)};
Constant SCENE_ARRAY_SIZE = (NUMBER_SCENES_CREATED+2);
Array scene_started --> SCENE_ARRAY_SIZE;
Array scene_ended --> SCENE_ARRAY_SIZE;
```

```
Array scene_status --> SCENE_ARRAY_SIZE;
Array scene_endings --> SCENE_ARRAY_SIZE;
Array scene_latest_ending --> SCENE_ARRAY_SIZE;
{-array:scene_recurs}
! [6]
Global score; ! = I7 "score"
Global last_score; ! = I7 "last notified score"
Global notify_mode = 1; ! score notification on or off
Global left_hand_status_line = SL_Location; ! = I7 "left hand status line"
Global right_hand_status_line = SL_Score_Moves; ! = I7 "right hand status line"
! [7]
Global player; ! = I7 "player"
Global real_location; ! = I7 "location"
Global visibility_ceiling; ! highest object in tree visible to player
Global visibility_levels; ! distance in tree to that
Global SACK_OBJECT; ! current player's holdall item in use
! [8]
Global act_requester;
Global actor; ! = I7 "person asked" = I7 "person reaching"
Global actors_location; ! like real_location, but for the actor
Global actor_location; ! = I7 "actor-location"
Global action;
Global meta; ! action is out of world
Global inp1;
Global inp2;
Array  multiple_object --> MATCH_LIST_WORDS; ! multiple-object list (I6 table array)
Global toomany_flag; ! multiple-object list overflowed
Global multiflag; ! multiple-object being processed
Global multiple_object_item; ! item currently being processed in multiple-object list
Global noun; ! = I7 "noun"
Global second; ! = I7 "second noun"
Global keep_silent; ! true if current action is being tried silently
Global etype; ! parser error number if command not recognised
Global trace_actions = 0;
Global untouchable_object;
Global untouchable_silence;
Global touch_persona;
Global special_word; ! dictionary address of first word in "[text]" token
Global consult_from; ! word number of start of "[text]" token
Global consult_words; ! number of words in "[text]" token
Global parsed_number; ! value from any token not an object
Global special_number1; ! first value, if token not an object
Global special_number2; ! second value, if token not an object
Array  parser_results --> 16; ! for parser to write its results in
Global parser_trace = 0; ! normally 0, but 1 to 5 traces parser workings
Global pronoun_word; ! records which pronoun ("it", "them", ...) caused an error
Global pronoun_obj; ! and what object it was thought to refer to
Global players_command; ! = I7 "player's command"
Global matched_text; ! = I7 "matched text"
Global reason_the_action_failed; ! = I7 "reason the action failed"
Global understand_as_mistake_number; ! which form of "Understand... as a mistake"
```

```
Global particular_possession; ! = I7 "particular possession"
! [9]
Global parser_action; ! written by the parser for the benefit of GPRs
Global parser_one;
Global parser_two;
Global parameters; ! number of I7 tokens parsed on the current line
Global action_to_be; ! (if the current line were accepted)
Global action_reversed; ! (parameters would be reversed in order)
Global wn; ! word number within "parse" buffer (from 1)
Global num_words; ! number of words in buffer
Global verb_word; ! dictionary address of command verb
Global verb_wordnum; ! word number of command verb
! [10]
Global scope_reason = PARSING_REASON; ! current reason for searching scope
Global scope_token; ! for "scope=Routine" grammar tokens
Global scope_error;
Global scope_stage; ! 1, 2 then 3
Global advance_warning; ! what a later-named thing will be
Global reason_code = NULL; ! for the I6 veneer
Global ats_flag = 0; ! for AddToScope routines
Global ats_hls;
! [11]
Global move_pushing;
Global move_from;
Global move_to;
Global move_by;
Global move_through;
! [12]
Global lookmode = 1; ! 1 = standard, 2 = verbose, 3 = brief room descriptions
Global c_style; ! current list-writer style
Global c_depth; ! current recursion depth
Global c_iterator; ! current iteration function
Global lt_value; ! common value of list_together
Global listing_together; ! object number of one member of a group being listed together
Global listing_size; ! size of such a group
Global c_margin; ! current level of indentation printed by WriteListFrom()
Global inventory_stage = 1; ! 1 or 2 according to the context in which list_together uses
! [13]
Global clr_fg = 1; ! foreground colour
Global clr_bg = 1; ! background colour
Global clr_fgstatus = 1; ! foreground colour of statusline
Global clr_bgstatus = 1; ! background colour of statusline
Global clr_on; ! has colour been enabled by the player?
Global statuswin_current; ! if writing to top window
! [14]
Global statuswin_cursize = 0;
Global statuswin_size = 1;
! [15]
Global lm_act; Global lm_n; Global lm_o; Global lm_o2;
! [16]
Global debug_flag = 0;
```

```
Global debug_rules = 0;
Global debug_scenes = 0;
Global debug_rule_nesting;
```

§**7. VM-Specific Code.** These sections of code contain different definitions of the same routines, and in some cases the same arrays, to handle low-level functions in the virtual machine – saving the game, performing UNDO, parsing typed text into dictionary word addresses and so on.

```
#Ifdef TARGET_GLULX;
{-segment:Glulx.i6t}
#Endif;
#Ifdef TARGET_ZCODE;
{-segment:ZMachine.i6t}
#Endif;
```

§**8. Compass.** I6 identified compass directions as being children of the pseudo-object `Compass`; I7 uses membership of the class `CompassDirection` instead. We need both ways here. (Note that `Compass` is not a valid I7 object, and is used for no other purpose.) Because of the traditional structure of language definitions, this needs to come first.

```
Class CompassDirection
with capacity 0, article "the", description [; L__M(##Look, 7, self); ],
has scenery;
Object Compass "compass" has concealed;
```

§**9. Language of Play.** The equivalent of I6's language definition file, though here the idea is that a translation should have an inclusion to replace the "Language.i6t" segment, which contains the English definition.

```
{-segment:Language.i6t}
Default LanguageCases 1;
#Ifndef LibraryMessages; Object LibraryMessages; #Endif;
```

§**10. The Old Library.** The I6 library consisted essentially of the parser, the verb routines, and a pile of utilities and world-modelling code, of which the biggest single component was the list-writer. The parser lives on below; the verb routines are gone, with the equivalent functionality having moved upstairs into I7 source text in the Standard Rules; and the rest of the library largely lives here:

```
{-segment:Light.i6t}
{-segment:ListWriter.i6t}
{-segment:Utilities.i6t}
```

§**11. Parser.** The largest single block of code in the traditional I6 library part of the template layer is the parser.

The two pseudo-objects `InformParser` and `InformLibrary` are relics of the object-oriented approach in I6, and are used only very slightly in the template layer; they are not used at all in I7, and are not valid for the "object" kind of value.

The parser includes arrays for typed text and some parsing information derived from it, and if these should overrun it would cause enigmatic bugs, as the next arrays in memory would be corrupted: as a tripwire, the `Protect_I7_Arrays` array consists of two magic values in sequence. If it is ever discovered to contain the wrong data, the alarm sounds.

```
Object InformParser "(Inform Parser)" has proper;
{-segment:Parser.i6t}
[ ParserError error_type;
    if (error_type ofclass String or Routine) PrintSingleParagraph(error_type);
    rfalse;
];
Object InformLibrary "(Inform Library)" has proper;
Array Protect_I7_Arrays --> 16339 12345;
```

§**12. Order of Play.** The `Main` routine, where execution begins, and the primitive rules in the principal rulebooks.

```
{-segment:OrderOfPlay.i6t}
```

§**13. Properties.** Some either/or properties are compiled to I6 attributes, which must be predeclared, so we do that first. (All other properties can simply be used without declaration.)

What then follows is a table of property metadata: in particular, specifying which properties can be used with which I6 classes or objects. Policing this at run-time costs a little speed, but traps many errors of programming, and keeps everything typesafe. It is the price we pay for the relatively lenient compile-time checking of I7's "object" kind of value. To make it as efficient as possible, we calculate offsets into the metadata: this has to be done (once) at run-time, with the routine compiled.

```
{-call:compile_attributes}
{-array:property_metadata}
Array attribute_offsets --> (50 + {-value:NUMBER_CREATED(property_name)});
Array property_offsets --> (100 + {-value:NUMBER_CREATED(property_name)} + INDIV_PROP_START-48);
{-routine:CreatePropertyOffsets}
```

§**14. Quantities.** Some of these are constants – the enumerated values of new kinds of value with names as values – while others are non-local variables in I7. NI can choose to translate an I7 variable into any I6 lvalue, and usually makes this an entry in an array called `Global_Vars` rather than an I6 global variable: the latter are limited in number, the former are not.

```
{-call:declare_quantities}
{-array:Global_Vars}
```

§**15. Activities.** These are numbered upwards from 0 in order of creation. The following arrays taken together provide, for each activity number: (i) the rulebook numbers for the before, for, and after stages of the activity, and (ii) a flag indicating whether the activity is "future action"-capable, that is, is a parsing activity allowed to make use of the action which conjecturally might result from the current grammar line being parsed. (This is called the "action to be", hence "atb".)

```
Constant NUMBER_RULEBOOKS_CREATED = {-value:NUMBER_CREATED(rulebook)};
{-call:compile_activity_constants}
{-array:Activity_before_rulebooks}
{-array:Activity_for_rulebooks}
{-array:Activity_after_rulebooks}
{-array:Activity_atb_rulebooks}
```

§**16. Relations.**

```
{-array:relation_metadata}
{-call:compile_defined_relations}
```

§**17. Printing Routines.**

```
{-call:compile_data_type_support_routines}
{-routine:I7_Kind_Name}
{-routine:RulebookOutcomePrintingRule}
```

§**18. Object Tree.** The I6 object tree contains `Class` definitions as well as objects, but we precede both with a pseudo-object called `property_numberspace_forcer`. It does nothing except to ensure that properties are declared in I6 in the same sequence as I7 (which need not otherwise happen); it plays no part in play, and is not a valid I7 "object" value.

```
{-log:Compiling the object tree}
{-call:compile_property_numberspace_forcer}
{-call:compile_object_tree}
{-call:compile_direction_object_constants}
{-call:compile_runtime_relation_storage}
```

§**19. Tables.** The initial state of the I6 arrays corresponding to each I7 table: see "Tables.i6t" for details.

```
{-log:Compiling the tables}
{-call:compile_tables}
```

§**20. Equations.** Routines to evaluate from equations.

```
{-log:Compiling the equations}
{-call:compile_equations}
```

§**21. Actions.**

```
{-log:Compiling the named action patterns}
{-call:compile_named_action_patterns}
{-log:Compiling the action routines}
{-array:ActionData}
{-array:ActionCoding}
{-array:ActionHappened}
{-call:compile_action_routines}
{-routine:MistakeActionSub}
```

§**22. Phrases.** The following innocent-looking commands tell NI to compile I6 definitions for all of the non-inline "To..." phrases, and all of the rules which are not I6-written primitives, so it results in a fairly enormous cataract of code.

```
{-log:Compiling the phrases}
{-call:compile_phrases}
{-call:compile_adjectival_phrases}
{-call:compile_list_together_routines}
{-call:compile_los_routines}
```

§**23. Timed Events.** Some of the phrases are simply called in the course of other phrases, but some are rules in rulebooks or in the table of timed events, so those come next:

```
{-array:TimedEventsTable}
{-array:TimedEventTimesTable}
```

§**24. Rulebooks.** The literally hundreds of rulebooks are set up here. (In the end a rulebook is only a (word) array of rule addresses, terminated with a `NULL`.)

```
{-log:Compiling the rulebooks}
{-array:rulebooks_array}
{-call:compile_rulebooks}
{-call:resolve_predeclared_booked_rules}
```

§**25. Scenes.**

```
{-log:Compiling scene details}
{-routine:DetectSceneChange}
{-routine:PrintSceneName}
#IFDEF DEBUG;
{-routine:ShowSceneStatus}
#ENDIF;
```

§**26. The New Library.** The gleaming, aluminium and glass extension to the library: almost all of it material new in I7 usage.

```
{-log:CTNL}
{-segment:Actions.i6t}
{-segment:Activities.i6t}
{-segment:Figures.i6t}
{-segment:FileIO.i6t}
{-segment:MStack.i6t}
{-segment:OutOfWorld.i6t}
{-segment:Printing.i6t}
{-segment:Relations.i6t}
{-segment:RTP.i6t}
{-segment:Rulebooks.i6t}
{-segment:Sort.i6t}
{-segment:Tables.i6t}
{-segment:WorldModel.i6t}
```

§**27. Parsing Tokens.** GPRs, scope and noun filters to be used in grammar lines, but no actual grammar lines as yet.

```
{-callv:prepare_grammar}
{-call:compile_grammar_conditions}
{-log:Compiling GPR tokens for parsing various kinds of value}
{-segment:Number.i6t}
{-segment:Time.i6t}
{-call:compile_type_gprs}
{-log:Compiling noun and scope filter tokens}
{-call:compile_filters}
```

§**28. Testing commands.**

```
#IFDEF DEBUG;
{-call:write_test_text}
{-segment:Tests.i6t}
{-routine:InternalTestCases}
#ENDIF; ! DEBUG
```

§**29. I6 Inclusions.** This paragraph contains no code, by default: it's a hook on which to hang verbatim I6 material.

```
! "Include (- ... -)" inclusions with no specified position appear here.
```

§**30. Entries in constant lists.** Well: most of them, anyway. In particular, all of those which are lists of texts with substitution will be swept up, which is important for timing reasons. A second round later on will catch any later ones.

```
{-call:compile_list_constants}
```

§**31. Text Substitutions.** We now have to be quite careful about the sequence of events. Compiling the text routines is an irrevocable step, after which we must not compile any new text with substitutions. On the other hand we mustn't leave it any later, because a text substitution might contain references to the past, or involve propositions which must be deferred into routines.

```
{-log-phase:Winding up}
{-log:Compiling routines to print text containing substitutions}
{-call:compile_text_routines}
{-callv:allow_no_further_text_subs}
```

§**32. Chronology.** Similarly, this is where we wrap up all references to past tenses: after this point, we cannot safely compile any I7 condition in the past tense.

```
{-log:Compiling chronology}
{-segment:Chronology.i6t}
{-callv:allow_no_further_past_tenses}
```

§**33. Grammar.** This is the trickiest matter of timing. We had to leave the grammar lines until now because the past-tense code above might have needed to investigate whether the player's command matched a given pattern at some time in the past (a case which arose naturally in one of the example games, so which should not be dismissed as an aberration). This is therefore the earliest point at which we can know for sure that no further grammar lines are needed.

```
{-log:Compiling I6 Verb directives}
{-call:compile_grammar_verbs}
#IFTRUE ({-value:no_verb_verb_defined} == 1);
[ UnknownVerb; verb_wordnum = 0; return 'no.verb'; ];
[ PrintVerb v;
    if (v == 'no.verb') { print "do something to"; rtrue; }
    rfalse;
];
#Ifnot;
[ UnknownVerb; rfalse; ]; [ PrintVerb v; rfalse; ];
#ENDIF;
```

§**34. Deferred Propositions.** Most conditions, such as "the score is 10", and descriptions, such as "open doors which are in lighted rooms", are translated by NI into propositions in a form of predicate calculus. Sometimes these can be compiled immediately to I6 code, but other times they involve complicated searches and have to be "deferred" into special routines which will perform them. This is where we compile those routines.

```
{-log:Compiling routines from predicate calculus}
{-call:compile_deferred_propositions}
{-callv:allow_no_further_deferrals}
```

§**35. Miscellaneous Loose Ends.** And we still aren't done, because we still have:

(1) Routines which switch between possible interpretations of phrases by performing run-time type checking. (Note that these cannot involve grammar, or the past tenses, or text substitutions, or deferred propositions.)

(2) Arrays holding constant lists, such as `{2, 3, 4}`, if any.

(3) The string constants, named in the pattern `SC_*`, in alphabetical order. (This ensures that their packed addresses will have unsigned comparison ordering equivalent to alphabetical order.)

(4) "Stub" I6 constants for property names where properties aren't used, to prevent them causing errors if they are referred to in code but not actually present in any object (as can easily happen with extensions presenting optional features which the user chooses not to employ). `cap_short_name` is similarly stubbed: this doesn't correspond to any I7 property, but is used by NI to record capitalised forms of the printed name (which in turn goes into `short_name`).

(5) Counters are used to allocate cells of storage to inline phrases which need a permanent state associated with them: see the Standard Rules. Since all I7 source text has been compiled by now, we know the final values of the counters, and therefore the amount of storage we need to allocate.

(6) Similarly, each "quotation" box needs its own cell of memory.

```
{-call:compile_resolver_routines}
{-call:compile_list_constants}
{-array:ConstantListPointers}
{-call:compile_string_constants}
{-call:compile_stub_properties}
#IFNDEF cap_short_name;
Constant cap_short_name = short_name;
#ENDIF;
{-call:compile_allocated_counter_storage}
Array Runtime_Quotations_Displayed --> {-value:extent_of_runtime_quotations_array};
```

§**36. Block Values.** These are values which are pointers to more elaborate data on the memory heap, rather than values in themselves: they point to "blocks". A section of code handles the heap, and there is then one further section to support each of the kinds of value in question.

```
{-call:compile_heap_allocator}
{-segment:Flex.i6t}
{-segment:BlockValues.i6t}
{-segment:IndexedText.i6t}
{-segment:RegExp.i6t}
{-segment:StoredAction.i6t}
{-segment:Lists.i6t}
{-array:tableoffigures}
{-array:tableofsounds}
{-call:create_block_constants}
```

§**37. Signing off.** And that's all, folks.

```
! End of automatically generated I6 source
! ------------------------------------------------------------------------
```

Definitions Template B/defnt

Purpose

Miscellaneous constant definitions, usually providing symbolic names for otherwise inscrutable numbers, which are used throughout the template layer.

---

§**1. VM Target Constants.** Inform can compile story files for four different virtual machines (or VMs): Z-machine versions 5, 6 and 8, and Glulx.

When compiling to Glulx, the I6 compiler predefines the constant `TARGET_GLULX` and also sets `WORDSIZE` to 4; but when compiling to Z, we shouldn't assume it has this modern habit, so we simulate the same effect.

```
#Ifndef WORDSIZE; ! compiling with Z-code only compiler
Constant TARGET_ZCODE;
Constant WORDSIZE 2;
#Endif;
```

§**2. Wordsize-Dependent Definitions.** The old I6 library used to confuse Z-vs-G with 16-vs-32-bit, but we try to separate these distinctions here, even though at present the Z-machine is our only 16-bit target and Glulx our only 32-bit one. The `WORDSIZE` constant is the word size in bytes, so is the multiplier between `->` and `-->` offsets in I6 pointer syntax.

(1) `NULL` is used, as in C, to represent a null value or pointer. In C, this is conventionally 0, but here it is the maximum unsigned value which can be stored, pointing to the topmost byte in the directly addressable memory map; this means it is also -1 when regarded as a signed twos-complement integer, but we write it as an unsigned hexadecimal address for clarity's sake.

(2) `WORD_HIGHBIT` is the most significant bit in the VM's data word.

(3) `IMPROBABLE_VALUE` is one which is *unlikely but still possible* to be a genuine I7 value. The efficiency of some algorithms depends on how well chosen this is: they would ran badly if we chose 1, for instance.

(4) `MAX_POSITIVE_NUMBER` is the largest representable positive (signed) integer, in twos-complement form.

(5) `REPARSE_CODE` is a magic value used in the I6 library's parser to signal that some code which ought to have been parsing a command has in fact rewritten it, so that the whole command must be re-parsed afresh. (Returning this value is like throwing an exception in a language like Java, though we don't implement it that way.) A comment in the 6/11 library reads: "The parser rather gunkily adds addresses to `REPARSE_CODE` for some purposes. And expects the result to be greater than `REPARSE_CODE` (signed comparison). So Glulx Inform is limited to a single gigabyte of storage, for the moment." Guilty as charged, but the gigabyte story file is a remote prospect for now: even megabyte story files are off the horizon. Anyway, it's this comparison issue which means we need a different value for each possible word size.

```
#Iftrue (WORDSIZE == 2);
Constant NULL = $ffff;
Constant WORD_HIGHBIT = $8000;
Constant WORD_NEXTTOHIGHBIT = $4000;
```

```
Constant IMPROBABLE_VALUE = $7fe3;
Constant MAX_POSITIVE_NUMBER 32767;
Constant MIN_NEGATIVE_NUMBER -32768;
Constant REPARSE_CODE = 10000;
#Endif;
#Iftrue (WORDSIZE == 4);
Constant NULL = $ffffffff;
Constant WORD_HIGHBIT = $80000000;
Constant WORD_NEXTTOHIGHBIT = $40000000;
Constant IMPROBABLE_VALUE = $deadce11;
Constant MAX_POSITIVE_NUMBER 2147483647;
Constant MIN_NEGATIVE_NUMBER -2147483648;
Constant REPARSE_CODE = $40000000;
#Endif;
```

§**3. Z-Machine Definitions.** The Z-machine contains certain special constants and variables at fixed position in its "header"; the addresses of these are given below. See *The Z-Machine Standards Document*, version 1.0, for details.

`INDIV_PROP_START` is the lowest number of any "individual property", an I6 internal quantity defined by the compiler when the target is Glulx but not for Z.

```
#Ifdef TARGET_ZCODE;
Global max_z_object;
Constant INDIV_PROP_START 64;
! Offsets into Z-machine header:
Constant HDR_ZCODEVERSION     = $00;     ! byte
Constant HDR_TERPFLAGS        = $01;     ! byte
Constant HDR_GAMERELEASE      = $02;     ! word
Constant HDR_HIGHMEMORY       = $04;     ! word
Constant HDR_INITIALPC        = $06;     ! word
Constant HDR_DICTIONARY       = $08;     ! word
Constant HDR_OBJECTS          = $0A;     ! word
Constant HDR_GLOBALS          = $0C;     ! word
Constant HDR_STATICMEMORY     = $0E;     ! word
Constant HDR_GAMEFLAGS        = $10;     ! word
Constant HDR_GAMESERIAL       = $12;     ! six ASCII characters
Constant HDR_ABBREVIATIONS    = $18;     ! word
Constant HDR_FILELENGTH       = $1A;     ! word
Constant HDR_CHECKSUM         = $1C;     ! word
Constant HDR_TERPNUMBER       = $1E;     ! byte
Constant HDR_TERPVERSION      = $1F;     ! byte
Constant HDR_SCREENHLINES     = $20;     ! byte
Constant HDR_SCREENWCHARS     = $21;     ! byte
Constant HDR_SCREENWUNITS     = $22;     ! word
Constant HDR_SCREENHUNITS     = $24;     ! word
Constant HDR_FONTWUNITS       = $26;     ! byte
Constant HDR_FONTHUNITS       = $27;     ! byte
Constant HDR_ROUTINEOFFSET    = $28;     ! word
Constant HDR_STRINGOFFSET     = $2A;     ! word
Constant HDR_BGCOLOUR         = $2C;     ! byte
Constant HDR_FGCOLOUR         = $2D;     ! byte
```

```
Constant HDR_TERMCHARS        = $2E;     ! word
Constant HDR_PIXELSTO3        = $30;     ! word
Constant HDR_TERPSTANDARD     = $32;     ! two bytes
Constant HDR_ALPHABET         = $34;     ! word
Constant HDR_EXTENSION        = $36;     ! word
Constant HDR_UNUSED           = $38;     ! two words
Constant HDR_INFORMVERSION    = $3C;     ! four ASCII characters
#Endif;
```

§**4. Glulx Definitions.** We make similar header definitions for Glulx. Extensive further definitions, of constants needed to handle the Glk I/O layer, can be found in the "Infglk" section of "Glulx.i6t"; they are not used in the rest of the template layer, and would only get in the way here.

```
#IFDEF TARGET_GLULX;
Global unicode_gestalt_ok; ! Set if interpreter supports Unicode
! Offsets into Glulx header and start of ROM:
Constant HDR_MAGICNUMBER      = $00;     ! long word
Constant HDR_GLULXVERSION     = $04;     ! long word
Constant HDR_RAMSTART         = $08;     ! long word
Constant HDR_EXTSTART         = $0C;     ! long word
Constant HDR_ENDMEM           = $10;     ! long word
Constant HDR_STACKSIZE        = $14;     ! long word
Constant HDR_STARTFUNC        = $18;     ! long word
Constant HDR_DECODINGTBL      = $1C;     ! long word
Constant HDR_CHECKSUM         = $20;     ! long word
Constant ROM_INFO             = $24;     ! four ASCII characters
Constant ROM_MEMORYLAYOUT     = $28;     ! long word
Constant ROM_INFORMVERSION    = $2C;     ! four ASCII characters
Constant ROM_COMPVERSION      = $30;     ! four ASCII characters
Constant ROM_GAMERELEASE      = $34;     ! short word
Constant ROM_GAMESERIAL       = $36;     ! six ASCII characters
#Endif;
```

§**5. Powers of Two.** I6 lacks support for logical shifts, and the Z-machine opcodes which bear on this are not always well supported, so the I6 library has traditionally used a lookup table for the values of 2^{15-n} where $0 \le n \le 11$.

```
Array PowersOfTwo_TB
--> $$100000000000
    $$010000000000
    $$001000000000
    $$000100000000
    $$000010000000
    $$000001000000
    $$000000100000
    $$000000010000
    $$000000001000
    $$000000000100
    $$000000000010
    $$000000000001;
Array IncreasingPowersOfTwo_TB
```

```
--> $$0000000000000001
    $$0000000000000010
    $$0000000000000100
    $$0000000000001000
    $$0000000000010000
    $$0000000000100000
    $$0000000001000000
    $$0000000010000000
    $$0000000100000000
    $$0000001000000000
    $$0000010000000000
    $$0000100000000000
    $$0001000000000000
    $$0010000000000000
    $$0100000000000000
    $$1000000000000000;
```

§**6. Text Styles.** These are the styles of text distinguished by the template layer, though they are not required to look different from each other on any given VM. The codes are independent of the VM targetted, though in fact they are equal to Glulx style numbers as conventionally used. (The Z-machine renders some as roman, some as bold, but for instance makes `HEADER_VMSTY` and `SUBHEADER_VMSTY` indistinguishable to the eye.) Glulx's system of styles is one of its weakest points from an IF author's perspective, since it is all but impossible to achieve the text effects one might want – boldface, for instance, or red text – and text rendering is almost the only area in which it is clearly inferior to the Z-machine, which it was designed to replace. Still, using these styles when we can will get the most out of it, and for unornamental works Glulx is fine in practice.

```
Constant NORMAL_VMSTY     = 0;
Constant HEADER_VMSTY     = 3;
Constant SUBHEADER_VMSTY  = 4;
Constant ALERT_VMSTY      = 5;
Constant NOTE_VMSTY       = 6;
Constant BLOCKQUOTE_VMSTY = 7;
Constant INPUT_VMSTY      = 8;
```

§**7. Colour Numbers.** These are traditional colour names: quite who it was who thought that azure was the same colour as cyan is now unclear. Colour is, again, not easy to arrange on Glulx, but there is some workaround code.

```
Constant CLR_DEFAULT = 1;
Constant CLR_BLACK   = 2;
Constant CLR_RED     = 3;
Constant CLR_GREEN   = 4;
Constant CLR_YELLOW  = 5;
Constant CLR_BLUE    = 6;
Constant CLR_MAGENTA = 7; Constant CLR_PURPLE  = 7;
Constant CLR_CYAN    = 8; Constant CLR_AZURE   = 8;
Constant CLR_WHITE   = 9;
```

§**8. Window Numbers.** Although Glulx can support elaborate tessalations of windows on screen (if the complexity of handling this can be mastered), the Z-machine has much more limited resources in general, so the template layer assumes a simple screen model: there are just two screen areas, the scrolling main window in which commands are typed and responses printed, and the fixed status line bar at the top of the screen.

```
Constant WIN_ALL     = 0; ! Both windows at once
Constant WIN_STATUS  = 1;
Constant WIN_MAIN    = 2;
```

§**9. Paragraphing Flags.** I am not sure many dictionaries would countenance "to paragraph" as a verb, but never mind: the reference here is to the algorithm used to place paragraph breaks within text, which uses bitmaps composed of the following.

```
Constant PARA_COMPLETED          = 1;
Constant PARA_PROMPTSKIP         = 2;
Constant PARA_SUPPRESSPROMPTSKIP = 4;
Constant PARA_NORULEBOOKBREAKS   = 8;
Constant PARA_CONTENTEXPECTED    = 16;
```

§**10. Descriptors in the Language of Play.** The following constants, which must be different in value from the number of any I6 attribute, are used in the `LanguageDescriptors` table found in the definition of the language of play.

```
Constant POSSESS_PK  = $100;
Constant DEFART_PK   = $101;
Constant INDEFART_PK = $102;
```

§**11. Run-Time Problem Numbers.** The enumeration sequence here must correspond to that in the file of RTP texts, which is used to generate the HTML pages displayed by the Inform user interface when a run-time problem has occurred. (For instance, the file of RTP texts includes the heading "P17 - Can't divide by zero", equivalent to `RTP_DIVZERO` being 17 below.) There is no significance to the sequence, which is simply the historical order in which they were added to I7.

```
Constant RTP_BACKDROP               = 1;
Constant RTP_EXITDOOR               = 2;
Constant RTP_NOEXIT                 = 3;
Constant RTP_CANTCHANGE             = 4;
Constant RTP_IMPREL                 = 5;
Constant RTP_RULESTACK              = 6;
Constant RTP_TOOMANYRULEBOOKS       = 7;
Constant RTP_TOOMANYEVENTS          = 8;
Constant RTP_BADPROPERTY            = 9;
Constant RTP_UNPROVIDED             = 10;
Constant RTP_UNSET                  = 11;
Constant RTP_TOOMANYACTS            = 12;
Constant RTP_CANTABANDON            = 13;
Constant RTP_CANTEND                = 14;
Constant RTP_CANTMOVENOTHING        = 15;
Constant RTP_CANTREMOVENOTHING      = 16;
Constant RTP_DIVZERO                = 17;
Constant RTP_BADVALUEPROPERTY       = 18;
Constant RTP_NOTBACKDROP            = 19;
```

```
Constant RTP_TABLE_NOCOL             = 20;
Constant RTP_TABLE_NOCORR            = 21;
Constant RTP_TABLE_NOROW             = 22;
Constant RTP_TABLE_NOENTRY           = 23;
Constant RTP_TABLE_NOTABLE           = 24;
Constant RTP_TABLE_NOMOREBLANKS      = 25;
Constant RTP_TABLE_NOROWS            = 26;
Constant RTP_TABLE_CANTSORT          = 27;
Constant RTP_NOTINAROOM              = 28;
Constant RTP_BADTOPIC                = 29;
Constant RTP_ROUTELESS               = 30;
Constant RTP_PROPOFNOTHING           = 31;
Constant RTP_DECIDEONWRONGKIND       = 32;
Constant RTP_DECIDEONNOTHING         = 33;
Constant RTP_TABLE_CANTSAVE          = 34;
Constant RTP_TABLE_WONTFIT           = 35;
Constant RTP_TABLE_BADFILE           = 36;
Constant RTP_LOWLEVELERROR           = 37;
Constant RTP_DONTIGNORETURNSEQUENCE  = 38;
Constant RTP_SAYINVALIDSNIPPET       = 39;
Constant RTP_SPLICEINVALIDSNIPPET    = 40;
Constant RTP_INCLUDEINVALIDSNIPPET   = 41;
Constant RTP_LISTWRITERMEMORY        = 42;
Constant RTP_CANTREMOVEPLAYER        = 43;
Constant RTP_CANTREMOVEDOORS         = 44;
Constant RTP_CANTCHANGEOFFSTAGE      = 45;
Constant RTP_MSTACKMEMORY            = 46;
Constant RTP_TYPECHECK               = 47;
Constant RTP_FILEIOERROR             = 48;
Constant RTP_HEAPERROR               = 49;
Constant RTP_LISTRANGEERROR          = 50;
Constant RTP_REGEXPSYNTAXERROR       = 51;
Constant RTP_NOGLULXUNICODE          = 52;
Constant RTP_BACKDROPONLY            = 53;
Constant RTP_NOTTHING                = 54;
Constant RTP_SCENEHASNTSTARTED       = 55;
Constant RTP_SCENEHASNTENDED         = 56;
Constant RTP_NEGATIVEROOT            = 57;
```

§**12. Relation Types.** These names are identical to symbolic names used in the NI source code, and are used to show whether a relation is one to one, various to one, and so forth. `Sym` means that it is forcibly symmetric, `Equiv` that it is an equivalence relation ("in groups"). An implicitly-defined relation is one which can't be used for practical purposes. A relation by routine is one for which $R(x, y)$ is true if and only if a given routine, called with arguments x and y, says so: it is therefore read only.

```
Constant Relation_Implicit   = -1;
Constant Relation_OtoO       = 1;
Constant Relation_OtoV       = 2;
Constant Relation_VtoO       = 3;
Constant Relation_VtoV       = 4;
Constant Relation_Sym_OtoO   = 5;
Constant Relation_Sym_VtoV   = 6;
Constant Relation_Equiv      = 7;
Constant Relation_ByRoutine  = 8;
```

§**13. Template Activities.** These are the activities defined in the Standard Rules. Most, though not all, are carried out by explicit function calls in the template layer, which is why we need their ID numbers: note that NI assigns each activity a unique ID number on creation, counting upwards from 0, and that it processes the Standard Rules before any other source text. (These numbers must correspond *both* to those in the source of NI, *and* to the creation sequence in the Standard Rules.)

```
Constant PRINTING_THE_NAME_ACT           = 0;
Constant PRINTING_THE_PLURAL_NAME_ACT    = 1;
Constant PRINTING_A_NUMBER_OF_ACT        = 2;
Constant PRINTING_ROOM_DESC_DETAILS_ACT  = 3;
Constant LISTING_CONTENTS_ACT            = 4;
Constant GROUPING_TOGETHER_ACT           = 5;
Constant WRITING_A_PARAGRAPH_ABOUT_ACT   = 6;
Constant LISTING_NONDESCRIPT_ITEMS_ACT   = 7;

Constant PRINTING_NAME_OF_DARK_ROOM_ACT  = 8;
Constant PRINTING_DESC_OF_DARK_ROOM_ACT  = 9;
Constant PRINTING_NEWS_OF_DARKNESS_ACT   = 10;
Constant PRINTING_NEWS_OF_LIGHT_ACT      = 11;
Constant REFUSAL_TO_ACT_IN_DARK_ACT      = 12;

Constant CONSTRUCTING_STATUS_LINE_ACT    = 13;
Constant PRINTING_BANNER_TEXT_ACT        = 14;

Constant READING_A_COMMAND_ACT           = 15;
Constant DECIDING_SCOPE_ACT              = 16;
Constant DECIDING_CONCEALED_POSSESS_ACT  = 17;
Constant DECIDING_WHETHER_ALL_INC_ACT    = 18;
Constant CLARIFYING_PARSERS_CHOICE_ACT   = 19;
Constant ASKING_WHICH_DO_YOU_MEAN_ACT    = 20;
Constant PRINTING_A_PARSER_ERROR_ACT     = 21;
Constant SUPPLYING_A_MISSING_NOUN_ACT    = 22;
Constant SUPPLYING_A_MISSING_SECOND_ACT  = 23;
Constant IMPLICITLY_TAKING_ACT           = 24;
Constant STARTING_VIRTUAL_MACHINE_ACT    = 25;

Constant AMUSING_A_VICTORIOUS_PLAYER_ACT = 26;
Constant PRINTING_PLAYERS_OBITUARY_ACT   = 27;
Constant DEALING_WITH_FINAL_QUESTION_ACT = 28;

Constant PRINTING_LOCALE_DESCRIPTION_ACT = 29;
```

```
Constant CHOOSING_NOTABLE_LOCALE_OBJ_ACT = 30;
Constant PRINTING_LOCALE_PARAGRAPH_ACT   = 31;
```

§**14. Template Rulebooks.** Rulebooks are created in a similar way, and again are numbered upwards from 0 in order of creation. These are the ones used in the template layer. (These numbers must correspond *both* to those in the source of NI, *and* to the creation sequence in the Standard Rules.)

```
Constant PROCEDURAL_RB                  = 0;
Constant STARTUP_RB                     = 1;
Constant TURN_SEQUENCE_RB               = 2;
Constant SHUTDOWN_RB                    = 3;
Constant WHEN_PLAY_BEGINS_RB            = 5;
Constant WHEN_PLAY_ENDS_RB              = 6;
Constant ACTION_PROCESSING_RB           = 8;
Constant SETTING_ACTION_VARIABLES_RB    = 9;
Constant SPECIFIC_ACTION_PROCESSING_RB  = 10;
Constant ACCESSIBILITY_RB               = 12;
Constant REACHING_INSIDE_RB             = 13;
Constant REACHING_OUTSIDE_RB            = 14;
Constant VISIBLE_RB                     = 15;
Constant PERSUADE_RB                    = 16;
Constant UNSUCCESSFUL_ATTEMPT_RB        = 17;
Constant AFTER_RB                       = 22;
Constant REPORT_RB                      = 23;
```

§**15. Data Type IDs.** These are filled in automatically by NI, and have the same names as are used in the NI source (and in the Types.i6t section): for instance `NUMBER_TY`.

```
{-call:compile_I6_constants_for_typenames}
```

§**16. Parser Error Numbers.** The traditional ways in which the I6 library's parser, which we adopt here more or less intact, can give up on a player's command. See the *Inform Designer's Manual*, 4th edition, for details.

```
Constant STUCK_PE     = 1;
Constant UPTO_PE      = 2;
Constant NUMBER_PE    = 3;
Constant ANIMA_PE     = 4;
Constant CANTSEE_PE   = 5;
Constant TOOLIT_PE    = 6;
Constant NOTHELD_PE   = 7;
Constant MULTI_PE     = 8;
Constant MMULTI_PE    = 9;
Constant VAGUE_PE     = 10;
Constant EXCEPT_PE    = 11;
Constant VERB_PE      = 12;
Constant SCENERY_PE   = 13;
Constant ITGONE_PE    = 14;
Constant JUNKAFTER_PE = 15;
Constant TOOFEW_PE    = 16;
```

```
Constant NOTHING_PE   = 17;
Constant ASKSCOPE_PE  = 18;
Constant NOTINCONTEXT_PE = 19;
Constant BLANKLINE_PE = 20; ! Not formally a parser error, but used by I7 as if
```

§**17. Scope Searching Reasons.** The parser makes use of a mechanism for searching through the objects currently "in scope", which basically means visible to the actor who would perform the action envisaged by the command being parsed. It is sometimes useful to behave differently depending on why this scope searching is being done, so the following constants enumerate the possibilities.

I6's `EACH_TURN_REASON`, `REACT_BEFORE_REASON` and `REACT_AFTER_REASON` have been removed from this list as no longer meaningful; hence the lacuna in numbering.

```
Constant PARSING_REASON       = 0;
Constant TALKING_REASON       = 1;
Constant EACH_TURN_REASON     = 2;
Constant LOOPOVERSCOPE_REASON = 5;
Constant TESTSCOPE_REASON     = 6;
```

§**18. Token Types.** Tokens are the indecomposable pieces of a grammar line making up a possible reading of a command; some are literal words, others stand for "any named object in scope", and so on. The following codes identify the possibilities. The `*_TOKEN` constants must not be altered without modifying the I6 compiler to match (so, basically, they must not be altered at all).

```
Constant ILLEGAL_TT         = 0;    ! Types of grammar token: illegal
Constant ELEMENTARY_TT      = 1;    !     (one of those below)
Constant PREPOSITION_TT     = 2;    !     e.g. 'into'
Constant ROUTINE_FILTER_TT  = 3;    !     e.g. noun=CagedCreature
Constant ATTR_FILTER_TT     = 4;    !     e.g. edible
Constant SCOPE_TT           = 5;    !     e.g. scope=Spells
Constant GPR_TT             = 6;    !     a general parsing routine

Constant NOUN_TOKEN         = 0;    ! The elementary grammar tokens, and
Constant HELD_TOKEN         = 1;    ! the numbers compiled by I6 to
Constant MULTI_TOKEN        = 2;    ! encode them
Constant MULTIHELD_TOKEN    = 3;
Constant MULTIEXCEPT_TOKEN  = 4;
Constant MULTIINSIDE_TOKEN  = 5;
Constant CREATURE_TOKEN     = 6;
Constant SPECIAL_TOKEN      = 7;
Constant NUMBER_TOKEN       = 8;
Constant TOPIC_TOKEN        = 9;
Constant ENDIT_TOKEN        = 15;   ! Value used to mean "end of grammar line"
```

§**19. GPR Return Values.** GRP stands for "General Parsing Routine", an I6 routine which acts as a grammar token: again, see the *Inform Designer's Manual*, 4th edition, for details.

In Library 6/11, `GPR_NOUN` is defined as `$ff00`, but this would fail on Glulx: it needs to be `$ffffff00` on 32-bit virtual machines. It appears that `GPR_NOUN` to `GPR_CREATURE`, though documented in the old *Inform Translator's Manual*, were omitted when this was consolidated into the DM4, so that they effectively disappeared from view. But they might still be useful for implementing inflected forms of nouns, so we have retained them here regardless.

```
Constant GPR_FAIL           = -1;   ! Return values from General Parsing
Constant GPR_PREPOSITION    = 0;    ! Routines
Constant GPR_NUMBER         = 1;
Constant GPR_MULTIPLE       = 2;
Constant GPR_REPARSE        = REPARSE_CODE;
Constant GPR_NOUN           = -256; ! Reparse, but as |NOUN_TOKEN| this time
Constant GPR_HELD           = GPR_NOUN + 1; ! And so on
Constant GPR_MULTI          = GPR_NOUN + 2;
Constant GPR_MULTIHELD      = GPR_NOUN + 3;
Constant GPR_MULTIEXCEPT    = GPR_NOUN + 4;
Constant GPR_MULTIINSIDE    = GPR_NOUN + 5;
Constant GPR_CREATURE       = GPR_NOUN + 6;
```

§**20. List Styles.** These constants make up bitmaps of the options in use when listing objects.

```
Constant NEWLINE_BIT        = $$0000000000000001; ! New-line after each entry
Constant INDENT_BIT         = $$0000000000000010; ! Indent each entry by depth
Constant FULLINV_BIT        = $$0000000000000100; ! Full inventory information after entry
Constant ENGLISH_BIT        = $$0000000000001000; ! English sentence style, with commas and and
Constant RECURSE_BIT        = $$0000000000010000; ! Recurse downwards with usual rules
Constant ALWAYS_BIT         = $$0000000000100000; ! Always recurse downwards
Constant TERSE_BIT          = $$0000000001000000; ! More terse English style
Constant PARTINV_BIT        = $$0000000010000000; ! Only brief inventory information after entry
Constant DEFART_BIT         = $$0000000100000000; ! Use the definite article in list
Constant WORKFLAG_BIT       = $$0000001000000000; ! At top level (only), only list objects
                                                  ! which have the "workflag" attribute
Constant ISARE_BIT          = $$0000010000000000; ! Print " is" or " are" before list
Constant CONCEAL_BIT        = $$0000100000000000; ! Omit objects with "concealed" or "scenery":
                                                  ! if WORKFLAG_BIT also set, then does not
                                                  ! apply at top level, but does lower down
Constant NOARTICLE_BIT      = $$0001000000000000; ! Print no articles, definite or not
Constant EXTRAINDENT_BIT    = $$0010000000000000; ! New in I7: extra indentation of 1 level
Constant CFIRSTART_BIT      = $$0100000000000000; ! Capitalise first article in list
```

§**21. Lengths Of Time.** Inform measures time in minutes.

```
Constant QUARTER_HOUR = 15;
Constant HALF_HOUR = 30;
Constant ONE_HOUR = 60;
Constant TWELVE_HOURS = 720;
Constant TWENTY_FOUR_HOURS = 1440;
```

§**22. Empty Text.** The I6 compiler does not optimise string compilation: if it needs to compile the (packed, read-only) string `"exemplum"` twice, it will compile two copies. This is slightly wasteful on memory, though in practice the loss is not enough for us to care. But we do want to avoid this in I7 because, to make string-sorting algorithms more efficient, we want direct numerical comparison of packed addresses to be equivalent to string comparison: and that means the text "exemplum" has to be compiled once and once only. There's a general mechanism for this in NI, but the single case most often needed is the empty text, since this is the default value for text variables and properties: we give it a name as follows.

(This works because I6 constant definition is not like C preprocessor macro expansion: `EMPTY_TEXT_VALUE` is equated with the address resulting from compiling `""`, rather than being replaced with the blank text to be recompiled each time.)

```
Constant EMPTY_TEXT_VALUE "";
```

§**23. Empty Table.** Similarly: the default value for the "table" kind of value, a Table containing no rows and no columns.

```
Array TheEmptyTable --> 0 0;
```

§**24. Empty Rulebook.** Similarly. An empty rulebook is one whose array has first word equal to `NULL`: we define a sequence of four `$ff` bytes so that the first word will be `NULL` on either 16-bit or 32-bit VMs. As with the empty text, this array is the value of many empty rulebooks: rulebook arrays are read-only, so there is no risk of a clash.

```
Array EMPTY_RULEBOOK -> $ff $ff $ff $ff;
```

§**25. Empty Set.** The falsity proposition describes the empty set of objects, and is the zero value for the "description" kind of value.

```
[ Prop_Falsity reason obj; return 0; ];
```

§**26. Score and Rankings Table.** The following command tells NI to compile constant definitions for `MAX_SCORE` and/or `RANKING_TABLE`, in cases where there are scores and rankings. If there's no score, we define `MAX_SCORE` as 0 anyway; if there's no ranking table, `RANKING_TABLE` is left undefined, so that we can `#ifdef` this possibility later.

```
{-call:compile_max_score}
Default MAX_SCORE 0;
```

§**27. Template Attributes.** An I6 attribute is equivalent to an I7 "either/or property", though the latter are not always translated into I6 attributes because the Z-machine has only a limited number of attributes to use. Here, we define attributes used by the template.

Many concepts in I6 correspond directly to their successors in I7, even if details may vary. (Value properties are a case in point.) Attributes are the opposite of this: indeed, no I6 concept is more fragmented in its I7 equivalents. All but one of the old I6 library attributes are still used (the `general` attribute, for miscellaneous use, has been removed: it more often invited abuse than use); and a few new attributes have been added. But they are used for a variety of purposes. Some do correspond exactly to either/or properties in I7, but others are a sort of signature for I7 kinds. (So that for I7 use they are read-only.) Others still are used by the template layer as part of the implementation of services for I7, but are not visible to I7 source text as storage.

```
Attribute absent; ! Used to mark objects removed from play
Attribute animate; ! I6-level marker for I7 kind "person"
Attribute clothing; ! = I7 "wearable"
Attribute concealed; ! = I7 "undescribed"
Attribute container; ! I6-level marker for I7 kind "container"
Attribute door; ! I6-level marker for I7 kind "door"
Attribute edible; ! = I7 "edible" vs "inedible"
Attribute enterable; ! = I7 "enterable"
Attribute light; ! = I7 "lighted" vs "dark"
Attribute lockable; ! = I7 "lockable"
Attribute locked; ! = I7 "locked"
Attribute moved; ! = I7 "handled"
Attribute on; ! = I7 "switched on" vs "switched off"
Attribute open; ! = I7 "open" vs "closed"
Attribute openable; ! = I7 "openable"
Attribute scenery; ! = I7 "scenery"
Attribute static; ! = I7 "fixed in place" vs "portable"
Attribute supporter; ! I6-level marker for I7 kind "supporter"
Attribute switchable; ! I6-level marker for I7 kind "device"
Attribute talkable; ! Not currently used by I7, but retained for possible future use
Attribute transparent; ! = I7 "transparent" vs "opaque"
Attribute visited; ! = I7 "visited"
Attribute worn; ! marks that an object tree edge represents wearing

Attribute male; ! not directly used by I7, but available for languages with genders
Attribute female; ! = I7 "female" vs "male"
Attribute neuter; ! = I7 "neuter"
Attribute pluralname; ! = I7 "plural-named"
Attribute proper; ! = I7 "proper-named"
Attribute remove_proper; ! remember to remove proper again when using ChangePlayer next

Attribute initially_carried; ! New in I7
Constant privately_named = initially_carried; ! = I7 "privately-named"; not used at run-time
Attribute mentioned; ! New in I7
Attribute pushable; ! New in I7

Attribute mark_as_room; ! Used in I7 to speed up testing "ofclass K1_room"
Attribute mark_as_thing; ! Used in I7 to speed up testing "ofclass K2_thing"

Attribute workflag; ! = I7 "marked for listing", but basically temporary workspace
Attribute workflag2; ! new in I7 and also temporary workspace
Constant list_filter_permits = initially_carried; ! another I7 listwriter convenience
```

§**28. Template Properties.** As remarked above, these more often correspond to value properties in I7. To an experienced I6 user, though, the death toll of abolished I6 properties in I7 is breathtaking: in alphabetical order, `after`, `cant_go`, `daemon`, `each_turn`, `invent`, `life`, `number`, `orders`, `react_after`, `react_before`, `time_left`, `time_out`, `when_closed`, `when_off`, `when_on`, `when_open`. In May 2008, the direction properties `n_to`, `s_to`, `e_to`, ..., `out_to` joined the list of the missing.

The losses are numerous because of the shift from I6's object orientation to I7's rule orientation: information about the behaviour of objects is no longer thought of as data attached to them. At that, it could have been worse: a few unused I6 library properties have been retained for possible future use.

```
Property add_to_scope; ! used as in I6 to place component parts in scope
Property article "a"; ! used as in I6 to implement articles
Property capacity 100; ! = I7 "carrying capacity"
Property component_child; ! new in I7: forest structure holding "part of" relation
Property component_parent; ! new in I7
Property component_sibling; ! new in I7
Property description; ! = I7 "description"
Property door_dir; ! used to implement two-sided doors, but holds direction object, not a property
Property door_to; ! used as in I6 to implement two-sided doors
Property found_in; ! used as in I6 to implement two-sided doors and backdrops
Property initial; ! = I7 "initial description"
Property list_together; ! used as in I6 to implement "grouping together" activity
Property map_region; ! new in I7
Property parse_name 0; ! used as in I6 to implement "Understand... as..." grammars
Property plural; ! used as in I6 to implement plural names for duplicate objects
Property regional_found_in; ! new in I7
Property room_index; ! new in I7: storage for route-finding
Property short_name 0; ! = I7 "printed name"
Property vector; ! new in I7: storage for route-finding
Property with_key; ! = I7 "matching key"

Property IK_0; ! Instance count of the kind of the current object
Property IK_1; ! These are instance counts within kinds K1, K2, ...
Property IK_2; ! and it is efficient to declare the common ones with Property
Property IK_4; ! since this results in a slightly smaller story file
Property IK_5;
Property IK_6;
Property IK_8;

Property IK1_link; ! These are for linked lists used to make searches faster
Property IK2_link; ! and again it's memory-efficient to declare the common ones
Property IK5_link; !
Property IK6_link; !
Property IK8_link; !

Property articles; ! not used by I7, but an interesting hook in the parser
Property grammar; ! not used by I7, but an interesting hook in the parser
Property inside_description; ! not used by I7, but an interesting hook in the locale code
Property short_name_indef 0; ! not used by I7, but an interesting hook in the listmaker
```

§**29. Loss of Life.** The loss of `life` is so appalling that I6 will not even compile a story file which doesn't define the property number `life` (well, strictly speaking, it checks the presence of constants suggesting the I6 library first, but the template layer does define constants like that). We define it as a null constant to be sure of avoiding any valid property number; I6 being typeless, that enables the veneer to compile again. (The relevant code is in `CA__Pr`, defined in the `veneer.c` section of I6.)

```
Constant life = NULL;
```

§**30. Action Count.** The number of valid I7 actions in existence.

```
Constant ActionCount = {-value:NUMBER_CREATED(action_name)};
```

§**31. Fake Actions.** Though sometimes useful for I6 coding tricks, fake actions – action numbers not corresponding to any action, but different from those of valid actions, and useable with a number of action syntaxes – are not conceptually present in I7 source text. They can only really exist at the I6 level because I6 is typeless; in I7 terms, there is no convenient kind of value which could represent both actions and fake actions while protecting each from confusion with the other.

See the *Inform Designer's Manual*, 4th edition, for what these are used for.

The following fake actions from the I6 library have been dropped here: `##LetGo`, `##Receive`, `##ThrownAt`, `##Prompt`, `##Places`, `##Objects`, `##Order`, `##NotUnderstood`.

```
Fake_Action ListMiscellany;
Fake_Action Miscellany;
Fake_Action PluralFound;
Fake_Action TheSame;
```

OrderOfPlay Template B/ordt

Purpose

The sequence of events in play: the Main routine which runs the startup rulebook, the turn sequence rulebook and the shutdown rulebook; and most of the I6 definitions of primitive rules in those rulebooks.

---

§**1. Main.** This is where every I6 story file begins execution: it can end either by returning, or by a `quit` statement or equivalent opcode. (In I7 this does indeed happen when the quitting the game action is carried out, or when QUIT is typed as a reply to the final question; it's only if the user has altered the shutdown rulebook that we might ever actually return from `Main`.) The return value from `Main` is not meaningful.

The `EarlyInTurnSequence` flag is used to enforce the requirement that the "parse command rule" and "generate action rule" do nothing unless the turn sequence rulebook is being followed directly from `Main`, an anomaly explained in the Standard Rules.

```
Global EarlyInTurnSequence;
[ Main;
    #ifdef TARGET_ZCODE; max_z_object = #largest_object - 255; #endif;
    ProcessRulebook(STARTUP_RB);
    #ifdef DEBUG; InternalTestCases(); #endif;
    while (true) {
        while (deadflag == false) {
            EarlyInTurnSequence = true;
            FollowRulebook(TURN_SEQUENCE_RB);
        }
        if (FollowRulebook(SHUTDOWN_RB) == false) return;
    }
];
```

§**2. Virtual Machine Startup Rule.** Note that we consider the three rulebooks for the "starting the virtual machine" activity, but do not formally carry it out, because that might invoke procedural rules: this early in the run, before the screen can accept text, for instance, procedural rules would be risky. We then delegate to the appropriate VM-specific section of code for the real work. The printing of three blank lines at the start of play is traditional: on early Z-machine interpreters such as InfoTaskForce and Zip it was a necessity because of the way they buffered output. On modern windowed ones it still helps to space the opening text better.

```
[ VIRTUAL_MACHINE_STARTUP_R;
    ProcessRulebook(Activity_before_rulebooks-->STARTING_VIRTUAL_MACHINE_ACT);
    ProcessRulebook(Activity_for_rulebooks-->STARTING_VIRTUAL_MACHINE_ACT);
    ProcessRulebook(Activity_after_rulebooks-->STARTING_VIRTUAL_MACHINE_ACT);
    VM_Initialise();
    print "^^^";
    rfalse;
];
```

§**3. Initial Situation.** The array `InitialSituation` is compiled by NI and contains:

(0) The object number for the player, which is usually `selfobj`.
(1) The object in or on which the player begins, if he does. (This will always be an enterable container or supporter, or `nothing`.)
(2) The room in which the player begins, which is usually the first room created in the source text.
(3) The initial time of day, which is usually 9 AM.

The start object and start room are meaningful only if the player's object is compiled outside of the object tree (as can happen if the source text reads, say, "Mrs Bridges is a woman. The player is Mrs Bridges."): in other circumstances they are often correct, but this must not be relied on.

```
Constant PLAYER_OBJECT_INIS = 0;
Constant START_OBJECT_INIS = 1;
Constant START_ROOM_INIS = 2;
Constant START_TIME_INIS = 3;
{-array:InitialSituation}
```

§**4. Initialise Memory Rule.** This rule amalgamates some minimal initialisations which all need to happen before we can risk using some of the more exotic I7 data types:

(a) The language definition might call for initialisation, although the default language of play (English) does not.
(b) A handful of variables are filled in. `I7_LOOKMODE` is a constant created by the use options "use full-length room descriptions" or "use abbreviated room descriptions", but otherwise not existing. It is particularly important that `player` have the correct value, as the process of initialising the memory heap uses the player as the presumed actor when creating memory representations of literal stored actions where no actor was specified; this is why `player` is initialised here and not in the "position player in model world rule" below. The other interesting point here is that we explicitly set `location` and `real_location` to `nothing`, which is certainly incorrect, even though we know better. We do this so that the "update chronological records rule" cannot see where the player is: see the Standard Rules for an explanation of why this is, albeit perhaps dubiously, a good thing.
(c) We start the machinery needed to check that property accesses are valid during play.
(d) And we initialise the memory allocation heap, and expand the literal constants, as hinted above: these are called "block constants" since they occupy blocks of memory.

The `not_yet_in_play` flag, which is cleared when the first command is about to be read from the keyboard, suppresses the standard status line text: thus, if there is some long text to read before the player finds out where he is, the surprise will not be spoiled.

```
[ INITIALISE_MEMORY_R;
    #Ifdef LanguageInitialise; LanguageInitialise(); #Endif;
    not_yet_in_play = true;
    #ifdef I7_LOOKMODE; lookmode = I7_LOOKMODE; #endif;
    player = InitialSituation-->PLAYER_OBJECT_INIS;
    the_time = InitialSituation-->START_TIME_INIS;
    real_location = nothing;
    location = nothing;
    CreatePropertyOffsets();
    HeapInitialise(); ! Create a completely unused memory allocation heap
    InitialHeapAllocation(); ! Allocate empty blocks for variables, properties, and such
    CreateBlockConstants(); ! Allocate and fill in blocks for constant values
    DistributeBlockConstants(); ! Ensure these exist in multiple independent copies when needed
    rfalse;
];
```

§**5. Seed Random Number Generator Rule.** Unless a seed is provided by NI, and it won't be for released story files, the VM's interpreter is supposed to start up with a good seed in its random number generator: something usually derived from, say, the milliseconds part of the current time of day, which is unlikely to repeat or show any pattern in real-world use. However, early Z-machine interpreters often did this quite badly, starting with poor seed values which meant that the first few random numbers always had something in common (being fairly small in their range, for instance). To obviate this we extract and throw away 100 random numbers to get the generator going, shaking out more obvious early patterns, but that cannot really help much if the VM interpreter's RNG is badly written. "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" (von Neumann).

```
[ SEED_RANDOM_NUMBER_GENERATOR_R i;
    if ({-value:rng_seed_at_start_of_play}) VM_Seed_RNG({-value:rng_seed_at_start_of_play});
    for (i=1: i<=100: i++) random(i);
    rfalse;
];
```

§**6. Position Player In Model World Rule.** This seems as good a place as any to write down the invariant we attempt to maintain for the player's position variables:

(1) The `player` variable is the object through which the player plays, which is always a person: its value is always set by starting from `selfobj` and then making a sequence of 0 or more `ChangePlayer(new_value)` calls. (This enables us to make sure it has the correct property values for printed name and so forth.)

(2) The `location` is always either the current room, a valid I7 room, or `thedark`, which is not a valid I7 object but is distinguishable both from all I7 objects and from `nothing`. The `real_location` is always the current room, which is always a valid I7 room. `location` equals `thedark` if and only if the player does not have light to see by; the routine `SilentlyConsiderLight` updates this without printing any messages to announce the fall or lifting of darkness (hence "silently").

(3) The `player` object is always in the subtree of `real_location`, and is always in a chain $O_1 < O_2 < ... < O_n$ where O_1 is the player, O_n is `real_location`, $n \geq 2$ and $O_2, ..., O_{n-1}$ are all either enterable containers, enterable supporters, or component parts of such. The routine `LocationOf`, applied to the player object, always agrees with `real_location`.

(4) "Floating" objects, such as backdrops and two-sided doors, are in theory present in more than one room at once. In practice they can only be in a single position in the I6 object tree at any one time. The rule is that if they are theoretically present in the `real_location`, then they are actually present in the subtree of `real_location`. The routine `MoveFloatingObjects` updates this, and must be called whenever the player moves from one room to another.

(5) Any objects carried by the player have the I6 `moved` attribute set. The `SACK_OBJECT` variable is always set to the object of kind "player's holdall" which the player has most recently been carrying, or had as a component part of himself. The "note object acquisitions" rule updates this.

These invariants are usually all false before the following rule is executed; they are all true once it has completed. In addition, because the global action variables usually hold details of the action most recently carried out, we initialise these as if the most recent action had been the player waiting. (Nobody ought to use these variables at this point, but in case they do use them by accident in a "when play begins" rule, we want Inform to behave predictably and without type-unsafe values entering code.)

```
[ POSITION_PLAYER_IN_MODEL_R player_to_be;
    player = selfobj;
    player_to_be = InitialSituation-->PLAYER_OBJECT_INIS;
    location = LocationOf(player_to_be);
    if (location == 0) {
        location = InitialSituation-->START_ROOM_INIS;
        if (InitialSituation-->START_OBJECT_INIS)
```

```
            move player_to_be to InitialSituation-->START_OBJECT_INIS;
        else move player_to_be to location;
    }
    if (player_to_be ~= player) { remove selfobj; ChangePlayer(player_to_be); }
    real_location = location; SilentlyConsiderLight();
    NOTE_OBJECT_ACQUISITIONS_R(); MoveFloatingObjects();
    actor = player; act_requester = nothing; actors_location = real_location; action = ##Wait;
    rfalse;
];
```

§**7. Parse Command Rule.** This section contains only two primitive rules from the turn sequence rulebook, the matched pair of the "parse command rule" and the "generate action rule"; the others are found in the sections on Light and Time.

We use almost identically the same parser as that in the I6 library, since it is a well-proven and understood algorithm. The I6 parser returns some of its results in a supplied array (here `parser_results`, though the I6 library used to call this `inputobjs`), but others are in global variables:

(1) The `parser_results` array holds four words, used as indexed by the constants below.
   (a) The action can be a valid I6 action number, or an I6 "fake action", a concept not used overtly in I7. Most valid I6 actions correspond exactly to I7 actions, but in principle it is possible to define (say) extra debugging commands entirely at the I6 level.
   (b) The count `NO_INPS_PRES` is always 0, 1 or 2, and then that many of the next two words are meaningful.
   (c) Each of the "inp" values is either 0, meaning "put the multiple object list here"; or 1, meaning "not an object but a value"; or a valid I6 object. (We use the scoping rules to ensure that any I6 object visible to the parser is also a valid I7 object, so – unlike with actions – we need not distinguish between the two.)

(2) The global variable `actor` is set to the person asked to carry out the command, or is the same as `player` if nobody was mentioned. Thus it will be the object for Floyd in the command FLOYD, GET PERMIT, but will be just `player` in the command EAST.

(3) The global variables `special_number1` and, if necessary, `special_number2` hold values corresponding to the first and second of the "inps" to be returned as 1. Thus, if one of the "inps" is a value and the other is an object, then `special_number1` is that value; only if both are values rather than objects will `special_number2` be used. There is no indication of the data type of these values: I6 is typeless.

(4) At most one of the "inps" is permitted to be 1, referring to a multiple object list. (And a multiple value list is forbidden.) If this happens, the list of objects is stored in an I6 table array (i.e., with the 0th word being the number of subsequent words) called `multiple_object`, and the parser will have set the `toomany_flag` if an overflow occurred – that is, if the list was truncated because it originally called for more than 63 objects.

(5) The global variable `meta` is set if the action is one marked as such in the I6 grammar. A confusion in the design of I6 is that being out of world, as we would say in I7 terms, is associated not with an action as such but with the command verb triggering it. (This in practice caused no trouble since we never used, say, the word SAVE for both saving the game and saving, I don't know, box top coupons.) The state of `meta` returned by the I6 parser does not quite correspond to I7's "out of world" concept, so we will alter it in a few cases.

Some of these conventions are a little odd-looking now: why not simply have a larger results array, rather than this pile of occasionally used variables? The reasons are purely historical: the I6 parser developed gradually over about a decade.

```
Constant ACTION_PRES = 0;
Constant NO_INPS_PRES = 1;
```

```
Constant INP1_PRES = 2;
Constant INP2_PRES = 3;

[ PARSE_COMMAND_R;
    if (EarlyInTurnSequence == false) rfalse; ! Prevent use outside top level
    not_yet_in_play = false;
    Parser__parse(parser_results);
    TreatParserResults(parser_results);
    rfalse;
];
```

§**8. Treat Parser Results.** We don't quite use the results exactly as they are returned by the parser: we make modifications in a few special cases.

(1) `##MistakeAction` is a valid I6 action, but not an I7 one. It is used to implement "Understand ... as a mistake", which provides a short-cut way for I7 source text to specify responses to mistaken guesses at the syntax expected for commands. It can therefore result from a whole variety of different commands, some of which might be flagged `meta`, others not. We forcibly set the `meta` flag: a mistake in guessing the command always happens out of world.

(2) A command in the form PERSON, TELL ME ABOUT SOMETHING is altered to the action resulting from ASK PERSON ABOUT SOMETHING, so that `##Tell` is converted to `##Ask` in these cases.

```
[ TreatParserResults results;
    if (results-->ACTION_PRES == ##MistakeAction) meta = true;
    if (results-->ACTION_PRES == ##Tell && results-->INP1_PRES == player && actor ~= player) {
        results-->ACTION_PRES = ##Ask;
        results-->INP1_PRES = actor; actor = player;
    }
];
```

§**9. Generate Action Rule.** For what are, again, historical reasons to do with the development of I6, the current action is recorded in a slate of global variables:

(1) `actor` is as above; `action` is the I6 action number or fake action number, though in I7 usage no fake actions should ever reach this point.

(2) `act_requester` is the person requesting that the actor should perform the action, or `nothing` if the action is the actor's own choice. In the command FLOYD, MOP FLOOR, the `act_requester` is the player and the actor is Floyd; but when an action arises from a try phrase in I7, such as "try Floyd mopping the floor", `act_requester` is `nothing` because it is Floyd's own decision to do this. (The computer, of course, represents the will-power of all characters other than the player.)

(3) `inp1` and `inp2` are global variables whose contents mean the same as those of `parser_results-->INP1_PRES` and `parser_results-->INP2_PRES`. (This is not duplication, because actions also arise from "try" rather than the parser.)

(4) The variable `multiflag` is set during the processing of a multiple object list, and clear otherwise. (It is used for instance by the Standard Rules to give more concise reports of some successful actions.) Note that it remains set during any knock-on actions caused by actions in the multiple object list: the following rule is the only place where `multiflag` is set or cleared.

(5) `noun` and `second` are global variables which are equal to `inp1` and `inp2` when the latter hold valid object numbers, and are equal to `nothing` otherwise. (This is not duplication either, because it provides us with type-safe access to objects: there is no KOV which can safely represent `inp1` and `inp2`, but `noun` and `second` are valid for the I7 kind of value "object".)

In the following rule, we create this set of variables for the action or multiple action(s) suggested by the parser: each action is sent on to `BeginAction` for processing. Once done, we reset the above variables in

what might seem an odd way: we allow straightforward actions by the player to remain in the variables, but convert requests to other people to the neutral "waiting" action carried out by the player (which is the zero value for actions). Now, in a better world, we would always erase the action like this, because an action once completed ought to be forgotten. The value of `noun` ought to be visible only during the action's processing.

But in practice many I7 users write "every turn" rules which are predicated on what the turn's main action was: say, "Every turn when going: ..." The every turn stage is not until later in the turn sequence, so such rules can only work if we keep the main parser-generated action of the turn in the action variables when we finish up here: so that's what we do. (Note that `BeginAction` preserves the values of the action variables, storing copies on the stack, so whatever may have happened during processing, we finish this routine with the same action variable values that we set at the beginning.)

Finally, note that an out of world action stops the turn sequence early, at the end of action generation: this is what prevents the time of day advancing, every turn rules from firing, and so forth – see the Standard Rules.

```
[ GENERATE_ACTION_R i j k l;
    if (EarlyInTurnSequence == false) rfalse; ! Prevent use outside top level
    EarlyInTurnSequence = false;
    action = parser_results-->ACTION_PRES;
    act_requester = nothing; if (actor ~= player) act_requester = player;
    inp1 = 0; inp2 = 0; multiflag = false;
    if (parser_results-->NO_INPS_PRES >= 1) {
        inp1 = parser_results-->INP1_PRES; if (inp1 == 0) multiflag = true;
    }
    if (parser_results-->NO_INPS_PRES >= 2) {
        inp2 = parser_results-->INP2_PRES; if (inp2 == 0) multiflag = true;
    }
    if (inp1 == 1) {
        noun = special_number1;
    } else noun = inp1;
    if (inp2 == 1) {
        if (inp1 == 1) second = special_number2; else second = special_number1;
    } else second = inp2;
    if (multiflag) {
        if (multiple_object-->0 == 0) { L__M(##Miscellany, 2); return; }
        if (toomany_flag) { toomany_flag = false; L__M(##Miscellany, 1); }
        GenerateMultipleActions();
        multiflag = false;
    } else BeginAction(action, noun, second);
    if ((actor ~= player) || (act_requester)) action = ##Wait;
    actor = player; act_requester = 0;
    if (meta) { RulebookSucceeds(); rtrue; }
    rfalse;
];
```

§**10. Generate Multiple Actions.** So this routine is used to issue the individual actions necessary when a multiple object list has been supplied as either the noun or second noun part of an action generated by the parser. Note that we stop processing the list in the event of the game ending, or of the `location` variable changing its value, which can happen either through movement of the player, or through passage from darkness to light or vice versa.

We use `RunParagraphOn` to omit skipped lines as paragraph breaks between the results from any item in the list: this is both more condensed on screen in ordinary lists, and might allow the user to play tricks such as gathering up reports from a list and delivering them later in some processed way.

```
[ GenerateMultipleActions initial_location k item;
    initial_location = location;
    for (k=1: k<=multiple_object-->0: k++) {
        item = multiple_object-->k;
        RunParagraphOn();
        if (inp1 == 0) { inp1 = item; BeginAction(action, item, second, item); inp1 = 0; }
        else { inp2 = item; BeginAction(action, noun, item, item); inp2 = 0; }
        if (deadflag) return;
        if (location ~= initial_location) { L__M(##Miscellany, 51); return; }
    }
];
```

§**11. Timed Events Rule.** A timed event is a rule stored in the `TimedEventsTable`, an I6 table array: zero entries in this table are ignored, and the sequence is significant only if more than one event goes off at the same moment, in which case earlier entries go off first. Each rule in the table has a corresponding timer value in `TimedEventTimesTable`. If this is negative, it represents a number of turns to go before the event happens – or properly speaking, the number of times the timed events rule is invoked. Otherwise the timer value must be a valid time of day at which the event happens (note that valid times are all non-negative integers). We allow a bracket of 30 minutes after the event time proper; this is designed to cope with situations in which the user sets some timed events, then advances the clock by hand (or uses a long step time, say in which each turn equates to 20 minutes).

Because an event is struck out of the table just before it is fired, it will not continue to go off the rest of the half-hour. Moreover, because the striking out happens *before* rather than after the rule fires, a rule can re-time itself to go off again later, somewhat like the snooze feature on an alarm clock, without the risk of it going off again immediately in the same use of the timed events rule: there is guaranteed to be a blank slot in the timer array at or before the current position because we have just blanked one.

```
[ TIMED_EVENTS_R i event_timer fire rule;
    for (i=1: i<=(TimedEventsTable-->0): i++)
        if ((rule=TimedEventsTable-->i) ~= 0) {
            event_timer = TimedEventTimesTable-->i; fire = false;
            if (event_timer<0) {
                (TimedEventTimesTable-->i)++;
                if (TimedEventTimesTable-->i == 0) fire = true;
            } else {
                if ((the_time >= event_timer) && (the_time < event_timer+30)) fire = true;
            }
            if (fire) {
                TimedEventsTable-->i = 0;
                ProcessRulebook(rule);
            }
        }
    rfalse;
];
```

§**12. Setting Timed Events.** This is the corresponding routine which adds events to the timer tables, and is used to define phrases like "the cuckoo clock explodes in 7 turns from now" or "the cuckoo clock explodes at 4 PM". Here the `rule` would be "cuckoo clock explodes", and the `event_time` would either be 4 PM with `absolute_time` set, or simply 7 with `absolute_time` clear.

Note that the same event can occur only once in the timer tables: a new setting for its firing overwrites an old one. (This ensures that the table does not slowly balloon in size if the user has not been careful to ensure that events always fire.)

```
[ SetTimedEvent rule event_time absolute_time i b;
    for (i=1: i<=(TimedEventsTable-->0): i++) {
        if (rule == TimedEventsTable-->i) { b=i; break; }
        if ((b==0) && (TimedEventsTable-->i == 0)) b=i;
    }
    if (b==0) return RunTimeProblem(RTP_TOOMANYEVENTS);
    TimedEventsTable-->b = rule;
    if (absolute_time) TimedEventTimesTable-->b = event_time;
    else TimedEventTimesTable-->b = -event_time;
];
```

§**13. Setting Time Of Day.** This is the old I6 library routine `SetTime`, which is no longer used in I7 at present; but might be, some day.

```
Global time_step;
[ SetTime t s;
    the_time = t; time_rate = s; time_step = 0;
    if (s < 0) time_step = 0-s;
];
```

§**14. Advance Time Rule.** This rule advances the two measures of the passing of time: the number of `turns` of play, and `the_time` of day.

```
[ ADVANCE_TIME_R;
    turns++;
    if (the_time ~= NULL) {
        if (time_rate >= 0) the_time = the_time+time_rate;
        else {
            time_step--;
            if (time_step == 0) {
                the_time++;
                time_step = -time_rate;
            }
        }
        the_time = the_time % TWENTY_FOUR_HOURS;
    }
    rfalse;
];
```

§**15. Note Object Acquisitions Rule.** See the Standard Rules for comment on this.

```
[ NOTE_OBJECT_ACQUISITIONS_R obj;
    objectloop (obj in player) give obj moved ~concealed;
    #Ifdef RUCKSACK_CLASS;
    objectloop (obj in player)
        if (obj ofclass RUCKSACK_CLASS)
            SACK_OBJECT = obj;
    objectloop (obj ofclass RUCKSACK_CLASS && obj provides component_parent
        && obj.component_parent == player)
        SACK_OBJECT = obj;
    #Endif;
    rfalse;
];
```

§**16. Resurrect Player If Asked Rule.** If a rule in the "when play ends" rulebook set `resurrect_please`, by executing the "resume the game" phrase, then this is where we notice that: making the shutdown rulebook succeed then tells `Main` to fall back into the turn sequence.

```
[ RESURRECT_PLAYER_IF_ASKED_R;
    if (resurrect_please) { RulebookSucceeds(); resurrect_please = false; deadflag = 0; rtrue; }
    rfalse;
];
```

§**17. Ask The Final Question Rule.** And so we come to the bittersweet end: we ask the final question endlessly, until the player gives a reply which takes drastic enough action to destroy the current execution context in the VM, for instance by typing QUIT, RESTART, UNDO or RESTORE. The question and answer are all managed by the activity, which is defined in I7 source text in the Standard Rules.

```
[ ASK_FINAL_QUESTION_R;
    print "^";
    while (true) {
        CarryOutActivity(DEALING_WITH_FINAL_QUESTION_ACT);
        DivideParagraphPoint();
    }
];
```

§**18. Read The Final Answer Rule.** This erases the current command, so is a technique we couldn't use during actual play, but here commands are but a distant memory. So we can use the same buffers for the final question as for game commands.

```
[ READ_FINAL_ANSWER_R;
    DrawStatusLine();
    KeyboardPrimitive(buffer, parse);
    players_command = 100 + WordCount();
    num_words = WordCount();
    wn = 1;
    rfalse;
];
```

§**19. Immediately Restart VM Rule.** Now for four rules acting on typical responses to the final question.

```
[ IMMEDIATELY_RESTART_VM_R; @restart; ];
```

§**20. Immediately Restore Saved Game Rule.** It is almost certainly unnecessary to set `actor` to `player` here, but we do so just in case, because `RESTORE_THE_GAME_R` is protected against doing anything when it thinks it might have been called erroneously through a command like "DAPHNE, RESTORE". (Out of world actions should never be carried out that way, but again, it's a precaution.)

```
[ IMMEDIATELY_RESTORE_SAVED_R; actor = player; RESTORE_THE_GAME_R(); ];
```

§**21. Immediately Quit Rule.**

```
[ IMMEDIATELY_QUIT_R; @quit; ];
```

§**22. Immediately Undo Rule.** An UNDO is disallowed when `turns` is 1, because there is nothing to revert to: but suppose the player died or won as a result of the very first command? Then the game will be over with `turns` equal to 1, but UNDO disallowed, even though there is a saved state to revert to, captured just before that command was typed. To prevent this, we increment `turns` to include the one only partially completed, but only if a command was actually typed. (If the player died as a result of a monstrously unfair rule applied before the very first command had been typed, UNDO is indeed impossible, and `turns` is left at 1.)

```
[ IMMEDIATELY_UNDO_R;
    if (not_yet_in_play == false) turns++;
    Perform_Undo();
    if (not_yet_in_play == false) turns--;
];
```

§**23. Print Obituary Headline Rule.** Finally, definitions of three primitive rules for the "printing the player's obituary" activity.

```
[ PRINT_OBITUARY_HEADLINE_R;
    print "^^    ";
    VM_Style(ALERT_VMSTY);
    print "***";
    if (deadflag == 1) L__M(##Miscellany, 3);
    if (deadflag == 2) L__M(##Miscellany, 4);
    if (deadflag ~= 0 or 1 or 2)  {
        print " ";
        if (deadflag ofclass Routine) (deadflag)();
        if (deadflag ofclass String) print (string) deadflag;
        print " ";
    }
    print "***";
    VM_Style(NORMAL_VMSTY);
    print "^^"; #Ifndef NO_SCORE; print "^"; #Endif;
    rfalse;
];
```

§24. Print Final Score Rule.

```
[ PRINT_FINAL_SCORE_R;
    #Ifndef NO_SCORING; ANNOUNCE_SCORE_R(); #Endif;
    rfalse;
];
```

§25. Display Final Status Line Rule.

```
[ DISPLAY_FINAL_STATUS_LINE_R;
    sline1 = score; sline2 = turns;
    rfalse;
];
```

Actions Template B/actt

Purpose

To try actions by people in the model world, processing the necessary rulebooks.

---

§**1. Summary.** To review: an action is an impulse to do something by a person in the model world. Commands such as DROP POTATO are converted into actions ("dropping the Idaho potato"); sometimes they succeed, sometimes they fail. While they run, the fairly complicated details are stored in a suite of I6 global variables such as `actor`, `noun`, `inp1`, and so on (see "OrderOfPlay.i6t" for details); the running of an action is mainly a matter of processing many rulebooks, chief among them they "action processing rules".

In general, actions can come from five different sources:

(i) As a result of parsing the player's command: there are actually two ways this can happen, one if the command calls for a single action, and another if it calls for a whole run of them (like TAKE ALL). See the rules in "OrderOfPlay.i6t".
(ii) From an I7 "try" phrase, in which case `TryAction` is called.
(iii) From an I6 angle-bracket-notation such as `<Wait>`, though this is a syntax which is deprecated now, and is never normally used in I7. The I6 compiler converts such a syntax into a call to the `R_Process` below.
(iv) Through conversion of an existing action. For instance, "removing the cup from the table" is converted in the Standard Rules to "taking the cup". This is done via a routine called `GVS_Convert`.
(v) When a request is successful, the "carry out requested actions" rule turns the original action – a request by the player, such as is produced by CHOPIN, PLAY POLONAISE – into an action by the person asked, such as "Chopin playing the Polonaise".

Certain exceptional cases can arise for other reasons:

(vi) Implicit taking actions are generated by the "carrying requirements rule" when the actor tries something which requires him to be holding an item which he can see, but is not currently holding.
(vii) A partial but intentionally incomplete form of the "looking" action is generated when describing the new location at the end of a "going" action.

In every case except (vii), the action is carried out by `BeginAction`, the single routine which unifies all of these approaches. Except the last one.

This segment of the template is divided into two: first, the I6 code needed for (i) to (vii), the alternative ways for actions to begin; and secondly the common machinery into which all actions eventually pass.

§**2. Action Data.** This is perhaps a good place to document the `ActionData` array, a convenient table of metadata about the actions. Since this is compiled by NI, the following structure can't be modified here without making matching changes in NI. `ActionData` is an I6 `table` containing a series of fixed-length records, one on each action.

The routine `FindAction` locates the record in this table for a given action number, returning its word offset within the table: the argument -1 means "the current action".

```
Constant AD_ACTION = 0; ! The I6 action number (0 to 4095)
Constant AD_REQUIREMENTS = 1; ! Such as requiring light; a bitmap, see below
Constant AD_NOUN_KOV = 2; ! Kind of value of the first noun
Constant AD_SECOND_KOV = 3; ! Kind of value of the second noun
Constant AD_NAME_BASE = 4; ! Up to five words of a name, padded with nulls
Constant AD_VARIABLES_CREATOR = 9; ! Routine to initialise variables owned
Constant AD_VARIABLES_ID = 10; ! Frame ID for variables owned by action

Constant AD_NAME_LENGTH = 5;
Constant AD_RECORD_SIZE = 11;

[ FindAction fa t;
    if (fa == -1) fa = action;
    t = 1;
    while (t <= ActionData-->0) {
        if (fa == ActionData-->t) return t;
        t = t + AD_RECORD_SIZE;
    }
    rfalse;
];
```

§**3. Requirements Bitmap.** As noted above, the `AD_REQUIREMENTS` field is a bitmap of flags for various possible action requirements:

```
Constant TOUCH_NOUN_ABIT   = $$00000001;
Constant TOUCH_SECOND_ABIT = $$00000010;
Constant LIGHT_ABIT        = $$00000100;
Constant NEED_NOUN_ABIT    = $$00001000;
Constant NEED_SECOND_ABIT  = $$00010000;
Constant OUT_OF_WORLD_ABIT = $$00100000;
Constant CARRY_NOUN_ABIT   = $$01000000;
Constant CARRY_SECOND_ABIT = $$10000000;

[ NeedToCarryNoun;       return TestActionMask(CARRY_NOUN_ABIT); ];
[ NeedToCarrySecondNoun; return TestActionMask(CARRY_SECOND_ABIT); ];
[ NeedToTouchNoun;       return TestActionMask(TOUCH_NOUN_ABIT); ];
[ NeedToTouchSecondNoun; return TestActionMask(TOUCH_SECOND_ABIT); ];
[ NeedLightForAction;    return TestActionMask(LIGHT_ABIT); ];

[ TestActionMask match mask at;
    at = FindAction(-1);
    if (at == 0) rfalse;
    mask = ActionData-->(at+AD_REQUIREMENTS);
    if (mask & match) rtrue;
    rfalse;
];
```

§**4. Try Action.** This is method (ii) in the summary above.

```
[ TryAction req by ac n s stora tbits saved_command text_of_command;
    if (stora) return STORED_ACTION_TY_New(ac, n, s, by, req, stora);
    tbits = req & (16+32);
    req = req & 1;
    @push actor; @push act_requester; @push inp1; @push inp2; @push parsed_number;
    actor = by; if (req) act_requester = player; else act_requester = 0;
    by = FindAction(ac);
    if (by) {
        if (ActionData-->(by+AD_NOUN_KOV) == OBJECT_TY) inp1 = n;
        else { inp1 = 1; parsed_number = n; }
        if (ActionData-->(by+AD_SECOND_KOV) == OBJECT_TY) inp2 = s;
        else { inp2 = 1; parsed_number = s; }
        if (((ActionData-->(by+AD_NOUN_KOV) == UNDERSTANDING_TY) ||
            (ActionData-->(by+AD_SECOND_KOV) == UNDERSTANDING_TY)) && (tbits)) {
            saved_command = INDEXED_TEXT_TY_Create();
            INDEXED_TEXT_TY_Cast(players_command, SNIPPET_TY, saved_command);
            text_of_command = INDEXED_TEXT_TY_Create();
            INDEXED_TEXT_TY_Cast(parsed_number, TEXT_TY, text_of_command);
            SetPlayersCommand(text_of_command);
            if (tbits == 16) {
                n = players_command; inp1 = 1; parsed_number = players_command;
            } else {
                s = players_command; inp2 = 1; parsed_number = players_command;
            }
            BlkFree(text_of_command);
            @push consult_from; @push consult_words;
            consult_from = 1; consult_words = parsed_number - 100;
        }
    }
    BeginAction(ac, n, s, 0, true);
    if (saved_command) {
        @pull consult_words; @pull consult_from;
        SetPlayersCommand(saved_command);
        BlkFree(saved_command);
    }
    @pull parsed_number; @pull inp2; @pull inp1; @pull act_requester; @pull actor;
    TrackActions(true);
];
```

§**5. I6 Angle Brackets.** This is method (iii) in the summary above. The routine here has slightly odd conventions and a curious name which would take too long to explain: neither can be changed without amending the veneer code within the I6 compiler.

```
[ R_Process a i j;
    @push inp1; @push inp2;
    inp1 = i; inp2 = j; BeginAction(a, i, j);
    @pull inp2; @pull inp1;
];
```

§**6. Conversion.** This is method (iv) in the summary above.

```
Global converted_action_outcome = -1;
[ GVS_Convert ac n s;
    converted_action_outcome = BeginAction(ac, n, s);
    rtrue;
];
[ ConvertToGoingWithPush i oldrm newrm infl;
    i=noun;
    if (IndirectlyContains(noun, actor) == false) { move i to actor; infl = true; }
    move_pushing = i;
    oldrm = LocationOf(noun);
    BeginAction(##Go, second);
    newrm = LocationOf(actor);
    move_pushing = nothing; move i to newrm;
    if (newrm ~= oldrm) {
        if (IndirectlyContains(i, player)) TryAction(0, player, ##Look, 0, 0);
        RulebookSucceeds();
    } else RulebookFails();
];
```

§**7. Implicit Take.** This is method (vi) in the summary above.

```
[ ImplicitTake obj ks;
    if (actor == player) L__M(##Miscellany, 69, obj);
    else L__M(##Miscellany, 68, obj);
    ClearParagraphing();
    @push keep_silent; keep_silent = true;
    if (act_requester) TryAction(true, actor, ##Take, obj, nothing);
    else TryAction(false, actor, ##Take, obj, nothing);
    @pull keep_silent;
    if (obj in actor) rtrue;
    rfalse;
];
```

§**8. Look After Going.** This is method (vii) in the summary above.

Fundamentally, room descriptions arise through looking actions, but they are also printed after successful going actions, with a special form of paragraph break (see "Printing.i6t" for an explanation of this). Room descriptions through looking are always given in full, unless we have SUPERBRIEF mode set.

```
[ LookAfterGoing;
    GoingLookBreak();
    AbbreviatedRoomDescription();
];
```

§**9. Abbreviated Room Description.** This is used when we want a room description with the same abbreviation conventions as after a going action, and we don't quite want a looking action fully to take place. We nevertheless want to be sure that the action variables for looking exist, and in particular, we want to set the "room-describing action" variable to the action which was prevailing when the room description was called for. We also set "abbreviated form allowed" to "true": when the ordinary looking action is running, this is "false".

The actual description occurs during `LookSub`, which is the specific action processing stage for the "looking" action: thus, we use the check, carry out, after and report rules as if we were "looking", but are unaffected by before or instead rules.

Uniquely, this pseudo-action does not use `BeginAction`: it works only through the specific action processing rules, not the main action-processing ones, though that is not easy to see from the code below because it is hidden in the call to `LookSub`. The `-Sub` suffix is an I6 usage identifying this as the routine to go along with the action `##Look`, and so it is, but it looks nothing like the `LookSub` of the old I6 library. NI compiles `-Sub` routines like so:

```
[ LookSub; return GenericVerbSub(153,154,155); ];
```

(with whatever rulebook numbers are appropriate). `GenericVerbSub` then runs through the specific action processing stage.

```
[ AbbreviatedRoomDescription  prior_action pos frame_id;
    prior_action = action;
    action = ##Look;
    pos = FindAction(##Look);
    if ((pos) && (ActionData-->(pos+AD_VARIABLES_CREATOR))) {
        frame_id = ActionData-->(pos+AD_VARIABLES_ID);
        Mstack_Create_Frame(ActionData-->(pos+AD_VARIABLES_CREATOR), frame_id);
        ProcessRulebook(SETTING_ACTION_VARIABLES_RB);
        (MStack-->MstVO(frame_id, 0)) = prior_action; ! "room-describing action"
        (MStack-->MstVO(frame_id, 1)) = true; ! "abbreviated form allowed"
    }
    LookSub(); ! The I6 verb routine for "looking"
    if (frame_id) Mstack_Destroy_Frame(ActionData-->(pos+AD_VARIABLES_CREATOR), frame_id);
    action = prior_action;
];
```

§**10. Begin Action.** We now begin the second half of the segment: the machinery which handles all actions.

The significance of 4096 here is that this is how I6 distinguishes genuine actions – numbered upwards in order of creation – from what I6 calls "fake actions" – numbered upwards from 4096. Fake actions are hardly used at all in I7, and certainly shouldn't get here, but it's possible nonetheless using I6 angled-brackets, so... In other respects all we do is to save details of whatever current action is happening onto the stack, and then call `ActionPrimitive`.

```
[ BeginAction a n s moi notrack  rv;
    ChronologyPoint();
    @push action; @push noun; @push second; @push self; @push multiple_object_item;
    action = a; noun = n; second = s; self = noun; multiple_object_item = moi;
    if (action < 4096) rv = ActionPrimitive();
    @pull multiple_object_item; @pull self; @pull second; @pull noun; @pull action;
    if (notrack == false) TrackActions(true);
    return rv;
];
```

§**11. Action Primitive.** This is somewhat different from the I6 library counterpart which gives it its name, but the idea is the same. It has no arguments at all: everything it needs to know is now stored in global variables. The routine looks long, but really contains little: it's all just book-keeping, printing debugging information if ACTIONS is in force, etc., with all of the actual work delegated to the action processing rulebook.

We use a rather sneaky device to handle out-of-world actions, those for which the `meta` flag is set: we make it look to the system as if the "action processing rulebook" is being followed, so that all its variables are created and placed in scope, but at the crucial moment we descend to the specific action processing rules directly instead of processing the main rulebook. This is what short-circuits out of world actions and protects them from before and instead rules: see the Standard Rules for more discussion of this.

```
[ ActionPrimitive  rv p1 p2 p3 p4 p5 frame_id;
    MStack_CreateRBVars(ACTION_PROCESSING_RB);
    if ((keep_silent == false) && (multiflag == false)) DivideParagraphPoint();
    reason_the_action_failed = 0;
    frame_id = -1;
    p1 = FindAction(action);
    if ((p1) && (ActionData-->(p1+AD_VARIABLES_CREATOR))) {
        frame_id = ActionData-->(p1+AD_VARIABLES_ID);
        Mstack_Create_Frame(ActionData-->(p1+AD_VARIABLES_CREATOR), frame_id);
    }
    if (ActionVariablesNotTypeSafe()) {
        if (frame_id ~= -1)
            Mstack_Destroy_Frame(ActionData-->(p1+AD_VARIABLES_CREATOR), frame_id);
        MStack_DestroyRBVars(ACTION_PROCESSING_RB);
        return;
    }
    ProcessRulebook(SETTING_ACTION_VARIABLES_RB);
    #IFDEF DEBUG;
    if ((trace_actions) && (FindAction(-1))) {
        print "["; p1=actor; p2=act_requester; p3=action; p4=noun; p5=second;
        DB_Action(p1,p2,p3,p4,p5);
        print "]^"; ClearParagraphing();
    }
    ++debug_rule_nesting;
    #ENDIF;
    TrackActions(false);
    BeginFollowRulebook();
    if ((meta) && (actor ~= player)) { L__M(##Miscellany, 74, actor); rv = RS_FAILS; }
    else if (meta) { DESCEND_TO_SPECIFIC_ACTION_R(); rv = RulebookOutcome(); }
    else { ProcessRulebook(ACTION_PROCESSING_RB); rv = RulebookOutcome(); }
    #IFDEF DEBUG;
    --debug_rule_nesting;
    if ((trace_actions) && (FindAction(-1))) {
        print "["; DB_Action(p1,p2,p3,p4,p5); print " - ";
        switch (rv) {
            RS_SUCCEEDS: print "succeeded";
            RS_FAILS: print "failed";
                #IFNDEF MEMORY_ECONOMY;
                if (reason_the_action_failed)
                    print " the ",
                        (RulePrintingRule) reason_the_action_failed;
```

```
                #ENDIF;
            default: print "ended without result";
        }
        print "]^"; say__p = 1;
        SetRulebookOutcome(rv); ! In case disturbed by printing activities
    }
    #ENDIF;
    if (rv == RS_SUCCEEDS) UpdateActionBitmap();
    EndFollowRulebook();
    if (frame_id ~= -1) {
        p1 = FindAction(action);
        Mstack_Destroy_Frame(ActionData-->(p1+AD_VARIABLES_CREATOR), frame_id);
    }
    MStack_DestroyRBVars(ACTION_PROCESSING_RB);
    if ((keep_silent == false) && (multiflag == false)) DivideParagraphPoint();
    if (rv == RS_SUCCEEDS) rtrue;
    rfalse;
];
```

§**12. Type Safety.** Some basic action requirements have to be met before we can go any further: if they aren't, then it isn't type-safe even to run the action processing rulebook.

(i) For an out of world action, we set the `meta` flag. Otherwise:

(ii) If either the noun or second noun is a topic, then this is an action arising from parsing (such actions do not arise through the "try" phrase, unless by stored actions in which case this has all happened before and doesn't need to be done again) – the parser places details of which words make up the topic in the I6 global variables `consult_words` and `consult_from`. We convert them to a valid I7 snippet value.

(iii) If either the first or second noun is supposed to be an object but seems here to be a value, or vice versa, we stop with a parser error. (This should be fairly difficult to provoke: NI's type-checking will make it difficult to arrange without I6 subterfuges.)

(iv) If either the first or second noun is supposed to be an object and required to exist, yet is missing, we use the "supplying a missing noun" or "supplying a missing second noun" activities to fill the void.

We return `true` if type safety is violated, `false` if all is well.

```
[ ActionVariablesNotTypeSafe mask noun_kova second_kova at;
    at = FindAction(-1); if (at == 0) rfalse; ! For any I6-defined actions
    mask = ActionData-->(at+AD_REQUIREMENTS);
    if (mask & OUT_OF_WORLD_ABIT) { meta = 1; rfalse; }
    noun_kova = ActionData-->(at+AD_NOUN_KOV);
    second_kova = ActionData-->(at+AD_SECOND_KOV);
    !print "at = ", at, " nst = ", noun_kova, "^";
    !print "consult_from = ", consult_from, " consult_words = ", consult_from, "^";
    !print "inp1 = ", inp1, " noun = ", noun, "^";
    !print "inp2 = ", inp2, " second = ", second, "^";
    !print "sst = ", second_kova, "^";
    if (noun_kova == {-value:SNIPPET_TY} or {-value:UNDERSTANDING_TY}) {
        if (inp1 ~= 1) { inp2 = inp1; second = noun; }
        parsed_number = 100*consult_from + consult_words;
        inp1 = 1; noun = parsed_number;
    }
    if (second_kova == {-value:SNIPPET_TY} or {-value:UNDERSTANDING_TY}) {
        parsed_number = 100*consult_from + consult_words;
```

```
        inp2 = 1; second = parsed_number;
    }
    if (inp1 == 1) {
        if (noun_kova == {-value:OBJECT_TY}) {
            return L__M(##Miscellany, 61); }
    } else {
        if (noun_kova ~= {-value:OBJECT_TY}) {
            return L__M(##Miscellany, 62); }
        if ((mask & NEED_NOUN_ABIT) && (noun == nothing)) {
            @push act_requester; act_requester = nothing;
            CarryOutActivity(SUPPLYING_A_MISSING_NOUN_ACT);
            @pull act_requester;
            if (noun == nothing) {
                if (say__p) rtrue;
                return L__M(##Miscellany, 59);
            }
        }
        if (((mask & NEED_NOUN_ABIT) == 0) && (noun ~= nothing)) {
            return L__M(##Miscellany, 60); }
    }
    if (inp2 == 1) {
        if (second_kova == {-value:OBJECT_TY}) {
            return L__M(##Miscellany, 63); }
    } else {
        if (second_kova ~= {-value:OBJECT_TY}) {
            return L__M(##Miscellany, 64); }
        if ((mask & NEED_SECOND_ABIT) && (second == nothing)) {
            @push act_requester; act_requester = nothing;
            CarryOutActivity(SUPPLYING_A_MISSING_SECOND_ACT);
            @pull act_requester;
            if (second == nothing) {
                if (say__p) rtrue;
                return L__M(##Miscellany, 65);
            }
        }
        if (((mask & NEED_SECOND_ABIT) == 0) && (second ~= nothing)) {
            return L__M(##Miscellany, 66); }
    }
    rfalse;
];
```

§**13. Basic Visibility Rule.** This is one of the I6 primitive rules in the action processing rulebook: see the account in the Standard Rules for details.

Note that this rule only blocks the player from acting in darkness: this is because light is only reckoned from the player's perspective in any case, so that it would be unfair to apply the rule to any other person.

```
[ BASIC_VISIBILITY_R;
    if (act_requester) rfalse;
    if ((NeedLightForAction()) &&
        (actor == player) &&
        (ProcessRulebook(VISIBLE_RB)) &&
        (RulebookSucceeded())) {
        BeginActivity(REFUSAL_TO_ACT_IN_DARK_ACT);
        if (ForActivity(REFUSAL_TO_ACT_IN_DARK_ACT)==false) L__M(##Miscellany, 17);
        EndActivity(REFUSAL_TO_ACT_IN_DARK_ACT);
        reason_the_action_failed = BASIC_VISIBILITY_R;
        RulebookFails();
        rtrue;
    }
    rfalse;
];
```

§**14. Basic Accessibility Rule.** This is one of the I6 primitive rules in the action processing rulebook: see the account in the Standard Rules for details.

```
[ BASIC_ACCESSIBILITY_R mask at;
    if (act_requester) rfalse;
    at = FindAction(-1);
    if (at == 0) rfalse;
    mask = ActionData-->(at+AD_REQUIREMENTS);
    if ((mask & TOUCH_NOUN_ABIT) && noun && (inp1 ~= 1)) {
        if (noun ofclass K3_direction) {
            RulebookFails();
            reason_the_action_failed = BASIC_ACCESSIBILITY_R;
            if (actor~=player) rtrue;
            return L__M(##Miscellany, 67);
        }
        if (ObjectIsUntouchable(noun, (actor~=player), FALSE, actor)) {
            RulebookFails();
            reason_the_action_failed = BASIC_ACCESSIBILITY_R;
            rtrue;
        }
    }
    if ((mask & TOUCH_SECOND_ABIT) && second && (inp2 ~= 1)) {
        if (second ofclass K3_direction) {
            RulebookFails();
            reason_the_action_failed = BASIC_ACCESSIBILITY_R;
            if (actor~=player) rtrue;
            return L__M(##Miscellany, 67);
        }
        if (ObjectIsUntouchable(second, (actor~=player), FALSE, actor)) {
            RulebookFails();
            reason_the_action_failed = BASIC_ACCESSIBILITY_R;
```

```
            rtrue;
        }
    }
    rfalse;
];
```

§**15. Carrying Requirements Rule.** This is one of the I6 primitive rules in the action processing rulebook: see the account in the Standard Rules for details.

```
[ CARRYING_REQUIREMENTS_R mask at;
    at = FindAction(-1);
    if (at == 0) rfalse;
    mask = ActionData-->(at+AD_REQUIREMENTS);
    if ((mask & TOUCH_NOUN_ABIT) && noun && (inp1 ~= 1)) {
        if ((mask & CARRY_NOUN_ABIT) && (noun notin actor)) {
            BeginActivity(IMPLICITLY_TAKING_ACT, noun);
            if (ForActivity(IMPLICITLY_TAKING_ACT, noun)==false)
                ImplicitTake(noun);
            EndActivity(IMPLICITLY_TAKING_ACT, noun);
            !if (act_requester) rfalse;
            if (noun notin actor) {
                RulebookFails();
                reason_the_action_failed = CARRYING_REQUIREMENTS_R;
                rtrue;
            }
        }
    }
    if ((mask & TOUCH_SECOND_ABIT) && second && (inp2 ~= 1)) {
        if ((mask & CARRY_SECOND_ABIT) && (second notin actor)) {
            BeginActivity(IMPLICITLY_TAKING_ACT, second);
            if (ForActivity(IMPLICITLY_TAKING_ACT, second)==false)
                ImplicitTake(second);
            EndActivity(IMPLICITLY_TAKING_ACT, second);
            !if (act_requester) rfalse;
            if (second notin actor) {
                RulebookFails();
                reason_the_action_failed = CARRYING_REQUIREMENTS_R;
                rtrue;
            }
        }
    }
    rfalse;
];
```

§16. Requested Actions Require Persuasion Rule. This is one of the I6 primitive rules in the action processing rulebook: see the account in the Standard Rules for details.

```
[ REQUESTED_ACTIONS_REQUIRE_R;
    if ((actor ~= player) && (act_requester)) {
        @push say__p;
        say__p = 0;
        ProcessRulebook(PERSUADE_RB);
        if (RulebookSucceeded() == false) {
            if (say__p == FALSE) L__M(##Miscellany, 72, actor);
            RulebookFails(); rtrue;
        }
        @pull say__p;
    }
    rfalse;
];
```

§17. Carry Out Requested Actions Rule. This is one of the I6 primitive rules in the action processing rulebook: see the account in the Standard Rules for details.

```
[ CARRY_OUT_REQUESTED_ACTIONS_R rv;
    if ((actor ~= player) && (act_requester)) {
        @push act_requester; act_requester = nothing;
        rv = BeginAction(action, noun, second);
        if ((meta) || (rv == false)) {
            if (ProcessRulebook(UNSUCCESSFUL_ATTEMPT_RB) == false) L__M(##Miscellany, 58);
        }
        @pull act_requester;
        ActRulebookSucceeds();
        rtrue;
    }
    rfalse;
];
```

§18. Generic Verb Subroutine. In I6, actions are carried out by routines with names like `TakeSub`, consisting of `-Sub` tacked on to the action name `Take`. `Sub` stands for "subroutine": this is all a convention going back to Inform 1, which was in 1993 practically an assembler. In the I6 code generated by I7, every `-Sub` routine corresponding to an I7 action consists only of a call to `GenericVerbSub` which specifies the three rulebooks it owns: its check, carry out and report rulebooks.

```
Array Details_of_Specific_Action-->5;
[ GenericVerbSub ch co re vis rv;
    @push converted_action_outcome;
    converted_action_outcome = -1;
    Details_of_Specific_Action-->0 = true;
    if (meta) Details_of_Specific_Action-->0 = false;
    Details_of_Specific_Action-->1 = keep_silent;
    Details_of_Specific_Action-->2 = ch; ! Check rules for the action
    Details_of_Specific_Action-->3 = co; ! Carry out rules for the action
    Details_of_Specific_Action-->4 = re; ! Report rules for the action
    ProcessRulebook(SPECIFIC_ACTION_PROCESSING_RB, 0, true);
```

```
    if ((RulebookFailed()) && (converted_action_outcome == 1)) ActRulebookSucceeds();
    @pull converted_action_outcome;
    rtrue;
];
```

§**19. Work Out Details Of Specific Action Rule.** This is one of the I6 primitive rules in the specific action processing rulebook, and it's basically a trick to allow information known to the `GenericVerbSub` routine to be passed down as rulebook variables for the specific action-processing rules – in effect allowing us to pass not one but five parameters to the rulebook: the out-of-world and silence flags, plus the three specific rulebooks needed to process the action.

```
[ WORK_OUT_DETAILS_OF_SPECIFIC_R;
    MStack-->MstVO(SPECIFIC_ACTION_PROCESSING_RB, 0) = Details_of_Specific_Action-->0;
    MStack-->MstVO(SPECIFIC_ACTION_PROCESSING_RB, 1) = Details_of_Specific_Action-->1;
    MStack-->MstVO(SPECIFIC_ACTION_PROCESSING_RB, 2) = Details_of_Specific_Action-->2;
    MStack-->MstVO(SPECIFIC_ACTION_PROCESSING_RB, 3) = Details_of_Specific_Action-->3;
    MStack-->MstVO(SPECIFIC_ACTION_PROCESSING_RB, 4) = Details_of_Specific_Action-->4;
    rfalse;
];
```

§**20. Actions Bitmap.** This is a fairly large bitmap recording which actions have succeeded thus far on which nouns. It was to some extent an early attempt at implementing a past-tense system; I'm not at all sure it was successful, since it is hindered by certain restrictions – it only records action/noun combinations, for instance, and the notion of "success" is a vexed one for actions anyway. There is a clearly defined meaning, but it doesn't always correspond to what the user might expect, which is unfortunate.

```
[ TestActionBitmap obj act i j k bitmap;
    if (obj == nothing) bitmap = ActionHappened;
    else {
        if (~~(obj provides action_bitmap)) rfalse;
        bitmap = obj.&action_bitmap;
    }
    if (act == -1) return (((bitmap->0) & 1) ~= 0);
    for (i=0, k=2; i<ActionCount; i++) {
        if (act == ActionCoding-->i) {
            return (((bitmap->j) & k) ~= 0);
        }
        k = k*2; if (k == 256) { k = 1; j++; }
    }
    rfalse;
];
[ UpdateActionBitmap;
    SetActionBitmap(noun, action);
    if (action == ##Go) SetActionBitmap(location, ##Enter);
];
[ SetActionBitmap obj act i j k bitmap;
    for (i=0, k=2; i<ActionCount; i++) {
        if (act == ActionCoding-->i) {
            if (obj provides action_bitmap) {
                bitmap = obj.&action_bitmap;
                bitmap->0 = (bitmap->0) | 1;
```

```
                bitmap->j = (bitmap->j) | k;
            }
            ActionHappened->0 = (ActionHappened->0) | 1;
            ActionHappened->j = (ActionHappened->j) | k;
        }
        k = k*2; if (k == 256) { k = 1; j++; }
    }
];
```

§**21. Printing Actions.** This is really for debugging purposes, but also provides us with a way to print a stored action, for instance, or to print an action-name value. (For instance, printing an action-name might result in "taking"; printing a whole action might produce "Henry taking the grapefruit".)

```
[ SayActionName act; DB_Action(0, 0, act, 0, 0, 2); ];
[ DA_Name n; if (n ofclass K3_direction) print (name) n; else print (the) n; ];
[ DA_Topic x a b c d i cf cw;
    cw = x%100; cf = x/100;
    print "~";
    for (a=cf:d<cw:d++,a++) {
        wn = a; b = WordAddress(a); c = WordLength(a);
        for (i=b:i<b+c:i++) {
            print (char) 0->i;
        }
        if (d<cw-1) print " ";
    }
    print "~";
];
[ DA_Number n; print n; ];
[ DA_TruthState n; if (n==0) print "false"; else print "true"; ];
[ DB_Action ac acr act n s for_say t l j v c clc;
    if ((for_say == 0) && (debug_rule_nesting > 0))
        print "(", debug_rule_nesting, ") ";
    if ((ac ~= player) && (for_say ~= 2)) {
        if (acr) print "asking ", (the) ac, " to try ";
        else print (the) ac, " ";
    }
    t = FindAction(act);
    if (t) {
        t = t + AD_NAME_BASE; l = t + AD_NAME_LENGTH;
        while ((v = ActionData-->t) ~= NULL) {
            if (v ofclass Routine) {
                if (for_say == 2) {
                    if (ActionData-->(t+1) ~= NULL) {
                        if (clc++ > 0) print " ";
                        print "it";
                    }
                } else {
                    if (clc++ > 0) print " ";
                    if (c==0) { (v)(n); }
                    else { (v)(s); }
                    c++;
                }
```

```
            } else {
                if (clc++ > 0) print " ";
                print (string) v;
            }
            t++;
            if (t == l) break;
        }
        if ((keep_silent) && (for_say == 0)) print " - silently";
    }
];
```

Activities Template B/acvt

Purpose

To run the necessary rulebooks to carry out an activity.

---

§**1. The Activities Stack.** Activities are more like nested function calls than independent processes; they finish in reverse order of starting, and are placed on a stack. This needs only very limited size in practice: 20 might seem a bit low, but making it much higher simply means that oddball bugs in the user's code – where activities recursively cause themselves ad infinitum – will be caught less efficiently.

```
Constant MAX_NESTED_ACTIVITIES = 20;
Global activities_sp = 0;
Array activities_stack --> MAX_NESTED_ACTIVITIES;
Array activity_parameters_stack --> MAX_NESTED_ACTIVITIES;
```

§**2. Rule Debugging Inhibition.** The output from RULES or RULES ALL becomes totally illegible if it is applied even to the activities printing names of objects, so this is inhibited when any such activity is running. `FixInhibitFlag` is called each time the stack changes and ensures that `inhibit_flag` has exactly this meaning.

```
Global inhibit_flag = 0;
Global saved_debug_rules = 0;
[ FixInhibitFlag n act inhibit_rule_debugging;
    for (n=0:n<activities_sp:n++) {
        act = activities_stack-->n;
        if (act == PRINTING_THE_NAME_ACT or PRINTING_THE_PLURAL_NAME_ACT or
            PRINTING_ROOM_DESC_DETAILS_ACT or LISTING_CONTENTS_ACT or
            GROUPING_TOGETHER_ACT) inhibit_rule_debugging = true;
    }
    if ((inhibit_flag == false) && (inhibit_rule_debugging)) {
        saved_debug_rules = debug_rules;
        debug_rules = 0;
    }
    if ((inhibit_flag) && (inhibit_rule_debugging == false)) {
        debug_rules = saved_debug_rules;
    }
    inhibit_flag = inhibit_rule_debugging;
];
```

§**3. Testing Activities.** The following tests whether a given activity `A` is currently running whose parameter-object matches description `desc`, where as usual the description is represented by a routine testing membership, and where zero `desc` means that any parameter is valid.

```
[ TestActivity A desc i;
    for (i=0:i<activities_sp:i++)
        if (activities_stack-->i == A) {
            if (desc == 0) rtrue;
            if ((desc)(activity_parameters_stack-->i)) rtrue;
        }
    rfalse;
];
```

§**4. Emptiness.** An activity is defined by its three rulebooks: it is empty if they are all empty.

```
[ ActivityEmpty A x;
    x = Activity_before_rulebooks-->A;
    if (((rulebooks_array-->x)-->0) ~= NULL) rfalse;
    x = Activity_for_rulebooks-->A;
    if (((rulebooks_array-->x)-->0) ~= NULL) rfalse;
    x = Activity_after_rulebooks-->A;
    if (((rulebooks_array-->x)-->0) ~= NULL) rfalse;
    rtrue;
];
[ RulebookEmpty rb;
    if (((rulebooks_array-->rb)-->0) ~= NULL) rfalse;
    rtrue;
];
```

§**5. Process Activity Rulebook.** This is really much like processing any rulebook, except that `self` is temporarily set to the parameter, and is preserved by the process.

```
[ ProcessActivityRulebook rulebook parameter  rv;
    @push self;
    if (parameter) self = parameter;
    rv = ProcessRulebook(rulebook, parameter, true);
    @pull self;
    if (rv) rtrue;
    rfalse;
];
```

§**6. Carrying Out Activities.** This is a three-stage process; most activities are run by calling the following simple routine, but some are run by calling the three subroutines independently.

```
[ CarryOutActivity A o rv;
    BeginActivity(A, o);
    rv = ForActivity(A, o);
    EndActivity(A, o);
    return rv;
];
```

§**7. Begin.** Note that when an activity based on the conjectural "future action" is being run – in a few parser-related cases, that is – the identity of this action is put temporarily into `action`, and the current value saved while this takes place. That allows rules in the activity rulebooks to have preambles based on the current action, and yet be tested against what is not yet the current action.

```
[ BeginActivity A o x;
    if (activities_sp == MAX_NESTED_ACTIVITIES) return RunTimeProblem(RTP_TOOMANYACTS);
    activity_parameters_stack-->activities_sp = o;
    activities_stack-->(activities_sp++) = A;
    FixInhibitFlag();
    MStack_CreateAVVars(A);
    if (Activity_atb_rulebooks->A) { x = action; action = action_to_be; }
    o = ProcessActivityRulebook(Activity_before_rulebooks-->A, o);
    if (Activity_atb_rulebooks->A) action = x;
    return o;
];
```

§**8. For.**

```
[ ForActivity A o x;
    if (Activity_atb_rulebooks->A) { x = action; action = action_to_be; }
    o = ProcessActivityRulebook(Activity_for_rulebooks-->A, o);
    if (Activity_atb_rulebooks->A) action = x;
    return o;
];
```

§**9. End.**

```
[ EndActivity A o rv x;
    if ((activities_sp > 0) && (activities_stack-->(activities_sp-1) == A)) {
        if (Activity_atb_rulebooks->A) { x = action; action = action_to_be; }
        rv = ProcessActivityRulebook(Activity_after_rulebooks-->A, o);
        if (Activity_atb_rulebooks->A) action = x;
        activities_sp--; FixInhibitFlag();
        MStack_DestroyAVVars(A);
        return rv;
    }
    return RunTimeProblem(RTP_CANTABANDON);
];
```

§**10. Abandon.** For (very) rare cases where an activity must be abandoned midway; such an activity must be being run by calling the three stages individually, and `EndActivity` must not have been called yet.

```
[ AbandonActivity A o;
    if ((activities_sp > 0) && (activities_stack-->(activities_sp-1) == A)) {
        activities_sp--; FixInhibitFlag();
        MStack_DestroyAVVars(A);
        return;
    }
    return RunTimeProblem(RTP_CANTEND);
];
```

Rulebooks Template B/rbt

Purpose

To work through the rules in a rulebook until a decision is made.

§**1. Rule Change Stack.** The rulebook is a fundamental data structure in Inform 7: it's the basic way in which code is organised. The metaphor is that a rulebook is a loose-leaf ring-binder rather than a bound volume: so we can not only tell NI how to bind the rulebooks at compile time, we can also rearrange the pages during use at run-time.

We keep track of such run-time changes not by actually changing the rulebook arrays but by using the "rule change stack". This is a tally of any temporary abolition or modification of rules: thus, before using any rule from a rulebook we need to check that there is no note of its abolition. As the rule change stack gets larger, rule processing gets slower: this is why it's always more efficient to change rulebooks at compile time than at run-time, if there's a choice between the two.

The rule change stack contains 3-word (6 byte) records, a usage code and two operands (normally both rules).

```
Constant RULECHANGE_STACK_SIZE = 501;
Global rulechange_sp = 0;
Array rulechange_stack --> RULECHANGE_STACK_SIZE;

[ PushRuleChange usage rule1 rule2;
    if (rulechange_sp >= RULECHANGE_STACK_SIZE) return RunTimeProblem(RTP_RULESTACK);
    rulechange_stack-->rulechange_sp++ = usage;
    rulechange_stack-->rulechange_sp++ = rule1;
    rulechange_stack-->rulechange_sp++ = rule2;
];
```

§**2. Usage Codes.** The first word of each record indicates a usage type, of which one value is special and indicates the start of a frame. A new frame is opened on the stack each time a new "follow" occurs; recall that this processes a rulebook after consulting procedural rules, which may in turn modify the behaviour of other rules. These frame divisions, therefore, mark local scopes for modifications: anything from the top of the stack down to the topmost frame marker is currently in force.

There are then seven kinds of frame marking different ways in which rules have been modified; and three further values which indicate possible outcomes. In the following, we refer to the record as containing word 1, one of the usage codes below, word 2 and word 3.

(0) `RS_FRAME` marks the lowest point on the stack of a frame. Words 2 and 3 are not used.

(1) `RS_DONOTRUN` marks the rule or rulebook in word 1 as to be ignored.

(2) `RS_RUN` reinstates the rule or rulebook in word 1. This is useful only to reverse the effect of an `RS_DONOTRUN` frame.

(3) `RS_MOVEBEFORE` moves the rule/rulebook in word 1 to immediately before the rule/rulebook in word 2. If the latter is never invoked, nor is the former.

(4) `RS_MOVEAFTER` moves the rule/rulebook in word 1 to immediately after the rule/rulebook in word 2. If the latter is never invoked, nor is the former.

(5) `RS_DONOTUSE` marks the rule/rulebook in word 1 as to be invoked in the normal way, but then have its result (if any) ignored.

(6) `RS_USE` reverses the effect of an earlier `RS_DONOTUSE` frame.

(7) `RS_SUBSTITUTE` moves the rule/rulebook in word 1 so that it is invoked instead of the rule/rulebook in word 2. If the latter is never invoked, nor is the former.

(8) `RS_SUCCEEDS` occurs only exactly above the top of the stack: it is an ephemeral frame wiped out as soon as the stack is used again. It indicates that the most recent rule or rulebook processed ended in success. Word 2 is a flag: `true` means that a value was returned, `false` that it wasn't. If this is `true` then word 3 contains the value.

(9) `RS_FAILS` is similar, but for a failure. Note that failures can also return values.

(10) `RS_NEITHER` is similar except that it cannot return any value, so that words 2 and 3 are meaningless.

```
Constant RS_FRAME       = -1;
Constant RS_DONOTRUN = 1;
Constant RS_RUN = 2;
Constant RS_MOVEBEFORE = 3;
Constant RS_MOVEAFTER = 4;
Constant RS_DONOTUSE = 5;
Constant RS_USE = 6;
Constant RS_SUBSTITUTE = 7;
Constant RS_SUCCEEDS = 8;
Constant RS_FAILS = 9;
Constant RS_NEITHER = 10;
```

§**3. Following.** At the I6 level, there are two ways to invoke a rulebook: we can "follow" it, or simply "process" it. The former is a grander and slightly slower method which contains the latter. The I7 language talks about "considering" and "abiding by" rules and rulebooks, but these aren't handled at the I6 level: they are both achieved by "process" and the difference between the two is a matter of what is done with the result. (See the Standard Rules for the definitions.)

To "follow" a rulebook, we start a new frame, process the procedural rules, then process the rulebook, then clear the frame back off the stack. (To avoid circularity, the procedural rulebook is the only one which is exempted from the procedural rules.)

```
Global rule_frames = 0; ! Number of frames currently in force
Constant MAX_SIMULTANEOUS_FRAMES = 20;
[ FollowRulebook rulebook parameter no_paragraph_skips  rv;
    @push self;
    if ((Protect_I7_Arrays-->0 ~= 16339) || (Protect_I7_Arrays-->1 ~= 12345)) {
        print "^^*** Fatal programming error: I7 arrays corrupted ***^^";
        @quit;
    }
    if (parameter) self = parameter;
    if (rulebook ~= PROCEDURAL_RB) BeginFollowRulebook();
    rv = ProcessRulebook(rulebook, parameter, no_paragraph_skips);
    if (rulebook ~= PROCEDURAL_RB) EndFollowRulebook();
    @pull self;
    if (rv) rtrue;
    rfalse;
];
[ BeginFollowRulebook;
    PushRuleChange(RS_FRAME, RS_FRAME, RS_FRAME);
    rule_frames++;
    if (rule_frames == MAX_SIMULTANEOUS_FRAMES) {
        RunTimeProblem(RTP_TOOMANYRULEBOOKS);
```

```
        rule_frames = -1; ! For recovery: this terminates rulebook processing
        return;
    }
    ProcessRulebook(PROCEDURAL_RB, 0, true);
];
[ EndFollowRulebook r x y;
    if (rulechange_stack-->rulechange_sp == RS_SUCCEEDS) r = 1;
    else if (rulechange_stack-->rulechange_sp == RS_FAILS) r = 0;
    else r = -1;
    if (r ~= -1) {
        x = rulechange_stack-->(rulechange_sp+1);
        y = rulechange_stack-->(rulechange_sp+2);
    }
    rule_frames--;
    while (rulechange_sp > 0) {
        rulechange_sp = rulechange_sp - 3;
        if (rulechange_stack-->rulechange_sp == RS_FRAME) break;
    }
    if (rulechange_sp == 0) rule_frames = 0;
    if (r == 1) rulechange_stack-->rulechange_sp = RS_SUCCEEDS;
    else if (r == 0) rulechange_stack-->rulechange_sp = RS_FAILS;
    if (r ~= -1) {
        rulechange_stack-->(rulechange_sp+1) = x;
        rulechange_stack-->(rulechange_sp+2) = y;
    }
];
```

§**4. Processing.** The routine `ProcessRulebook` is arguably the most important in the whole of I7. It does something essentially simple but has deceptively complicated implications. To complicate matters, it reuses its variables to keep the virtual machine stack usage to an absolute minimum – here we use 10 locals per call to `ProcessRulebook`, which is the fewest I can comfortably manage. In the early days of I7, stack usage became a serious issue since some forms of the Frotz Z-machine interpreter provided only 4K of stack by default. ("Only" 4K. In the mid-1980s, one of the obstacles facing IF authors at Infocom was the need to get the stack usage down to fewer than 600 bytes in order that the story file could be run on the smaller home computers of the day.)

`ProcessRulebook` takes three arguments, of which only the first is compulsory:

(a) The `rulebook` is an I7 value of kind "rule", which means it can be either the ID number of a rulebook – from 0 up to $N-1$, where N is the number of rulebooks compiled by NI, typically about 600 – or else the address of a routine representing an individual rule.

(b) The `parameter` supplied to the rulebook. Much as arguments can be supplied to a function in a conventional language's function call, so a parameter can be supplied whenever a rulebook is invoked.

(c) The value `bits` is initially a flag: if not supplied, this is `false`; if explicitly set `true`, then the rulebook is run with paragraph breaking suppressed. This is the process by which paragraph division points are placed between rules, so that if two rules both print text then a paragraph break appears between. While that is appropriate for rulebooks attached to actions or for "every turn" rules, it is disastrous for rulebooks attached to activities such as "printing the name of something". Once the routine is running, however, `bits` becomes a bitmap containing five flags, made up from the `RS_*_BIT` values defined below.

`ProcessRulebook` returns `true` if the rulebook or rule chose to "succeed" or "fail", and `false` if it made no choice. (To repeat: if the rule explicitly fails, then `ProcessRulebook` returns `true`. It's easy to write plausible-looking code which goes wrong because it assumes that the return value is success vs. failure.) The outcome of `ProcessRulebook` is lodged just above the top of this stack: thus the most recent rule or rulebook succeeded or failed if –

```
    (rulechange_stack-->rulechange_sp == RS_SUCCEEDS)
    (rulechange_stack-->rulechange_sp == RS_FAILS)
```

and otherwise there was no decision. If there was indeed a decision, then the second word of this record is a flag: `true` means that a value was returned, in which case the third word is that value, and `false` means that no value was returned, so that the third word is meaningless.

```
Constant RS_ACTIVE_BIT = 1;
Constant RS_MOVED_BIT = 2;
Constant RS_USERESULT_BIT = 4;
Constant RS_ACTIVITY = 8;
Constant RS_NOSKIPS = 16;
Constant RS_AFFECTED_BIT = 32;

Global process_rulebook_count; ! Depth of processing recursion
Global debugging_rules = false; ! Are we tracing rule invocations?

[ ProcessRulebook rulebook parameter bits rv
    x frame_base substituted_rule usage original_deadflag rbaddress ra acf gc ga;
    if (bits) bits = RS_ACTIVITY + RS_NOSKIPS;
    if (say__pc & PARA_NORULEBOOKBREAKS) bits = bits | RS_NOSKIPS;
    if (rule_frames<0) rfalse;
    if (parameter) parameter_object = parameter;
    for (x = rulechange_sp-3: x>=0: x = x - 3) {
        usage = rulechange_stack-->x;
        if (usage == RS_FRAME) { x=x+3; break; }
        if (rulechange_stack-->(x+1) == rulebook) {
            bits = bits | (RS_AFFECTED_BIT);
            if (usage == RS_MOVEBEFORE or RS_MOVEAFTER)
                bits = bits | (RS_MOVED_BIT);
        }
        if (rulechange_stack-->(x+2) == rulebook) {
            bits = bits | (RS_AFFECTED_BIT);
        }
    } if (x<0) x=0; frame_base = x;
    if ((bits & RS_MOVED_BIT) && (rv == false)) { rfalse; }
    ! rv was a call parameter: it's no longer needed and is now reused
    bits = bits | (RS_ACTIVE_BIT + RS_USERESULT_BIT);
    substituted_rule = rulebook; rv = 0;
    if (bits & RS_AFFECTED_BIT)
        for (: x<rulechange_sp: x = x + 3) {
            usage = rulechange_stack-->x;
            if (rulechange_stack-->(x+1) == rulebook) {
                if (usage == RS_DONOTRUN) bits = bits & (~RS_ACTIVE_BIT);
                if (usage == RS_RUN) bits = bits | (RS_ACTIVE_BIT);
                if (usage == RS_DONOTUSE) bits = bits & (~RS_USERESULT_BIT);
                if (usage == RS_USE) bits = bits | (RS_USERESULT_BIT);
                if (usage == RS_SUBSTITUTE)
                    substituted_rule = rulechange_stack-->(x+2);
            }
            if ((usage == RS_MOVEBEFORE) && (rulechange_stack-->(x+2) == rulebook)) {
                rv = ProcessRulebook(rulechange_stack-->(x+1),
                    parameter, (bits & RS_ACTIVITY ~= 0), true);
                if (rv) return rv;
            }
        }
```

```
if ((bits & RS_ACTIVE_BIT) == 0) rfalse;
! We now reuse usage to keep the stack frame slimmer
usage = debugging_rules;
#ifndef MEMORY_ECONOMY;
if (debugging_rules) DebugRulebooks(substituted_rule, parameter);
#endif;
! (A routine defined in the I7 code generator)
process_rulebook_count = process_rulebook_count + debugging_rules;
if ((substituted_rule >= 0) && (substituted_rule < NUMBER_RULEBOOKS_CREATED)) {
    rbaddress = rulebooks_array-->substituted_rule;
    ra = rbaddress-->0; x = 0; original_deadflag = deadflag;
    if (ra ~= NULL) {
        acf = (bits & RS_ACTIVITY ~= 0);
        if (substituted_rule ~= ACTION_PROCESSING_RB) MStack_CreateRBVars(substituted_rule);
        if (ra == (-2)) {
            for (x=1: original_deadflag == deadflag: x++) {
                ra = rbaddress-->x;
                if (ra == NULL) break;
                if (gc == 0) {
                    ga = ra; x++; gc = rbaddress-->x;
                    if ((gc<1) || (gc>31)) { gc = 1; x--; }
                    x++; ra = rbaddress-->x;
                }
                gc--;
                if (ga ~= (-2) or action) continue;
                if ((rv = (ProcessRulebook(ra, parameter, acf)))
                    && (bits & RS_USERESULT_BIT)) jump NonNullResult;
            }
        } else {
            for (: original_deadflag == deadflag: x++) {
                ra = rbaddress-->x;
                if (ra == NULL) break;
                if ((rv = (ProcessRulebook(ra, parameter, acf)))
                    && (bits & RS_USERESULT_BIT)) jump NonNullResult;
            }
        }
        rv = 0;
        .NonNullResult;
        if (substituted_rule ~= ACTION_PROCESSING_RB) MStack_DestroyRBVars(substituted_rule);
    }
} else {
    if ((say__p) && (bits & RS_NOSKIPS == 0)) DivideParagraphPoint();
    rv = indirect(substituted_rule);
    if (rv) rv = substituted_rule;
}
if (rv && (bits & RS_USERESULT_BIT)) {
    process_rulebook_count = process_rulebook_count - debugging_rules;
    if (process_rulebook_count < 0) process_rulebook_count = 0;
    #ifndef MEMORY_ECONOMY;
    if (debugging_rules) {
        spaces(2*process_rulebook_count);
        if (rulechange_stack-->rulechange_sp == RS_SUCCEEDS)
            print "[stopped: success]^";
```

```
            if (rulechange_stack-->rulechange_sp == RS_FAILS)
                print "[stopped: fail]^";
        }
        #endif;
        debugging_rules = usage;
        return rv;
    }
    if (bits & RS_AFFECTED_BIT)
        for (x=rulechange_sp-3: x>=frame_base: x = x-3) {
            if ((rulechange_stack-->x == RS_MOVEAFTER) &&
                (rulechange_stack-->(x+2) == rulebook)) {
                rv = ProcessRulebook(rulechange_stack-->(x+1),
                    parameter, (bits & RS_ACTIVITY ~= 0), true);
                if (rv) {
                    process_rulebook_count--;
                    debugging_rules = usage;
                    return rv;
                }
            }
        }
    process_rulebook_count = process_rulebook_count - debugging_rules;
    rulechange_stack-->rulechange_sp = 0;
    debugging_rules = usage;
    rfalse;
];
```

§**5. Specifying Outcomes.** The following provide ways for rules to succeed, fail or decline to do either.

`SetRulebookOutcome` is a little different: it changes the outcome state of the most recent rule completed, not the current one. (It's used only when saving and restoring this in the actions machinery: rules should not call it.)

```
[ ActRulebookSucceeds rule_id;
    if (rule_id) reason_the_action_failed = rule_id;
    RulebookSucceeds();
];
[ ActRulebookFails rule_id;
    if (rule_id) reason_the_action_failed = rule_id;
    RulebookFails();
];
[ RulebookSucceeds valueflag value;
    PushRuleChange(RS_SUCCEEDS, valueflag, value);
    rulechange_sp = rulechange_sp - 3;
];
[ RulebookFails valueflag value;
    PushRuleChange(RS_FAILS, valueflag, value);
    rulechange_sp = rulechange_sp - 3;
];
[ RuleHasNoOutcome;
    PushRuleChange(RS_NEITHER, 0, 0);
    rulechange_sp = rulechange_sp - 3;
];
```

```
[ SetRulebookOutcome a;
    rulechange_stack-->rulechange_sp = a;
];
```

§**6. Discovering Outcomes.** And here is how to tell what the results were.

```
[ RulebookOutcome a;
    a = rulechange_stack-->rulechange_sp;
    if ((a == RS_FAILS) || (a == RS_SUCCEEDS)) return a;
    return RS_NEITHER;
];
[ RulebookFailed;
    if (rulechange_stack-->rulechange_sp == RS_FAILS) rtrue; rfalse;
];
[ RulebookSucceeded;
    if (rulechange_stack-->rulechange_sp == RS_SUCCEEDS) rtrue; rfalse;
];
[ ResultOfRule a;
    a = rulechange_stack-->rulechange_sp;
    if ((a == RS_FAILS) || (a == RS_SUCCEEDS)) {
        a = rulechange_stack-->(rulechange_sp + 1);
        if (a) return rulechange_stack-->(rulechange_sp + 2);
    }
    return 0;
];
```

§**7. Procedural Rule Changes.** The following routines provide a sort of rule-changing API, and correspond closely to the I7 phrases documented in *Writing with Inform*, so they won't be discussed in any detail here.

```
Global DITS_said = false;
[ SuppressRule rule;
    if (rule == TURN_SEQUENCE_RB) {
        if (DITS_said == false) RunTimeProblem(RTP_DONTIGNORETURNSEQUENCE);
        DITS_said = true;
    } else PushRuleChange(RS_DONOTRUN, rule, 0);
];
[ ReinstateRule rule; PushRuleChange(RS_RUN, rule, 0); ];
[ DonotuseRule rule; PushRuleChange(RS_DONOTUSE, rule, 0); ];
[ UseRule rule; PushRuleChange(RS_USE, rule, 0); ];
[ SubstituteRule rule1 rule2; PushRuleChange(RS_SUBSTITUTE, rule2, rule1); ];
[ MoveRuleBefore rule1 rule2; PushRuleChange(RS_MOVEBEFORE, rule1, rule2); ];
[ MoveRuleAfter rule1 rule2; PushRuleChange(RS_MOVEAFTER, rule1, rule2); ];
```

§**8. Printing Rule Names.** This is the I6 printing rule used for a value of kind "rule", which as noted above can either be rulebook ID numbers in the range 0 to $N-1$ or are addresses of individual rules.

Names of rules and rulebooks take up a fair amount of space, and one of the main memory economies enforced by the "Use memory economy" option is to omit the necessary arrays. (It's not the text which is the problem so much as the table of addresses pointing to that text, which has to live in precious readable memory on the Z-machine.)

```
#IFNDEF MEMORY_ECONOMY;
{-array:RulebookNames}
#ENDIF; ! MEMORY_ECONOMY
[ RulePrintingRule R p1;
#ifndef MEMORY_ECONOMY;
    if ((R>=0) && (R<NUMBER_RULEBOOKS_CREATED)) {
        print (string) (RulebookNames-->R);
    } else {
{-call:compile_rule_printing_switch}
        print "(nameless rule at address ", R, ")";
    }
#ifnot;
    if ((R>=0) && (R<NUMBER_RULEBOOKS_CREATED)) {
        print "(rulebook ", R, ")";
    } else {
        print "(rule at address ", R, ")";
    }
#endif;
];
```

§**9. Debugging.** Two modest routines to print out the names of rules and rulebooks when they occur, in so far as memory economy allows this.

```
[ DebugRulebooks subs parameter i;
    spaces(2*process_rulebook_count);
    print "[", (RulePrintingRule) subs;
    if (parameter) print " / on O", parameter;
    print "]^";
];
[ DB_Rule R N blocked;
    if (R==0) return;
    print "[Rule ~", (RulePrintingRule) R, "~ ";
    #ifdef NUMBERED_RULES; print "(", N, ") "; #endif;
    if (blocked == false) "applies.]";
    "does not apply.]";
];
```

Parser Template B/parst

Purpose

The parser for turning the text of the typed command into a proposed action by the player.

---

§**1. Grammar Line Variables.** This is the I6 library parser in mostly untouched form: reformatted for template file use, and with paragraph divisions, but otherwise hardly changed at all. It is a complex algorithm but one which is known to produce good results for the most part, and it is well understood from (at time of writing) fifteen years of use. A few I7 additions have been made, but none disrupting the basic method. For instance, I7's system for resolving ambiguities is implemented by providing a `ChooseObjects` routine, just as a user of the I6 library would do.

The I6 parser uses a huge number of global variables, which is not to modern programming tastes: in the early days of Inform, the parser was essentially written in assembly-language only lightly structured by C-like syntaxes, and the Z-machine's 240 globals were more or less registers. The I6 library made no distinction between which were "private" to the parser and which allowed to be accessed by the user's code at large. The I7 template does impose that boundary, though not very strongly: the variables defined in "Output.i6t" are for general access, while the ones below should only be read or written by the parser.

```
Global best_etype;                  ! Preferred error number so far
Global nextbest_etype;              ! Preferred one, if ASKSCOPE_PE disallowed
Global parser_inflection;           ! A property (usually "name") to find object names in
Array pattern --> 32;               ! For the current pattern match
Global pcount;                      ! and a marker within it
Array pattern2 --> 32;              ! And another, which stores the best match
Global pcount2;                     ! so far
Array  line_ttype-->32;             ! For storing an analysed grammar line
Array  line_tdata-->32;
Array  line_token-->32;
Global nsns;                        ! Number of special_numbers entered so far
Global params_wanted;               ! Number of parameters needed (which may change in parsing)
Global inferfrom;                   ! The point from which the rest of the command must be inferred
Global inferword;                   ! And the preposition inferred
Global dont_infer;                  ! Another dull flag
Global cobj_flag = 0;
Global oops_from;                   ! The "first mistake" word number
Global saved_oops;                  ! Used in working this out
Array  oops_workspace -> 64;        ! Used temporarily by "oops" routine
```

```
Global held_back_mode;              ! Flag: is there some input from last time
Global hb_wn;                       ! left over?  (And a save value for wn.)
                                      ! (Used for full stops and "then".)
Global usual_grammar_after;         ! Point from which usual grammar is parsed (it may vary from
                                      ! the above if user's routines match multi-word verbs)
```

§**2. Grammar Token Variables.** More globals, but dealing at the level of individual tokens now.

```
Constant PATTERN_NULL = $ffff;      ! Entry for a token producing no text
Global found_ttype;                 ! Used to break up tokens into type
Global found_tdata;                 ! and data (by AnalyseToken)
Global token_filter;                ! For noun filtering by user routines
Global length_of_noun;              ! Set by NounDomain to no of words in noun
Global lookahead;                   ! The token after the one now being matched
Global multi_mode;                  ! Multiple mode
Global multi_wanted;                ! Number of things needed in multitude
Global multi_had;                   ! Number of things actually found
Global multi_context;               ! What token the multi-obj was accepted for
Global indef_mode;                  ! "Indefinite" mode - ie, "take a brick"
                                      ! is in this mode
Global indef_type;                  ! Bit-map holding types of specification
Global indef_wanted;                ! Number of items wanted (INDEF_ALL_WANTED for all)
Constant INDEF_ALL_WANTED = 32767;
Global indef_guess_p;               ! Plural-guessing flag
Global indef_owner;                 ! Object which must hold these items
Global indef_cases;                 ! Possible gender and numbers of them
Global indef_possambig;             ! Has a possibly dangerous assumption
                                      ! been made about meaning of a descriptor?
Global indef_nspec_at;              ! Word at which a number like "two" was parsed
                                      ! (for backtracking)
Global allow_plurals;               ! Whether plurals presently allowed or not
Global take_all_rule;               ! Slightly different rules apply to "take all" than other uses
                                      ! of multiple objects, to make adjudication produce more
                                      ! pragmatically useful results
                                      ! (Not a flag: possible values 0, 1, 2)
Global dict_flags_of_noun;          ! Of the noun currently being parsed
                                      ! (a bitmap in #dict_par1 format)
Global pronoun__word;               ! Saved value
Global pronoun__obj;                ! Saved value
Constant comma_word = 'comma,';     ! An "untypeable word" used to substitute
                                      ! for commas in parse buffers
```

§**3. Match List Variables.** The most difficult tokens to match are those which refer to objects, since there is such a variety of names which can be given to any individual object, and we don't of course know which object or objects are meant. We store the possibilities (up to `MATCH_LIST_WORDS`, anyway) in a data structure called the match list.

```
Array  match_list --> MATCH_LIST_WORDS;    ! An array of matched objects so far
Array  match_classes --> MATCH_LIST_WORDS; ! An array of equivalence classes for them
Array  match_scores --> MATCH_LIST_WORDS;  ! An array of match scores for them
Global number_matched;              ! How many items in it?  (0 means none)
Global number_of_classes;           ! How many equivalence classes?
Global match_length;                ! How many words long are these matches?
Global match_from;                  ! At what word of the input do they begin?
```

§**4. Words.** The player's command is broken down into a numbered sequence of words, which break at spaces or certain punctuation (see the DM4). The numbering runs upwards from 1 to `WordCount()`. The following utility routines provide access to words in the current command; because buffers have different definitions in Z and Glulx, so these routines must vary also.

The actual text of each word is stored as a sequence of ZSCII values in a `->` (byte) array, with address `WordAddress(x)` and length `WordLength(x)`.

We picture the command as a stream of words to be read one at a time, with the global variable `wn` being the "current word" marker. `NextWord`, which takes no arguments, returns:

(a) 0 if the word at `wn` is unrecognised by the dictionary or `wn` is out of range,
(b) `comma_word` if the word was a comma,
(c) `THEN1__WD` if it was a full stop (because of the Infocom tradition that a full stop abbreviates for the word "then": e.g., TAKE BOX. EAST was read as two commands in succession),
(d) or the dictionary address if the word was recognised.

The current word marker `wn` is always advanced.

`NextWordStopped` does the same, but returns −1 when `wn` is out of range (e.g., by having advanced past the last word in the command).

`MoveWord(at1, b2, at2)` copies word `at2` from parse buffer `b2` – which doesn't need to be `buffer` – to word `at1` in `parse`.

```
#Ifdef TARGET_ZCODE;
[ WordCount; return parse->1; ];
[ WordAddress wordnum; return buffer + parse->(wordnum*4+1); ];
[ WordLength wordnum; return parse->(wordnum*4); ];
[ MoveWord at1 b2 at2 x y;
    x = at1*2-1; y = at2*2-1;
    parse-->x++ = b2-->y++;
    parse-->x = b2-->y;
];
#Ifnot;
[ WordCount; return parse-->0; ];
[ WordAddress wordnum; return buffer + parse-->(wordnum*3); ];
[ WordLength wordnum; return parse-->(wordnum*3-1); ];
[ MoveWord at1 b2 at2 x y;
    x = at1*3-2; y = at2*3-2;
    parse-->x++ = b2-->y++;
    parse-->x++ = b2-->y++;
    parse-->x = b2-->y;
];
#Endif;
```

```
[ NextWord i j wc;
    #Ifdef TARGET_ZCODE; wc = parse->1; i = wn*2-1;
    #Ifnot; wc = parse-->0; i = wn*3-2; #Endif;
    wn++;
    if ((wn < 2) || (wn > wc+1)) return 0;
    j = parse-->i;
    if (j == ',//') j = comma_word;
    if (j == './/') j = THEN1__WD;
    return j;
];
[ NextWordStopped wc;
    #Ifdef TARGET_ZCODE; wc = parse->1; #Ifnot; wc = parse-->0; #Endif;
    if ((wn < 1) || (wn > wc)) { wn++; return -1; }
    return NextWord();
];
```

§**5. Snippets.** Although the idea is arguably implicit in I6, the formal concept of "snippet" is new in I7. A snippet is a value which represents a word range in the command most recently typed by the player. These words number consecutively upwards from 1, as noted above. The correspondence between (w_1, w_2), the word range, and V, the number used to represent it as an I6 value, is:

$$V = 100w_1 + (w_2 - w_1 + 1)$$

so that the remainder mod 100 is the number of words in the range. We require that $1 \leq w_1 \leq w_2 \leq N$, where N is the number of words in the current player's command. The entire command is therefore represented by:

$$C = 100 + N$$

```
[ PrintSnippet snip from to i w1 w2;
    w1 = snip/100; w2 = w1 + (snip%100) - 1;
    if ((w2<w1) || (w1<1) || (w2>WordCount()))
        return RunTimeProblem(RTP_SAYINVALIDSNIPPET, w1, w2);
    from = WordAddress(w1); to = WordAddress(w2) + WordLength(w2) - 1;
    for (i=from: i<=to: i++) print (char) i->0;
];
[ SpliceSnippet snip t i w1 w2 nextw at endsnippet newlen;
    w1 = snip/100; w2 = w1 + (snip%100) - 1;
    if ((w2<w1) || (w1<1) || (w2>WordCount()))
        return RunTimeProblem(RTP_SPLICEINVALIDSNIPPET, w1, w2);
    @push say__p; @push say__pc;
    nextw = w2 + 1;
    at = WordAddress(w1) - buffer;
    if (nextw <= WordCount()) endsnippet = 100*nextw + (WordCount() - nextw + 1);
    buffer2-->0 = 120;
    newlen = VM_PrintToBuffer(buffer2, 120, SpliceSnippet__TextPrinter, t, endsnippet);
    for (i=0: (i<newlen) && (at+i<120): i++) buffer->(at+i) = buffer2->(WORDSIZE+i);
    #Ifdef TARGET_ZCODE; buffer->1 = at+i; #ifnot; buffer-->0 = at+i; #endif;
    for (:at+i<120:i++) buffer->(at+i) = ' ';
    VM_Tokenise(buffer, parse);
    players_command = 100 + WordCount();
    @pull say__pc; @pull say__p;
];
```

```
[ SpliceSnippet__TextPrinter t endsnippet;
    PrintText(t);
    if (endsnippet) { print " "; PrintSnippet(endsnippet); }
];
[ SnippetIncludes test snippet w1 w2 wlen i j;
    w1 = snippet/100; w2 = w1 + (snippet%100) - 1;
    if ((w2<w1) || (w1<1))
        return RunTimeProblem(RTP_INCLUDEINVALIDSNIPPET, w1, w2);
    if (metaclass(test) == Routine) {
        wlen = snippet%100;
        for (i=w1, j=wlen: j>0: i++, j--) {
            if (((test)(i, 0)) ~= GPR_FAIL) return i*100+wn-i;
        }
    }
    rfalse;
];
[ SnippetMatches snippet topic_gpr rv;
    wn=1;
    if (topic_gpr == 0) rfalse;
    if (metaclass(topic_gpr) == Routine) {
        rv = (topic_gpr)(snippet/100, snippet%100);
        if (rv ~= GPR_FAIL) rtrue;
        rfalse;
    }
    RunTimeProblem(RTP_BADTOPIC);
    rfalse;
];
```

§**6. Unpacking Grammar Lines.** Grammar lines are sequences of tokens in an array built into the story file, but in a format which differs depending on the virtual machine in use, so the following code unpacks the data into more convenient if larger arrays which are VM-independent.

```
[ UnpackGrammarLine line_address i size;
    for (i=0 : i<32 : i++) {
        line_token-->i = ENDIT_TOKEN;
        line_ttype-->i = ELEMENTARY_TT;
        line_tdata-->i = ENDIT_TOKEN;
    }
#Ifdef TARGET_ZCODE;
    action_to_be = 256*(line_address->0) + line_address->1;
    action_reversed = ((action_to_be & $400) ~= 0);
    action_to_be = action_to_be & $3ff;
    line_address--;
    size = 3;
#Ifnot; ! GLULX
    @aloads line_address 0 action_to_be;
    action_reversed = (((line_address->2) & 1) ~= 0);
    line_address = line_address - 2;
    size = 5;
#Endif;
    params_wanted = 0;
    for (i=0 : : i++) {
```

```
        line_address = line_address + size;
        if (line_address->0 == ENDIT_TOKEN) break;
        line_token-->i = line_address;
        AnalyseToken(line_address);
        if (found_ttype ~= PREPOSITION_TT) params_wanted++;
        line_ttype-->i = found_ttype;
        line_tdata-->i = found_tdata;
    }
    return line_address + 1;
];
[ AnalyseToken token;
    if (token == ENDIT_TOKEN) {
        found_ttype = ELEMENTARY_TT;
        found_tdata = ENDIT_TOKEN;
        return;
    }
    found_ttype = (token->0) & $$1111;
    found_tdata = (token+1)-->0;
];
```

§**7. Extracting Verb Numbers.** A long tale of woe lies behind the following. Infocom games stored verb numbers in a single byte in dictionary entries, but they did so counting downwards, so that verb number 0 was stored as 255, 1 as 254, and so on. Inform followed suit so that debugging of Inform 1 could be aided by using the then-available tools for dumping dictionaries from Infocom story files; by using the Infocom format for dictionary tables, Inform's life was easier.

But there was an implicit restriction there of 255 distinct verbs (not 256 since not all words were verbs). When Glulx raised almost all of the Z-machine limits, it made space for 65535 verbs instead of 255, but it appears that nobody remembered to implement this in I6-for-Glulx and the Glulx form of the I6 library. This was only put right in March 2009, and the following routine was added to concentrate lookups of this field in one place.

```
[ DictionaryWordToVerbNum dword verbnum;
#Ifdef TARGET_ZCODE;
    verbnum = $ff-(dword->#dict_par2);
#Ifnot; ! GLULX
    dword = dword + #dict_par2 - 1;
    @aloads dword 0 verbnum;
    verbnum = $ffff-verbnum;
#Endif;
    return verbnum;
];
```

§**8. Keyboard Primitive.** This is the primitive routine to read from the keyboard: it usually delegates this to a routine specific to the virtual machine being used, but sometimes uses a hacked version to allow TEST commands to work. (When a TEST is running, the text in the walk-through provided is fed into the buffer as if it had been typed at the keyboard.)

```
[ KeyboardPrimitive a_buffer a_table;
#Ifdef DEBUG; #Iftrue ({-value:NUMBER_CREATED(test_scenario)} > 0);
    return TestKeyboardPrimitive(a_buffer, a_table);
#Endif; #Endif;
    return VM_ReadKeyboard(a_buffer, a_table);
];
```

§**9. Reading the Command.** The `Keyboard` routine actually receives the player's words, putting the words in `a_buffer` and their dictionary addresses in `a_table`. It is assumed that the table is the same one on each (standard) call. Much of the code handles the OOPS and UNDO commands, which are not actions and do not pass through the rest of the parser. The undo state is saved – it is essentially an internal saved game, in the VM interpreter's memory rather than in an external file – and note that this is therefore also where execution picks up if an UNDO has been typed. Since UNDO recreates the former machine state perfectly, it might seem impossible to tell that an UNDO had occurred, but in fact the VM passes information back in the form of a return code from the relevant instruction, and this allows us to detect an undo. (We deal with it by printing the current location and asking another command.)

`Keyboard` can also be used by miscellaneous routines in the game to ask yes/no questions and the like, without invoking the rest of the parser.

The return value is the number of words typed.

```
[ Keyboard  a_buffer a_table  nw i w w2 x1 x2;
    sline1 = score; sline2 = turns;
    while (true) {
        ! Save the start of the buffer, in case "oops" needs to restore it
        for (i=0 : i<64 : i++) oops_workspace->i = a_buffer->i;
        ! In case of an array entry corruption that shouldn't happen, but would be
        ! disastrous if it did:
        #Ifdef TARGET_ZCODE;
        a_buffer->0 = INPUT_BUFFER_LEN;
        a_table->0 = 15;  ! Allow to split input into this many words
        #Endif; ! TARGET_
        ! Print the prompt, and read in the words and dictionary addresses
        PrintPrompt();
        DrawStatusLine();
        KeyboardPrimitive(a_buffer, a_table);
        ! Set nw to the number of words
        #Ifdef TARGET_ZCODE; nw = a_table->1; #Ifnot; nw = a_table-->0; #Endif;
        ! If the line was blank, get a fresh line
        if (nw == 0) {
            @push etype; etype = BLANKLINE_PE;
            players_command = 100;
            BeginActivity(PRINTING_A_PARSER_ERROR_ACT);
            if (ForActivity(PRINTING_A_PARSER_ERROR_ACT) == false) L__M(##Miscellany,10);
            EndActivity(PRINTING_A_PARSER_ERROR_ACT);
            @pull etype;
            continue;
```

```
}
! Unless the opening word was OOPS, return
! Conveniently, a_table-->1 is the first word on both the Z-machine and Glulx
w = a_table-->1;
if (w == OOPS1__WD or OOPS2__WD or OOPS3__WD) {
    if (oops_from == 0) { L__M(##Miscellany, 14); continue; }
    if (nw == 1) { L__M(##Miscellany, 15); continue; }
    if (nw > 2) { L__M(##Miscellany, 16); continue; }
    ! So now we know: there was a previous mistake, and the player has
    ! attempted to correct a single word of it.
    for (i=0 : i<INPUT_BUFFER_LEN : i++) buffer2->i = a_buffer->i;
    #Ifdef TARGET_ZCODE;
    x1 = a_table->9;  ! Start of word following "oops"
    x2 = a_table->8;  ! Length of word following "oops"
    #Ifnot; ! TARGET_GLULX
    x1 = a_table-->6; ! Start of word following "oops"
    x2 = a_table-->5; ! Length of word following "oops"
    #Endif; ! TARGET_
    ! Repair the buffer to the text that was in it before the "oops"
    ! was typed:
    for (i=0 : i<64 : i++) a_buffer->i = oops_workspace->i;
    VM_Tokenise(a_buffer,a_table);
    ! Work out the position in the buffer of the word to be corrected:
    #Ifdef TARGET_ZCODE;
    w = a_table->(4*oops_from + 1); ! Start of word to go
    w2 = a_table->(4*oops_from);    ! Length of word to go
    #Ifnot; ! TARGET_GLULX
    w = a_table-->(3*oops_from);      ! Start of word to go
    w2 = a_table-->(3*oops_from - 1); ! Length of word to go
    #Endif; ! TARGET_
    ! Write spaces over the word to be corrected:
    for (i=0 : i<w2 : i++) a_buffer->(i+w) = ' ';
    if (w2 < x2) {
        ! If the replacement is longer than the original, move up...
        for (i=INPUT_BUFFER_LEN-1 : i>=w+x2 : i--)
            a_buffer->i = a_buffer->(i-x2+w2);
        ! ...increasing buffer size accordingly.
        #Ifdef TARGET_ZCODE;
        a_buffer->1 = (a_buffer->1) + (x2-w2);
        #Ifnot; ! TARGET_GLULX
        a_buffer-->0 = (a_buffer-->0) + (x2-w2);
        #Endif; ! TARGET_
    }
    ! Write the correction in:
    for (i=0 : i<x2 : i++) a_buffer->(i+w) = buffer2->(i+x1);
    VM_Tokenise(a_buffer, a_table);
    #Ifdef TARGET_ZCODE; nw = a_table->1; #Ifnot; nw = a_table-->0; #Endif;
    return nw;
}
! Undo handling
```

```
        if ((w == UNDO1__WD or UNDO2__WD or UNDO3__WD) && (nw==1)) {
            Perform_Undo();
            continue;
        }
        i = VM_Save_Undo();
        #ifdef PREVENT_UNDO; undo_flag = 0; #endif;
        #ifndef PREVENT_UNDO; undo_flag = 2; #endif;
        if (i == -1) undo_flag = 0;
        if (i == 0) undo_flag = 1;
        if (i == 2) {
            VM_RestoreWindowColours();
            VM_Style(SUBHEADER_VMSTY);
            SL_Location(); print "^";
            ! print (name) location, "^";
            VM_Style(NORMAL_VMSTY);
            L__M(##Miscellany, 13);
            continue;
        }
        return nw;
    }
];
```

§**10. Parser Proper.** The main parser routine is something of a leviathan, and it has traditionally been divided into 11 lettered parts:

(A) Get the input, do OOPS and AGAIN
(B) Is it a direction, and so an implicit GO? If so go to (K)
(C) Is anyone being addressed?
(D) Get the command verb: try all the syntax lines for that verb
(E) Break down a syntax line into analysed tokens
(F) Look ahead for advance warning for `multiexcept`/`multiinside`
(G) Parse each token in turn (calling `ParseToken` to do most of the work)
(H) Cheaply parse otherwise unrecognised conversation and return
(I) Print best possible error message
(J) Retry the whole lot
(K) Last thing: check for THEN and further instructions(s), return.

This lettering has been preserved here, with the code under each letter now being the body of "Parser Letter A", "Parser Letter B" and so on.

Note that there are three different places where a return can happen. The routine returns only when a sensible request has been made; for a fairly thorough description of its output, which is written into the `results` array but also into several globals, see "OrderOfPlay.i6t".

```
[ Parser__parse results
    syntax line num_lines line_address i j k token l m;

    cobj_flag = 0;
    results-->0 = 0; results-->1 = 0; results-->2 = 0; results-->3 = 0;
    meta = false;
```

§**11. Parser Letter A.** Get the input, do OOPS and AGAIN.

```
    if (held_back_mode == 1) {
        held_back_mode = 0;
        VM_Tokenise(buffer, parse);
        jump ReParse;
    }
.ReType;
    cobj_flag = 0;
    BeginActivity(READING_A_COMMAND_ACT); if (ForActivity(READING_A_COMMAND_ACT)==false) {
        Keyboard(buffer,parse);
        players_command = 100 + WordCount();
        num_words = WordCount();
    } if (EndActivity(READING_A_COMMAND_ACT)) jump ReType;
.ReParse;
    parser_inflection = name;
    ! Initially assume the command is aimed at the player, and the verb
    ! is the first word
    num_words = WordCount();
    wn = 1;
    #Ifdef LanguageToInformese;
    LanguageToInformese();
    ! Re-tokenise:
    VM_Tokenise(buffer,parse);
    #Endif; ! LanguageToInformese
    num_words = WordCount();
    k=0;
    #Ifdef DEBUG;
    if (parser_trace >= 2) {
        print "[ ";
        for (i=0 : i<num_words : i++) {
            #Ifdef TARGET_ZCODE;
            j = parse-->(i*2 + 1);
            #Ifnot; ! TARGET_GLULX
            j = parse-->(i*3 + 1);
            #Endif; ! TARGET_
            k = WordAddress(i+1);
            l = WordLength(i+1);
            print "~"; for (m=0 : m<l : m++) print (char) k->m; print "~ ";
            if (j == 0) print "?";
            else {
                #Ifdef TARGET_ZCODE;
                if (UnsignedCompare(j, HDR_DICTIONARY-->0) >= 0 &&
                    UnsignedCompare(j, HDR_HIGHMEMORY-->0) < 0)
                    print (address) j;
                else print j;
                #Ifnot; ! TARGET_GLULX
                if (j->0 == $60) print (address) j;
                else print j;
                #Endif; ! TARGET_
            }
```

```
            if (i ~= num_words-1) print " / ";
        }
        print " ]^";
    }
    #Endif; ! DEBUG
    verb_wordnum = 1;
    actor = player;
    actors_location = ScopeCeiling(player);
    usual_grammar_after = 0;
.AlmostReParse;
    scope_token = 0;
    action_to_be = NULL;
    ! Begin from what we currently think is the verb word
.BeginCommand;
    wn = verb_wordnum;
    verb_word = NextWordStopped();
    ! If there's no input here, we must have something like "person,".
    if (verb_word == -1) {
        best_etype = STUCK_PE;
        jump GiveError;
    }
    ! Now try for "again" or "g", which are special cases: don't allow "again" if nothing
    ! has previously been typed; simply copy the previous text across
    if (verb_word == AGAIN2__WD or AGAIN3__WD) verb_word = AGAIN1__WD;
    if (verb_word == AGAIN1__WD) {
        if (actor ~= player) {
            L__M(##Miscellany, 20);
            jump ReType;
        }
        #Ifdef TARGET_ZCODE;
        if (buffer3->1 == 0) {
            L__M(##Miscellany, 21);
            jump ReType;
        }
        #Ifnot; ! TARGET_GLULX
        if (buffer3-->0 == 0) {
            L__M(##Miscellany, 21);
            jump ReType;
        }
        #Endif; ! TARGET_
        for (i=0 : i<INPUT_BUFFER_LEN : i++) buffer->i = buffer3->i;
        VM_Tokenise(buffer,parse);
        num_words = WordCount();
        players_command = 100 + WordCount();
        jump ReParse;
    }
    ! Save the present input in case of an "again" next time
    if (verb_word ~= AGAIN1__WD)
        for (i=0 : i<INPUT_BUFFER_LEN : i++) buffer3->i = buffer->i;
    if (usual_grammar_after == 0) {
```

```
    j = verb_wordnum;
    i = RunRoutines(actor, grammar);
    #Ifdef DEBUG;
    if (parser_trace >= 2 && actor.grammar ~= 0 or NULL)
        print " [Grammar property returned ", i, "]^";
    #Endif; ! DEBUG
    if ((i ~= 0 or 1) && (VM_InvalidDictionaryAddress(i))) {
        usual_grammar_after = verb_wordnum; i=-i;
    }
    if (i == 1) {
        results-->0 = action;
        results-->1 = noun;
        results-->2 = second;
        rtrue;
    }
    if (i ~= 0) { verb_word = i; wn--; verb_wordnum--; }
    else { wn = verb_wordnum; verb_word = NextWord(); }
}
else usual_grammar_after = 0;
```

§**12. Parser Letter B.** Is the command a direction name, and so an implicit GO? If so, go to (K).

```
#Ifdef LanguageIsVerb;
if (verb_word == 0) {
    i = wn; verb_word = LanguageIsVerb(buffer, parse, verb_wordnum);
    wn = i;
}
#Endif; ! LanguageIsVerb
! If the first word is not listed as a verb, it must be a direction
! or the name of someone to talk to
if (verb_word == 0 || ((verb_word->#dict_par1) & 1) == 0) {
    ! So is the first word an object contained in the special object "compass"
    ! (i.e., a direction)?  This needs use of NounDomain, a routine which
    ! does the object matching, returning the object number, or 0 if none found,
    ! or REPARSE_CODE if it has restructured the parse table so the whole parse
    ! must be begun again...
    wn = verb_wordnum; indef_mode = false; token_filter = 0; parameters = 0;
    @push actor; @push action; @push action_to_be;
    actor = player; meta = false; action = ##Go; action_to_be = ##Go;
    l = NounDomain(compass, 0, 0);
    @pull action_to_be; @pull action; @pull actor;
    if (l == REPARSE_CODE) jump ReParse;
    ! If it is a direction, send back the results:
    ! action=GoSub, no of arguments=1, argument 1=the direction.
    if ((l~=0) && (l ofclass K3_direction)) {
        results-->0 = ##Go;
        results-->1 = 1;
        results-->2 = l;
        jump LookForMore;
    }
```

§**13. Parser Letter C.** Is anyone being addressed?

```
    ! Only check for a comma (a "someone, do something" command) if we are
    ! not already in the middle of one.  (This simplification stops us from
    ! worrying about "robot, wizard, you are an idiot", telling the robot to
    ! tell the wizard that she is an idiot.)
    if (actor == player) {
        for (j=2 : j<=num_words : j++) {
            i=NextWord();
            if (i == comma_word) jump Conversation;
        }
        verb_word = UnknownVerb(verb_word);
        if (verb_word ~= 0) jump VerbAccepted;
    }
    best_etype = VERB_PE;
    jump GiveError;
    ! NextWord nudges the word number wn on by one each time, so we've now
    ! advanced past a comma.  (A comma is a word all on its own in the table.)
.Conversation;
    j = wn - 1;
    if (j == 1) {
        L__M(##Miscellany, 22);
        jump ReType;
    }
    ! Use NounDomain (in the context of "animate creature") to see if the
    ! words make sense as the name of someone held or nearby
    wn = 1; lookahead = HELD_TOKEN;
    scope_reason = TALKING_REASON;
    l = NounDomain(player,actors_location,6);
    scope_reason = PARSING_REASON;
    if (l == REPARSE_CODE) jump ReParse;
    if (l == 0) {
        L__M(##Miscellany, 23);
        jump ReType;
    }
.Conversation2;
    ! The object addressed must at least be "talkable" if not actually "animate"
    ! (the distinction allows, for instance, a microphone to be spoken to,
    ! without the parser thinking that the microphone is human).
    if (l hasnt animate && l hasnt talkable) {
        L__M(##Miscellany, 24, l);
        jump ReType;
    }
    ! Check that there aren't any mystery words between the end of the person's
    ! name and the comma (eg, throw out "dwarf sdfgsdgs, go north").
    if (wn ~= j) {
        L__M(##Miscellany, 25);
        jump ReType;
    }
    ! The player has now successfully named someone.  Adjust "him", "her", "it":
```

```
        PronounNotice(l);
        ! Set the global variable "actor", adjust the number of the first word,
        ! and begin parsing again from there.
        verb_wordnum = j + 1;
        ! Stop things like "me, again":
        if (l == player) {
            wn = verb_wordnum;
            if (NextWordStopped() == AGAIN1__WD or AGAIN2__WD or AGAIN3__WD) {
                L__M(##Miscellany, 20);
                jump ReType;
            }
        }
        actor = l;
        actors_location = ScopeCeiling(l);
        #Ifdef DEBUG;
        if (parser_trace >= 1)
            print "[Actor is ", (the) actor, " in ", (name) actors_location, "]^";
        #Endif; ! DEBUG
        jump BeginCommand;
    } ! end of first-word-not-a-verb
```

§**14. Parser Letter D.** Get the verb: try all the syntax lines for that verb.

```
.VerbAccepted;
    ! We now definitely have a verb, not a direction, whether we got here by the
    ! "take ..." or "person, take ..." method.  Get the meta flag for this verb:
    meta = ((verb_word->#dict_par1) & 2)/2;
    ! You can't order other people to "full score" for you, and so on...
    if (meta == 1 && actor ~= player) {
        best_etype = VERB_PE;
        meta = 0;
        jump GiveError;
    }
    ! Now let i be the corresponding verb number...
    i = DictionaryWordToVerbNum(verb_word);
    ! ...then look up the i-th entry in the verb table, whose address is at word
    ! 7 in the Z-machine (in the header), so as to get the address of the syntax
    ! table for the given verb...
    #Ifdef TARGET_ZCODE;
    syntax = (HDR_STATICMEMORY-->0)-->i;
    #Ifnot; ! TARGET_GLULX
    syntax = (#grammar_table)-->(i+1);
    #Endif; ! TARGET_
    ! ...and then see how many lines (ie, different patterns corresponding to the
    ! same verb) are stored in the parse table...
    num_lines = (syntax->0) - 1;
    ! ...and now go through them all, one by one.
    ! To prevent pronoun_word 0 being misunderstood,
```

```
pronoun_word = NULL; pronoun_obj = NULL;
#Ifdef DEBUG;
if (parser_trace >= 1)
    print "[Parsing for the verb '", (address) verb_word, "' (", num_lines+1, " lines)]^";
#Endif; ! DEBUG
best_etype = STUCK_PE; nextbest_etype = STUCK_PE;
multiflag = false;
! "best_etype" is the current failure-to-match error - it is by default
! the least informative one, "don't understand that sentence".
! "nextbest_etype" remembers the best alternative to having to ask a
! scope token for an error message (i.e., the best not counting ASKSCOPE_PE).
! multiflag is used here to prevent inappropriate MULTI_PE errors
! in addition to its unrelated duties passing information to action routines
```

§**15. Parser Letter E.** Break down a syntax line into analysed tokens.

```
line_address = syntax + 1;
for (line=0 : line<=num_lines : line++) {
    for (i=0 : i<32 : i++) {
        line_token-->i = ENDIT_TOKEN;
        line_ttype-->i = ELEMENTARY_TT;
        line_tdata-->i = ENDIT_TOKEN;
    }
    ! Unpack the syntax line from Inform format into three arrays; ensure that
    ! the sequence of tokens ends in an ENDIT_TOKEN.
    line_address = UnpackGrammarLine(line_address);
    #Ifdef DEBUG;
    if (parser_trace >= 1) {
        if (parser_trace >= 2) new_line;
        print "[line ", line; DebugGrammarLine();
        print "]^";
    }
    #Endif; ! DEBUG
    ! We aren't in "not holding" or inferring modes, and haven't entered
    ! any parameters on the line yet, or any special numbers; the multiple
    ! object is still empty.
    inferfrom = 0;
    parameters = 0;
    nsns = 0; special_word = 0;
    multiple_object-->0 = 0;
    multi_context = 0;
    etype = STUCK_PE;
    ! Put the word marker back to just after the verb
    wn = verb_wordnum+1;
```

§16. Parser Letter F. Look ahead for advance warning for `multiexcept`/`multiinside`.

There are two special cases where parsing a token now has to be affected by the result of parsing another token later, and these two cases (multiexcept and multiinside tokens) are helped by a quick look ahead, to work out the future token now. We can only carry this out in the simple (but by far the most common) case:

```
multiexcept <one or more prepositions> noun
```

and similarly for `multiinside`.

```
advance_warning = -1; indef_mode = false;
for (i=0,m=false,pcount=0 : line_token-->pcount ~= ENDIT_TOKEN : pcount++) {
    scope_token = 0;
    if (line_ttype-->pcount ~= PREPOSITION_TT) i++;
    if (line_ttype-->pcount == ELEMENTARY_TT) {
        if (line_tdata-->pcount == MULTI_TOKEN) m = true;
        if (line_tdata-->pcount == MULTIEXCEPT_TOKEN or MULTIINSIDE_TOKEN  && i == 1) {
            ! First non-preposition is "multiexcept" or
            ! "multiinside", so look ahead.
            #Ifdef DEBUG;
            if (parser_trace >= 2) print " [Trying look-ahead]^";
            #Endif; ! DEBUG
            ! We need this to be followed by 1 or more prepositions.
            pcount++;
            if (line_ttype-->pcount == PREPOSITION_TT) {
                ! skip ahead to a preposition word in the input
                do {
                    l = NextWord();
                } until ((wn > num_words) ||
                        (l && (l->#dict_par1) & 8 ~= 0));
                if (wn > num_words) {
                    #Ifdef DEBUG;
                    if (parser_trace >= 2)
                        print " [Look-ahead aborted: prepositions missing]^";
                    #Endif;
                    jump LineFailed;
                }
                do {
                    if (PrepositionChain(l, pcount) ~= -1) {
                        ! advance past the chain
                        if ((line_token-->pcount)->0 & $20 ~= 0) {
                            pcount++;
                            while ((line_token-->pcount ~= ENDIT_TOKEN) &&
                                ((line_token-->pcount)->0 & $10 ~= 0))
                                pcount++;
                        } else {
                            pcount++;
                        }
                    } else {
                        ! try to find another preposition word
                        do {
                            l = NextWord();
                        } until ((wn >= num_words) ||
                                (l && (l->#dict_par1) & 8 ~= 0));
```

```
                        if (l && (l->#dict_par1) & 8) continue;
                        ! lookahead failed
                        #Ifdef DEBUG;
                        if (parser_trace >= 2)
                            print " [Look-ahead aborted: prepositions don't match]^";
                        #endif;
                        jump LineFailed;
                    }
                    l = NextWord();
                } until (line_ttype-->pcount ~= PREPOSITION_TT);
                ! put back the non-preposition we just read
                wn--;
                if ((line_ttype-->pcount == ELEMENTARY_TT) &&
                    (line_tdata-->pcount == NOUN_TOKEN)) {
                    l = Descriptors();  ! skip past THE etc
                    if (l~=0) etype=l;  ! don't allow multiple objects
                    k = parser_results-->2; @push k; @push parameters;
                    parameters = 1; parser_results-->2 = 0;
                    l = NounDomain(actors_location, actor, NOUN_TOKEN);
                    @pull parameters; @pull k; parser_results-->2 = k;
                    #Ifdef DEBUG;
                    if (parser_trace >= 2) {
                        print " [Advanced to ~noun~ token: ";
                        if (l == REPARSE_CODE) print "re-parse request]^";
                        else {
                            if (l == 1) print "but multiple found]^";
                            if (l == 0) print "error ", etype, "]^";
                            if (l >= 2) print (the) l, "]^";
                        }
                    }
                    #Endif; ! DEBUG
                    if (l == REPARSE_CODE) jump ReParse;
                    if (l >= 2) advance_warning = l;
                }
            }
            break;
        }
    }
}
! Slightly different line-parsing rules will apply to "take multi", to
! prevent "take all" behaving correctly but misleadingly when there's
! nothing to take.
take_all_rule = 0;
if (m && params_wanted == 1 && action_to_be == ##Take)
    take_all_rule = 1;
! And now start again, properly, forearmed or not as the case may be.
! As a precaution, we clear all the variables again (they may have been
! disturbed by the call to NounDomain, which may have called outside
! code, which may have done anything!).
inferfrom = 0;
parameters = 0;
nsns = 0; special_word = 0;
```

```
multiple_object-->0 = 0;
etype = STUCK_PE;
wn = verb_wordnum+1;
```

§**17. Parser Letter G.** Parse each token in turn (calling `ParseToken` to do most of the work).

The `pattern` gradually accumulates what has been recognised so far, so that it may be reprinted by the parser later on.

```
for (pcount=1 : : pcount++) {
    pattern-->pcount = PATTERN_NULL; scope_token = 0;
    token = line_token-->(pcount-1);
    lookahead = line_token-->pcount;
    #Ifdef DEBUG;
    if (parser_trace >= 2)
        print " [line ", line, " token ", pcount, " word ", wn, " : ", (DebugToken) token,
        "]^";
    #Endif; ! DEBUG
    if (token ~= ENDIT_TOKEN) {
        scope_reason = PARSING_REASON;
        AnalyseToken(token);
        l = ParseToken(found_ttype, found_tdata, pcount-1, token);
        while ((l >= GPR_NOUN) && (l < -1)) l = ParseToken(ELEMENTARY_TT, l + 256);
        scope_reason = PARSING_REASON;
        if (l == GPR_PREPOSITION) {
            if (found_ttype~=PREPOSITION_TT && (found_ttype~=ELEMENTARY_TT ||
                found_tdata~=TOPIC_TOKEN)) params_wanted--;
            l = true;
        }
        else
            if (l < 0) l = false;
            else
                if (l ~= GPR_REPARSE) {
                    if (l == GPR_NUMBER) {
                        if (nsns == 0) special_number1 = parsed_number;
                        else special_number2 = parsed_number;
                        nsns++; l = 1;
                    }
                    if (l == GPR_MULTIPLE) l = 0;
                    results-->(parameters+2) = l;
                    parameters++;
                    pattern-->pcount = l;
                    l = true;
                }
        #Ifdef DEBUG;
        if (parser_trace >= 3) {
            print "  [token resulted in ";
            if (l == REPARSE_CODE) print "re-parse request]^";
            if (l == 0) print "failure with error type ", etype, "]^";
            if (l == 1) print "success]^";
        }
        #Endif; ! DEBUG
```

```
    if (l == REPARSE_CODE) jump ReParse;
    if (l == false) break;
}
else {
    ! If the player has entered enough already but there's still
    ! text to wade through: store the pattern away so as to be able to produce
    ! a decent error message if this turns out to be the best we ever manage,
    ! and in the mean time give up on this line
    ! However, if the superfluous text begins with a comma or "then" then
    ! take that to be the start of another instruction
    if (wn <= num_words) {
        l = NextWord();
        if (l == THEN1__WD or THEN2__WD or THEN3__WD or comma_word) {
            held_back_mode = 1; hb_wn = wn-1;
        }
        else {
            for (m=0 : m<32 : m++) pattern2-->m = pattern-->m;
            pcount2 = pcount;
            etype = UPTO_PE;
            break;
        }
    }
    ! Now, we may need to revise the multiple object because of the single one
    ! we now know (but didn't when the list was drawn up).
    if (parameters >= 1 && results-->2 == 0) {
        l = ReviseMulti(results-->3);
        if (l ~= 0) { etype = l; results-->0 = action_to_be; break; }
    }
    if (parameters >= 2 && results-->3 == 0) {
        l = ReviseMulti(results-->2);
        if (l ~= 0) { etype = l; break; }
    }
    ! To trap the case of "take all" inferring only "yourself" when absolutely
    ! nothing else is in the vicinity...
    if (take_all_rule == 2 && results-->2 == actor) {
        best_etype = NOTHING_PE;
        jump GiveError;
    }
    #Ifdef DEBUG;
    if (parser_trace >= 1) print "[Line successfully parsed]^";
    #Endif; ! DEBUG
    ! The line has successfully matched the text.  Declare the input error-free...
    oops_from = 0;
    ! ...explain any inferences made (using the pattern)...
    if (inferfrom ~= 0) {
        PrintInferredCommand(inferfrom);
        ClearParagraphing();
    }
    ! ...copy the action number, and the number of parameters...
    results-->0 = action_to_be;
```

```
            results-->1 = parameters;
            ! ...reverse first and second parameters if need be...
            if (action_reversed && parameters == 2) {
                i = results-->2; results-->2 = results-->3;
                results-->3 = i;
                if (nsns == 2) {
                    i = special_number1; special_number1 = special_number2;
                    special_number2 = i;
                }
            }
            ! ...and to reset "it"-style objects to the first of these parameters, if
            ! there is one (and it really is an object)...
            if (parameters > 0 && results-->2 >= 2)
                PronounNotice(results-->2);
            ! ...and return from the parser altogether, having successfully matched
            ! a line.
            if (held_back_mode == 1) {
                wn=hb_wn;
                jump LookForMore;
            }
            rtrue;
        } ! end of if(token ~= ENDIT_TOKEN) else
    } ! end of for(pcount++)
    .LineFailed;
    ! The line has failed to match.
    ! We continue the outer "for" loop, trying the next line in the grammar.
    if (etype > best_etype) best_etype = etype;
    if (etype ~= ASKSCOPE_PE && etype > nextbest_etype) nextbest_etype = etype;
    ! ...unless the line was something like "take all" which failed because
    ! nothing matched the "all", in which case we stop and give an error now.
    if (take_all_rule == 2 && etype==NOTHING_PE) break;
} ! end of for(line++)
! The grammar is exhausted: every line has failed to match.
```

§**18. Parser Letter H.** Cheaply parse otherwise unrecognised conversation and return.

(Errors are handled differently depending on who was talking. If the command was addressed to somebody else (eg, DWARF, SFGH) then it is taken as conversation which the parser has no business in disallowing.)

The parser used to return the fake action `##NotUnderstood` when a command in the form PERSON, ARFLE BARFLE GLOOP is parsed, where a character is addressed but with an instruction which the parser can't understand. (If a command such as ARFLE BARFLE GLOOP is not an instruction to someone else, the parser prints an error and requires the player to type another command: thus `##NotUnderstood` was only returned when `actor` is not the player.) And I6 had elaborate object-oriented ways to deal with this, but we won't use any of that: we simply convert to a `##Answer` action, which communicates a snippet of words to another character, just as if the player had typed ANSWER ARFLE BARFLE GLOOP TO PERSON. For I7 purposes, the fake action `##NotUnderstood` does not exist.

```
.GiveError;
    etype = best_etype;
    if (actor ~= player) {
        if (usual_grammar_after ~= 0) {
            verb_wordnum = usual_grammar_after;
            jump AlmostReParse;
        }
        wn = verb_wordnum;
        special_word = NextWord();
        if (special_word == comma_word) {
            special_word = NextWord();
            verb_wordnum++;
        }
        results-->0 = ##Answer;
        results-->1 = 2;
        results-->2 = actor;
        results-->3 = 1; special_number1 = special_word;
        actor = player;
        consult_from = verb_wordnum; consult_words = num_words-consult_from+1;
        rtrue;
    }
```

§**19. Parser Letter I.** Print best possible error message.

```
    ! If the player was the actor (eg, in "take dfghh") the error must be printed,
    ! and fresh input called for.  In three cases the oops word must be jiggled.
    if ((etype ofclass Routine) || (etype ofclass String)) {
        if (ParserError(etype) ~= 0) jump ReType;
    } else {
        if (verb_wordnum == 0 && etype == CANTSEE_PE) etype = VERB_PE;
        players_command = 100 + WordCount(); ! The snippet variable ‘‘player’s command’’
        BeginActivity(PRINTING_A_PARSER_ERROR_ACT);
        if (ForActivity(PRINTING_A_PARSER_ERROR_ACT)) jump SkipParserError;
    }
    pronoun_word = pronoun__word; pronoun_obj = pronoun__obj;
    if (etype == STUCK_PE) {    L__M(##Miscellany, 27); oops_from = 1; }
    if (etype == UPTO_PE) {     L__M(##Miscellany, 28);
        for (m=0 : m<32 : m++) pattern-->m = pattern2-->m;
        pcount = pcount2; PrintCommand(0); L__M(##Miscellany, 56);
    }
```

```
if (etype == NUMBER_PE)     L__M(##Miscellany, 29);
if (etype == CANTSEE_PE) {  L__M(##Miscellany, 30); oops_from=saved_oops; }
if (etype == TOOLIT_PE)     L__M(##Miscellany, 31);
if (etype == NOTHELD_PE) {  L__M(##Miscellany, 32); oops_from=saved_oops; }
if (etype == MULTI_PE)      L__M(##Miscellany, 33);
if (etype == MMULTI_PE)     L__M(##Miscellany, 34);
if (etype == VAGUE_PE)      L__M(##Miscellany, 35);
if (etype == EXCEPT_PE)     L__M(##Miscellany, 36);
if (etype == ANIMA_PE)      L__M(##Miscellany, 37);
if (etype == VERB_PE)       L__M(##Miscellany, 38);
if (etype == SCENERY_PE)    L__M(##Miscellany, 39);
if (etype == ITGONE_PE) {
    if (pronoun_obj == NULL)
                             L__M(##Miscellany, 35);
    else                    L__M(##Miscellany, 40);
}
if (etype == JUNKAFTER_PE)  L__M(##Miscellany, 41);
if (etype == TOOFEW_PE)     L__M(##Miscellany, 42, multi_had);
if (etype == NOTHING_PE) {
    if (results-->0 == ##Remove && results-->3 ofclass Object) {
        noun = results-->3; ! ensure valid for messages
        if (noun has animate) L__M(##Take, 6, noun);
        else if (noun hasnt container or supporter) L__M(##Insert, 2, noun);
        else if (noun has container && noun hasnt open) L__M(##Take, 9, noun);
        else if (children(noun)==0) L__M(##Search, 6, noun);
        else results-->0 = 0;
        }
    if (results-->0 ~= ##Remove) {
        if (multi_wanted==100)  L__M(##Miscellany, 43);
        else                    L__M(##Miscellany, 44);
    }
}
if (etype == ASKSCOPE_PE) {
    scope_stage = 3;
    if (indirect(scope_error) == -1) {
        best_etype = nextbest_etype;
        if (~~((etype ofclass Routine) || (etype ofclass String)))
            EndActivity(PRINTING_A_PARSER_ERROR_ACT);
        jump GiveError;
    }
}
if (etype == NOTINCONTEXT_PE) L__M(##Miscellany, 73);
.SkipParserError;
if ((etype ofclass Routine) || (etype ofclass String)) jump ReType;
say__p = 1;
EndActivity(PRINTING_A_PARSER_ERROR_ACT);
```

§**20. Parser Letter J.** Retry the whole lot.

```
    ! And go (almost) right back to square one...
    jump ReType;
    ! ...being careful not to go all the way back, to avoid infinite repetition
    ! of a deferred command causing an error.
```

§**21. Parser Letter K.** Last thing: check for THEN and further instructions(s), return.

```
    ! At this point, the return value is all prepared, and we are only looking
    ! to see if there is a "then" followed by subsequent instruction(s).
.LookForMore;
    if (wn > num_words) rtrue;
    i = NextWord();
    if (i == THEN1__WD or THEN2__WD or THEN3__WD or comma_word) {
        if (wn > num_words) {
        held_back_mode = false;
        return;
        }
        i = WordAddress(verb_wordnum);
        j = WordAddress(wn);
        for (: i<j : i++) i->0 = ' ';
        i = NextWord();
        if (i == AGAIN1__WD or AGAIN2__WD or AGAIN3__WD) {
            ! Delete the words "then again" from the again buffer,
            ! in which we have just realised that it must occur:
            ! prevents an infinite loop on "i. again"
            i = WordAddress(wn-2)-buffer;
            if (wn > num_words) j = INPUT_BUFFER_LEN-1;
            else j = WordAddress(wn)-buffer;
            for (: i<j : i++) buffer3->i = ' ';
        }
        VM_Tokenise(buffer,parse);
        held_back_mode = true;
        return;
    }
    best_etype = UPTO_PE;
    jump GiveError;
```

§**22. End of Parser Proper.**

```
]; ! end of Parser__parse
```

§**23. Parse Token.** The main parsing routine above tried a sequence of "grammar lines" in turn, matching each against the text typed until one fitted. A grammar line is itself a sequence of "grammar tokens". Here we have to parse the tokens.

`ParseToken(type, data)` tries the match text beginning at the current word marker `wn` against a token of the given `type`, with the given `data`. The optional further arguments `token_n` and `token` supply the token number in the current grammar line (because some tokens do depend on what has happened before or is needed later) and the address of the dictionary word which makes up the `token`, in the case where it's a "preposition".

The return values are:
(a) `GPR_REPARSE` for "I have rewritten the command, please re-parse from scratch";
(b) `GPR_PREPOSITION` for "token accepted with no result";
(c) $-256 + x$ for "please parse `ParseToken(ELEMENTARY_TT, x)` instead";
(d) 0 for "token accepted, result is the multiple object list";
(e) 1 for "token accepted, result is the number in parsed$_n umber''$";
(f) an object number for "token accepted with this object as result";
(g) -1 for "token rejected".

Strictly speaking `ParseToken` is a shell routine which saves the current state on the stack, and calling `ParseToken__` to do the actual work.

Once again the routine is traditionally divided into six letters, here named under paragraphs "Parse Token Letter A", and so on.

(A) Analyse the token; handle all tokens not involving object lists and break down others into elementary tokens
(B) Begin parsing an object list
(C) Parse descriptors (articles, pronouns, etc.) in the list
(D) Parse an object name
(E) Parse connectives (AND, BUT, etc.) and go back to (C)
(F) Return the conclusion of parsing an object list

```
[ ParseTokenStopped x y;
    if (wn>WordCount()) return GPR_FAIL;
    return ParseToken(x,y);
];

Global parsetoken_nesting = 0;
[ ParseToken given_ttype given_tdata token_n token  i t rv;
    if (parsetoken_nesting > 0) {
        ! save match globals
        @push match_from; @push token_filter; @push match_length;
        @push number_of_classes; @push oops_from;
        for (i=0: i<number_matched: i++) {
            t = match_list-->i; @push t;
            t = match_classes-->i; @push t;
            t = match_scores-->i; @push t;
        }
        @push number_matched;
    }

    parsetoken_nesting++;
    rv = ParseToken__(given_ttype, given_tdata, token_n, token);
    parsetoken_nesting--;

    if (parsetoken_nesting > 0) {
        ! restore match globals
        @pull number_matched;
```

```
        for (i=0: i<number_matched: i++) {
            @pull t; match_scores-->i = t;
            @pull t; match_classes-->i = t;
            @pull t; match_list-->i = t;
        }
        @pull oops_from; @pull number_of_classes;
        @pull match_length; @pull token_filter; @pull match_from;
    }
    return rv;
];
[ ParseToken__ given_ttype given_tdata token_n token
    l o i j k and_parity single_object desc_wn many_flag
    token_allows_multiple prev_indef_wanted;
```

§**24. Parse Token Letter A.** Analyse token; handle all not involving object lists, break down others.

```
    token_filter = 0;
    parser_inflection = name;
    switch (given_ttype) {
    ELEMENTARY_TT:
        switch (given_tdata) {
        SPECIAL_TOKEN:
            l = TryNumber(wn);
            special_word = NextWord();
            #Ifdef DEBUG;
            if (l ~= -1000)
                if (parser_trace >= 3) print "  [Read special as the number ", l, "]^";
            #Endif; ! DEBUG
            if (l == -1000) {
                #Ifdef DEBUG;
                if (parser_trace >= 3) print "  [Read special word at word number ", wn, "]^";
                #Endif; ! DEBUG
                l = special_word;
            }
            parsed_number = l;
            return GPR_NUMBER;
        NUMBER_TOKEN:
            l=TryNumber(wn++);
            if (l == -1000) {
                etype = NUMBER_PE;
                return GPR_FAIL;
            }
            #Ifdef DEBUG;
            if (parser_trace>=3) print "  [Read number as ", l, "]^";
            #Endif; ! DEBUG
            parsed_number = l;
            return GPR_NUMBER;
        CREATURE_TOKEN:
            if (action_to_be == ##Answer or ##Ask or ##AskFor or ##Tell)
                scope_reason = TALKING_REASON;
        TOPIC_TOKEN:
```

```
        consult_from = wn;
        if ((line_ttype-->(token_n+1) ~= PREPOSITION_TT) &&
        (line_token-->(token_n+1) ~= ENDIT_TOKEN))
            RunTimeError(13);
        do o = NextWordStopped();
        until (o == -1 || PrepositionChain(o, token_n+1) ~= -1);
        wn--;
        consult_words = wn-consult_from;
        if (consult_words == 0) return GPR_FAIL;
        if (action_to_be == ##Ask or ##Answer or ##Tell) {
            o = wn; wn = consult_from; parsed_number = NextWord();
            wn = o; return 1;
        }
        if (o==-1 && (line_ttype-->(token_n+1) == PREPOSITION_TT))
            return GPR_FAIL;    ! don't infer if required preposition is absent
        return GPR_PREPOSITION;
    }
PREPOSITION_TT:
    ! Is it an unnecessary alternative preposition, when a previous choice
    ! has already been matched?
    if ((token->0) & $10) return GPR_PREPOSITION;
    ! If we've run out of the player's input, but still have parameters to
    ! specify, we go into "infer" mode, remembering where we are and the
    ! preposition we are inferring...
    if (wn > num_words) {
        if (inferfrom==0 && parameters<params_wanted) {
            inferfrom = pcount; inferword = token;
            pattern-->pcount = REPARSE_CODE + VM_DictionaryAddressToNumber(given_tdata);
        }
        ! If we are not inferring, then the line is wrong...
        if (inferfrom == 0) return -1;
        ! If not, then the line is right but we mark in the preposition...
        pattern-->pcount = REPARSE_CODE + VM_DictionaryAddressToNumber(given_tdata);
        return GPR_PREPOSITION;
    }
    o = NextWord();
    pattern-->pcount = REPARSE_CODE + VM_DictionaryAddressToNumber(o);
    ! Whereas, if the player has typed something here, see if it is the
    ! required preposition... if it's wrong, the line must be wrong,
    ! but if it's right, the token is passed (jump to finish this token).
    if (o == given_tdata) return GPR_PREPOSITION;
    if (PrepositionChain(o, token_n) ~= -1) return GPR_PREPOSITION;
    return -1;
GPR_TT:
    l = indirect(given_tdata);
    #Ifdef DEBUG;
    if (parser_trace >= 3) print "  [Outside parsing routine returned ", l, "]^";
    #Endif; ! DEBUG
    return l;
SCOPE_TT:
```

```
    scope_token = given_tdata;
    scope_stage = 1;
    #Ifdef DEBUG;
    if (parser_trace >= 3) print "  [Scope routine called at stage 1]^";
    #Endif; ! DEBUG
    l = indirect(scope_token);
    #Ifdef DEBUG;
    if (parser_trace >= 3) print "  [Scope routine returned multiple-flag of ", l, "]^";
    #Endif; ! DEBUG
    if (l == 1) given_tdata = MULTI_TOKEN; else given_tdata = NOUN_TOKEN;
ATTR_FILTER_TT:
    token_filter = 1 + given_tdata;
    given_tdata = NOUN_TOKEN;
ROUTINE_FILTER_TT:
    token_filter = given_tdata;
    given_tdata = NOUN_TOKEN;
} ! end of switch(given_ttype)
token = given_tdata;
```

§**25. Parse Token Letter B.** Begin parsing an object list.

```
! There are now three possible ways we can be here:
!     parsing an elementary token other than "special" or "number";
!     parsing a scope token;
!     parsing a noun-filter token (either by routine or attribute).
!
! In each case, token holds the type of elementary parse to
! perform in matching one or more objects, and
! token_filter is 0 (default), an attribute + 1 for an attribute filter
! or a routine address for a routine filter.
token_allows_multiple = false;
if (token == MULTI_TOKEN or MULTIHELD_TOKEN or MULTIEXCEPT_TOKEN or MULTIINSIDE_TOKEN)
    token_allows_multiple = true;
many_flag = false; and_parity = true; dont_infer = false;
```

§**26. Parse Token Letter C.** Parse descriptors (articles, pronouns, etc.) in the list.

```
    ! We expect to find a list of objects next in what the player's typed.
.ObjectList;
    #Ifdef DEBUG;
    if (parser_trace >= 3) print "  [Object list from word ", wn, "]^";
    #Endif; ! DEBUG
    ! Take an advance look at the next word: if it's "it" or "them", and these
    ! are unset, set the appropriate error number and give up on the line
    ! (if not, these are still parsed in the usual way - it is not assumed
    ! that they still refer to something in scope)
    o = NextWord(); wn--;
    pronoun_word = NULL; pronoun_obj = NULL;
    l = PronounValue(o);
    if (l ~= 0) {
        pronoun_word = o; pronoun_obj = l;
        if (l == NULL) {
            ! Don't assume this is a use of an unset pronoun until the
            ! descriptors have been checked, because it might be an
            ! article (or some such) instead
            for (l=1 : l<=LanguageDescriptors-->0 : l=l+4)
                if (o == LanguageDescriptors-->l) jump AssumeDescriptor;
            pronoun__word = pronoun_word; pronoun__obj = pronoun_obj;
            etype = VAGUE_PE;
            if (parser_trace >= 3) print "  [Stop: unset pronoun]^";
            return GPR_FAIL;
        }
    }
.AssumeDescriptor;
    if (o == ME1__WD or ME2__WD or ME3__WD) { pronoun_word = o; pronoun_obj = player; }
    allow_plurals = true; desc_wn = wn;
.TryAgain;
    ! First, we parse any descriptive words (like "the", "five" or "every"):
    l = Descriptors(token_allows_multiple);
    if (l ~= 0) { etype = l; return 0; }
.TryAgain2;
```

§**27. Parse Token Letter D.** Parse an object name.

```
! This is an actual specified object, and is therefore where a typing error
! is most likely to occur, so we set:
oops_from = wn;
! So, two cases.  Case 1: token not equal to "held" (so, no implicit takes)
! but we may well be dealing with multiple objects
! In either case below we use NounDomain, giving it the token number as
! context, and two places to look: among the actor's possessions, and in the
! present location.  (Note that the order depends on which is likeliest.)
if (token ~= HELD_TOKEN) {
    i = multiple_object-->0;
    #Ifdef DEBUG;
    if (parser_trace >= 3) print "  [Calling NounDomain on location and actor]^";
    #Endif; ! DEBUG
    l = NounDomain(actors_location, actor, token);
    if (l == REPARSE_CODE) return l;                    ! Reparse after Q&A
    if (indef_wanted == INDEF_ALL_WANTED && l == 0 && number_matched == 0)
        l = 1;  ! ReviseMulti if TAKE ALL FROM empty container
    if (token_allows_multiple && ~~multiflag) {
        if (best_etype==MULTI_PE) best_etype=STUCK_PE;
        multiflag = true;
    }
    if (l == 0) {
        if (indef_possambig) {
            ResetDescriptors();
            wn = desc_wn;
            jump TryAgain2;
        }
        if (etype == MULTI_PE or TOOFEW_PE && multiflag) etype = STUCK_PE;
        etype=CantSee();
        jump FailToken;
    } ! Choose best error
    #Ifdef DEBUG;
    if (parser_trace >= 3) {
        if (l > 1) print "  [ND returned ", (the) l, "]^";
        else {
            print "  [ND appended to the multiple object list:^";
            k = multiple_object-->0;
            for (j=i+1 : j<=k : j++)
                print "  Entry ", j, ": ", (The) multiple_object-->j,
                    " (", multiple_object-->j, ")^";
            print "  List now has size ", k, "]^";
        }
    }
    #Endif; ! DEBUG
    if (l == 1) {
        if (~~many_flag) many_flag = true;
        else {                                  ! Merge with earlier ones
            k = multiple_object-->0;              ! (with either parity)
            multiple_object-->0 = i;
            for (j=i+1 : j<=k : j++) {
```

```
                if (and_parity) MultiAdd(multiple_object-->j);
                else            MultiSub(multiple_object-->j);
            }
            #Ifdef DEBUG;
            if (parser_trace >= 3)
                print "  [Merging ", k-i, " new objects to the ", i, " old ones]^";
            #Endif; ! DEBUG
        }
    }
    else {
        ! A single object was indeed found
        if (match_length == 0 && indef_possambig) {
            ! So the answer had to be inferred from no textual data,
            ! and we know that there was an ambiguity in the descriptor
            ! stage (such as a word which could be a pronoun being
            ! parsed as an article or possessive).  It's worth having
            ! another go.
            ResetDescriptors();
            wn = desc_wn;
            jump TryAgain2;
        }
        if ((token == CREATURE_TOKEN) && (CreatureTest(l) == 0)) {
            etype = ANIMA_PE;
            jump FailToken;
        } !  Animation is required
        if (~~many_flag) single_object = l;
        else {
            if (and_parity) MultiAdd(l); else MultiSub(l);
            #Ifdef DEBUG;
            if (parser_trace >= 3) print "  [Combining ", (the) l, " with list]^";
            #Endif; ! DEBUG
        }
    }
}
else {
! Case 2: token is "held" (which fortunately can't take multiple objects)
! and may generate an implicit take
    l = NounDomain(actor,actors_location,token);       ! Same as above...
    if (l == REPARSE_CODE) return l;
    if (l == 0) {
        if (indef_possambig) {
            ResetDescriptors();
            wn = desc_wn;
            jump TryAgain2;
        }
        etype = CantSee(); jump FailToken;            ! Choose best error
    }
    ! ...until it produces something not held by the actor.  Then an implicit
    ! take must be tried.  If this is already happening anyway, things are too
    ! confused and we have to give up (but saving the oops marker so as to get
    ! it on the right word afterwards).
```

```
        ! The point of this last rule is that a sequence like
        !
        !     > read newspaper
        !     (taking the newspaper first)
        !     The dwarf unexpectedly prevents you from taking the newspaper!
        !
        ! should not be allowed to go into an infinite repeat - read becomes
        ! take then read, but take has no effect, so read becomes take then read...
        ! Anyway for now all we do is record the number of the object to take.
        o = parent(l);
        if (o ~= actor) {
            #Ifdef DEBUG;
            if (parser_trace >= 3) print "  [Allowing object ", (the) l, " for now]^";
            #Endif; ! DEBUG
        }
        single_object = l;
    } ! end of if (token ~= HELD_TOKEN) else
    ! The following moves the word marker to just past the named object...
    wn = oops_from + match_length;
```

§**28. Parse Token Letter E.** Parse connectives (AND, BUT, etc.) and go back to (C).

```
    ! Object(s) specified now: is that the end of the list, or have we reached
    ! "and", "but" and so on?  If so, create a multiple-object list if we
    ! haven't already (and are allowed to).
.NextInList;
    o = NextWord();
    if (o == AND1__WD or AND2__WD or AND3__WD or BUT1__WD or BUT2__WD or BUT3__WD or comma_word) {
        #Ifdef DEBUG;
        if (parser_trace >= 3) print "  [Read connective '", (address) o, "']^";
        #Endif; ! DEBUG
        if (~~token_allows_multiple) {
            if (multiflag) jump PassToken; ! give UPTO_PE error
            etype=MULTI_PE;
            jump FailToken;
        }
        if (o == BUT1__WD or BUT2__WD or BUT3__WD) and_parity = 1-and_parity;
        if (~~many_flag) {
            multiple_object-->0 = 1;
            multiple_object-->1 = single_object;
            many_flag = true;
            #Ifdef DEBUG;
            if (parser_trace >= 3) print "  [Making new list from ", (the) single_object, "]^";
            #Endif; ! DEBUG
        }
        dont_infer = true; inferfrom=0;           ! Don't print (inferences)
        jump ObjectList;                          ! And back around
    }
    wn--;   ! Word marker back to first not-understood word
```

§**29. Parse Token Letter F.** Return the conclusion of parsing an object list.

```
    ! Happy or unhappy endings:
.PassToken;
    if (many_flag) {
        single_object = GPR_MULTIPLE;
        multi_context = token;
    }
    else {
        if (indef_mode == 1 && indef_type & PLURAL_BIT ~= 0) {
            if (indef_wanted < INDEF_ALL_WANTED && indef_wanted > 1) {
                multi_had = 1; multi_wanted = indef_wanted;
                etype = TOOFEW_PE;
                jump FailToken;
            }
        }
    }
    return single_object;
.FailToken;
    ! If we were only guessing about it being a plural, try again but only
    ! allowing singulars (so that words like "six" are not swallowed up as
    ! Descriptors)
    if (allow_plurals && indef_guess_p == 1) {
        #Ifdef DEBUG;
        if (parser_trace >= 4) print "   [Retrying singulars after failure ", etype, "]^";
        #Endif;
        prev_indef_wanted = indef_wanted;
        allow_plurals = false;
        wn = desc_wn;
        jump TryAgain;
    }
    if ((indef_wanted > 0 || prev_indef_wanted > 0) && (~~multiflag)) etype = MULTI_PE;
    return GPR_FAIL;
]; ! end of ParseToken__
```

§**30. Descriptors.** In grammatical terms, a descriptor appears at the front of an English noun phrase and clarifies the quantity or specific identity of what is referred to: for instance, *my* mirror, *the* dwarf, *that* woman. (Numbers, as in *four* duets, are also descriptors in linguistics: but the I6 parser doesn't handle them that way.)

Slightly unfortunately, the bitmap constants used for descriptors in the I6 parser have names in the form `*_BIT`, coinciding with the names of style bits in the list-writer: but they never occur in the same context.

The actual words used as descriptors are read from tables in the language definition. `ArticleDescriptors` uses this table to move current word marker past a run of one or more descriptors which refer to the definite or indefinite article.

```
Constant OTHER_BIT  =   1;     !  These will be used in Adjudicate()
Constant MY_BIT     =   2;     !  to disambiguate choices
Constant THAT_BIT   =   4;
Constant PLURAL_BIT =   8;
Constant LIT_BIT    =  16;
Constant UNLIT_BIT  =  32;

[ ResetDescriptors;
    indef_mode = 0; indef_type = 0; indef_wanted = 0; indef_guess_p = 0;
    indef_possambig = false;
    indef_owner = nothing;
    indef_cases = $$111111111111;
    indef_nspec_at = 0;
];

[ ArticleDescriptors  o x flag cto type n;
    if (wn > num_words) return 0;
    for (flag=true : flag :) {
        o = NextWordStopped(); flag = false;
    for (x=1 : x<=LanguageDescriptors-->0 : x=x+4)
            if (o == LanguageDescriptors-->x) {
                type = LanguageDescriptors-->(x+2);
                if (type == DEFART_PK or INDEFART_PK) flag = true;
            }
    }
    wn--;
    return 0;
];
```

§**31. Parsing Descriptors.** The `Descriptors()` routine parses the descriptors at the head of a noun phrase, leaving the current word marker `wn` at the first word of the noun phrase's body. It is allowed to set up for a plural only if `allow_p` is set; it returns a parser error number, or 0 if no error occurred.

```
[ Descriptors  o x flag cto type n;
    ResetDescriptors();
    if (wn > num_words) return 0;
    for (flag=true : flag :) {
        o = NextWordStopped(); flag = false;
    for (x=1 : x<=LanguageDescriptors-->0 : x=x+4)
            if (o == LanguageDescriptors-->x) {
                flag = true;
                type = LanguageDescriptors-->(x+2);
                if (type ~= DEFART_PK) indef_mode = true;
                indef_possambig = true;
                indef_cases = indef_cases & (LanguageDescriptors-->(x+1));
                if (type == POSSESS_PK) {
                    cto = LanguageDescriptors-->(x+3);
                    switch (cto) {
                    0: indef_type = indef_type | MY_BIT;
                    1: indef_type = indef_type | THAT_BIT;
                    default:
                        indef_owner = PronounValue(cto);
                        if (indef_owner == NULL) indef_owner = InformParser;
                    }
                }
                if (type == light)  indef_type = indef_type | LIT_BIT;
                if (type == -light) indef_type = indef_type | UNLIT_BIT;
            }
        if (o == OTHER1__WD or OTHER2__WD or OTHER3__WD) {
            indef_mode = 1; flag = 1;
            indef_type = indef_type | OTHER_BIT;
        }
        if (o == ALL1__WD or ALL2__WD or ALL3__WD or ALL4__WD or ALL5__WD) {
            indef_mode = 1; flag = 1; indef_wanted = INDEF_ALL_WANTED;
            if (take_all_rule == 1) take_all_rule = 2;
            indef_type = indef_type | PLURAL_BIT;
        }
        if (allow_plurals) {
            n = NumberWord(o);
            if (n == 1) { indef_mode = 1; flag = 1; }
            if (n > 1) {
                indef_guess_p = 1;
                indef_mode = 1; flag = 1; indef_wanted = n;
                indef_nspec_at = wn-1;
                indef_type = indef_type | PLURAL_BIT;
            }
        }
        if (flag == 1 && NextWordStopped() ~= OF1__WD or OF2__WD or OF3__WD or OF4__WD)
            wn--;  ! Skip 'of' after these
    }
    wn--;
```

```
    return 0;
];
[ SafeSkipDescriptors;
    @push indef_mode; @push indef_type; @push indef_wanted;
    @push indef_guess_p; @push indef_possambig; @push indef_owner;
    @push indef_cases; @push indef_nspec_at;
    Descriptors();
    @pull indef_nspec_at; @pull indef_cases;
    @pull indef_owner; @pull indef_possambig; @pull indef_guess_p;
    @pull indef_wanted; @pull indef_type; @pull indef_mode;
];
```

§**32. Preposition Chain.** A small utility for runs of prepositions.

```
[ PrepositionChain wd index;
    if (line_tdata-->index == wd) return wd;
    if ((line_token-->index)->0 & $20 == 0) return -1;
    do {
        if (line_tdata-->index == wd) return wd;
        index++;
    } until ((line_token-->index == ENDIT_TOKEN) || (((line_token-->index)->0 & $10) == 0));
    return -1;
];
```

§**33. Creature.** Will this object do for an I6 `creature` token? (In I7 terms, this affects the tokens "[someone]", "[somebody]", "[anyone]" and "[anybody]".)

```
[ CreatureTest obj;
    if (obj has animate) rtrue;
    if (obj hasnt talkable) rfalse;
    if (action_to_be == ##Ask or ##Answer or ##Tell or ##AskFor) rtrue;
    rfalse;
];
```

§**34. Noun Domain.** `NounDomain` does the most substantial part of parsing an object name. It is given two "domains" – usually a location and then the actor who is looking – and a context (i.e. token type), and returns:

(a) 0 if no match at all could be made,
(b) 1 if a multiple object was made,
(c) k if object k was the one decided upon,
(d) `REPARSE_CODE` if it asked a question of the player and consequently rewrote the player's input, so that the whole parser should start again on the rewritten input.

In case (c), `NounDomain` also sets the variable `length_of_noun` to the number of words in the input text matched to the noun. In case (b), the multiple objects are added to `multiple_object` by hand (not by `MultiAdd`, because we want to allow duplicates).

```
[ NounDomain domain1 domain2 context
    first_word i j k l answer_words marker;
    #Ifdef DEBUG;
    if (parser_trace >= 4) {
        print "   [NounDomain called at word ", wn, "^";
        print "   ";
        if (indef_mode) {
            print "seeking indefinite object: ";
            if (indef_type & OTHER_BIT)  print "other ";
            if (indef_type & MY_BIT)     print "my ";
            if (indef_type & THAT_BIT)   print "that ";
            if (indef_type & PLURAL_BIT) print "plural ";
            if (indef_type & LIT_BIT)    print "lit ";
            if (indef_type & UNLIT_BIT)  print "unlit ";
            if (indef_owner ~= 0) print "owner:", (name) indef_owner;
            new_line;
            print "   number wanted: ";
            if (indef_wanted == INDEF_ALL_WANTED) print "all"; else print indef_wanted;
            new_line;
            print "   most likely GNAs of names: ", indef_cases, "^";
        }
        else print "seeking definite object^";
    }
    #Endif; ! DEBUG

    match_length = 0; number_matched = 0; match_from = wn;

    SearchScope(domain1, domain2, context);

    #Ifdef DEBUG;
    if (parser_trace >= 4) print "   [ND made ", number_matched, " matches]^";
    #Endif; ! DEBUG

    wn = match_from+match_length;

    ! If nothing worked at all, leave with the word marker skipped past the
    ! first unmatched word...

    if (number_matched == 0) { wn++; rfalse; }

    ! Suppose that there really were some words being parsed (i.e., we did
    ! not just infer).  If so, and if there was only one match, it must be
    ! right and we return it...

    if (match_from <= num_words) {
        if (number_matched == 1) {
            i=match_list-->0;
```

```
        return i;
    }
    ! ...now suppose that there was more typing to come, i.e. suppose that
    ! the user entered something beyond this noun.  If nothing ought to follow,
    ! then there must be a mistake, (unless what does follow is just a full
    ! stop, and or comma)
    if (wn <= num_words) {
        i = NextWord(); wn--;
        if (i ~=  AND1__WD or AND2__WD or AND3__WD or comma_word
            or THEN1__WD or THEN2__WD or THEN3__WD
            or BUT1__WD or BUT2__WD or BUT3__WD) {
            if (lookahead == ENDIT_TOKEN) rfalse;
        }
    }
}
! Now look for a good choice, if there's more than one choice...
number_of_classes = 0;
if (number_matched == 1) i = match_list-->0;
if (number_matched > 1) {
    i = Adjudicate(context);
    if (i == -1) rfalse;
    if (i == 1) rtrue;       !  Adjudicate has made a multiple
                         !  object, and we pass it on
}
! If i is non-zero here, one of two things is happening: either
! (a) an inference has been successfully made that object i is
!     the intended one from the user's specification, or
! (b) the user finished typing some time ago, but we've decided
!     on i because it's the only possible choice.
! In either case we have to keep the pattern up to date,
! note that an inference has been made and return.
! (Except, we don't note which of a pile of identical objects.)
if (i ~= 0) {
    if (dont_infer) return i;
    if (inferfrom == 0) inferfrom=pcount;
    pattern-->pcount = i;
    return i;
}
! If we get here, there was no obvious choice of object to make.  If in
! fact we've already gone past the end of the player's typing (which
! means the match list must contain every object in scope, regardless
! of its name), then it's foolish to give an enormous list to choose
! from - instead we go and ask a more suitable question...
if (match_from > num_words) jump Incomplete;
! Now we print up the question, using the equivalence classes as worked
! out by Adjudicate() so as not to repeat ourselves on plural objects...
BeginActivity(ASKING_WHICH_DO_YOU_MEAN_ACT);
if (ForActivity(ASKING_WHICH_DO_YOU_MEAN_ACT)) jump SkipWhichQuestion;
if (context==CREATURE_TOKEN) L__M(##Miscellany, 45);
else                         L__M(##Miscellany, 46);
```

```
    j = number_of_classes; marker = 0;
    for (i=1 : i<=number_of_classes : i++) {
        while (((match_classes-->marker) ~= i) && ((match_classes-->marker) ~= -i)) marker++;
        k = match_list-->marker;
        if (match_classes-->marker > 0) print (the) k; else print (a) k;
        if (i < j-1)  print (string) COMMA__TX;
        if (i == j-1) {
            #Ifdef SERIAL_COMMA;
            print ",";
            #Endif; ! SERIAL_COMMA
            print (string) OR__TX;
        }
    }
    L__M(##Miscellany, 57);
    .SkipWhichQuestion; EndActivity(ASKING_WHICH_DO_YOU_MEAN_ACT);
    ! ...and get an answer:
.WhichOne;
    #Ifdef TARGET_ZCODE;
    for (i=2 : i<INPUT_BUFFER_LEN : i++) buffer2->i = ' ';
    #Endif; ! TARGET_ZCODE
    answer_words=Keyboard(buffer2, parse2);
    ! Conveniently, parse2-->1 is the first word in both ZCODE and GLULX.
    first_word = (parse2-->1);
    ! Take care of "all", because that does something too clever here to do
    ! later on:
    if (first_word == ALL1__WD or ALL2__WD or ALL3__WD or ALL4__WD or ALL5__WD) {
        if (context == MULTI_TOKEN or MULTIHELD_TOKEN or MULTIEXCEPT_TOKEN or MULTIINSIDE_TOKEN) {
            l = multiple_object-->0;
            for (i=0 : i<number_matched && l+i<MATCH_LIST_WORDS : i++) {
                k = match_list-->i;
                multiple_object-->(i+1+l) = k;
            }
            multiple_object-->0 = i+l;
            rtrue;
        }
        L__M(##Miscellany, 47);
        jump WhichOne;
    }
    ! If the first word of the reply can be interpreted as a verb, then
    ! assume that the player has ignored the question and given a new
    ! command altogether.
    ! (This is one time when it's convenient that the directions are
    ! not themselves verbs - thus, "north" as a reply to "Which, the north
    ! or south door" is not treated as a fresh command but as an answer.)
    #Ifdef LanguageIsVerb;
    if (first_word == 0) {
        j = wn; first_word = LanguageIsVerb(buffer2, parse2, 1); wn = j;
    }
    #Endif; ! LanguageIsVerb
    if (first_word ~= 0) {
        j = first_word->#dict_par1;
```

```
        if ((0 ~= j&1) && ~~LanguageVerbMayBeName(first_word)) {
            VM_CopyBuffer(buffer, buffer2);
            jump RECONSTRUCT_INPUT;
        }
    }
    ! Now we insert the answer into the original typed command, as
    ! words additionally describing the same object
    ! (eg, > take red button
    !      Which one, ...
    !      > music
    ! becomes "take music red button".  The parser will thus have three
    ! words to work from next time, not two.)
    #Ifdef TARGET_ZCODE;
    k = WordAddress(match_from) - buffer; l=buffer2->1+1;
    for (j=buffer + buffer->0 - 1 : j>=buffer+k+l : j--) j->0 = 0->(j-l);
    for (i=0 : i<l : i++) buffer->(k+i) = buffer2->(2+i);
    buffer->(k+l-1) = ' ';
    buffer->1 = buffer->1 + l;
    if (buffer->1 >= (buffer->0 - 1)) buffer->1 = buffer->0;
    #Ifnot; ! TARGET_GLULX
    k = WordAddress(match_from) - buffer;
    l = (buffer2-->0) + 1;
    for (j=buffer+INPUT_BUFFER_LEN-1 : j>=buffer+k+l : j--) j->0 = j->(-l);
    for (i=0 : i<l : i++) buffer->(k+i) = buffer2->(WORDSIZE+i);
    buffer->(k+l-1) = ' ';
    buffer-->0 = buffer-->0 + l;
    if (buffer-->0 > (INPUT_BUFFER_LEN-WORDSIZE)) buffer-->0 = (INPUT_BUFFER_LEN-WORDSIZE);
    #Endif; ! TARGET_
    ! Having reconstructed the input, we warn the parser accordingly
    ! and get out.
    .RECONSTRUCT_INPUT;
    num_words = WordCount();
    wn = 1;
    #Ifdef LanguageToInformese;
    LanguageToInformese();
    ! Re-tokenise:
    VM_Tokenise(buffer,parse);
    #Endif; ! LanguageToInformese
    num_words = WordCount();
    players_command = 100 + WordCount();
    FollowRulebook(Activity_after_rulebooks-->READING_A_COMMAND_ACT, true);
    return REPARSE_CODE;
    ! Now we come to the question asked when the input has run out
    ! and can't easily be guessed (eg, the player typed "take" and there
    ! were plenty of things which might have been meant).
.Incomplete;
    if (context == CREATURE_TOKEN) L__M(##Miscellany, 48);
    else                           L__M(##Miscellany, 49);
    #Ifdef TARGET_ZCODE;
    for (i=2 : i<INPUT_BUFFER_LEN : i++) buffer2->i=' ';
    #Endif; ! TARGET_ZCODE
```

```
answer_words = Keyboard(buffer2, parse2);
first_word=(parse2-->1);
#Ifdef LanguageIsVerb;
if (first_word==0) {
    j = wn; first_word=LanguageIsVerb(buffer2, parse2, 1); wn = j;
}
#Endif; ! LanguageIsVerb
! Once again, if the reply looks like a command, give it to the
! parser to get on with and forget about the question...
if (first_word ~= 0) {
    j = first_word->#dict_par1;
    if (0 ~= j&1) {
        VM_CopyBuffer(buffer, buffer2);
        return REPARSE_CODE;
    }
}
! ...but if we have a genuine answer, then:
!
! (1) we must glue in text suitable for anything that's been inferred.
if (inferfrom ~= 0) {
    for (j=inferfrom : j<pcount : j++) {
        if (pattern-->j == PATTERN_NULL) continue;
        #Ifdef TARGET_ZCODE;
        i = 2+buffer->1; (buffer->1)++; buffer->(i++) = ' ';
        #Ifnot; ! TARGET_GLULX
        i = WORDSIZE + buffer-->0;
        (buffer-->0)++; buffer->(i++) = ' ';
        #Endif; ! TARGET_
        #Ifdef DEBUG;
        if (parser_trace >= 5)
            print "[Gluing in inference with pattern code ", pattern-->j, "]^";
        #Endif; ! DEBUG
        ! Conveniently, parse2-->1 is the first word in both ZCODE and GLULX.
        parse2-->1 = 0;
        ! An inferred object.  Best we can do is glue in a pronoun.
        ! (This is imperfect, but it's very seldom needed anyway.)
        if (pattern-->j >= 2 && pattern-->j < REPARSE_CODE) {
            PronounNotice(pattern-->j);
            for (k=1 : k<=LanguagePronouns-->0 : k=k+3)
                if (pattern-->j == LanguagePronouns-->(k+2)) {
                    parse2-->1 = LanguagePronouns-->k;
                    #Ifdef DEBUG;
                    if (parser_trace >= 5)
                        print "[Using pronoun '", (address) parse2-->1, "']^";
                    #Endif; ! DEBUG
                    break;
                }
        }
        else {
            ! An inferred preposition.
            parse2-->1 = VM_NumberToDictionaryAddress(pattern-->j - REPARSE_CODE);
```

```
                #Ifdef DEBUG;
                if (parser_trace >= 5)
                    print "[Using preposition '", (address) parse2-->1, "']^";
                #Endif; ! DEBUG
            }
            ! parse2-->1 now holds the dictionary address of the word to glue in.
            if (parse2-->1 ~= 0) {
                k = buffer + i;
                #Ifdef TARGET_ZCODE;
                @output_stream 3 k;
                print (address) parse2-->1;
                @output_stream -3;
                k = k-->0;
                for (l=i : l<i+k : l++) buffer->l = buffer->(l+2);
                i = i + k; buffer->1 = i-2;
                #Ifnot; ! TARGET_GLULX
                k = Glulx_PrintAnyToArray(buffer+i, INPUT_BUFFER_LEN-i, parse2-->1);
                i = i + k; buffer-->0 = i - WORDSIZE;
                #Endif; ! TARGET_
            }
        }
    }
    ! (2) we must glue the newly-typed text onto the end.
    #Ifdef TARGET_ZCODE;
    i = 2+buffer->1; (buffer->1)++; buffer->(i++) = ' ';
    for (j=0 : j<buffer2->1 : i++,j++) {
        buffer->i = buffer2->(j+2);
        (buffer->1)++;
        if (buffer->1 == INPUT_BUFFER_LEN) break;
    }
    #Ifnot; ! TARGET_GLULX
    i = WORDSIZE + buffer-->0;
    (buffer-->0)++; buffer->(i++) = ' ';
    for (j=0 : j<buffer2-->0 : i++,j++) {
        buffer->i = buffer2->(j+WORDSIZE);
        (buffer-->0)++;
        if (buffer-->0 == INPUT_BUFFER_LEN) break;
    }
    #Endif; ! TARGET_
    ! (3) we fill up the buffer with spaces, which is unnecessary, but may
    !     help incorrectly-written interpreters to cope.
    #Ifdef TARGET_ZCODE;
    for (: i<INPUT_BUFFER_LEN : i++) buffer->i = ' ';
    #Endif; ! TARGET_ZCODE
    return REPARSE_CODE;
]; ! end of NounDomain
```

§**35. Adjudicate.** The `Adjudicate` routine tries to see if there is an obvious choice, when faced with a list of objects (the `match_list`) each of which matches the player's specification equally well. To do this it makes use of the `context` (the token type being worked on).

It counts up the number of obvious choices for the given context – all to do with where a candidate is, except for 6 (`animate`) which is to do with whether it is animate or not – and then:

(a) if only one obvious choice is found, that is returned;
(b) if we are in indefinite mode (don't care which) one of the obvious choices is returned, or if there is no obvious choice then an unobvious one is made;
(c) at this stage, we work out whether the objects are distinguishable from each other or not: if they are all indistinguishable from each other, then choose one, it doesn't matter which;
(d) otherwise, 0 (meaning, unable to decide) is returned (but remember that the equivalence classes we've just worked out will be needed by other routines to clear up this mess, so we can't economise on working them out).

`Adjudicate` returns -1 if an error occurred.

```
[ Adjudicate context i j k good_ones last n ultimate flag offset;
    #Ifdef DEBUG;
    if (parser_trace >= 4) {
        print "   [Adjudicating match list of size ", number_matched,
            " in context ", context, "^";
        print "   ";
        if (indef_mode) {
            print "indefinite type: ";
            if (indef_type & OTHER_BIT)  print "other ";
            if (indef_type & MY_BIT)     print "my ";
            if (indef_type & THAT_BIT)   print "that ";
            if (indef_type & PLURAL_BIT) print "plural ";
            if (indef_type & LIT_BIT)    print "lit ";
            if (indef_type & UNLIT_BIT)  print "unlit ";
            if (indef_owner ~= 0) print "owner:", (name) indef_owner;
            new_line;
            print "   number wanted: ";
            if (indef_wanted == INDEF_ALL_WANTED) print "all"; else print indef_wanted;
            new_line;
            print "   most likely GNAs of names: ", indef_cases, "^";
        }
        else print "definite object^";
    }
    #Endif; ! DEBUG

    j = number_matched-1; good_ones = 0; last = match_list-->0;
    for (i=0 : i<=j : i++) {
        n = match_list-->i;
        match_scores-->i = good_ones;
        ultimate = ScopeCeiling(n);

        if (context==HELD_TOKEN && parent(n)==actor)
        {   good_ones++; last=n; }
        if (context==MULTI_TOKEN && ultimate==ScopeCeiling(actor)
            && n~=actor && n hasnt concealed && n hasnt scenery)
        {   good_ones++; last=n; }
        if (context==MULTIHELD_TOKEN && parent(n)==actor)
        {   good_ones++; last=n; }

        if (context==MULTIEXCEPT_TOKEN or MULTIINSIDE_TOKEN)
```

```
    {   if (advance_warning==-1)
        {   if (context==MULTIEXCEPT_TOKEN)
            {   good_ones++; last=n;
            }
            if (context==MULTIINSIDE_TOKEN)
            {   if (parent(n)~=actor) { good_ones++; last=n; }
            }
        }
        else
        {   if (context==MULTIEXCEPT_TOKEN && n~=advance_warning)
            {   good_ones++; last=n; }
            if (context==MULTIINSIDE_TOKEN && n in advance_warning)
            {   good_ones++; last=n; }
        }
    }
    if (context==CREATURE_TOKEN && CreatureTest(n)==1)
    {   good_ones++; last=n; }
    match_scores-->i = 1000*(good_ones - match_scores-->i);
}
if (good_ones == 1) return last;
! If there is ambiguity about what was typed, but it definitely wasn't
! animate as required, then return anything; higher up in the parser
! a suitable error will be given.  (This prevents a question being asked.)
if (context == CREATURE_TOKEN && good_ones == 0) return match_list-->0;
if (indef_mode == 0) indef_type=0;
ScoreMatchL(context);
if (number_matched == 0) return -1;
if (indef_mode == 0) {
    !  Is there now a single highest-scoring object?
    i = SingleBestGuess();
    if (i >= 0) {
        #Ifdef DEBUG;
        if (parser_trace >= 4) print "   Single best-scoring object returned.]^";
        #Endif; ! DEBUG
        return i;
    }
}
if (indef_mode == 1 && indef_type & PLURAL_BIT ~= 0) {
    if (context ~= MULTI_TOKEN or MULTIHELD_TOKEN or MULTIEXCEPT_TOKEN
                or MULTIINSIDE_TOKEN) {
        etype = MULTI_PE;
        return -1;
    }
    i = 0; offset = multiple_object-->0;
    for (j=BestGuess(): j~=-1 && i<indef_wanted && i+offset<MATCH_LIST_WORDS-1: j=BestGuess()) {
        flag = 0;
        BeginActivity(DECIDING_WHETHER_ALL_INC_ACT, j);
        if ((ForActivity(DECIDING_WHETHER_ALL_INC_ACT, j)) == 0) {
            if (j hasnt concealed && j hasnt worn) flag = 1;
            if (context == MULTIHELD_TOKEN or MULTIEXCEPT_TOKEN && parent(j) ~= actor)
                flag = 0;
```

```
            if (action_to_be == ##Take or ##Remove && parent(j) == actor)
                flag = 0;
            k = ChooseObjects(j, flag);
            if (k == 1)
                flag = 1;
            else {
                if (k == 2) flag = 0;
            }
        } else {
            flag = 0; if (RulebookSucceeded()) flag = 1;
        }
        EndActivity(DECIDING_WHETHER_ALL_INC_ACT, j);
        if (flag == 1) {
            i++; multiple_object-->(i+offset) = j;
            #Ifdef DEBUG;
            if (parser_trace >= 4) print "   Accepting it^";
            #Endif; ! DEBUG
        }
        else {
            i = i;
            #Ifdef DEBUG;
            if (parser_trace >= 4) print "   Rejecting it^";
            #Endif; ! DEBUG
        }
    }
    if (i < indef_wanted && indef_wanted < INDEF_ALL_WANTED) {
        etype = TOOFEW_PE; multi_wanted = indef_wanted;
        multi_had=i;
        return -1;
    }
    multiple_object-->0 = i+offset;
    multi_context = context;
    #Ifdef DEBUG;
    if (parser_trace >= 4)
        print "   Made multiple object of size ", i, "]^";
    #Endif; ! DEBUG
    return 1;
}
for (i=0 : i<number_matched : i++) match_classes-->i = 0;
n = 1;
for (i=0 : i<number_matched : i++)
    if (match_classes-->i == 0) {
        match_classes-->i = n++; flag = 0;
        for (j=i+1 : j<number_matched : j++)
            if (match_classes-->j == 0 && Identical(match_list-->i, match_list-->j) == 1) {
                flag=1;
                match_classes-->j = match_classes-->i;
            }
        if (flag == 1) match_classes-->i = 1-n;
    }
n--; number_of_classes = n;
#Ifdef DEBUG;
```

```
    if (parser_trace >= 4) {
        print "   Grouped into ", n, " possibilities by name:^";
        for (i=0 : i<number_matched : i++)
            if (match_classes-->i > 0)
                print "   ", (The) match_list-->i, " (", match_list-->i, ")  ---  group ",
                match_classes-->i, "^";
    }
    #Endif; ! DEBUG
    if (indef_mode == 0) {
        if (n > 1) {
            k = -1;
            for (i=0 : i<number_matched : i++) {
                if (match_scores-->i > k) {
                    k = match_scores-->i;
                    j = match_classes-->i; j = j*j;
                    flag = 0;
                }
                else
                    if (match_scores-->i == k) {
                        if ((match_classes-->i) * (match_classes-->i) ~= j)
                            flag = 1;
                    }
            }
        if (flag) {
            #Ifdef DEBUG;
            if (parser_trace >= 4) print "   Unable to choose best group, so ask player.]^";
            #Endif; ! DEBUG
            return 0;
        }
        #Ifdef DEBUG;
        if (parser_trace >= 4) print "   Best choices are all from the same group.^";
        #Endif; ! DEBUG
        }
    }
    !  When the player is really vague, or there's a single collection of
    !  indistinguishable objects to choose from, choose the one the player
    !  most recently acquired, or if the player has none of them, then
    !  the one most recently put where it is.
    if (n == 1) dont_infer = true;
    return BestGuess();
]; ! Adjudicate
```

§**36. ReviseMulti.** `ReviseMulti` revises the multiple object which already exists, in the light of information which has come along since then (i.e., the second parameter). It returns a parser error number, or else 0 if all is well. This only ever throws things out, never adds new ones.

```
[ ReviseMulti second_p  i low;
    #Ifdef DEBUG;
    if (parser_trace >= 4)
        print "   Revising multiple object list of size ", multiple_object-->0,
        " with 2nd ", (name) second_p, "^";
    #Endif; ! DEBUG
    if (multi_context == MULTIEXCEPT_TOKEN or MULTIINSIDE_TOKEN) {
        for (i=1,low=0 : i<=multiple_object-->0 : i++) {
            if ( (multi_context==MULTIEXCEPT_TOKEN && multiple_object-->i ~= second_p) ||
                 (multi_context==MULTIINSIDE_TOKEN && multiple_object-->i in second_p)) {
                low++;
                multiple_object-->low = multiple_object-->i;
            }
        }
        multiple_object-->0 = low;
    }
    if (multi_context == MULTI_TOKEN && action_to_be == ##Take) {
        #Ifdef DEBUG;
        if (parser_trace >= 4) print "   Token 2 plural case: number with actor ", low, "^";
        #Endif; ! DEBUG
        if (take_all_rule == 2) {
            for (i=1,low=0 : i<=multiple_object-->0 : i++) {
                if (ScopeCeiling(multiple_object-->i) == ScopeCeiling(actor)) {
                    low++;
                    multiple_object-->low = multiple_object-->i;
                }
            }
            multiple_object-->0 = low;
        }
    }
    i = multiple_object-->0;
    #Ifdef DEBUG;
    if (parser_trace >= 4) print "   Done: new size ", i, "^";
    #Endif; ! DEBUG
    if (i == 0) return NOTHING_PE;
    return 0;
];
```

§**37. Match List.** The match list is an array, `match_list-->`, which holds the current best guesses at what object(s) a portion of the command refers to. The global `number_matched` is set to the current length of the `match_list`.

When the parser sees a possible match of object `obj` at quality level `q`, it calls `MakeMatch(obj, q)`. If this is the best quality match so far, then we wipe out all the previous matches and start a new list with this one. If it's only as good as the best so far, we add it to the list (provided we haven't run out of space, and provided it isn't in the list already). If it's worse, we ignore it altogether.

I6 tokens in the form `noun=Filter` or `Attribute` are "noun filter tokens", and mean that the match list should be filtered to accept only nouns which are acceptable to the given routine, or have the given attribute. Such a token is in force if `token_filter` is used. (I7 makes no use of this in the attribute case, which is deprecated nowadays.)

Quality is essentially the number of words in the command referring to the object: the idea is that "red panic button" is better than "red button" or "panic".

```
[ MakeMatch obj quality i;
    #Ifdef DEBUG;
    if (parser_trace >= 6) print "    Match with quality ",quality,"^";
    #Endif; ! DEBUG
    if (token_filter ~= 0 && ConsultNounFilterToken(obj) == 0) {
        #Ifdef DEBUG;
        if (parser_trace >= 6) print "    Match filtered out: token filter ", token_filter, "^";
        #Endif; ! DEBUG
        rtrue;
    }
    if (quality < match_length) rtrue;
    if (quality > match_length) { match_length = quality; number_matched = 0; }
    else {
        if (number_matched >= MATCH_LIST_WORDS) rtrue;
        for (i=0 : i<number_matched : i++)
            if (match_list-->i == obj) rtrue;
    }
    match_list-->number_matched++ = obj;
    #Ifdef DEBUG;
    if (parser_trace >= 6) print "    Match added to list^";
    #Endif; ! DEBUG
];
[ ConsultNounFilterToken obj;
    if (token_filter ofclass Routine) {
        noun = obj;
        return indirect(token_filter);
    }
    if (obj has (token_filter-1)) rtrue;
    rfalse;
];
```

§**38. ScoreMatchL.** `ScoreMatchL` scores the match list for quality in terms of what the player has vaguely asked for. Points are awarded for conforming with requirements like "my", and so on. Remove from the match list any entries which fail the basic requirements of the descriptors. (The scoring system used to evaluate the possibilities is discussed in detail in the DM4.)

```
Constant SCORE__CHOOSEOBJ = 1000;
Constant SCORE__IFGOOD = 500;
Constant SCORE__UNCONCEALED = 100;
Constant SCORE__BESTLOC = 60;
Constant SCORE__NEXTBESTLOC = 40;
Constant SCORE__NOTCOMPASS = 20;
Constant SCORE__NOTSCENERY = 10;
Constant SCORE__NOTACTOR = 5;
Constant SCORE__GNA = 1;
Constant SCORE__DIVISOR = 20;

Constant PREFER_HELD;
[ ScoreMatchL context its_owner its_score obj i j threshold met a_s l_s;
!   if (indef_type & OTHER_BIT ~= 0) threshold++;
    if (indef_type & MY_BIT ~= 0)    threshold++;
    if (indef_type & THAT_BIT ~= 0)  threshold++;
    if (indef_type & LIT_BIT ~= 0)   threshold++;
    if (indef_type & UNLIT_BIT ~= 0) threshold++;
    if (indef_owner ~= nothing)      threshold++;

    #Ifdef DEBUG;
    if (parser_trace >= 4) print "   Scoring match list: indef mode ", indef_mode, " type ",
    indef_type, ", satisfying ", threshold, " requirements:^";
    #Endif; ! DEBUG

    #ifdef PREFER_HELD;
    a_s = SCORE__BESTLOC; l_s = SCORE__NEXTBESTLOC;
    if (action_to_be == ##Take or ##Remove) {
        a_s = SCORE__NEXTBESTLOC; l_s = SCORE__BESTLOC;
    }
    context = context;  ! silence warning
    #ifnot;
    a_s = SCORE__NEXTBESTLOC; l_s = SCORE__BESTLOC;
    if (context == HELD_TOKEN or MULTIHELD_TOKEN or MULTIEXCEPT_TOKEN) {
        a_s = SCORE__BESTLOC; l_s = SCORE__NEXTBESTLOC;
    }
    #endif; ! PREFER_HELD

    for (i=0 : i<number_matched : i++) {
        obj = match_list-->i; its_owner = parent(obj); its_score=0; met=0;
        !      if (indef_type & OTHER_BIT ~= 0
        !          &&  obj ~= itobj or himobj or herobj) met++;
        if (indef_type & MY_BIT ~= 0 && its_owner == actor) met++;
        if (indef_type & THAT_BIT ~= 0 && its_owner == actors_location) met++;
        if (indef_type & LIT_BIT ~= 0 && obj has light) met++;
        if (indef_type & UNLIT_BIT ~= 0 && obj hasnt light) met++;
        if (indef_owner ~= 0 && its_owner == indef_owner) met++;

        if (met < threshold) {
            #Ifdef DEBUG;
            if (parser_trace >= 4)
                print "   ", (The) match_list-->i, " (", match_list-->i, ") in ",
```

```
                (the) its_owner, " is rejected (doesn't match descriptors)^";
            #Endif; ! DEBUG
            match_list-->i = -1;
        }
        else {
            its_score = 0;
            if (obj hasnt concealed) its_score = SCORE__UNCONCEALED;
            if (its_owner == actor) its_score = its_score + a_s;
            else
                if (its_owner == actors_location) its_score = its_score + l_s;
                else
                    if (its_owner ~= compass) its_score = its_score + SCORE__NOTCOMPASS;
            its_score = its_score + SCORE__CHOOSEOBJ * ChooseObjects(obj, 2);
            if (obj hasnt scenery) its_score = its_score + SCORE__NOTSCENERY;
            if (obj ~= actor) its_score = its_score + SCORE__NOTACTOR;
            !   A small bonus for having the correct GNA,
            !   for sorting out ambiguous articles and the like.
            if (indef_cases & (PowersOfTwo_TB-->(GetGNAOfObject(obj))))
                its_score = its_score + SCORE__GNA;
            match_scores-->i = match_scores-->i + its_score;
            #Ifdef DEBUG;
            if (parser_trace >= 4) print "     ", (The) match_list-->i, " (", match_list-->i,
            ") in ", (the) its_owner, " : ", match_scores-->i, " points^";
            #Endif; ! DEBUG
        }
    }
    for (i=0 : i<number_matched : i++) {
        while (match_list-->i == -1) {
            if (i == number_matched-1) { number_matched--; break; }
            for (j=i : j<number_matched-1 : j++) {
                match_list-->j = match_list-->(j+1);
                match_scores-->j = match_scores-->(j+1);
            }
            number_matched--;
        }
    }
];
```

§**39. BestGuess.** `BestGuess` makes the best guess it can out of the match list, assuming that everything in the match list is textually as good as everything else; however it ignores items marked as −1, and so marks anything it chooses. It returns −1 if there are no possible choices.

```
[ BestGuess  earliest its_score best i;
    earliest = 0; best = -1;
    for (i=0 : i<number_matched : i++) {
        if (match_list-->i >= 0) {
            its_score = match_scores-->i;
            if (its_score > best) { best = its_score; earliest = i; }
        }
    }
    #Ifdef DEBUG;
    if (parser_trace >= 4)
    if (best < 0) print "   Best guess ran out of choices^";
    else print "   Best guess ", (the) match_list-->earliest,
        " (", match_list-->earliest, ")^";
    #Endif; ! DEBUG
    if (best < 0) return -1;
    i = match_list-->earliest;
    match_list-->earliest = -1;
    return i;
];
```

§**40. SingleBestGuess.** `SingleBestGuess` returns the highest-scoring object in the match list if it is the clear winner, or returns −1 if there is no clear winner.

```
[ SingleBestGuess  earliest its_score best i;
    earliest = -1; best = -1000;
    for (i=0 : i<number_matched : i++) {
        its_score = match_scores-->i;
        if (its_score == best) earliest = -1;
        if (its_score > best) { best = its_score; earliest = match_list-->i; }
    }
    return earliest;
];
```

§**41. Identical.** `Identical` decides whether or not two objects can be distinguished from each other by anything the player can type. If not, it returns `true`. (This routine is critical to the handling of plurals, and the list-writer requires it to be an equivalence relation between objects: but it is, because it is equivalent to $O_1 \sim O_2$ if and only if $f(O_1) = f(O_2)$ for some function f.)

```
[ Identical o1 o2 p1 p2 n1 n2 i j flag;
    if (o1 == o2) rtrue;  ! This should never happen, but to be on the safe side
    if (o1 == 0 || o2 == 0) rfalse;  ! Similarly
    if (o1 ofclass K3_direction || o2 ofclass K3_direction) rfalse; ! Saves time
    !  What complicates things is that o1 or o2 might have a parsing routine,
    !  so the parser can't know from here whether they are or aren't the same.
    !  If they have different parsing routines, we simply assume they're
    !  different.  If they have the same routine (which they probably got from
    !  a class definition) then the decision process is as follows:
    !
    !     the routine is called (with self being o1, not that it matters)
    !       with noun and second being set to o1 and o2, and action being set
    !       to the fake action TheSame.  If it returns -1, they are found
    !       identical; if -2, different; and if >=0, then the usual method
    !       is used instead.
    if (o1.parse_name ~= 0 || o2.parse_name ~= 0) {
    if (o1.parse_name ~= o2.parse_name) rfalse;
    parser_action = ##TheSame; parser_one = o1; parser_two = o2;
    j = wn; i = RunRoutines(o1,parse_name); wn = j;
    if (i == -1) rtrue;
    if (i == -2) rfalse;
    }
    !  This is the default algorithm: do they have the same words in their
    !  "name" (i.e. property no. 1) properties.  (Note that the following allows
    !  for repeated words and words in different orders.)
    p1 = o1.&1; n1 = (o1.#1)/WORDSIZE;
    p2 = o2.&1; n2 = (o2.#1)/WORDSIZE;
    !  for (i=0 : i<n1 : i++) { print (address) p1-->i, " "; } new_line;
    !  for (i=0 : i<n2 : i++) { print (address) p2-->i, " "; } new_line;
    for (i=0 : i<n1 : i++) {
        flag = 0;
        for (j=0 : j<n2 : j++)
            if (p1-->i == p2-->j) flag = 1;
        if (flag == 0) rfalse;
    }
    for (j=0 : j<n2 : j++) {
        flag = 0;
        for (i=0 : i<n1 : i++)
            if (p1-->i == p2-->j) flag = 1;
        if (flag == 0) rfalse;
    }
    !  print "Which are identical!^";
    rtrue;
];
```

§**42. Print Command.** `PrintCommand` reconstructs the command as it presently reads, from the pattern which has been built up.

If `from` is 0, it starts with the verb: then it goes through the pattern.

The other parameter is `emptyf` – a flag: if 0, it goes up to `pcount`: if 1, it goes up to `pcount`-1.

Note that verbs and prepositions are printed out of the dictionary: and that since the dictionary may only preserve the first six characters of a word (in a V3 game), we have to hand-code the longer words needed. At present, I7 doesn't do this, but it probably should.

(Recall that pattern entries are 0 for "multiple object", 1 for "special word", 2 to `REPARSE_CODE-1` are object numbers and `REPARSE_CODE+n` means the preposition `n`.)

```
[ PrintInferredCommand from singleton_noun;
    singleton_noun = FALSE;
    if ((from ~= 0) && (from == pcount-1) &&
        (pattern-->from > 1) && (pattern-->from < REPARSE_CODE))
            singleton_noun = TRUE;
    if (singleton_noun) {
        BeginActivity(CLARIFYING_PARSERS_CHOICE_ACT, pattern-->from);
        if (ForActivity(CLARIFYING_PARSERS_CHOICE_ACT, pattern-->from) == 0) {
            print "("; PrintCommand(from); print ")^";
        }
        EndActivity(CLARIFYING_PARSERS_CHOICE_ACT, pattern-->from);
    } else {
        print "("; PrintCommand(from); print ")^";
    }
];
[ PrintCommand from i k spacing_flag;
    if (from == 0) {
        i = verb_word;
        if (LanguageVerb(i) == 0)
            if (PrintVerb(i) == 0) print (address) i;
        from++; spacing_flag = true;
    }
    for (k=from : k<pcount : k++) {
        i = pattern-->k;
        if (i == PATTERN_NULL) continue;
        if (spacing_flag) print (char) ' ';
        if (i == 0) { print (string) THOSET__TX; jump TokenPrinted; }
        if (i == 1) { print (string) THAT__TX;   jump TokenPrinted; }
        if (i >= REPARSE_CODE)
            print (address) VM_NumberToDictionaryAddress(i-REPARSE_CODE);
        else
            if (i ofclass K3_direction)
                print (LanguageDirection) i; ! the direction name as adverb
            else
                print (the) i;
    .TokenPrinted;
        spacing_flag = true;
    }
];
```

§**43. CantSee.** The `CantSee` routine returns a good error number for the situation where the last word looked at didn't seem to refer to any object in context.

The idea is that: if the actor is in a location (but not inside something like, for instance, a tank which is in that location) then an attempt to refer to one of the words listed as meaningful-but-irrelevant there will cause "you don't need to refer to that in this game" rather than "no such thing" or "what's 'it'?".

(The advantage of not having looked at "irrelevant" local nouns until now is that it stops them from clogging up the ambiguity-resolving process. Thus game objects always triumph over scenery.)

```
[ CantSee  i w e;
    saved_oops=oops_from;
    if (scope_token ~= 0) {
        scope_error = scope_token; return ASKSCOPE_PE;
    }
    wn--; w = NextWord();
    e = CANTSEE_PE;
    if (w == pronoun_word) {
        pronoun__word = pronoun_word; pronoun__obj = pronoun_obj;
        e = ITGONE_PE;
    }
    if (etype > e) return etype;
    return e;
];
```

§**44. Multiple Object List.** The `MultiAdd` routine adds object `o` to the multiple-object-list. This is only allowed to hold `MATCH_LIST_WORDS` minus one objects at most, at which point it ignores any new entries (and sets a global flag so that a warning may later be printed if need be).

The `MultiSub` routine deletes object `o` from the multiple-object-list. It returns 0 if the object was there in the first place, and 9 (because this is the appropriate error number in `Parser()`) if it wasn't.

The `MultiFilter` routine goes through the multiple-object-list and throws out anything without the given attribute `attr` set.

```
[ MultiAdd o i j;
    i = multiple_object-->0;
    if (i == MATCH_LIST_WORDS-1) { toomany_flag = 1; rtrue; }
    for (j=1 : j<=i : j++)
        if (o == multiple_object-->j) rtrue;
    i++;
    multiple_object-->i = o;
    multiple_object-->0 = i;
];
[ MultiSub o i j k;
    i = multiple_object-->0;
    for (j=1 : j<=i : j++)
        if (o == multiple_object-->j) {
            for (k=j : k<=i : k++) multiple_object-->k = multiple_object-->(k+1);
            multiple_object-->0 = --i;
            return 0;
        }
    return VAGUE_PE;
];
[ MultiFilter attr  i j o;
```

```
    .MFiltl;
    i = multiple_object-->0;
    for (j=1 : j<=i : j++) {
        o = multiple_object-->j;
        if (o hasnt attr) { MultiSub(o); jump Mfiltl; }
    }
];
```

§**45. Scope.** The scope of an actor is the set of objects which he can refer to in typed commands, which is normally the same as the set of visible objects; but this can be modified. This is how I7 handles tokens like "[any room]".

Scope determination is done by calling `SearchScope` to iterate through the objects in scope, and "visit" each one: which means, carry out some task for each as we get there. The task depends on the current value of `scope_reason`, which is `PARSING_REASON` when the parser is matching command text against object names.

The scope machinery is built on a number of levels, each making use only of lower levels:

(0) Either `NounDomain`, `TestScope` or `LoopOverScope` makes one or more calls to `SearchScope` (on level 1). The point of making multiple calls is to influence the order in which items in scope are visited, which improves the quality of "take all"-style multiple object lists, for instance.

(1) `SearchScope` searches for the objects in scope which are within first one domain, and then another: for instance, first within the current room but not within the current actor, and then within the current actor. It can be called either from level 0, or externally from the choose-objects machinery, but is not recursive. It works within the context of a given token in the parser (when called for `PARSING_REASON`) and in particular the `multiinside` token, and also handles testing commands, scope tokens, scope in darkness, and intervention by the I7 "deciding the scope of" activity. Most of its actual searches are delegated to `ScopeWithin` (level 2), but it also uses `DoScopeActionAndRecurse` (level 3) and `DoScopeAction` (level 4) as necessary.

(2) `ScopeWithin` iterates through the objects in scope which are within one supplied domain, but not within another. It can be called either from level 1, or independently from rules in the "deciding the scope of" activity via the I7 "place the contents of X in scope" phrase. It calls `DoScopeActionAndRecurse` (level 3) on any unconcealed objects it finds.

(3) `DoScopeActionAndRecurse` visits a given object by calling down to `DoScopeAction` (level 4), and recurses to all unconcealed object-tree contents and component parts of the object. The I7 phrase "place X in scope" uses this routine.

(4) `DoScopeAction` simply visits a single object, taking whatever action is needed there – which will depend on the `scope_reason`. The only use made by the parser of `TryGivenObject`, which tries to match command text against the name of a given object, is from here. The I7 phrase "place X in scope, but not its contents" uses this routine.

Two routines are provided for code external to the parser to modify the scope. They should be called only during scope deliberations – i.e., in `scope=...` tokens or in rules for the "deciding the scope of" activity. (At present, `AddToScope` is not used in I7 at all.) Note that this I7 form of `PlaceInScope` has a slightly different specification to its I6 library counterpart of the same name: it can place a room in scope. (In I6, room names were not normally parsed.)

```
[ PlaceInScope O opts; ! If opts is set, do not place contents in scope
    wn = match_from;
    if (opts == false) DoScopeActionAndRecurse(O);
    else DoScopeAction(O);
    return;
];
[ AddToScope obj;
    if (ats_flag >= 2) DoScopeActionAndRecurse(obj, 0, ats_flag-2);
    if (ats_flag == 1) { if (HasLightSource(obj)==1) ats_hls = 1; }
```

```
];
```

§**46. Scope Level 0.** The two ways of starting up the scope machinery other than via the parser code above.

```
[ TestScope obj act a al sr x y;
    x = parser_one; y = parser_two;
    parser_one = obj; parser_two = 0; a = actor; al = actors_location;
    sr = scope_reason; scope_reason = TESTSCOPE_REASON;
    if (act == 0) actor = player; else actor = act;
    actors_location = ScopeCeiling(actor);
    SearchScope(actors_location, actor, 0); scope_reason = sr; actor = a;
    actors_location = al; parser_one = x; x = parser_two; parser_two = y;
    return x;
];
[ LoopOverScope routine act x y a al;
    x = parser_one; y = scope_reason; a = actor; al = actors_location;
    parser_one = routine;
    if (act == 0) actor = player; else actor = act;
    actors_location = ScopeCeiling(actor);
    scope_reason = LOOPOVERSCOPE_REASON;
    SearchScope(actors_location, actor, 0);
    parser_one = x; scope_reason = y; actor = a; actors_location = al;
];
```

§**47. SearchScope.** Level 1. The method is:

(a) If the context is a `scope=...` token, then the search is delegated to "stage 2" of the scope routine. This was the old I6 way to override the searching behaviour: while users probably won't be using it any more, the template does, in order to give testing commands universal scope which is exempt from the activity below; and the NI compiler creates `scope=...` tokens to handle Understand grammar such as "[any room]". So the feature remains very much still in use.

(b) The "deciding the scope of" activity is given the chance to intervene. This is the I7 way to override the searching behaviour, and is the one taken by users.

(c) And otherwise:

(1) The I6 `multiinside` token, used as the first noun of its grammar line, has as its scope all of the objects which are inside or on top of the *second* noun of the grammar line. This provides a neat scope for the ALL in a command like GET ALL FROM CUPBOARD, where the player clearly does not intend ALL to refer to the cupboard itself, for instance. The difficulty is that we don't yet know what the second object is, if we are parsing left to right. But the parser code above has taken care of all of that, and the `advance_warning` global is set to the object number of the second noun, or to -1 if that is not yet known. Note that we check that the contents are visible before adding them to scope, because otherwise an unscrupulous player could use such a command to detect the contents of an opaque locked box. If this rule applies, we skip (c.2), (c.3) and (c.4).

(2) For all other tokens except `creature`, searching scope for the room holding the current actor always catches the compass directions unless a definite article has already been typed. (Thus OPEN THE EAST would match an object called "east door", but not the compass direction "east".)

(3) The contents of `domain1` which are not contents of `domain2` are placed in scope, and so are any component parts of `domain1`. If `domain1` is a container or supporter, it is placed in scope itself.

(4) The contents and component parts of `domain2` are placed in scope. If `domain2` is a container or supporter, it is placed in scope itself.

(5) In darkness, the actor and his component parts are in scope. If the actor is inside or on top of something, then that thing is also in scope. (This avoids a situation where the player gets into an

opaque box, then pulls it closed from the inside, plunging himself into darkness, then types OPEN BOX only to be told that he can't see any such thing.)

```
[ SearchScope domain1 domain2 context i;
    if (domain1 == 0) return;
    ! (a)
    if (scope_token) {
        scope_stage = 2;
        #Ifdef DEBUG;
        if (parser_trace >= 3) print "  [Scope routine called at stage 2]^";
        #Endif;
        if (indirect(scope_token) ~= 0) rtrue;
    }
    ! (b)
    BeginActivity(DECIDING_SCOPE_ACT, actor);
    if (ForActivity(DECIDING_SCOPE_ACT, actor) == false) {
        ! (c.1)
        if ((scope_reason == PARSING_REASON) && (context == MULTIINSIDE_TOKEN) &&
            (advance_warning ~= -1)) {
            if (IsSeeThrough(advance_warning) == 1)
                ScopeWithin(advance_warning, 0, context);
        } else {
            ! (c.2)
            if ((scope_reason == PARSING_REASON) && (context ~= CREATURE_TOKEN) &&
                (indef_mode == 0) && (domain1 == actors_location))
                    ScopeWithin(compass);
            ! (c.3)
            if (domain1 has supporter or container) DoScopeAction(domain1);
            ScopeWithin(domain1, domain2, context);
            ! (c.4)
            if (domain2) {
                if (domain2 has supporter or container) DoScopeAction(domain2);
                ScopeWithin(domain2, 0, context);
            }
        }
        ! (c.5)
        if (thedark == domain1 or domain2) {
            DoScopeActionAndRecurse(actor, actor, context);
            if (parent(actor) has supporter or container)
                DoScopeActionAndRecurse(parent(actor), parent(actor), context);
        }
    }
    EndActivity(DECIDING_SCOPE_ACT, actor);
];
```

§**48. ScopeWithin.** Level 2. `ScopeWithin` puts objects visible from within the `domain` into scope. An item belonging to the `domain` is placed in scope unless it is being concealed by the `domain`: and even then, if the `domain` is the current actor. Suppose Zorro conceals a book beneath his cloak: then the book is not in scope to his lady friend The Black Whip, but it is in scope to Zorro himself. (Thus an actor is not allowed to conceal anything from himself.)

Note that the `domain` object itself, and its component parts if any, are not placed in scope by this routine, though nothing prevents some other code doing so.

```
[ ScopeWithin domain nosearch context obj next_obj;
    if (domain == 0) rtrue;
    ! Look through the objects in the domain, avoiding "objectloop" in case
    ! movements occur.
    obj = child(domain);
    while (obj) {
        next_obj = sibling(obj);
        if ((domain == actor) || (TestConcealment(domain, obj) == false))
            DoScopeActionAndRecurse(obj, nosearch, context);
        obj = next_obj;
    }
];
```

§**49. DoScopeActionAndRecurse.** Level 3. In all cases, the `domain` itself is visited. There are then three possible forms of recursion:

(a) To unconcealed objects which are inside, on top of, carried or worn by the `domain`: this is called "searching" in traditional I6 language and is suppressed if `domain` is the special value `nosearch`.
(b) To unconcealed component parts of the `domain`.
(c) To any other objects listed in the `add_to_scope` property array, or supplied by the `add_to_scope` property routine, if it has one. (I7 does not usually use `add_to_scope`, but it remains a useful hook in the parser, so it retains its old I6 library interpretation.)

```
[ DoScopeActionAndRecurse domain nosearch context i ad n obj next_obj;
    DoScopeAction(domain);
    ! (a)
    if ((domain ~= nosearch) &&
        ((domain ofclass K1_room or K8_person) || (IsSeeThrough(domain) == 1))) {
        obj = child(domain);
        while (obj) {
            next_obj = sibling(obj);
            if ((domain == actor) || (TestConcealment(domain, obj) == false))
                DoScopeActionAndRecurse(obj, nosearch, context);
            obj = next_obj;
        }
    }
    ! (b)
    if (domain provides component_child) {
        obj = domain.component_child;
        while (obj) {
            next_obj = obj.component_sibling;
            if ((domain == actor) || (TestConcealment(domain, obj) == false))
                DoScopeActionAndRecurse(obj, 0, context);
            obj = next_obj;
        }
```

```
    }
    ! (c)
    ad = domain.&add_to_scope;
    if (ad ~= 0) {
        ! Test if the property value is not an object.
        #Ifdef TARGET_ZCODE;
        i = (UnsignedCompare(ad-->0, top_object) > 0);
        #Ifnot; ! TARGET_GLULX
        i = (((ad-->0)->0) ~= $70);
        #Endif; ! TARGET_
        if (i) {
            ats_flag = 2+context;
            RunRoutines(domain, add_to_scope);
            ats_flag = 0;
        }
        else {
            n = domain.#add_to_scope;
            for (i=0 : (WORDSIZE*i)<n : i++)
                if (ad-->i)
                    DoScopeActionAndRecurse(ad-->i, 0, context);
        }
    }
];
```

§**50. DoScopeAction.** Level 4. This is where we take whatever action is to be performed as the "visit" to each scoped object, and it's the bottom at last of the scope mechanism.

```
[ DoScopeAction item;
    #Ifdef DEBUG;
    if (parser_trace >= 6)
        print "[DSA on ", (the) item, " with reason = ", scope_reason,
            " p1 = ", parser_one, " p2 = ", parser_two, "]^";
    #Endif; ! DEBUG
    @push parser_one; @push scope_reason;
    switch(scope_reason) {
        TESTSCOPE_REASON: if (item == parser_one) parser_two = 1;
        LOOPOVERSCOPE_REASON: if (parser_one ofclass Routine) indirect(parser_one, item);
        PARSING_REASON, TALKING_REASON: MatchTextAgainstObject(item);
    }
    @pull scope_reason; @pull parser_one;
];
```

§**51. Parsing Object Names.** We now reach the final major block of code in the parser: the part which tries to match a given object's name(s) against the text at word position `match_from` in the player's command, and calls `MakeMatch` if it succeeds. There are basically four possibilities: ME, a pronoun such as IT, a name which doesn't begin misleadingly with a number, and a name which does. In the latter two cases, we pass the job down to `TryGivenObject`.

```
[ MatchTextAgainstObject item i;
    if (match_from <= num_words) { ! If there's any text to match, that is
        wn = match_from;
        i = NounWord();
        if ((i == 1) && (player == item)) MakeMatch(item, 1); ! "me"
        if ((i >= 2) && (i < 128) && (LanguagePronouns-->i == item)) MakeMatch(item, 1);
    }

    ! Construing the current word as the start of a noun, can it refer to the
    ! object?

    wn = match_from;
    if (TryGivenObject(item) > 0)
        if (indef_nspec_at > 0 && match_from ~= indef_nspec_at) {
            ! This case arises if the player has typed a number in
            ! which is hypothetically an indefinite descriptor:
            ! e.g. "take two clubs".  We have just checked the object
            ! against the word "clubs", in the hope of eventually finding
            ! two such objects.  But we also backtrack and check it
            ! against the words "two clubs", in case it turns out to
            ! be the 2 of Clubs from a pack of cards, say.  If it does
            ! match against "two clubs", we tear up our original
            ! assumption about the meaning of "two" and lapse back into
            ! definite mode.

            wn = indef_nspec_at;
            if (TryGivenObject(item) > 0) {
                match_from = indef_nspec_at;
                ResetDescriptors();
            }
            wn = match_from;
        }
];
```

§**52. TryGivenObject.** `TryGivenObject` tries to match as many words as possible in what has been typed to the given object, `obj`. If it manages any words matched at all, it calls `MakeMatch` to say so, then returns the number of words (or 1 if it was a match because of inadequate input).

```
[ TryGivenObject obj nomatch threshold k w j;
    #Ifdef DEBUG;
    if (parser_trace >= 5) print "    Trying ", (the) obj, " (", obj, ") at word ", wn, "^";
    #Endif; ! DEBUG
    if (nomatch && obj == 0) return 0;
! if (nomatch) print "*** TryGivenObject *** on ", (the) obj, " at wn = ", wn, "^";
    dict_flags_of_noun = 0;
!  If input has run out then always match, with only quality 0 (this saves
!  time).
    if (wn > num_words) {
        if (nomatch) return 0;
        if (indef_mode ~= 0)
            dict_flags_of_noun = $$01110000;  ! Reject "plural" bit
        MakeMatch(obj,0);
        #Ifdef DEBUG;
        if (parser_trace >= 5) print "    Matched (0)^";
        #Endif; ! DEBUG
        return 1;
    }
!  Ask the object to parse itself if necessary, sitting up and taking notice
!  if it says the plural was used:
    if (obj.parse_name~=0) {
        parser_action = NULL; j=wn;
        k = RunRoutines(obj,parse_name);
        if (k > 0) {
            wn=j+k;
        .MMbyPN;
            if (parser_action == ##PluralFound)
                dict_flags_of_noun = dict_flags_of_noun | 4;
            if (dict_flags_of_noun & 4) {
                if (~~allow_plurals) k = 0;
                else {
                    if (indef_mode == 0) {
                        indef_mode = 1; indef_type = 0; indef_wanted = 0;
                    }
                    indef_type = indef_type | PLURAL_BIT;
                    if (indef_wanted == 0) indef_wanted = INDEF_ALL_WANTED;
                }
            }
            #Ifdef DEBUG;
            if (parser_trace >= 5) print "    Matched (", k, ")^";
            #Endif; ! DEBUG
            if (nomatch == false) MakeMatch(obj,k);
            return k;
        }
        if (k == 0) jump NoWordsMatch;
    }
```

```
    ! The default algorithm is simply to count up how many words pass the
    ! Refers test:
    parser_action = NULL;
    w = NounWord();
    if (w == 1 && player == obj) { k=1; jump MMbyPN; }
    if (w >= 2 && w < 128 && (LanguagePronouns-->w == obj)) { k = 1; jump MMbyPN; }
    if (Refers(obj, wn-1) == 0) {
        .NoWordsMatch;
        if (indef_mode ~= 0) { k = 0; parser_action = NULL; jump MMbyPN; }
        rfalse;
    }
    threshold = 1;
    dict_flags_of_noun = (w->#dict_par1) & $$01110100;
    w = NextWord();
    while (Refers(obj, wn-1)) {
        threshold++;
        if (w)
        dict_flags_of_noun = dict_flags_of_noun | ((w->#dict_par1) & $$01110100);
        w = NextWord();
    }
    k = threshold;
    jump MMbyPN;
];
```

§**53. Refers.** `Refers` works out whether the word at number wnum can refer to the object `obj`, returning true or false. The standard method is to see if the word is listed under the `name` property for the object, but this is more complex in languages other than English.

```
[ Refers obj wnum   wd k l m;
    if (obj == 0) rfalse;
    #Ifdef LanguageRefers;
    k = LanguageRefers(obj,wnum); if (k >= 0) return k;
    #Endif; ! LanguageRefers
    k = wn; wn = wnum; wd = NextWordStopped(); wn = k;
    if (parser_inflection >= 256) {
        k = indirect(parser_inflection, obj, wd);
        if (k >= 0) return k;
        m = -k;
    }
    else
        m = parser_inflection;
    k = obj.&m; l = (obj.#m)/WORDSIZE-1;
    for (m=0 : m<=l : m++)
        if (wd == k-->m) rtrue;
    rfalse;
];
[ WordInProperty wd obj prop k l m;
    k = obj.&prop; l = (obj.#prop)/WORDSIZE-1;
    for (m=0 : m<=l : m++)
        if (wd == k-->m) rtrue;
```

```
    rfalse;
];
```

§**54. NounWord.** `NounWord` (which takes no arguments) returns:
(a) 0 if the next word is not in the dictionary or is but does not carry the "noun" bit in its dictionary entry,
(b) 1 if it is a word meaning "me",
(c) the index in the pronoun table (plus 2) of the value field of a pronoun, if it is a pronoun,
(d) the address in the dictionary if it is a recognised noun.

```
[ NounWord i j s;
    i = NextWord();
    if (i == 0) rfalse;
    if (i == ME1__WD or ME2__WD or ME3__WD) return 1;
    s = LanguagePronouns-->0;
    for (j=1 : j<=s : j=j+3)
        if (i == LanguagePronouns-->j)
            return j+2;
    if ((i->#dict_par1)&128 == 0) rfalse;
    return i;
];
```

§**55. TryNumber.** `TryNumber` takes word number `wordnum` and tries to parse it as an (unsigned) decimal number or the name of a small number, returning

(a) −1000 if it is not a number
(b) the number, if it has between 1 and 4 digits
(c) 10000 if it has 5 or more digits.

(The danger of allowing 5 digits is that Z-machine integers are only 16 bits long, and anyway this routine isn't meant to be perfect: it only really needs to be good enough to handle numeric descriptors such as those in TAKE 31 COINS or DROP FOUR DAGGERS. In particular, it is not the way I7 "[number]" tokens are parsed.)

```
[ TryNumber wordnum   i j c num len mul tot d digit;
    i = wn; wn = wordnum; j = NextWord(); wn = i;
    j = NumberWord(j); ! Test for verbal forms ONE to TWENTY
    if (j >= 1) return j;
    #Ifdef TARGET_ZCODE;
    i = wordnum*4+1; j = parse->i; num = j+buffer; len = parse->(i-1);
    #Ifnot; ! TARGET_GLULX
    i = wordnum*3; j = parse-->i; num = j+buffer; len = parse-->(i-1);
    #Endif; ! TARGET_
    if (len >= 4) mul=1000;
    if (len == 3) mul=100;
    if (len == 2) mul=10;
    if (len == 1) mul=1;
    tot = 0; c = 0; len = len-1;
    for (c=0 : c<=len : c++) {
        digit=num->c;
        if (digit == '0') { d = 0; jump digok; }
        if (digit == '1') { d = 1; jump digok; }
        if (digit == '2') { d = 2; jump digok; }
        if (digit == '3') { d = 3; jump digok; }
```

```
        if (digit == '4') { d = 4; jump digok; }
        if (digit == '5') { d = 5; jump digok; }
        if (digit == '6') { d = 6; jump digok; }
        if (digit == '7') { d = 7; jump digok; }
        if (digit == '8') { d = 8; jump digok; }
        if (digit == '9') { d = 9; jump digok; }
        return -1000;
    .digok;
        tot = tot+mul*d; mul = mul/10;
    }
    if (len > 3) tot=10000;
    return tot;
];
```

§**56. Extended TryNumber.** The same, but recognising verbal forms up to 30.

```
[ I7_ExtendedTryNumber wordnum i j;
    i = wn; wn = wordnum; j = NextWordStopped(); wn = i;
    switch (j) {
        'twenty-one': return 21;
        'twenty-two': return 22;
        'twenty-three': return 23;
        'twenty-four': return 24;
        'twenty-five': return 25;
        'twenty-six': return 26;
        'twenty-seven': return 27;
        'twenty-eight': return 28;
        'twenty-nine': return 29;
        'thirty': return 30;
        default: return TryNumber(wordnum);
    }
];
```

§**57. Gender.** `GetGender` returns 0 if the given animate object is female, and 1 if male, and is abstracted as a routine in case something more elaborate is ever needed.

For GNAs – gender/noun/animation combinations – see the *Inform Designer's Manual*, 4th edition.

```
[ GetGender person;
    if (person hasnt female) rtrue;
    rfalse;
];
[ GetGNAOfObject obj case gender;
    if (obj hasnt animate) case = 6;
    if (obj has male) gender = male;
    if (obj has female) gender = female;
    if (obj has neuter) gender = neuter;
    if (gender == 0) {
        if (case == 0) gender = LanguageAnimateGender;
        else gender = LanguageInanimateGender;
    }
    if (gender == female)   case = case + 1;
    if (gender == neuter)   case = case + 2;
```

```
    if (obj has pluralname) case = case + 3;
    return case;
];
```

§58. Noticing Plurals.

```
[ DetectPluralWord at n i w swn outcome;
    swn = wn; wn = at;
    for (i=0:i<n:i++) {
        w = NextWordStopped();
        if (w == 0 or THEN1__WD or COMMA_WORD or -1) break;
        if ((w->#dict_par1) & $$00000100) {
            parser_action = ##PluralFound;
            outcome = true;
        }
    }
    wn = swn;
    return outcome;
];
```

§59. Pronoun Handling.

```
[ SetPronoun dword value x;
    for (x=1 : x<=LanguagePronouns-->0 : x=x+3)
        if (LanguagePronouns-->x == dword) {
            LanguagePronouns-->(x+2) = value; return;
        }
    RunTimeError(14);
];
[ PronounValue dword x;
    for (x=1 : x<=LanguagePronouns-->0 : x=x+3)
        if (LanguagePronouns-->x == dword)
            return LanguagePronouns-->(x+2);
    return 0;
];
[ ResetVagueWords obj; PronounNotice(obj); ];
[ PronounNotice obj x bm;
    if (obj == player) return;
    bm = PowersOfTwo_TB-->(GetGNAOfObject(obj));
    for (x=1 : x<=LanguagePronouns-->0 : x=x+3)
        if (bm & (LanguagePronouns-->(x+1)) ~= 0)
            LanguagePronouns-->(x+2) = obj;
];
[ PronounNoticeHeldObjects x;
#IFNDEF MANUAL_PRONOUNS;
    objectloop(x in player) PronounNotice(x);
#ENDIF;
    x = 0; ! To prevent a "not used" error
    rfalse;
];
```

§60. Yes/No Questions.

```
[ YesOrNo i j;
    for (::) {
        #Ifdef TARGET_ZCODE;
        if (location == nothing || parent(player) == nothing) read buffer parse;
        else read buffer parse DrawStatusLine;
        j = parse->1;
        #Ifnot; ! TARGET_GLULX;
        KeyboardPrimitive(buffer, parse);
        j = parse-->0;
        #Endif; ! TARGET_
        if (j) { ! at least one word entered
            i = parse-->1;
            if (i == YES1__WD or YES2__WD or YES3__WD) rtrue;
            if (i == NO1__WD or NO2__WD or NO3__WD) rfalse;
        }
        L__M(##Quit, 1); print "> ";
    }
];
```

§61. Number Words. Not much of a parsing routine: we look through an array of pairs of number words (single words) and their numeric equivalents.

```
[ NumberWord o i n;
    n = LanguageNumbers-->0;
    for (i=1 : i<=n : i=i+2)
        if (o == LanguageNumbers-->i) return LanguageNumbers-->(i+1);
    return 0;
];
```

§62. Choose Objects. This material, the final body of code in the parser, is an I7 addition. The I6 parser leaves it to the user to provide a `ChooseObjects` routine to decide between possibilities when the situation is ambiguous. For I7 use, we provide a `ChooseObjects` which essentially runs the "does the player mean" rulebook to decide, though this is not obvious from the code below because it is hidden in the `CheckDPMR` routine – which is defined in the Standard Rules, not here.

```
!Constant COBJ_DEBUG;

! the highest value returned by CheckDPMR (see the Standard Rules)
Constant HIGHEST_DPMR_SCORE = 4;

Array alt_match_list --> (MATCH_LIST_WORDS+1);

#ifdef TARGET_GLULX;
[ COBJ__Copy words from to  i;
    for (i=0: i<words: i++)
        to-->i = from-->i;
];
#ifnot;
[ COBJ__Copy words from to  bytes;
    bytes = words * 2;
    @copy_table from to bytes;
];
```

```
#endif;
! swap alt_match_list with match_list/number_matched
[ COBJ__SwapMatches i x;
    ! swap the counts
    x = number_matched;
    number_matched = alt_match_list-->0;
    alt_match_list-->0 = x;
    ! swap the values
    if (x < number_matched) x = number_matched;
    for (i=x: i>0: i--) {
        x = match_list-->(i-1);
        match_list-->(i-1) = alt_match_list-->i;
        alt_match_list-->i = x;
    }
];
[ ChooseObjects obj code  l i swn spcount;
    if (code<2) rfalse;
    if (cobj_flag == 1) {
        .CodeOne;
        if (parameters > 0) {
            #ifdef COBJ_DEBUG;
            print "[scoring ", (the) obj, " (second)]^";
            #endif;
            return ScoreDabCombo(parser_results-->2, obj);
        } else {
            #ifdef COBJ_DEBUG;
            print "[scoring ", (the) obj, " (first) in ",
                alt_match_list-->0, " combinations]^";
            #endif;
            l = 0;
            for (i=1: i<=alt_match_list-->0: i++) {
                spcount = ScoreDabCombo(obj, alt_match_list-->i);
                if (spcount == HIGHEST_DPMR_SCORE) {
                    #ifdef COBJ_DEBUG;
                    print "[scored ", spcount, " - best possible]^";
                    #endif;
                    return spcount;
                }
                if (spcount>l) l = spcount;
            }
            return l;
        }
    }
    if (cobj_flag == 2) {
        .CodeTwo;
        #ifdef COBJ_DEBUG;
        print "[scoring ", (the) obj, " (simple); parameters = ", parameters,
            " aw = ", advance_warning, "]^";
        #endif;
        @push action_to_be;
        if (parameters==0) {
            if (advance_warning > 0)
                l = ScoreDabCombo(obj, advance_warning);
```

```
            else
                l = ScoreDabCombo(obj, 0);
        } else {
            l = ScoreDabCombo(parser_results-->2, obj);
        }
        @pull action_to_be;
        return l;
    }
    #ifdef COBJ_DEBUG;
    print "[choosing a cobj strategy: ";
    #endif;
    swn = wn;
    spcount = pcount;
    while (line_ttype-->pcount == PREPOSITION_TT) pcount++;
    if (line_ttype-->pcount == ELEMENTARY_TT) {
        while (wn <= num_words) {
            l = NextWordStopped(); wn--;
            if ( (l ~= -1 or 0) && (l->#dict_par1) &8 ) { wn++; continue; } ! if preposition
            if (l == ALL1__WD or ALL2__WD or ALL3__WD or ALL4__WD or ALL5__WD) { wn++; continue; }
            SafeSkipDescriptors();
            ! save the current match state
            @push match_length; @push token_filter; @push match_from;
            alt_match_list-->0 = number_matched;
            COBJ__Copy(number_matched, match_list, alt_match_list+WORDSIZE);
            ! now get all the matches for the second noun
            match_length = 0; number_matched = 0; match_from = wn;
            token_filter = 0;
            SearchScope(actor, actors_location, line_tdata-->pcount);
            #ifdef COBJ_DEBUG;
            print number_matched, " possible second nouns]^";
            #endif;
            wn = swn;
            cobj_flag = 1;
            ! restore match variables
            COBJ__SwapMatches();
            @pull match_from; @pull token_filter; @pull match_length;
            pcount = spcount;
            jump CodeOne;
        }
    }
    pcount = spcount;
    wn = swn;
    #ifdef COBJ_DEBUG;
    print "nothing interesting]^";
    #endif;
    cobj_flag = 2;
    jump CodeTwo;
];
[ ScoreDabCombo a b  result;
    @push action; @push act_requester; @push noun; @push second;
    action = action_to_be;
    act_requester = player;
    if (action_reversed) { noun = b; second = a; }
```

```
    else { noun = a; second = b; }
    result = CheckDPMR();
    @pull second; @pull noun; @pull act_requester; @pull action;
    #ifdef COBJ_DEBUG;
    print "[", (the) a, " / ", (the) b, " => ", result, "]^";
    #endif;
    return result;
];
```

ListWriter Template B/lwt

Purpose

A flexible object-lister taking care of plurals, inventory information, various formats and so on.

---

§**1. Specification.** The list-writer is called by one of the following function calls:

(1) `WriteListOfMarkedObjects(style)`, where the set of objects listed is understood to be exactly those with the `workflag2` attribute set, and
   (a) the `style` is a sum of `*_BIT` constants as defined in "Definitions.i6t".

(2) `WriteListFrom(obj, style, depth, noactivity, iterator)`, where only the first two parameters are compulsory:
   (a) the set of objects listed begins with `obj`;
   (b) the `style` is a sum of `*_BIT` constants as defined in "Definitions.i6t";
   (c) the `depth` is the recursion depth within the list – ordinarily 0, but by supplying a higher value, we can simulate a sublist of another list;
   (d) `noactivity` is a flag which forces the list-writer to ignore the "listing the contents of" activity (in cases where it would otherwise consult this): by default this is `false`;
   (e) `iterator` is an iterator function which provides the objects in sequence.

`WriteListOfMarkedObjects` is simply a front-end which supplies suitable parameters for `WriteListFrom`.

The iterator function is by default `ObjectTreeIterator`. This defines the sequence of objects as being the children of the parent of `obj`, in object tree sequence (that is: `child(parent(obj))` is first). Moreover, when using `ObjectTreeIterator`, the "listing the contents of" activity is carried out, unless `noactivity` is set.

We also provide the iterator function `MarkedListIterator`, which defines the sequence of objects as being the list in the word array `MarkedObjectArray` with length `MarkedObjectLength`. Here the "listing the contents of" activity is never used, since the objects are not necessarily the contents of any single thing. This of course is the iterator used by `WriteListOfMarkedObjects(style)`: it works by filling this array with all the objects having `workflag2` set.

The full specification for iterator functions is given below.

The list-writer automatically groups adjacent and indistinguishable terms in the sequence into plurals, and carries out the "printing the plural name" activity to handle these. Doing this alone would result in text such as "You can see a cake, three coins, an onion, and two coins here", where the five coins are mentioned in two clauses because they happen not to be adjacent in the list. The list-writer therefore carries out the activity "grouping together" to see if the user would like to tie certain objects together into a single entry: what this does is to set suitable `list_together` properties for the objects. NI has already given plural objects a similar `list_together` property. The net effect is that any entries in the list with a non-zero value of `list_together` must be adjacent to each other.

We could achieve that by sorting the list in order of `list_together` value, but that would result in drastic movements, whereas we want to upset the original ordering as little as possible. So instead we use a process called *coalescing* the list. This is such that for every value $L \neq 0$ of `list_together`, every entry with that value is moved back in the list to follow the first entry with that value. Thus if the original order is $x_1, x_2, ..., x_N$ then x_j still precedes x_k in coalesced order unless there exists $i < j < k$ such that $L(i) = L(k) \neq 0$ and $L(j) \neq L(i)$. This is as stable as it can be while achieving the "interval" property that non-zero L values occur in contiguous runs.

We therefore obtain text such as "You can see a cake, five coins, the tiles W, X, Y and Z from a Scrabble set, and an onion here." where the coins and the Scrabble tiles have been coalesced together in the list.

It's important to note that the default `ObjectTreeIterator` has the side-effect of actually reordering the object tree: it rearranges the children being listed so that they appear in the tree in coalesced order. The `MarkedListIterator` used by `WriteListOfMarkedObjects` has the same side-effect if the marked objects all happen to share a common parent. It might seem odd for a list-writer to have side effects at all, but the idea is that occasional coalescing improves the quality of play in many small ways – for instance, the sequence of matches to TAKE ALL is tidier – and that coalescing has a small speed cost, so we want to do it as little as possible. (The latter was more of a consideration for I6: interpreters are faster nowadays.)

This specification is somewhat stronger than that of the I6 library's traditional list-writer, because

(i) it supports arbitrary lists of objects, not just children of specific parents, while still allowing coalesced and grouped lists,
(ii) it can be used recursively in all cases,
(iii) it uses its own memory, rather than borrowing memory from the parser, so that it can safely be used while the parser is working, and
(iv) it manages this memory more flexibly and without silently failing by buffer overruns on unexpectedly large lists.

The I7 version of `WriteListFrom`, when using `ObjectTreeIterator`, differs from the I6 version in that the object `o` is required to be the `child` of its `parent`, that is, to be the eldest child. (So in effect it's a function of parents, not children, but we retain the form for familiarity's sake.) In practice this was invariably the way `WriteListFrom` was used even in I6 days.

Finally, the `ISARE_BIT` is differently interpreted in I7: instead of printing something like " are ducks and drakes", as it would have done in I6, the initial space is now suppressed and we instead print "are ducks and drakes".

§**2. Memory.** The list-writer needs to dynamically allocate temporary array space of a known size, in such a way that the array is effectively on the local stack frame: if only either the Z-machine or Glulx supported a stack in memory, this would be no problem, but they do not and we must therefore use the following.

The size of the stack is such that it can support a list which includes every object and recurses in turn to most other objects: in practice, this never happens. It would be nice to allocate more just in case, but the Z-machine is desperately short of array memory.

```
Constant REQUISITION_STACK_SIZE = 3*{-value:NUMBER_CREATED(world_object)};
Array requisition_stack --> REQUISITION_STACK_SIZE;
Global requisition_stack_pointer = 0;

[ RequisitionStack len top addr;
    top = requisition_stack_pointer + len;
    if (top > REQUISITION_STACK_SIZE) return false;
    addr = requisition_stack + requisition_stack_pointer*WORDSIZE;
    ! print "Allocating ", addr, " at pointer ", requisition_stack_pointer, "^";
    requisition_stack_pointer = top;
    return addr;
];

[ FreeStack addr;
    if (addr == 0) return;
    requisition_stack_pointer = (addr - requisition_stack)/WORDSIZE;
];
```

§**3. WriteListOfMarkedObjects.** This routine will use the `MarkedListIterator`. That means it has to create an array containing the object numbers of every object with the `workflag2` attribute set, placing the address of this array in `MarkedObjectArray` and the length in `MarkedObjectLength`. Note that we preserve any existing marked list on the stack (using the assembly-language instructions `@push` and `@pull`) for the duration of our use.

While the final order of this list will depend on what it looks like after coalescing, the initial order is also important. If all of the objects have a common parent in the object tree, then we coalesce those objects and place the list in object tree order. But if not, we place the list in object number order (which is essentially the order of traversal of the initial state of the object tree: thus objects in Room A will all appear before objects in Room B if A was created before B in the source text).

```
Global MarkedObjectArray = 0;
Global MarkedObjectLength = 0;

[ WriteListOfMarkedObjects style
    obj common_parent first mixed_parentage length;

    objectloop (obj ofclass Object && obj has workflag2) {
        length++;
        if (first == nothing) { first = obj; common_parent = parent(obj); }
        else { if (parent(obj) ~= common_parent) mixed_parentage = true; }
    }
    if (mixed_parentage) common_parent = nothing;

    if (length == 0) print (string) NOTHING__TX;
    else {

        @push MarkedObjectArray; @push MarkedObjectLength;
        MarkedObjectArray = RequisitionStack(length);
        MarkedObjectLength = length;
        if (MarkedObjectArray == 0) return RunTimeProblem(RTP_LISTWRITERMEMORY);

        if (common_parent) {
            ObjectTreeCoalesce(child(common_parent));
            length = 0;
            objectloop (obj in common_parent) ! object tree order
                if (obj has workflag2) MarkedObjectArray-->length++ = obj;
        } else {
            length = 0;
            objectloop (obj ofclass Object) ! object number order
                if (obj has workflag2) MarkedObjectArray-->length++ = obj;
        }

        WriteListFrom(first, style, 0, false, MarkedListIterator);

        FreeStack(MarkedObjectArray);
        @pull MarkedObjectLength; @pull MarkedObjectArray;
    }
    return;
];
```

§**4. About Iterator Functions.** Suppose `Iter` is an iterator function and that we have a "raw list" $x_1, x_2, ..., x_n$ of objects. Of these, the iterator function will choose a sublist of "qualifying" objects. It is called with arguments

```
Iter(obj, depth, L, function)
```

where the `obj` is required to be x_j for some j and the function is one of the `*_ITF` constants defined below:

(a) On `START_ITF`, we return x_1, or `nothing` if the list is empty.

(b) On `SEEK_ITF`, we return the smallest $k \geq j$ such that x_k qualifies, or `nothing` if none of $x_j, x_{j+1}, ..., x_n$ qualify.

(c) On `ADVANCE_ITF`, we return the smallest $k > j$ such that x_k qualifies, or `nothing` if none of $x_{j+1}, x_{j+2}..., x_n$ qualify.

(d) On `COALESCE_ITF`, we coalesce the entire list (not merely the qualifying entries) and return the new x_1, or `nothing` if the list is empty.

Thus, given any x_i, we can produce the sublist of qualifying entries by performing `START_ITF` on x_i, then `SEEK_ITF`, to produce q_1; then `ADVANCE_ITF` to produce q_2, and so on until `nothing` is returned, when there are no more qualifying entries. `SEEK_ITF` and `ADVANCE_ITF` always return qualifying objects, or `nothing`; but `START_ITF` and `COALESCE_ITF` may return a non-qualifying object, since there's no reason why x_1 should qualify in any ordering.

The iterator can make its own choice of which entries qualify, and of what the raw list is; `depth` is supplied to help in that decision. But if `L` is non-zero then the iterator is required to disqualify any entry whose `list_together` value is not `L`.

```
Constant SEEK_ITF = 0;
Constant ADVANCE_ITF = 1;
Constant COALESCE_ITF = 2;
Constant START_ITF = 3;
! Constant DBLW; ! Uncomment this to provide debugging information at run-time
```

§**5. Marked List Iterator.** Here the raw list is provided by the `MarkedObjectArray`, which is convenient for coalescing, but not so helpful for translating the `obj` parameter into the i such that it is x_i. We simply search from the beginning to do this, which combined with other code using the iterator makes for some unnecessary $O(n^2)$ calculations. But it saves memory and nuisance, and the iterator is used only with printing to the screen, which never needs to be very rapid anyway (because the player can only read very slowly).

```
[ MarkedListIterator obj depth required_lt function i;
    if (obj == nothing) return nothing;
    switch(function) {
        START_ITF: return MarkedObjectArray-->0;
        COALESCE_ITF: return MarkedListCoalesce();
        SEEK_ITF, ADVANCE_ITF:
            for (i=0: i<MarkedObjectLength: i++)
                if (MarkedObjectArray-->i == obj) {
                    if (function == ADVANCE_ITF) i++;
                    for (:i<MarkedObjectLength: i++) {
                        obj = MarkedObjectArray-->i;
                        if ((required_lt) && (obj.list_together ~= required_lt)) continue;
                        if ((c_style & WORKFLAG_BIT) && (depth==0) && (obj hasnt workflag))
                            continue;
                        if ((c_style & CONCEAL_BIT) &&
                            ((obj has concealed) || (obj has scenery))) continue;
                        return obj;
```

```
                    }
                    return nothing;
                }
    }
    return nothing;
];
```

§**6. Coalesce Marked List.** The return value is the new first entry in the raw list.

```
[ MarkedListCoalesce o i lt l swap m;
    for (i=0: i<MarkedObjectLength: i++) {
        lt = (MarkedObjectArray-->i).list_together;
        if (lt ~= 0) {
            ! Find first object in list after contiguous run with this list_together value:
            for (i++: (i<MarkedObjectLength)&&((MarkedObjectArray-->i).list_together==lt): i++) ;
            ! If the contiguous run extends to end of list, the list is now perfect:
            if (i == MarkedObjectLength) return MarkedObjectArray-->0;
            ! And otherwise we look to see if any future entries belong in the earlier run:
            for (l=i+1: l<MarkedObjectLength: l++)
                if ((MarkedObjectArray-->l).list_together == lt) {
                    ! Yes, they do: so we perform a rotation to insert it before element i:
                    swap = MarkedObjectArray-->l;
                    for (m=l: m>i: m--) MarkedObjectArray-->m = MarkedObjectArray-->(m-1);
                    MarkedObjectArray-->i = swap;
                    ! And now the run is longer:
                    i++;
                    if (i == MarkedObjectLength) return MarkedObjectArray-->0;
                }
            i--;
        }
    }
    return MarkedObjectArray-->0;
];
```

§**7. Object Tree Iterator.** Here the raw list is the list of all children of a given parent: since the argument `obj` is required to be a member of the raw list, we can use `parent(obj)` to find it. Now seeking and advancing are fast, but coalescing is slower.

```
Global list_filter_routine;

[ ObjectTreeIterator obj depth required_lt function;
    if ((obj == nothing) || (parent(obj) == nothing)) return nothing;
    if (function == START_ITF) return child(parent(obj));
    if (function == COALESCE_ITF) return ObjectTreeCoalesce(obj);
    if (function == ADVANCE_ITF) obj = sibling(obj);
    for (:: obj = sibling(obj)) {
        if (obj == nothing) return nothing;
        if ((required_lt) && (obj.list_together ~= required_lt)) continue;
        if ((c_style & WORKFLAG_BIT) && (depth==0) && (obj hasnt workflag)) continue;
        if (obj hasnt list_filter_permits) continue;
        if ((c_style & CONCEAL_BIT) &&
            ((obj has concealed) || (obj has scenery))) continue;
        return obj;
```

```
    }
];
```

§**8. Coalesce Object Tree.** Again, the return value is the new first entry in the raw list.

```
[ ObjectTreeCoalesce obj memb lt later;
    #Ifdef DBLW; print "^^Sorting out: "; DiagnoseSortList(obj); #Endif;
    .StartAgain;
    for (memb=obj: memb~=nothing: memb=sibling(memb)) {
        lt = memb.list_together;
        if (lt ~= 0) {
            ! Find first object in list after contiguous run with this list_together value:
            for (memb=sibling(memb): (memb) && (memb.list_together == lt): memb = sibling(memb)) ;
            ! If the contiguous run extends to end of list, the list is now perfect:
            if (memb == 0) return obj;
            ! And otherwise we look to see if any future entries belong in the earlier run:
            for (later=sibling(memb): later: later=sibling(later))
                if (later.list_together == lt) {
                    ! Yes, they do: so we perform a regrouping of the list and start again:
                    obj = GroupChildren(parent(obj), list_together, lt);
                    #Ifdef DBLW; print "^^Sorted to: "; DiagnoseSortList(obj); #Endif;
                    jump StartAgain;
                }
        }
    }
    return obj;
];
#Ifdef DBLW;
[ DiagnoseSortList obj memb;
    for (memb=child(obj): memb~=nothing: memb=sibling(memb)) print memb, " --> "; new_line;
];
#Endif;
```

§**9. WriteListFrom.** And here we go at last. Or at any rate we initialise the quartet of global variables detailing the current list-writing process, and begin.

```
[ WriteListFrom first style depth noactivity iter a ol;
    @push c_iterator; @push c_style; @push c_depth; @push c_margin;
    if (iter) c_iterator = iter; else c_iterator = ObjectTreeIterator;
    c_style = style; c_depth = depth;
    c_margin = 0; if (style & EXTRAINDENT_BIT) c_margin = 1;
    objectloop (a ofclass Object) {
        give a list_filter_permits;
        if ((list_filter_routine) && (list_filter_routine(a) == false))
            give a ~list_filter_permits;
    }
    first = c_iterator(first, depth, 0, START_ITF);
    if (first == nothing) {
        print (string) NOTHING__TX;
        if (style & NEWLINE_BIT ~= 0) new_line;
    } else {
        if ((noactivity) || (iter)) {
```

```
            WriteListR(first, c_depth, true);
            say__p = 1;
        } else {
            objectloop (ol provides list_together) ol.list_together = 0;
            CarryOutActivity(LISTING_CONTENTS_ACT, parent(first));
        }
    }
    @pull c_margin; @pull c_depth; @pull c_style; @pull c_iterator;
];
```

§**10. Standard Contents Listing Rule.** The default for the listing contents activity is to call this rule in its "for" stage: note that this suppresses the use of the activity, to avoid an infinite regress. The activity is used only for the default `ObjectTreeIterator`, so there is no need to specify which is used.

```
[ STANDARD_CONTENTS_LISTING_R;
    WriteListFrom(child(parameter_object), c_style, c_depth, true);
];
```

§**11. Partitioning.** Given qualifying objects $x_1, ..., x_j$, we partition them into classes of the equivalence relation $x_i \sim x_j$ if and only

(i) they both have a `plural` property (not necessarily the same), and
(ii) neither will cause the list-maker to recurse downwards to show the objects inside or on top of them, and
(iii) if they cause the list-maker to add information about whether they are worn, lighted or open, then it will add the same information about both, and
(iv) they are considered identical by the parser, in that they respond to the same syntaxes of name: so that it is impossible to find a TAKE X command such that X matches one and not the other.

The equivalence classes are numbered consecutively upwards from 1 to n, in order of first appearance in the list. For each object x_i, `partition_classes->(i-1)` is the number of its equivalence class. For each equivalence class number c, `partition_class_sizes->c` is the number of objects in this class.

```
#Ifdef DBLW;
Global DBLW_no_classes; Global DBLW_no_objs;
[ DebugPartition partition_class_sizes partition_classes first depth i k o;
    print "[Length of list is ", DBLW_no_objs, " with ", k, " plural.]^";
    print "[Partitioned into ", DBLW_no_classes, " equivalence classes.]^";
    for (i=1: i<=DBLW_no_classes : i++) {
        print "Class ", i, " has size ", partition_class_sizes->i, "^";
    }
    for (k=0, o=first: k<DBLW_no_objs : k++, o = c_iterator(o, depth, lt_value, ADVANCE_ITF)) {
        print "Entry ", k, " has class ", partition_classes->k,
            " represented by ", o, " with L=", o.list_together, "^";
    }
];
#Endif;
```

§**12. Partition List.** The following creates the `partition_classes` and `partition_class_sizes` accordingly. We return n, the number of classes.

```
[ PartitionList first no_objs depth partition_classes partition_class_sizes
    i k l n m;
    for (i=0: i<no_objs: i++) partition_classes->i = 0;
    n = 1;
    for (i=first, k=0: k<no_objs: i=c_iterator(i, depth, lt_value, ADVANCE_ITF), k++)
        if (partition_classes->k == 0) {
            partition_classes->k = n; partition_class_sizes->n = 1;
            for (l=c_iterator(i, depth, lt_value, ADVANCE_ITF), m=k+1:
                (l~=0) && (m<no_objs):
                l=c_iterator(l, depth, lt_value, ADVANCE_ITF), m++) {
                if ((partition_classes->m == 0) && (ListEqual(i, l))) {
                    if (partition_class_sizes->n < 255) (partition_class_sizes->n)++;
                    partition_classes->m = n;
                }
            }
            if (n < 255) n++;
        }
    n--;
    #Ifdef DBLW;
    DBLW_no_classes = n; DBLW_no_objs = no_objs;
    DebugPartition(partition_class_sizes, partition_classes, first, depth);
    DebugPartition(partition_class_sizes, partition_classes, first, depth);
    DebugPartition(partition_class_sizes, partition_classes, first, depth);
    #Endif;
    return n;
];
```

§**13. Equivalence Relation.** The above algorithm will fail unless `ListEqual` is indeed reflexive, symmetric and transitive, which ultimately depends on the care with which `Identical` is implemented, which in turn hangs on the `parse_noun` properties compiled by NI. But this seems to be sound.

```
[ ListEqual o1 o2;
    if ((o1.plural == 0) || (o2.plural == 0)) rfalse;
    if (child(o1) ~= 0 && WillRecurs(o1) ~= 0) rfalse;
    if (child(o2) ~= 0 && WillRecurs(o2) ~= 0) rfalse;
    if (c_style & (FULLINV_BIT + PARTINV_BIT) ~= 0) {
        if ((o1 hasnt worn && o2 has worn) || (o2 hasnt worn && o1 has worn)) rfalse;
        if ((o1 hasnt light && o2 has light) || (o2 hasnt light && o1 has light)) rfalse;
        if (o1 has container) {
            if (o2 hasnt container) rfalse;
            if ((o1 has open && o2 hasnt open) || (o2 has open && o1 hasnt open))
                rfalse;
        }
        else if (o2 has container)
            rfalse;
    }
    return Identical(o1, o2);
];
[ WillRecurs o;
```

```
    if (c_style & ALWAYS_BIT ~= 0) rtrue;
    if (c_style & RECURSE_BIT == 0) rfalse;
    if ((o has supporter) || ((o has container) && (o has open or transparent))) rtrue;
    rfalse;
];
```

§**14. Grouping.** A "group" is a maximally-sized run of one or more adjacent partition classes in the list whose first members have a common value of `list_together` which is a routine or string, and which is not equal to `lt_value`, the current grouping value. (As we see below, it's by setting `lt_value` that we restrict attention to a particular group: if we reacted to that as a `list_together` value here, then we would simply go around in circles, and never be able to see the individual objects in the group.)

For instance, suppose we have objects with `list_together` values as follows, where R_1 and R_2 are addresses of routines:

`coin` (R_1), `coin` (R_1), `box` (R_2), `statuette` (0), `coin` (R_1), `box` (R_2)

Then the partition is 1, 1, 2, 3, 1, 2, so that the final output will be something like "three coins, two boxes and a statuette" – with three grouped terms. Here each partition class is the only member in its group, so the number of groups is the same as the number of classes. (The above list is not coalesced, so the R_1 values, for instance, are not contiguous: we need to work in general with non-coalesced lists because not every iterator function will be able to coalesce fully.)

But if we have something like this:

`coin` (R_1), `Q` (R_2), `W` (R_2), `coin` (R_1), `statuette` (0), `E` (R_2), `R` (R_2)

then the partition is 1, 2, 3, 1, 4, 5, 6 and we have six classes in all. But classes 2 and 3 are grouped together, as are classes 5 and 6, so we end up with a list having just four groups: "two coins, the letters Q and W from a Scrabble set, a statuette and the letters E and R from a Scrabble set". (Again, this list has not been fully coalesced: if it had been, it would be reordered `coin`, `coin`, `Q`, `W`, `E`, `R`, `statuette`, with partition 1, 1, 2, 3, 4, 5, 6, and three groups: "two coins, the letters Q, W, E and R from a Scrabble set and a statuette".)

The reason we do not group together classes with a common non-zero `list_together` which is *not* a routine or string is that low values of `list_together` are used in coalescing the list into a pleasing order (say, moving all of the key-like items together) but not in grouping: see `list_together` in the *Inform Designer's Manual*, 4th edition.

We calculate the number of groups by starting with the number of classes and then subtracting one each time two adjacent classes share `list_together` in the above sense.

```
[ NumberOfGroupsInList o no_classes depth partition_classes partition_class_sizes
    no_groups cl memb k current_lt lt;
    no_groups = no_classes;
    for (cl=1, memb=o, k=0: cl<=no_classes: cl++) {
        ! Advance to first member of class number cl
        while (partition_classes->k ~= cl) {
            k++; memb = c_iterator(memb, depth, lt_value, ADVANCE_ITF);
        }
        if (memb) { ! In case of accidents, but should always happen
            lt = memb.list_together;
            if ((lt ~= lt_value) && (lt ofclass Routine or String) && (lt == current_lt))
                no_groups--;
            current_lt = lt;
        }
    }
    #Ifdef DBLW; print "[There are ", no_groups, " groups.]^"; #Endif;
    return no_groups;
];
```

§**15. Write List Recursively.** The big one: `WriteListR` is the heart of the list-writer.

```
[ WriteListR o depth from_start
    partition_classes partition_class_sizes
    cl memb index k2 l m no_classes q groups_to_do current_lt;
    if (o == nothing) return; ! An empty list: no output
    if (from_start) {
        o = c_iterator(o, depth, 0, COALESCE_ITF); ! Coalesce list and choose new start
    }
    o = c_iterator(o, depth, 0, SEEK_ITF); ! Find first entry in list from o
    if (o == nothing) return;
    ! Count index = length of list
    for (memb=o, index=0: memb: memb=c_iterator(memb, depth, lt_value, ADVANCE_ITF)) index++;
    if (c_style & ISARE_BIT ~= 0) {
        if (index == 1 && o hasnt pluralname) print (string) IS3__TX;
        else                                  print (string) ARE3__TX;
        if (c_style & NEWLINE_BIT ~= 0)       print ":^";
        else                                  print (char) ' ';
        c_style = c_style - ISARE_BIT;
    }
    partition_classes = RequisitionStack(index/WORDSIZE + 2);
    partition_class_sizes = RequisitionStack(index/WORDSIZE + 2);
    if ((partition_classes == 0) || (partition_class_sizes == 0))
        return RunTimeProblem(RTP_LISTWRITERMEMORY);
    no_classes =
        PartitionList(o, index, depth, partition_classes, partition_class_sizes);
    groups_to_do =
        NumberOfGroupsInList(o, no_classes, depth, partition_classes, partition_class_sizes);
    for (cl=1, memb=o, index=0, current_lt=0: groups_to_do>0: cl++) {
        ! Set memb to first object of partition class cl
        while (partition_classes->index ~= cl) {
            index++; memb=c_iterator(memb, depth, lt_value, ADVANCE_ITF);
            if (memb==0) { print "*** Error in list-writer ***^"; break; }
        }
        #Ifdef DBLW;
        ! DebugPartition(partition_class_sizes, partition_classes, o, depth);
        print "^[Class ", cl, " of ", no_classes, ": first object ", memb,
            " (", memb.list_together, "); groups_to_do ", groups_to_do, ",
            current_lt=", current_lt, " listing_size=", listing_size, "]^";
        #Endif;
        if ((memb.list_together == lt_value) ||
            (~~(memb.list_together ofclass Routine or String))) current_lt = 0;
        else {
            if (memb.list_together == current_lt) continue;
            ! Otherwise this class begins a new group
            @push listing_size;
            q = memb; listing_size = 1; l = index; m = cl;
            while (m < no_classes && q.list_together == memb.list_together) {
                m++;
                while (partition_classes->l ~= m) {
                    l++; q = c_iterator(q, depth, lt_value, ADVANCE_ITF);
```

```
                }
                if (q.list_together == memb.list_together) listing_size++;
            }
            if (listing_size > 1) {
                ! The new group contains more than one partition class
                WriteMultiClassGroup(cl, memb, depth, partition_class_sizes);
                current_lt = memb.list_together;
                jump GroupComplete;
            }
            current_lt = 0;
            @pull listing_size;
        }
        WriteSingleClassGroup(cl, memb, depth, partition_class_sizes->cl);
        .GroupComplete;
        groups_to_do--;
        if (c_style & ENGLISH_BIT ~= 0) {
            if (groups_to_do == 1) {
                if (cl <= 1) print (string) LISTAND2__TX;
                else print (string) LISTAND__TX;
            }
            if (groups_to_do > 1) print (string) COMMA__TX;
        }
    }
    FreeStack(partition_class_sizes);
    FreeStack(partition_classes);
]; ! end of WriteListR
```

§**16. Write Multiple Class Group.** The text of a single group which contains more than one partition class. We carry out the "grouping together" activity, so that the user can add text fore and aft – this is how groups of objects such as "X, Y and Z" can be fluffed up to "the letters X, Y and Z from a Scrabble set" – but the default is simple: we print the grouped items by calling `WriteListR` once again, but this time starting from X, and where `lt_value` is set to the common `list_together` value of X, Y and Z. (That restricts the list to just the objects in this group.) Because `lt_value` is set to this value, the grouping code won't then group X, Y and Z again, and they will instead be individual classes in the new list – so each will end up being sent, in turn, to `WriteSingleClassGroup` below, and *that* is where they are printed.

```
[ WriteMultiClassGroup cl memb depth partition_class_sizes q k2 l;
    ! Save the style, because the activity below is allowed to change it
    q = c_style;
    if (c_style & INDENT_BIT ~= 0) PrintSpaces(2*(depth+c_margin));
    BeginActivity(GROUPING_TOGETHER_ACT, memb);
    if (ForActivity(GROUPING_TOGETHER_ACT, memb)) {
        c_style = c_style &~ NEWLINE_BIT;
    } else {
        if (memb.list_together ofclass String) {
            ! Set k2 to the number of objects covered by the group
            k2 = 0;
            for (l=0 : l<listing_size : l++) k2 = k2 + partition_class_sizes->(l+cl);
            EnglishNumber(k2); print " ";
            print (string) memb.list_together;
            if (c_style & ENGLISH_BIT ~= 0) print " (";
```

```
            if (c_style & INDENT_BIT ~= 0)  print ":^";
        } else {
            inventory_stage = 1;
            parser_one = memb; parser_two = depth + c_margin;
            if (RunRoutines(memb, list_together) == 1) jump Omit__Sublist2;
        }
        c_margin++;
        @push lt_value; @push listing_together; @push listing_size;
        lt_value = memb.list_together; listing_together = memb;
        #Ifdef DBLW; print "^^DOWN lt_value = ", lt_value, " listing_together = ", memb, "^^";
        @push DBLW_no_classes; @push DBLW_no_objs; #Endif;
        WriteListR(memb, depth, false);
        #Ifdef DBLW; print "^^UP^^"; @pull DBLW_no_objs; @pull DBLW_no_classes; #Endif;
        @pull listing_size; @pull listing_together; @pull lt_value;
        c_margin--;
        if (memb.list_together ofclass String) {
            if (q & ENGLISH_BIT ~= 0) print ")";
        } else {
            inventory_stage = 2;
            parser_one = memb; parser_two = depth+c_margin;
            RunRoutines(memb, list_together);
        }
        .Omit__Sublist2;
    }
    EndActivity(GROUPING_TOGETHER_ACT, memb);
    ! If the NEWLINE_BIT has been forced by the activity, act now
    ! before it vanishes...
    if (q & NEWLINE_BIT ~= 0 && c_style & NEWLINE_BIT == 0) new_line;
    ! ...when the original style is restored again:
    c_style = q;
];
```

§**17. Write Single Class Group.** The text of a single group which contains exactly one partition class. Because of the way the multiple-class case recurses, every class ends up in this routine sooner or later; this is the place where the actual name of an object is printed, at long last.

```
[ WriteSingleClassGroup cl memb depth size q;
    q = c_style;
    if (c_style & INDENT_BIT) PrintSpaces(2*(depth+c_margin));
    if (size == 1) {
        if (c_style & NOARTICLE_BIT ~= 0) print (name) memb;
        else {
            if (c_style & DEFART_BIT) {
                if ((cl == 1) && (c_style & CFIRSTART_BIT)) print (The) memb;
                else print (the) memb;
            } else {
                if ((cl == 1) && (c_style & CFIRSTART_BIT)) print (CIndefArt) memb;
                else print (a) memb;
            }
        }
    } else {
```

```
        if (c_style & DEFART_BIT) {
            if ((cl == 1) && (c_style & CFIRSTART_BIT)) PrefaceByArticle(memb, 0, size);
            else PrefaceByArticle(memb, 1, size);
        }
        @push listing_size; listing_size = size;
        CarryOutActivity(PRINTING_A_NUMBER_OF_ACT, memb);
        @pull listing_size;
    }
    if ((size > 1) && (memb hasnt pluralname)) {
        give memb pluralname;
        WriteAfterEntry(memb, depth);
        give memb ~pluralname;
    } else WriteAfterEntry(memb, depth);
    c_style = q;
];
```

§**18. Write After Entry.** Each entry can be followed by supplementary, usually parenthetical, information: exactly what, depends on the style. The extreme case is when the style, and the object, call for recursion to list the object-tree contents: this is achieved by calling `WriteListR`, using the `ObjectTreeIterator` (whatever the iterator used at the top level) and increasing the depth by 1.

```
[ WriteAfterEntry o depth
    p recurse_flag parenth_flag eldest_child child_count combo;
    inventory_stage = 2;
    if (c_style & PARTINV_BIT) {
        BeginActivity(PRINTING_ROOM_DESC_DETAILS_ACT);
        if (ForActivity(PRINTING_ROOM_DESC_DETAILS_ACT) == false) {
        combo = 0;
        if (o has light && location hasnt light) combo=combo+1;
        if (o has container && o hasnt open)     combo=combo+2;
        if ((o has container && (o has open || o has transparent))
            && (child(o)==0))                    combo=combo+4;
        if (combo) L__M(##ListMiscellany, combo, o);
        }
        EndActivity(PRINTING_ROOM_DESC_DETAILS_ACT);
    }   ! end of PARTINV_BIT processing
    if (c_style & FULLINV_BIT) {
        if (o has light && o has worn) { L__M(##ListMiscellany, 8);     parenth_flag = true; }
        else {
            if (o has light)           { L__M(##ListMiscellany, 9, o);  parenth_flag = true; }
            if (o has worn)            { L__M(##ListMiscellany, 10, o); parenth_flag = true; }
        }
        if (o has container)
            if (o has openable) {
                if (parenth_flag) {
                    #Ifdef SERIAL_COMMA; print ","; #Endif;
                    print (string) AND__TX;
                } else            L__M(##ListMiscellany, 11, o);
                if (o has open)
                    if (child(o)) L__M(##ListMiscellany, 12, o);
                    else          L__M(##ListMiscellany, 13, o);
                else
```

```
                if (o has lockable && o has locked) L__M(##ListMiscellany, 15, o);
                else                                L__M(##ListMiscellany, 14, o);
            parenth_flag = true;
        }
        else
            if (child(o)==0 && o has transparent)
                if (parenth_flag) L__M(##ListMiscellany, 16, o);
                else              L__M(##ListMiscellany, 17, o);
    if (parenth_flag) print ")";
}   ! end of FULLINV_BIT processing
child_count = 0;
eldest_child = nothing;
objectloop (p in o)
    if ((c_style & CONCEAL_BIT == 0) || (p hasnt concealed && p hasnt scenery))
        if (p has list_filter_permits) {
            child_count++;
            if (eldest_child == nothing) eldest_child = p;
        }
if (child_count && (c_style & ALWAYS_BIT)) {
    if (c_style & ENGLISH_BIT) L__M(##ListMiscellany, 18, o);
    recurse_flag = true;
}
if (child_count && (c_style & RECURSE_BIT)) {
    if (o has supporter) {
        if (c_style & ENGLISH_BIT) {
            if (c_style & TERSE_BIT) L__M(##ListMiscellany, 19, o);
            else                     L__M(##ListMiscellany, 20, o);
            if (o has animate)       print (string) WHOM__TX;
            else                     print (string) WHICH__TX;
        }
        recurse_flag = true;
    }
    if (o has container && (o has open || o has transparent)) {
        if (c_style & ENGLISH_BIT) {
            if (c_style & TERSE_BIT) L__M(##ListMiscellany, 21, o);
            else                     L__M(##ListMiscellany, 22, o);
            if (o has animate)       print (string) WHOM__TX;
            else                     print (string) WHICH__TX;
        }
        recurse_flag = true;
    }
}
if (recurse_flag && (c_style & ENGLISH_BIT))
    if (child_count > 1 || eldest_child has pluralname) print (string) ARE2__TX;
    else                                                print (string) IS2__TX;
if (c_style & NEWLINE_BIT) new_line;
if (recurse_flag) {
    o = child(o);
    @push lt_value; @push listing_together; @push listing_size;
    @push c_iterator;
    c_iterator = ObjectTreeIterator;
    lt_value = 0;   listing_together = 0;   listing_size = 0;
```

```
        WriteListR(o, depth+1, true);
        @pull c_iterator;
        @pull listing_size; @pull listing_together; @pull lt_value;
        if (c_style & TERSE_BIT) print ")";
    }
];
```

OutOfWorld Template B/oowt

Purpose
To implement some of the out of world actions.

---

§1. Perform Undo.

```
[ Perform_Undo;
    #ifdef PREVENT_UNDO; L__M(##Miscellany, 70); return; #endif;
    if (turns == 1) { L__M(##Miscellany, 11); return; }
    if (undo_flag == 0) { L__M(##Miscellany, 6); return; }
    if (undo_flag == 1) { L__M(##Miscellany, 7); return; }
    if (VM_Undo() == 0) L__M(##Miscellany, 7);
];
```

§2. Announce Score Rule.

```
[ ANNOUNCE_SCORE_R;
    if (actor ~= player) rfalse;
    #ifdef NO_SCORING; L__M(##Score, 2);
    #ifnot; GL__M(##Score); PrintRank();
    #endif;
];
```

§3. Switch Score Notification On Rule.

```
[ SWITCH_SCORE_NOTIFY_ON_R;
    if (actor ~= player) rfalse;
    #ifdef NO_SCORING; ANNOUNCE_SCORE_R();
    #ifnot; notify_mode=1; #endif;
];
```

§4. Standard Report Switching Score Notification On Rule.

```
[ REP_SWITCH_NOTIFY_ON_R;
    if (actor ~= player) rfalse;
    #ifndef NO_SCORING; GL__M(##NotifyOn); #endif;
];
```

§5. Switch Score Notification Off Rule.

```
[ SWITCH_SCORE_NOTIFY_OFF_R;
    if (actor ~= player) rfalse;
    #ifdef NO_SCORING; ANNOUNCE_SCORE_R();
    #ifnot; notify_mode=0; #endif;
];
```

§6. Standard Report Switching Score Notification Off Rule.

```
[ REP_SWITCH_NOTIFY_OFF_R;
    if (actor ~= player) rfalse;
    #ifndef NO_SCORING; GL__M(##NotifyOff); #endif;
];
```

§7. Prefer Sometimes Abbreviated Room Descriptions Rule.

```
[ PREFER_SOMETIMES_ABBREVIATED_R;
    if (actor ~= player) rfalse;
    lookmode=1;
]; ! Brief
```

§8. Standard Report Prefer Sometimes Abbreviated Room Descriptions Rule.

```
[ REP_PREFER_SOMETIMES_ABBR_R;
    if (actor ~= player) rfalse;
    print (string) Story; GL__M(##LMode1);
]; ! Brief
```

§9. Prefer Unabbreviated Room Descriptions Rule.

```
[ PREFER_UNABBREVIATED_R;
    if (actor ~= player) rfalse;
    lookmode=2;
]; ! Verbose
```

§10. Standard Report Prefer Unabbreviated Room Descriptions Rule.

```
[ REP_PREFER_UNABBREVIATED_R;
    if (actor ~= player) rfalse;
    print (string) Story; GL__M(##LMode2);
]; ! Verbose
```

§11. Prefer Abbreviated Room Descriptions Rule.

```
[ PREFER_ABBREVIATED_R;
    if (actor ~= player) rfalse;
    lookmode=3;
]; ! Superbrief
```

§12. Standard Report Prefer Abbreviated Room Descriptions Rule.

```
[ REP_PREFER_ABBREVIATED_R;
    if (actor ~= player) rfalse;
    print (string) Story; GL__M(##LMode3);
]; ! Superbrief
```

§13. Announce Pronoun Meanings Rule.

```
[ ANNOUNCE_PRONOUN_MEANINGS_R x y c d;
    if (actor ~= player) rfalse;
    GL__M(##Pronouns, 1);
    c = (LanguagePronouns-->0)/3;
    if (player ~= selfobj) c++;
    if (c==0) return GL__M(##Pronouns, 4);
    for (x = 1, d = 0 : x <= LanguagePronouns-->0: x = x+3) {
        print "~", (address) LanguagePronouns-->x, "~ ";
        y = LanguagePronouns-->(x+2);
        if (y == NULL) GL__M(##Pronouns, 3);
        else { GL__M(##Pronouns, 2); print (the) y; }
        d++;
        if (d < c-1) print ", ";
        if (d == c-1) print (string) LISTAND__TX;
    }
    if (player ~= selfobj) {
        print "~", (address) ME1__WD, "~ "; GL__M(##Pronouns, 2);
        c = player; player = selfobj;
        print (the) c; player = c;
    }
    ".";
];
```

WorldModel Template B/wmt

Purpose

Testing and changing the fundamental spatial relations.

---

§**1. The Core Tree.** Whereas I6 traditionally has a simple object tree hierarchy for containment, support, carrying and so on, the I7 template also uses the `component_*` properties to provide a second tree which defines the "part of" relation. These two trees interact in a subtle way, and it took a very long time to work out the simplest way to express this.

The *core* of an object is the root of its subtree in the component relation tree. So for a television set with a control panel which has a button on, the core for the button (and also for the panel and the set) will be the set. When X is a part of Y, it must be spatially in the same position as Y, and so the spatial location of X is determined by the position in the object tree of its core. (In effect, the spatial situation can be found by contracting together all nodes corresponding to objects which are parts of each other, but it would waste memory to construct such a tree: `CoreOfParentOfCoreOf` simulates what the `parent` operation would be in such a tree if it existed, wasting a little time instead.)

The *holder* of an object is its component parent, if it is part of something, or its object-tree parent if it has one. It is illegal for both of these to be non-`nothing`, so this is unambiguous.

```
[ HolderOf o;
    if (o && (o.component_parent)) return o.component_parent;
    if (o && (parent(o))) return parent(o);
    return nothing;
];
[ ParentOf o;
    if (o) o = parent(o);
    return o;
];
[ CoreOf o;
    while (o && (o provides component_parent) && (o.component_parent)) o = o.component_parent;
    return o;
];
[ CoreOfParentOfCoreOf o;
    while (o && (o provides component_parent) && (o.component_parent)) o = o.component_parent;
    if (o) o = parent(o);
    while (o && (o provides component_parent) && (o.component_parent)) o = o.component_parent;
    return o;
];
```

§**2. Climbing the Core Tree.** `LocationOf` returns the room in which an object can be found, or `nothing` if it is out of play. For this purpose a backdrop has no location, there being no canonical choice to make, while a two-sided door is in its "front side" (see below). Directions and regions are always out of play. For a room, `LocationOf` is necessarily itself.

`CommonAncestor` finds the nearest object indirectly containing `o1` and `o2`, or returns `nothing` if there is no common ancestor. (This is a port of `CommonAncestor` from the I6 library, adapted to work on the core tree.)

```
[ LocationOf o;
    if (~~(O ofclass K1_room or K2_thing)) return nothing;
    if (O ofclass K4_door) return FrontSideOfDoor(O);
    if (O ofclass K7_backdrop) return nothing;
    while (o) {
        if (o ofclass K1_room) return o;
        o = CoreOfParentOfCoreOf(o);
    }
    return nothing;
];
[ CommonAncestor o1 o2 i j;
    o1 = CoreOf(o1);
    o2 = CoreOf(o2);
    for (i=o1: i: i = CoreOfParentOfCoreOf(i))
        for (j=o2: j: j = CoreOfParentOfCoreOf(j))
            if (j == i) return j;
    return nothing;
];
[ IndirectlyContains o1 o2;
    if ((o1 == nothing) || (o2 == nothing)) rfalse;
    if ((o1 ofclass K1_room) && (o2 ofclass K4_door)) {
        if (o1 == FrontSideOfDoor(o2)) rtrue;
        if (o1 == BackSideOfDoor(o2)) rtrue;
        rfalse;
    }
    if (o2 ofclass K7_backdrop) rfalse;
    for (o2 = HolderOf(o2) : o2: o2 = HolderOf(o2)) if (o2 == o1) rtrue;
    rfalse;
];
```

§**3. To Decide Whether In.** A curiosity, this: the I6 definition of "To decide whether in (obj - object)". As can be seen, there are three possible interpretations of "in", depending on the kind of object, so this could all be done with three different definitions in the Standard Rules, so that run-time type checking would decide which to apply: but that would produce a little overhead and make the code for this rather common operation a bit complicated. Besides, this way we can produce a respectable run-time problem message when the phrase is misapplied.

Note that "in X" is not equivalent to "the player is in X": the latter uses direct containment, whereas we use indirect containment. Thus "in the Hall" and "in the laundry-basket" are both true if the player is in a laundry-basket in the Hall.

```
[ WhetherIn obj;
    if (obj has enterable) {
        if (IndirectlyContains(obj, player)) rtrue;
        rfalse;
    }
    if (obj ofclass K9_region) return TestRegionalContainment(real_location, obj);
    if (obj ofclass K1_room) {
        if (obj == real_location) rtrue;
        rfalse;
    }
    RunTimeProblem(RTP_NOTINAROOM, obj);
    rfalse;
];
```

§**4. Containment Relation.** This is the single most important relation in I7: direct containment. It is complicated by the fact that "A is in B" is represented differently at run-time when A is a room and B is a region, and when it isn't.

For each A there is at most one B such that "A is in B" is true. Because of this we test the relation with a function of one variable which turns A into B, rather than a function of two variables returning true or false. `ContainerOf` is that function.

I7 frequently needs to compile loops over all A such that "A is in B". In simpler relations (such as the support relation below) it can do this efficiently by iterating through the object-tree children of B, but for containment we have to provide an iterator function `TestContainmentRange`, as otherwise we would get the wrong result when B is a region.

```
[ ContainerOf A p;
    if (A ofclass K1_room) return A.map_region;
    p = parent(A);
    if (p == nothing) return nothing;
    if (p ofclass K5_container) return p;
    if (p ofclass K1_room) return p;
    if (p ofclass K9_region) return p;
    return nothing;
];
[ TestContainmentRange obj e f;
    if (obj ofclass K9_region) {
        objectloop (f ofclass K1_room && f.map_region == obj)
            if (f > e) return f;
        return nothing;
    }
    if (obj ofclass K5_container or K1_room) {
        if (e == nothing) return child(obj);
```

```
        return sibling(e);
    }
    return nothing;
];
```

§**5. Support Relation.** This then is simpler, with no need for an iterator governing searches.

```
[ SupporterOf obj p;
    p = parent(obj);
    if (p == nothing) return nothing;
    if (p ofclass K6_supporter) return p;
    return nothing;
];
```

§**6. Carrying Relation.** Only people may carry, and something worn is not in this sense carried.

```
[ CarrierOf obj p;
    p = parent(obj);
    if (p && (p ofclass K8_person) && (obj hasnt worn)) return p;
    return nothing;
];
```

§**7. Wearing Relation.** Only people may wear.

```
[ WearerOf obj p;
    p = parent(obj);
    if (p && (p ofclass K8_person) && (obj has worn)) return p;
    return nothing;
];
```

§**8. Having Relation.** A person has something if and only if he either wears or carries it.

```
[ OwnerOf obj p;
    p = parent(obj);
    if (p && (p ofclass K8_person)) return p;
    return nothing;
];
```

§**9. Making Parts.** Note that `MakePart` removes the part-to-be from the object tree before attaching it: it cannot, of course, be the core of the resulting object.

```
[ MakePart P Of First;
    if (parent(P)) remove P; give P ~worn;
    if (Of == nothing) { DetachPart(P); return; }
    if (P.component_parent) DetachPart(P);
    P.component_parent = Of;
    First = Of.component_child;
    Of.component_child = P; P.component_sibling = First;
];
[ DetachPart P From Daddy O;
    Daddy = P.component_parent; P.component_parent = nothing;
    if (Daddy == nothing) { P.component_sibling = nothing; return; }
    if (Daddy.component_child == P) {
        Daddy.component_child = P.component_sibling;
        P.component_sibling = nothing; return;
    }
    for (O = Daddy.component_child: O: O = O.component_sibling)
        if (O.component_sibling == P) {
            O.component_sibling = P.component_sibling;
            P.component_sibling = nothing; return;
        }
];
```

§**10. Movements.** Note that an object is detached from its component parent, if it has one, when moved.

```
[ MoveObject F T opt going_mode was;
    if (F == nothing) return RunTimeProblem(RTP_CANTMOVENOTHING);
    if (F ofclass K7_backdrop) {
        if (T ofclass K9_region) {
            give F ~absent; F.found_in = T.regional_found_in;
            if (TestRegionalContainment(LocationOf(player), T)) move F to LocationOf(player);
            else remove F;
            return; }
        return RunTimeProblem(RTP_BACKDROP, F, T);
    }
    if (~~(F ofclass K2_thing)) return RunTimeProblem(RTP_NOTTHING, F, T);
    if (T ofclass K9_region) return RunTimeProblem(RTP_NOTBACKDROP, F, T);
    if (F has worn) {
        give F ~worn;
        if (F in T) return;
    }
    DetachPart(F);
    if (going_mode == false) {
        if (F == player) { PlayerTo(T, opt); return; }
        if ((IndirectlyContains(F, player)) && (LocationOf(player) ~= LocationOf(T))) {
            was = parent(player);
            move player to real_location;
            move F to T;
            PlayerTo(was, true);
            return;
```

```
        }
    }
    move F to T;
];
[ RemoveFromPlay F;
    if (F == nothing) return RunTimeProblem(RTP_CANTREMOVENOTHING);
    if (F == player) return RunTimeProblem(RTP_CANTREMOVEPLAYER);
    if (F ofclass K4_door) return RunTimeProblem(RTP_CANTREMOVEDOORS);
    give F ~worn; DetachPart(F);
    if (F ofclass K7_backdrop) give F absent;
    remove F;
];
```

§**11. On Stage.** The following implements the "on-stage" and "off-stage" adjectives provided by the Standard Rules. Here, as above, note that the I6 attribute `absent` marks a floating object (see below) which has been removed from play; in I7 only doors and backdrops are allowed to float, and only backdrops are allowed to be removed from play.

```
[ OnStage O x;
    if (O ofclass K1_room) rfalse;
    while (metaclass(O) == Object) {
        if (O ofclass K1_room) rtrue;
        if (O ofclass K9_region) rfalse;
        if (O ofclass K4_door) rtrue;
        if (O ofclass K7_backdrop) { if (O has absent) rfalse; rtrue; }
        x = O.component_parent; if (x) { O = x; continue; }
        x = parent(O); if (x) { O = x; continue; }
        rfalse;
    }
    rfalse;
];
```

§**12. Moving the Player.** Note that the player object can only be moved by this routine: this allows us to maintain the invariant for `real_location` and `location` (for which, see "Light.i6t") and to ensure that multiply-present objects can be witnessed where they need to be.

```
[ PlayerTo newplace flag;
    @push actor; actor = player;
    move player to newplace;
    location = LocationOf(newplace);
    real_location = location;
    MoveFloatingObjects();
    ADJUST_LIGHT_R(1);
    DivideParagraphPoint();
    if (flag == 0) <Look>;
    if (flag == 1) give location visited;
    if (flag == 2) AbbreviatedRoomDescription();
    @pull actor;
];
```

§13. Move During Going. The following routine preserves the invariant for `real_location`, but gets `location` wrong since it doesn't adjust for light. Nor are floating objects moved. It should be used only in the course of other operations which get the rest of this right. (I7 uses it only for the "going" action, where these various operations are each handled by different named rules to increase the flexibility of the system.)

```
[ MoveDuringGoing F T;
    MoveObject(F, T, 0, true);
    if (actor == player) {
        location = LocationOf(player);
        real_location = location;
    }
];
```

§14. Being Everywhere. The following is used as the `found_in` property for any backdrop which is "everywhere", that is, which is found in every room.

```
[ FoundEverywhere; rtrue; ];
```

§15. Changing the Player. It is very important that nobody simply change the value of the `player` variable, because so much else must be updated when the identity of the player changes: the light situation, floating objects, the `real_location` and so on. Because of this, the NI compiler contains code which compiles an assertion of the proposition "is(`player`, X)" into a function call `ChangePlayer(X)` rather than a variable assignment `player = X`. So we cannot catch out the system by writing "now the player is Mr Henderson".

We must ensure that if `player` is initially X, is changed to Y, and is then changed back to X, that both X and Y end exactly as they began – hence the flummery below with using `remove_proper` to ensure that `proper` is left with the correct value. Note that:

(1) at any given time exactly one person has the I6 `concealed` attribute: the current player;
(2) the `selfobj` is the default initial value of `player`, and because it has as its actual printed name "yourself", we need to override this when something else takes over as player: we change to "your former self", in fact. No such device is needed for other people being changed from because they are explicitly given printed names – say "Mr Darcy", "the sous-chef", etc. – in the source text.

```
[ ChangePlayer obj flag i;
    if (~~(obj ofclass K8_person)) return RunTimeProblem(RTP_CANTCHANGE, obj);
    if (~~(OnStage(obj))) return RunTimeProblem(RTP_CANTCHANGEOFFSTAGE, obj);
    if (obj == player) return;
    give player ~concealed;
    if (player has remove_proper) give player ~proper;
    if (player == selfobj) {
        player.saved_short_name = player.short_name; player.short_name = FORMER__TX;
    }
    player = obj;
    if (player == selfobj) {
        player.short_name = player.saved_short_name;
    }
    if (player hasnt proper) give player remove_proper; ! when changing out again
    give player concealed proper;
    location = LocationOf(player); real_location = location;
    MoveFloatingObjects();
    SilentlyConsiderLight();
];
```

§**16. Floating Objects.** A single object can only be in one position in the object tree, yet backdrops and doors must be present in multiple rooms. This is accomplished by making them, in I6 jargon, "floating objects": objects which move in the tree whenever the player does, so that – from the player's perspective – they are always present when they should be. In I6, almost anything can be made a floating object, but in I7 this is strictly and only used for backdrops and two-sided doors.

There are several conceptual problems with this scheme: chiefly that it assumes that the only witness to the spatial arrangement of objects is the player. In I6 that was usually true, but in I7, where every person can undertake actions, it really isn't true any longer: if the objects float to follow the player, it means that they are not present with other people who might need to interact with them. This is why the accessibility rules are somewhat hacked for backdrops and doors (see "Light.i6t"). In fact we generally achieve the illusion we want, but this is largely because it is difficult to catch out all the exceptions for backdrops and doors, and because in practice authors tend not to set things up so that the presence or absence of backdrops much affects what non-player characters do.

All the same, the scheme is not logically defensible, and this is why we do not allow the user to create new categories of floating objects in I7.

The I6 implementation of `MoveFloatingObjects` acted on the `location` rather than the `real_location`, which (a) meant that multiply-present objects could include `thedark` – the generic Darkness place – as one possible location, but (b) assumed that the only sense by which the player could witness an object was sight. In I7, `thedark` is not a valid room, and we are a bit more careful about the senses.

```
[ MoveFloatingObjects i k l m address flag;
    if (real_location == nothing) return;
    objectloop (i) {
        address = i.&found_in;
        if (address ~= 0 && i hasnt absent) {
            if (ZRegion(address-->0) == 2) {
                m = address-->0;
                .TestPropositionally;
                if (m.call(real_location) ~= 0) move i to real_location;
                else remove i;
            }
            else {
                k = i.#found_in;
                for (l=0 : l<k/WORDSIZE : l++) {
                    m = address-->l;
                    if (ZRegion(m) == 2) jump TestPropositionally;
                    if (m == real_location || m in real_location) {
                        if (i notin real_location) move i to real_location;
                        flag = true;
                    }
                }
                if (flag == false) { if (parent(i)) remove i; }
            }
        }
    }
];
[ MoveBackdrop bd D x address;
    if (~~(bd ofclass K7_backdrop)) return RunTimeProblem(RTP_BACKDROPONLY, bd);
    if (bd.#found_in > WORDSIZE) {
        address = bd.&found_in;
        address-->0 = D;
    } else bd.found_in = D;
```

```
    give bd ~absent;
    MoveFloatingObjects();
];
```

§**17. Wearing Clothes.** An object X is worn by a person P if and only if (i) the object tree `parent` of X is P, and (ii) X has the `worn` attribute. In fact for I7 purposes we are careful to ensure that (ii) happens only when (i) does in any case: if X is moved in the object tree, or made a part of something, or removed from play, then `worn` is removed.

```
[ WearObject X P opt;
    if (X == false) rfalse;
    if (X notin P) MoveObject(X, P, opt);
    give X worn;
];
```

§**18. Map Connections.** `MapConnection` returns the room which is in the given direction (as an object) from the given room: it is used, among other things, to test the "mapped east of" and similar relations. (The same relations are asserted true with `AssertMapConnection` and false with `AssertMapUnconnection`.)

`RoomOrDoorFrom` returns either the room or door which is in the given direction, and is thus simpler (since it doesn't have to investigate what is through such a door).

Both routines return values which are type-safe in I7 provided the kind of value they are assigned to is "object": neither returns any specific kind of object without fail. (`MapConnection` is always either a door or `nothing`, but `nothing` is not a typesafe value for "door".)

Note that map connections via doors are immutable.

```
[ MapConnection from_room dir
    in_direction through_door;
    if ((from_room ofclass K1_room) && (dir ofclass K3_direction)) {
        in_direction = Map_Storage-->
            ((from_room.IK_1)*No_Directions + dir.IK_3);
        if (in_direction ofclass K1_room) return in_direction;
        if (in_direction ofclass K4_door) {
            @push location;
            location = from_room;
            through_door = in_direction.door_to();
            @pull location;
            if (through_door ofclass K1_room) return through_door;
        }
    }
    return nothing;
];
[ DoorFrom obj dir rv;
    rv = RoomOrDoorFrom(obj, dir);
    if (rv ofclass K4_door) return rv;
    return nothing;
];
[ RoomOrDoorFrom obj dir use_doors in_direction sl through_door;
    if ((obj ofclass K1_room) && (dir ofclass K3_direction)) {
        in_direction = Map_Storage-->
            ((obj.IK_1)*No_Directions + dir.IK_3);
        if (in_direction ofclass K1_room or K4_door) return in_direction;
```

```
    }
    return nothing;
];
[ AssertMapConnection r1 dir r2 in_direction;
    SignalMapChange();
    in_direction = Map_Storage-->
        ((r1.IK_1)*No_Directions + dir.IK_3);
    if ((in_direction == 0) || (in_direction ofclass K1_room)) {
        Map_Storage-->((r1.IK_1)*No_Directions + dir.IK_3) = r2;
        return;
    }
    if (in_direction ofclass K4_door) {
        RunTimeProblem(RTP_EXITDOOR, r1, dir);
        return;
    }
    RunTimeProblem(RTP_NOEXIT, r1, dir);
];
[ AssertMapUnconnection r1 dir r2 in_direction;
    SignalMapChange();
    in_direction = Map_Storage-->
        ((r1.IK_1)*No_Directions + dir.IK_3);
    if (r1 ofclass K4_door) {
        RunTimeProblem(RTP_EXITDOOR, r1, dir);
        return;
    }
    if (in_direction == r2)
        Map_Storage-->((r1.IK_1)*No_Directions + dir.IK_3) = 0;
    return;
];
```

§**19. Adjacency Relation.** A relation between two rooms which, note, does not see connections through doors.

```
[ TestAdjacency R1 R2 i row;
    row = (R1.IK_1)*No_Directions;
    for (i=0: i<No_Directions: i++, row++)
        if (Map_Storage-->row == R2) rtrue;
    rfalse;
];
```

§**20. Regional Containment Relation.** This tests whether an object with a definite physical location is in a given region. (For a two-sided door which straddles regions, the front side is what counts; for a backdrop, a direction or another region, the answer is always no.) We rely on the fact that every room object has a `map_region` property which is the smallest region containing it, if any does, and `nothing` otherwise; region objects are then in the object tree such that parenthood corresponds to spatial containment. (Note that in I7, a region must either completely contain another region, or else have no overlap with it.)

```
[ TestRegionalContainment obj region o;
    if ((obj == nothing) || (region == nothing)) rfalse;
    if (~~(obj ofclass K1_room)) obj = LocationOf(obj);
    if (obj == nothing) rfalse;
    o = obj.map_region;
    while (o) {
        if (o == region) rtrue;
        o = parent(o);
    }
    rfalse;
];
```

§**21. Doors.** There are two sorts of door: one-sided and two-sided.

A two-sided door is in two rooms at once: one is called the front side, the other the back side. The front side is the one declared first in the source. Note that a one-sided door is also an I6 object of class `K4_door`, and then the front side is the room holding it, while the back side is `nothing`.

The `door_to` property calculates the room which the door leads to; that of course depends on which side of it the player is standing. Similarly, `door_dir` calculates the direction it leads in. Unlike the I6 setup, where `door_dir` returned the direction property (`n_to`, `s_to`, etc.), here in I7's template it returns the direction object (`n_obj`, `s_obj`, etc.)

```
[ FrontSideOfDoor D; if (~~(D ofclass K4_door)) rfalse;
    if (D provides found_in) return (D.&found_in)-->0; ! Two-sided
    return parent(D); ! One-sided
];
[ BackSideOfDoor D; if (~~(D ofclass K4_door)) rfalse;
    if (D provides found_in) return (D.&found_in)-->1; ! Two-sided
    return nothing; ! One-sided
];
[ OtherSideOfDoor D from_room rv;
    if (D ofclass K4_door) {
        @push location;
        location = LocationOf(from_room);
        rv = D.door_to();
        @pull location;
    }
    return rv;
];
[ DirectionDoorLeadsIn D from_room rv dir;
    if (D ofclass K4_door) {
        @push location;
        location = LocationOf(from_room);
        rv = D.door_dir();
        @pull location;
    }
```

```
    return rv;
];
```

§**22. Visibility Relation.** We use `TestScope` to decide whether there is a line of sight from A to B; it's a relation which cannot be asserted true or false.

```
[ TestVisibility A B;
    if (~~OffersLight(parent(CoreOf(A)))) rfalse;
    if (suppress_scope_loops) rtrue;
    return TestScope(B, A);
];
```

§**23. Touchability Relation.** We use `ObjectIsUntouchable` to decide whether there is physical access from A to B; it's a relation which cannot be asserted true or false.

```
[ TestTouchability A B;
    if (TestScope(B,A) == false) rfalse;
    if (ObjectIsUntouchable(B, 1, 0, A)) rfalse;
    rtrue;
];
```

§**24. Concealment Relation.** An activity determines whether one object conceals another; it's a relation which cannot be asserted true or false.

```
[ TestConcealment A B;
    if (A ofclass K2_thing && B ofclass K2_thing) {
        particular_possession = B;
        if (CarryOutActivity(DECIDING_CONCEALED_POSSESS_ACT, A)) rtrue;
    }
    rfalse;
];
```

Light Template B/light

Purpose

The determination of light, visibility and physical access.

§**1. Darkness.** "Darkness" is not really a place: but in I6 it has to be an object so that the location-name on the status line can be "Darkness". In I7 we use it as little as possible: note that it has no properties.

```
Object thedark "(darkness object)";
```

§**2. Light Measurement.** These two routines, `OffersLight` and `HasLightSource`, are largely unchanged from their I6 definitions; see the *Inform Designer's Manual*, 4th edition, for a commentary. In terms of how they are used in I7, `OffersLight` is called only by the two rules below; `HasLightSource` is called also in determining the scope, that is, what is visible to the player.

```
[ OffersLight obj j;
    while (obj) {
        if (obj has light) rtrue;
        objectloop (j in obj) if (HasLightSource(j)) rtrue;
        if ((obj has container) && (obj hasnt open) && (obj hasnt transparent)) rfalse;
        if ((obj provides component_parent) && (obj.component_parent))
            obj = obj.component_parent;
        else
            obj = parent(obj);
    }
    rfalse;
];
[ HasLightSource i j ad sr po;
    if (i == 0) rfalse;
    if (i has light) rtrue;
    if ((IsSeeThrough(i)) && (~~(HidesLightSource(i))))
        objectloop (j in i)
            if (HasLightSource(j)) rtrue;
    ad = i.&add_to_scope;
    if (parent(i) ~= 0 && ad ~= 0) {
        if (metaclass(ad-->0) == Routine) {
            ats_hls = 0; ats_flag = 1;
            sr = scope_reason; po = parser_one;
            scope_reason = LOOPOVERSCOPE_REASON; parser_one = 0;
            RunRoutines(i, add_to_scope);
            scope_reason = sr; parser_one = po;
            ats_flag = 0; if (ats_hls == 1) rtrue;
        }
        else {
            for (j=0 : (WORDSIZE*j)<i.#add_to_scope : j++)
                if ((ad-->j) && (HasLightSource(ad-->j) == 1)) rtrue;
```

```
        }
    }
    if (ComponentHasLight(i)) rtrue;
    rfalse;
];
[ ComponentHasLight o obj next_obj;
    if (o provides component_child) {
        obj = o.component_child;
        while (obj) {
            next_obj = obj.component_sibling;
            if (obj has light) rtrue;
            if (HasLightSource(obj)) rtrue;
            if ((obj provides component_child) && (ComponentHasLight(obj))) rtrue;
            obj = next_obj;
        }
    }
    rfalse;
];
[ HidesLightSource obj;
    if (obj == player) rfalse;
    if (obj has transparent or supporter) rfalse;
    if (obj has animate) rfalse;
    if (obj has container) return (obj hasnt open);
    return (obj hasnt enterable);
];
```

§**3. Invariant.** The following routines maintain two variables about the light condition of the player:

(a) If on the most recent check the player was in light, then `location` equals `real_location` and `lightflag` is false.

(b) If on the most recent check the player was in darkness, then `location` equals `thedark` and `lightflag` is true.

Note that they are not allowed to alter `real_location`, whose definition has nothing to do with light.

```
Global lightflag = false;
```

§**4. Adjust Light Rule.** This rule fires at least once a turn, and more often when the player moves location, since that's likely to invalidate any previous assumptions. It compares the state of light now with the last time it ran, and gives instructions on what to do in each of the four possibilities.

```
[ ADJUST_LIGHT_R previous_light_condition;
    previous_light_condition = lightflag;
    lightflag = OffersLight(parent(player));
    if ((previous_light_condition == false) && (lightflag == false)) {
        location = thedark;
        rfalse;
    }
    if ((previous_light_condition == false) && (lightflag == true)) {
        location = real_location;
        CarryOutActivity(PRINTING_NEWS_OF_LIGHT_ACT);
        rfalse;
    }
    if ((previous_light_condition == true) && (lightflag == false)) {
        location = thedark;
        DivideParagraphPoint();
        BeginActivity(PRINTING_NEWS_OF_DARKNESS_ACT);
        if (ForActivity(PRINTING_NEWS_OF_DARKNESS_ACT) == false) L__M(##Miscellany, 9);
        EndActivity(PRINTING_NEWS_OF_DARKNESS_ACT);
        rfalse;
    }
    if ((previous_light_condition == true) && (lightflag == true)) {
        location = real_location;
        rfalse;
    }
    rfalse;
];
```

§**5. Silent Light Consideration.** The Adjust Light Rule makes a fuss when light changes: it prints messages, for instance. This rule is silent instead, and simply does the minimum necessary to maintain the light invariant. It is used in only three circumstances:

(a) To determine the initial light condition at start of play.
(b) When the player moves from one room to another via the "going" action.
(c) When the player changes from one persona to another.

Perhaps case (b) is surprising. Why not simply use the adjust light rule, as we do on an instance of "move player to ...", for instance? The answer is that the going action is just about to print details of the new location anyway, so it would be redundant to have the adjust light rule print out details as well.

```
[ SilentlyConsiderLight;
    lightflag = OffersLight(parent(player));
    if (lightflag) location = real_location; else location = thedark;
    rfalse;
];
```

§**6. Translucency.** `IsSeeThrough` is used at various places: roughly speaking, it determines whether `obj` being in scope means that the object-tree contents of `obj` are in scope.

```
[ IsSeeThrough obj;
    if ((obj has supporter)
        || (obj has transparent)
        || (obj has animate)
        || ((obj has container) && (obj has open)))
        rtrue;
rfalse;
];
```

§**7. Visibility Parent.** The idea of `VisibilityParent` is that it takes us from a given position in the object tree, `o`, to the next visible position above. Note that

(1) A container has an inside and an outside: this routine calculates from the "inside of `o`", which is why it returns `nothing` from an opaque closed container;
(2) Component parts are (for purposes of this routine) attached to the outside surface of a container, so that if `o` is part of a closed opaque container then the visibility parent of `o` is its actual parent.

```
[ VisibilityParent o;
    if (o && (o has container) && (o hasnt open) && (o hasnt transparent)) return nothing;
    if (o) o = CoreOfParentOfCoreOf(o);
    return o;
];
```

§**8. Find Visibility Levels.** The following routine sets the pair of variables `visibility_ceiling`, the highest visible point in the object tree above the player – or `thedark` if the player cannot see at all – and `visibility_levels`, the number of steps of `VisibilityParent` needed to reach this ceiling; or 0 if the player cannot see at all.

```
[ FindVisibilityLevels lc up;
    if (location == thedark) {
        visibility_ceiling = thedark;
        visibility_levels = 0;
    } else {
        visibility_ceiling = player;
        while (true) {
            up = VisibilityParent(visibility_ceiling);
            if (up == 0) break;
            visibility_ceiling = up;
            lc++;
        }
        visibility_levels = lc;
    }
];
```

§**9. Scope Ceiling.** Scope is almost the same thing as visibility, but not quite, and the following routine does not quite duplicate the calculation of `FindVisibilityLevels`. The difference arises in the first step, where we take the `parent` of `pos`, not the core of the `parent` of the core of `pos`: this makes a difference if `pos` is inside a container which is itself part of something else.

```
[ ScopeCeiling pos c;
    if (pos == player && location == thedark) return thedark;
    c = parent(pos);
    if (c == 0) return pos;
    while (VisibilityParent(c)) c = VisibilityParent(c);
    return c;
];
```

§**10. Object Is Untouchable.** The following routine imitates the I6 library one of the same name, but works instead by delegating the decision to the accessibility rulebook.

It is not easy to answer the question of whether someone other than the player, but in another room, can touch a multiply-present object (a two-sided door or a backdrop) and we err on the side of caution by saying yes in such cases. The question would only be asked in the case of a "try" action deliberately caused by the author, or by an instruction made by the player to someone not in the same room: in the first case, we will assume that the author knows what he is doing, and in the second case, the circumstances in which saying yes would be a wrong call are *highly* improbable.

```
[ ObjectIsUntouchable item silent_flag flag2 p save_sp decision;
    if ((p ~= player) && (LocationOf(p) ~= LocationOf(player)) &&
        ((item ofclass K4_door) || (item ofclass K7_backdrop))) {
        decision = false;
    } else {
        untouchable_object = item; untouchable_silence = silent_flag;
        touch_persona = p; if (p == actor) touch_persona = 0;
        save_sp = say__p; say__p = 0;
        if (ProcessRulebook(ACCESSIBILITY_RB, 0, true)) {
            if (RulebookSucceeded()) decision = false;
            else decision = true;
        } else decision = false;
        if (say__p == false) say__p = save_sp;
    }
    untouchable_silence = 0;
    return decision;
];
```

§11. Access Through Barriers Rule.

```
[ ACCESS_THROUGH_BARRIERS_R ancestor i j external p;
    p = touch_persona; if (p == 0) p = actor;
    ancestor = CommonAncestor(p, untouchable_object);
    if ((ancestor == 0) && (LocationOf(untouchable_object) == nothing)
        && ((untouchable_object ofclass K4_door or K7_backdrop) == false)) {
        if (touch_persona == 0) GL__M(##Take,8,untouchable_object);
        RulebookFails();
        rtrue;
    }
    ! First, a barrier between the player and the ancestor.
    if (CoreOf(p) ~= ancestor) {
        i = parent(CoreOf(p)); j = CoreOf(i); external = false;
        if (j ~= i) { i = j; external = true; }
        while (i~=ancestor && i) {
            if ((external == false)
                && (ProcessRulebook(REACHING_OUTSIDE_RB, i))
                && (RulebookFailed())) rtrue; ! Barrier
            i = parent(CoreOf(i)); j = CoreOf(i); external = false;
            if (j ~= i) { i = j; external = true; }
        }
    }
    ! Second, a barrier between the item and the ancestor.
    if (CoreOf(untouchable_object) ~= ancestor) {
        ! We can always get to the core of the item.
        i = CoreOf(untouchable_object);
        ! This will be on the inside of its parent, if its parent is a
        ! container, so there should be no exemption.
        i = parent(i); external = false;
        ! j = CoreOf(i); if (j ~= i) { i = j; external = true; }
        while (i~=ancestor && i) {
            if ((external == false) &&
                (ProcessRulebook(REACHING_INSIDE_RB, i)) &&
                (RulebookFailed())) rtrue; ! Barrier
            i = CoreOf(i);
            if (i == ancestor) break;
            i = parent(i); j = CoreOf(i); external = false;
            if (j ~= i) { i = j; external = true; }
        }
    }
    RulebookSucceeds(); ! No barrier
    rtrue;
];
```

§12. Can't Reach Inside Closed Containers Rule.

```
[ CANT_REACH_INSIDE_CLOSED_R;
    if (parameter_object has container && parameter_object hasnt open) {
        if (touch_persona == 0) GL__M(##Take,9,parameter_object);
        RulebookFails(); rtrue;
    }
    rfalse;
];
```

§13. Can't Reach Outside Closed Containers Rule.

```
[ CANT_REACH_OUTSIDE_CLOSED_R;
    if (parameter_object has container && parameter_object hasnt open) {
        if (touch_persona == 0) GL__M(##Take,9,parameter_object);
        RulebookFails(); rtrue;
    }
    rfalse;
];
```

§14. Can't Reach Inside Rooms Rule.

```
[ CANT_REACH_INSIDE_ROOMS_R;
    if (parameter_object && parameter_object ofclass K1_room) {
        if (touch_persona == 0) GL__M(##Take,14,parameter_object);
        RulebookFails(); rtrue;
    }
    rfalse;
];
```

Tests Template B/testt

Purpose

The command grammar and I6 implementation for testing commands such as TEST, ACTIONS and PURLOIN.

---

§**1. Abstract Command.** The code below is compiled only if the symbol `DEBUG` is defined, which it always is for normal runs in the Inform user interface, but not for Release runs.

Not all of these commands are documented; this is intentional. They may be changed in name or function. This is all of the testing commands except for the GLKLIST command, which is in Glulx.i6t (and does not exist when the target VM is the Z-machine).

We take the commands in alphabetical order, beginning with ABSTRACT, which moves an object to a new position in the object tree.

```
[ XAbstractSub;
    if (XTestMove(noun, second)) return;
    move noun to second;
    "[Abstracted.]";
];
[ XTestMove obj dest;
    if ((obj <= InformLibrary) || (obj == LibraryMessages))
        "[Can't move ", (name) obj, ": it's a system object.]";
    if (obj.component_parent)
        "[Can't move ", (name) obj, ": it's part of ",
        (the) obj.component_parent, ".]";
    while (dest) {
        if (dest == obj) "[Can't move ", (name) obj, ": it would contain itself.]";
        dest = CoreOfParentOfCoreOf(dest);
    }
    rfalse;
];
```

§**2. Actions Command.** ACTIONS turns tracing of actions on.

```
[ ActionsOnSub; trace_actions = 1; say__p = 1; "Actions listing on."; ];
[ ActionsOffSub; trace_actions = 0; say__p = 1; "Actions listing off."; ];
```

§**3. Gonear Command.** GONEAR teleports the player to the vicinity of some named item.

```
[ GonearSub;
    PlayerTo(LocationOf(noun));
];
```

§**4. Purloin Command.** To PURLOIN is to acquire something without reference to any rules on accessibility.

```
[ XPurloinSub;
    if (XTestMove(noun, player)) return;
    move noun to player; give noun moved ~concealed;
    say__p = 1;
    "[Purloined.]";
];
```

§**5. Random Command.** RANDOM forces the random-number generator to a predictable seed value.

```
[ PredictableSub;
    VM_Seed_RNG(-100);
    say__p = 1;
    "[Random number generator now predictable.]";
];
```

§**6. Relations Command.** RELATIONS lists the current state of the mutable relations.

```
[ ShowRelationsSub rc tot;
    for (rc=0:rc<{-value:NUMBER_CREATED(binary_predicate)}:rc++)
        tot = tot + Relation_ShowR(rc);
    say__p = 1;
    if (tot == 0) "No new relations have been created.";
];
```

§**7. Rules Command.** RULES changes the level of rule tracing.

```
[ RulesOnSub;
    debug_rules = 1; say__p = 1;
    "Rules tracing now switched on. Type ~rules off~ to switch it off again,
    or ~rules all~ to include even rules which do not apply.";
];
[ RulesAllSub;
    debug_rules = 2; say__p = 1;
    "Rules tracing now switched to ~all~. Type ~rules off~ to switch it off again.";
];
[ RulesOffSub;
    debug_rules = 0; say__p = 1;
    "Rules tracing now switched off. Type ~rules~ to switch it on again.";
];
```

§**8. Scenes Command.** SCENES switches scene-change tracing on or off, and also shows the current position.

```
[ ScenesOnSub;
    debug_scenes = 1;
    ShowSceneStatus(); say__p = 1;
    "(Scene monitoring now switched on. Type ~scenes off~ to switch it off again.)";
];
[ ScenesOffSub;
    debug_scenes = 0; say__p = 1;
    "(Scene monitoring now switched off. Type ~scenes~ to switch it on again.)";
];
```

§**9. Scope Command.** SCOPE prints a numbered list of all objects in scope to the player.

```
Global x_scope_count;
[ ScopeSub;
    x_scope_count = 0;
    LoopOverScope(Print_ScL, noun);
    if (x_scope_count == 0) "Nothing is in scope.";
];
[ Print_ScL obj; print_ret ++x_scope_count, ": ", (a) obj, " (", obj, ")"; ];
```

§**10. Showheap Command.** SHOWHEAP is for debugging the memory heap, and is intended for Inform maintainers rather than users.

```
[ ShowHeapSub;
    DebugHeap();
];
```

§**11. ShowMe Command.** SHOWME is probably the most useful testing command: it shows the state of the current room, or a named item.

```
[ ShowMeSub view na;
    view = noun;
    if (noun == nothing) noun = real_location;
    if (ShowMeRecursively(noun, 0, (noun == real_location))) {
        if (noun == real_location)
            print "* denotes things which are not in scope^";
    }
    if (view ofclass K2_thing) {
        print "location:"; ShowRLocation(noun, true); print "^";
    }
    {-call:add_showme_details}
];
[ ShowRLocation obj top;
    if (obj ofclass K1_room) return;
    print " ";
    if (parent(obj)) {
        if (obj has worn) print "worn by ";
        else {
```

```
            if (parent(obj) has animate) print "carried by ";
            if (parent(obj) has container) print "in ";
            if (parent(obj) ofclass K1_room) print "in ";
            if (parent(obj) has supporter) print "on ";
        }
        print (the) parent(obj);
        ShowRLocation(parent(obj));
    } else {
        if (obj.component_parent) {
            if (top == false) print ", which is ";
            print "part of ", (the) obj.component_parent;
            ShowRLocation(obj.component_parent);
        }
        else print "out of play";
    }
];
[ ShowMeRecursively obj depth f c i k;
    spaces(2*depth);
    if (f && (depth > 0) && (TestScope(obj, player) == false)) { print "*"; c = true; }
    print (name) obj;
    if (depth > 0) {
        if (obj.component_parent) print " (part of ", (name) obj.component_parent, ")";
        if (obj has worn) print " (worn)";
    }
    if (obj provides IK_0) {
        k = KindHierarchy-->((obj.IK_0)*2);
        if ((k ~= K2_thing) || (depth==0)) {
            print " - ";
            if (k == K4_door or K5_container) {
                if (obj has transparent) print "transparent ";
                if (obj has locked) print "locked ";
                else if (obj has open) print "open ";
                else print "closed ";
            }
            print (I7_Kind_Name) k;
        }
    }
    print "^";
    if (obj.component_child) c = c | ShowMeRecursively(obj.component_child, depth+2, f);
    if ((depth>0) && (obj.component_sibling))
        c = c | ShowMeRecursively(obj.component_sibling, depth, f);
    if (child(obj)) c = c | ShowMeRecursively(child(obj), depth+2, f);
    if ((depth>0) && (sibling(obj))) c = c | ShowMeRecursively(sibling(obj), depth, f);
    return c;
];
```

§**12. Showverb Command.** SHOWVERB is a holdover from old I6 days, but still quite useful. It writes out the I6 command verb grammar for the supplied command.

```
[ ShowVerbSub address lines meta i;
    if (noun == 0 || ((noun->#dict_par1) & 1) == 0)
        "Try typing ~showverb~ and then the name of a verb.";
    meta = ((noun->#dict_par1) & 2)/2;
    i = DictionaryWordToVerbNum(noun);
    address = VM_CommandTableAddress(i);
    lines = address->0;
    address++;
    print "Verb ";
    if (meta) print "meta ";
    VM_PrintCommandWords(i);
    new_line;
    if (lines == 0) "has no grammar lines.";
    for (: lines>0 : lines--) {
        address = UnpackGrammarLine(address);
        print "    "; DebugGrammarLine(); new_line;
    }
];
[ DebugGrammarLine pcount;
    print " * ";
    for (: line_token-->pcount ~= ENDIT_TOKEN : pcount++) {
        if ((line_token-->pcount)->0 & $10) print "/ ";
        print (DebugToken) line_token-->pcount, " ";
    }
    print "-> ", (DebugAction) action_to_be;
    if (action_reversed) print " reverse";
];
[ DebugToken token;
    AnalyseToken(token);
    switch (found_ttype) {
    ILLEGAL_TT:
        print "<illegal token number ", token, ">";
    ELEMENTARY_TT:
        switch (found_tdata) {
        NOUN_TOKEN:           print "noun";
        HELD_TOKEN:           print "held";
        MULTI_TOKEN:          print "multi";
        MULTIHELD_TOKEN:      print "multiheld";
        MULTIEXCEPT_TOKEN:    print "multiexcept";
        MULTIINSIDE_TOKEN:    print "multiinside";
        CREATURE_TOKEN:       print "creature";
        SPECIAL_TOKEN:        print "special";
        NUMBER_TOKEN:         print "number";
        TOPIC_TOKEN:          print "topic";
        ENDIT_TOKEN:          print "END";
        }
    PREPOSITION_TT:
        print "'", (address) found_tdata, "'";
    ROUTINE_FILTER_TT:
        print "noun=Routine(", found_tdata, ")";
```

```
    ATTR_FILTER_TT:
        print (DebugAttribute) found_tdata;
    SCOPE_TT:
        print "scope=Routine(", found_tdata, ")";
    GPR_TT:
        print "Routine(", found_tdata, ")";
    }
];
```

§**13. Test Command.** TEST runs a short script of commands from the source text.

```
#Iftrue ({-value:NUMBER_CREATED(test_scenario)} > 0);
[ TestScriptSub;
    switch(special_word) {
{-call:compile_test_switch}
    default:
        print ">--> The following tests are available:^";
{-call:compile_test_printout}
    }
];
Constant TEST_STACK_SIZE = 40;
Array test_stack --> TEST_STACK_SIZE;
Global test_sp = 0;
[ TestStart T R l k;
    if (test_sp >= TEST_STACK_SIZE) ">--> Testing too many levels deep";
    test_stack-->test_sp = T;
    test_stack-->(test_sp+1) = 0;
    test_stack-->(test_sp+3) = l;
    test_sp = test_sp + 4;
    if ((R-->0) && (R-->0 ~= real_location)) {
        print "(first moving to ", (name) R-->0, ")^";
        PlayerTo(R-->0, 1);
    }
    k=1;
    while (R-->k) {
        if (R-->k notin player) {
            print "(first acquiring ", (the) R-->k, ")^";
            move R-->k to player;
        }
        k++;
    }
    print "(Testing.)^"; say__p = 1;
];
[ TestKeyboardPrimitive a_buffer a_table p i j l spaced ch;
    if (test_sp == 0) {
        test_stack-->2 = 1;
        return VM_ReadKeyboard(a_buffer, a_table);
    }
    else {
        p = test_stack-->(test_sp-4);
        i = test_stack-->(test_sp-3);
        l = test_stack-->(test_sp-1);
```

```
        print "[";
        print test_stack-->2;
        print "] ";
        test_stack-->2 = test_stack-->2 + 1;
        style bold;
        while ((i < l) && (p->i ~= '/')) {
            ch = p->i;
            if (spaced || (ch ~= ' ')) {
                if ((p->i == '[') && (p->(i+1) == '/') && (p->(i+2) == ']')) {
                    ch = '/'; i = i+2;
                }
                a_buffer->(j+WORDSIZE) = ch;
                print (char) ch;
                i++; j++;
                spaced = true;
            } else i++;
        }
        style roman;
        print "^";
        #ifdef TARGET_ZCODE;
        a_buffer->1 = j;
        #ifnot; ! TARGET_GLULX
        a_buffer-->0 = j;
        #endif;
        VM_Tokenise(a_buffer, a_table);
        if (p->i == '/') i++;
        if (i >= l) {
            test_sp = test_sp - 4;
        } else test_stack-->(test_sp-3) = i;
    }
];
#IFNOT;
[ TestScriptSub;
    ">--> No test scripts exist for this game.";
];
#ENDIF;
```

§**14. Trace Command.** Another holdover from I6: TRACE sets the level of parser tracing, on a scale of 0 (off, the default) to 5.

```
[ TraceOnSub; parser_trace=1; say__p = 1; "[Trace on.]"; ];
[ TraceLevelSub;
    parser_trace = noun; say__p = 1;
    print "[Parser tracing set to level ", parser_trace, ".]^";
];
[ TraceOffSub; parser_trace=0; say__p = 1; "Trace off."; ];
```

§**15. Tree Command.** TREE prints out the I6 object tree, though this is not always very helpful in I7 terms. It should arguably be withdrawn, but doesn't seem to do any harm.

```
[ XTreeSub i;
    if (noun == 0) {
        objectloop (i)
            if (i ofclass Object && parent(i) == 0) XObj(i);
    }
    else XObj(noun,1);
];
[ XObj obj f;
    if (parent(obj) == 0) print (name) obj; else print (a) obj;
    print " (", obj, ") ";
    if (f == 1 && parent(obj) ~= 0)
        print "(in ", (name) parent(obj), " ", parent(obj), ")";
    new_line;
    if (child(obj) == 0) rtrue;
    if (obj == Class)
        WriteListFrom(child(obj), NEWLINE_BIT+INDENT_BIT+ALWAYS_BIT+NOARTICLE_BIT, 1);
    else
        WriteListFrom(child(obj), NEWLINE_BIT+INDENT_BIT+ALWAYS_BIT+FULLINV_BIT, 1);
];
```

§**16. Grammar.** In the old I6 parser, testing commands had their own scope hardwired in to the code: this worked by comparing the verb command word directly against `'scope'` and the like. That would go wrong if the testing commands were translated into other languages, and was a crude design at best. The following scope token is better: using this token instead of `multi` provides a noun with universal scope (but restricted to I7 objects, so I6 pseudo-objects like `compass` are not picked up) and able to accept multiple objects.

```
[ testcommandnoun obj o2;
    switch (scope_stage) {
        1: rtrue; ! allow multiple objects
        2: objectloop (obj)
            if ((obj ofclass Object) && (obj provides IK_0))
                PlaceInScope(obj, true);
        3: print "There seems to be no such object anywhere in the model world.^";
    }
];
{-testing-command:abstract}
    * scope=testcommandnoun 'to' scope=testcommandnoun -> XAbstract;
{-testing-command:actions}
    *                                       -> ActionsOn
    * 'on'                                  -> ActionsOn
    * 'off'                                 -> ActionsOff;
{-testing-command:gonear}
    * scope=testcommandnoun                 -> Gonear;
{-testing-command:purloin}
    * scope=testcommandnoun                 -> XPurloin;
{-testing-command:random}
    *                                       -> Predictable;
{-testing-command:relations}
```

```
    *                                   -> ShowRelations;
{-testing-command:rules}
    *                                   -> RulesOn
    * 'all'                             -> RulesAll
    * 'off'                             -> RulesOff;
{-testing-command:scenes}
    *                                   -> ScenesOn
    * 'off'                             -> ScenesOff;
{-testing-command:scope}
    *                                   -> Scope
    * scope=testcommandnoun             -> Scope;
{-testing-command:showheap}
    *                                   -> ShowHeap;
{-testing-command:showme}
    *                                   -> ShowMe
    * scope=testcommandnoun             -> ShowMe;
{-testing-command:showverb}
    * special                           -> Showverb;
{-testing-command:test}
    *                                   -> TestScript
    * special                           -> TestScript;
{-testing-command:trace}
    *                                   -> TraceOn
    * number                            -> TraceLevel
    * 'on'                              -> TraceOn
    * 'off'                             -> TraceOff;
{-testing-command:tree}
    *                                   -> XTree
    * scope=testcommandnoun             -> XTree;
```

Language Template B/langt

Purpose

The fundamental definitions needed by the parser and the verb library in order to specify the language of play – that is, the language used for communications between the story file and the player.

§1. Vocabulary.

```
Constant AGAIN1__WD     = 'again';
Constant AGAIN2__WD     = 'g//';
Constant AGAIN3__WD     = 'again';
Constant OOPS1__WD      = 'oops';
Constant OOPS2__WD      = 'o//';
Constant OOPS3__WD      = 'oops';
Constant UNDO1__WD      = 'undo';
Constant UNDO2__WD      = 'undo';
Constant UNDO3__WD      = 'undo';

Constant ALL1__WD       = 'all';
Constant ALL2__WD       = 'each';
Constant ALL3__WD       = 'every';
Constant ALL4__WD       = 'everything';
Constant ALL5__WD       = 'both';
Constant AND1__WD       = 'and';
Constant AND2__WD       = 'and';
Constant AND3__WD       = 'and';
Constant BUT1__WD       = 'but';
Constant BUT2__WD       = 'except';
Constant BUT3__WD       = 'but';
Constant ME1__WD        = 'me';
Constant ME2__WD        = 'myself';
Constant ME3__WD        = 'self';
Constant OF1__WD        = 'of';
Constant OF2__WD        = 'of';
Constant OF3__WD        = 'of';
Constant OF4__WD        = 'of';
Constant OTHER1__WD     = 'another';
Constant OTHER2__WD     = 'other';
Constant OTHER3__WD     = 'other';
Constant THEN1__WD      = 'then';
Constant THEN2__WD      = 'then';
Constant THEN3__WD      = 'then';

Constant NO1__WD        = 'n//';
Constant NO2__WD        = 'no';
Constant NO3__WD        = 'no';
Constant YES1__WD       = 'y//';
Constant YES2__WD       = 'yes';
Constant YES3__WD       = 'yes';

Constant AMUSING__WD    = 'amusing';
```

```
Constant FULLSCORE1__WD = 'fullscore';
Constant FULLSCORE2__WD = 'full';
Constant QUIT1__WD      = 'q//';
Constant QUIT2__WD      = 'quit';
Constant RESTART__WD    = 'restart';
Constant RESTORE__WD    = 'restore';
```

§2. Pronouns.

```
Array LanguagePronouns table
! word        possible GNAs                  connected
!             to follow:                     to:
!             a     i
!             s  p  s  p
!             mfnmfnmfnmfn
    'it'      $$001000111000                   NULL
    'him'     $$100000000000                   NULL
    'her'     $$010000000000                   NULL
    'them'    $$000111000111                   NULL;
```

§3. Descriptors.

```
Array LanguageDescriptors table
! word        possible GNAs   descriptor      connected
!             to follow:      type:           to:
!             a     i
!             s  p  s  p
!             mfnmfnmfnmfn
    'my'      $$111111111111    POSSESS_PK      0
    'this'    $$111111111111    POSSESS_PK      0
    'these'   $$000111000111    POSSESS_PK      0
    'that'    $$111111111111    POSSESS_PK      1
    'those'   $$000111000111    POSSESS_PK      1
    'his'     $$111111111111    POSSESS_PK      'him'
    'her'     $$111111111111    POSSESS_PK      'her'
    'their'   $$111111111111    POSSESS_PK      'them'
    'its'     $$111111111111    POSSESS_PK      'it'
    'the'     $$111111111111    DEFART_PK       NULL
    'a//'     $$111000111000    INDEFART_PK     NULL
    'an'      $$111000111000    INDEFART_PK     NULL
    'some'    $$000111000111    INDEFART_PK     NULL
    'lit'     $$111111111111    light           NULL
    'lighted' $$111111111111    light           NULL
    'unlit'   $$111111111111    (-light)        NULL;
```

§4. Numbers.

```
Array LanguageNumbers table
    'one' 1 'two' 2 'three' 3 'four' 4 'five' 5
    'six' 6 'seven' 7 'eight' 8 'nine' 9 'ten' 10
    'eleven' 11 'twelve' 12 'thirteen' 13 'fourteen' 14 'fifteen' 15
    'sixteen' 16 'seventeen' 17 'eighteen' 18 'nineteen' 19 'twenty' 20
    'twenty-one' 21 'twenty-two' 22 'twenty-three' 23 'twenty-four' 24
    'twenty-five' 25 'twenty-six' 26 'twenty-seven' 27 'twenty-eight' 28
    'twenty-nine' 29 'thirty' 30
;
[ LanguageNumber n f;
    if (n == 0)    { print "zero"; rfalse; }
    if (n < 0)     { print "minus "; n = -n; }
    if (n >= 1000) { print (LanguageNumber) n/1000, " thousand"; n = n%1000; f = 1; }
    if (n >= 100)  {
        if (f == 1) print ", ";
        print (LanguageNumber) n/100, " hundred"; n = n%100; f = 1;
    }
    if (n == 0) rfalse;
    #Ifdef DIALECT_US;
    if (f == 1) print " ";
    #Ifnot;
    if (f == 1) print " and ";
    #Endif;
    switch (n) {
    1:    print "one";
    2:    print "two";
    3:    print "three";
    4:    print "four";
    5:    print "five";
    6:    print "six";
    7:    print "seven";
    8:    print "eight";
    9:    print "nine";
    10:   print "ten";
    11:   print "eleven";
    12:   print "twelve";
    13:   print "thirteen";
    14:   print "fourteen";
    15:   print "fifteen";
    16:   print "sixteen";
    17:   print "seventeen";
    18:   print "eighteen";
    19:   print "nineteen";
    20 to 99: switch (n/10) {
        2:  print "twenty";
        3:  print "thirty";
        4:  print "forty";
        5:  print "fifty";
        6:  print "sixty";
        7:  print "seventy";
        8:  print "eighty";
```

```
        9:  print "ninety";
        }
        if (n%10 ~= 0) print "-", (LanguageNumber) n%10;
    }
];
```

§5. Time.

```
[ LanguageTimeOfDay hours mins i;
    i = hours%12;
    if (i == 0) i = 12;
    if (i < 10) print " ";
    print i, ":", mins/10, mins%10;
    if ((hours/12) > 0) print " pm"; else print " am";
];
```

§6. Directions.

```
[ LanguageDirection d;
    print (name) d;
];
```

§7. Translation.

```
[ LanguageToInformese; ];
```

§8. Articles.

```
Constant LanguageAnimateGender   = male;
Constant LanguageInanimateGender = neuter;
Constant LanguageContractionForms = 2;     ! English has two:
                                           ! 0 = starting with a consonant
                                           ! 1 = starting with a vowel
[ LanguageContraction text;
    if (text->0 == 'a' or 'e' or 'i' or 'o' or 'u'
                or 'A' or 'E' or 'I' or 'O' or 'U') return 1;
    return 0;
];
Array LanguageArticles -->
!   Contraction form 0:     Contraction form 1:
!   Cdef   Def    Indef     Cdef   Def    Indef
    "The " "the " "a "      "The " "the " "an "          ! Articles 0
    "The " "the " "some "   "The " "the " "some ";       ! Articles 1
                !             a           i
                !             s     p     s     p
                !             m f n m f n m f n m f n
Array LanguageGNAsToArticles --> 0 0 0 1 1 1 0 0 0 1 1 1;
```

§**9. Commands.** `LanguageVerbLikesAdverb` is called by `PrintCommand` when printing an `UPTO_PE` error or an inference message. Words which are intransitive verbs, i.e., which require a direction name as an adverb ("walk west"), not a noun ("I only understood you as far as wanting to touch the ground"), should cause the routine to return `true`.

`LanguageVerbMayBeName` is called by `NounDomain` when dealing with the player's reply to a "Which do you mean, the short stick or the long stick?" prompt from the parser. If the reply is another verb (for example, LOOK) then then previous ambiguous command is discarded unless it is one of these words which could be both a verb and an adjective in a `name` property.

```
[ LanguageVerb i;
    switch (i) {
    'i//','inv','inventory':
            print "take inventory";
    'l//':   print "look";
    'x//':   print "examine";
    'z//':   print "wait";
    default: rfalse;
    }
    rtrue;
];
[ LanguageVerbLikesAdverb w;
    if (w == 'look' or 'go' or 'push' or 'walk')
        rtrue;
    rfalse;
];
[ LanguageVerbMayBeName w;
    if (w == 'long' or 'short' or 'normal'
                    or 'brief' or 'full' or 'verbose')
        rtrue;
    rfalse;
];
```

§**10. Short Texts.**

```
Constant NKEY__TX       = "N = next subject";
Constant PKEY__TX       = "P = previous";
Constant QKEY1__TX      = "  Q = resume game";
Constant QKEY2__TX      = "Q = previous menu";
Constant RKEY__TX       = "RETURN = read subject";
Constant NKEY1__KY      = 'N';
Constant NKEY2__KY      = 'n';
Constant PKEY1__KY      = 'P';
Constant PKEY2__KY      = 'p';
Constant QKEY1__KY      = 'Q';
Constant QKEY2__KY      = 'q';
Constant SCORE__TX      = "Score: ";
Constant MOVES__TX      = "Moves: ";
Constant TIME__TX       = "Time: ";
Global CANTGO__TX     = "You can't go that way.";
Global FORMER__TX     = "your former self";
Global YOURSELF__TX   = "yourself";
Constant YOU__TX        = "You";
```

```
Constant DARKNESS__TX   = "Darkness";
Constant THOSET__TX     = "those things";
Constant THAT__TX       = "that";
Constant OR__TX         = " or ";
Constant NOTHING__TX    = "nothing";
Global IS__TX         = " is";
Global ARE__TX        = " are";
Global IS2__TX        = "is ";
Global ARE2__TX       = "are ";
Global IS3__TX        = "is";
Global ARE3__TX       = "are";
Constant AND__TX        = " and ";
#ifdef SERIAL_COMMA;
Constant LISTAND__TX   = ", and ";
Constant LISTAND2__TX  = " and ";
#ifnot;
Constant LISTAND__TX   = " and ";
Constant LISTAND2__TX  = " and ";
#endif; ! SERIAL_COMMA
Constant WHOM__TX       = "whom ";
Constant WHICH__TX      = "which ";
Constant COMMA__TX      = ", ";
```

§11. Printed Inflections.

```
[ ThatorThose obj;      ! Used in the accusative
    if (obj == player)            { print "you"; return; }
    if (obj has pluralname)       { print "those"; return; }
    if (obj has animate) {
        if (obj has female)       { print "her"; return; }
        else
            if (obj hasnt neuter) { print "him"; return; }
    }
    print "that";
];
[ ItorThem obj;
    if (obj == player)            { print "yourself"; return; }
    if (obj has pluralname)       { print "them"; return; }
    if (obj has animate) {
        if (obj has female)       { print "her"; return; }
        else
            if (obj hasnt neuter) { print "him"; return; }
    }
    print "it";
];
[ IsorAre obj;
    if (obj has pluralname || obj == player) print "are"; else print "is";
];
[ HasorHave obj;
    if (obj has pluralname || obj == player) print "have"; else print "has";
];
```

```
[ CThatorThose obj;     ! Used in the nominative
    if (obj == player)            { print "You"; return; }
    if (obj has pluralname)       { print "Those"; return; }
    if (obj has animate) {
        if (obj has female)       { print "She"; return; }
        else
            if (obj hasnt neuter) { print "He"; return; }
    }
    print "That";
];
[ CTheyreorThats obj;
    if (obj == player)             { print "You're"; return; }
    if (obj has pluralname)        { print "They're"; return; }
    if (obj has animate) {
        if (obj has female)        { print "She's"; return; }
        else if (obj hasnt neuter) { print "He's"; return; }
    }
    print "That's";
];
[ HisHerTheir o; if (o has pluralname) { print "their"; return; }
    if (o has female) { print "her"; return; }
    if (o has neuter) { print "its"; return; }
    print "his";
];
[ HimHerItself o; if (o has pluralname) { print "theirselves"; return; }
    if (o has female) { print "herself"; return; }
    if (o has neuter) { print "itself"; return; }
    print "himself";
];
```

§**12. Long Texts.** The messages here are expected eventually to move into I7 tables, where they will be more easily dealt with. But for now, the old-fashioned way:

```
[ LanguageLM n x1 x2;
say__p = 1;
Answer,Ask:
             "There is no reply.";
! Ask:      see Answer
Attack:   "Violence isn't the answer to this one.";
Burn:     "This dangerous act would achieve little.";
Buy:      "Nothing is on sale.";
Climb:    "I don't think much is to be achieved by that.";
Close: switch (n) {
        1:  print_ret (ctheyreorthats) x1, " not something you can close.";
        2:  print_ret (ctheyreorthats) x1, " already closed.";
        3:  "You close ", (the) x1, ".";
        4: print (The) actor, " closes ", (the) x1, ".^";
        5: print (The) x1, " close"; if (x1 hasnt pluralname) print "s";
            print ".^";
    }
Consult: switch (n) {
        1: "You discover nothing of interest in ", (the) x1, ".";
```

```
        2: print (The) actor, " looks at ", (the) x1, ".^";
    }
Cut:      "Cutting ", (thatorthose) x1, " up would achieve little.";
Disrobe: switch (n) {
        1:  "You're not wearing ", (thatorthose) x1, ".";
        2:  "You take off ", (the) x1, ".";
        3: print (The) actor, " takes off ", (the) x1, ".^";
    }
Drink:    "There's nothing suitable to drink here.";
Drop: switch (n) {
        1:  if (x1 has pluralname) print (The) x1, " are "; else print (The) x1, " is ";
            "already here.";
        2:  "You haven't got ", (thatorthose) x1, ".";
        3:  print "(first taking ", (the) x1, " off)^"; say__p = 0; return;
        4:  "Dropped.";
        5: "There is no more room on ", (the) x1, ".";
        6: "There is no more room in ", (the) x1, ".";
        7: print (The) actor, " puts down ", (the) x1, ".^";
    }
Eat: switch (n) {
        1:  print_ret (ctheyreorthats) x1, " plainly inedible.";
        2:  "You eat ", (the) x1, ". Not bad.";
        3: print (The) actor, " eats ", (the) x1, ".^";
    }
Enter: switch (n) {
        1:  print "But you're already ";
            if (x1 has supporter) print "on "; else print "in ";
            print_ret (the) x1, ".";
        2:  if (x1 has pluralname) print "They're"; else print "That's";
            print " not something you can ";
            switch (verb_word) {
            'stand':  "stand on.";
            'sit':    "sit down on.";
            'lie':    "lie down on.";
            default:  "enter.";
            }
        3:  "You can't get into the closed ", (name) x1, ".";
        4:  "You can only get into something free-standing.";
        5:  print "You get ";
            if (x1 has supporter) print "onto "; else print "into ";
            print_ret (the) x1, ".";
        6:  print "(getting ";
            if (x1 has supporter) print "off "; else print "out of ";
            print (the) x1; print ")^"; say__p = 0; return;
        7:  ! say__p = 0;
            if (x1 has supporter) "(getting onto ", (the) x1, ")";
            if (x1 has container) "(getting into ", (the) x1, ")";
            "(entering ", (the) x1, ")";
        8: print (The) actor, " gets into ", (the) x1, ".^";
        9:  print (The) actor, " gets onto ", (the) x1, ".^";
    }
Examine: switch (n) {
        1:  "Darkness, noun.  An absence of light to see by.";
```

```
        2:  "You see nothing special about ", (the) x1, ".";
        3:  print (The) x1, " ", (isorare) x1, " currently switched ";
            if (x1 has on) "on."; else "off.";
        4: print (The) actor, " looks closely at ", (the) x1, ".^";
        5: "You see nothing unexpected in that direction.";
    }
Exit: switch (n) {
        1:  "But you aren't in anything at the moment.";
        2:  "You can't get out of the closed ", (name) x1, ".";
        3:  print "You get ";
            if (x1 has supporter) print "off "; else print "out of ";
            print_ret (the) x1, ".";
        4:  print "But you aren't ";
            if (x1 has supporter) print "on "; else print "in ";
            print_ret (the) x1, ".";
        5: print (The) actor, " gets off ", (the) x1, ".^";
        6: print (The) actor, " gets out of ", (the) x1, ".^";
    }
GetOff:   "But you aren't on ", (the) x1, " at the moment.";
Give: switch (n) {
        1:  "You aren't holding ", (the) x1, ".";
        2:  "You juggle ", (the) x1, " for a while, but don't achieve much.";
        3:  print (The) x1;
            if (x1 has pluralname) print " don't"; else print " doesn't";
            " seem interested.";
        4:  print (The) x1;
            if (x1 has pluralname) print " aren't";
            else print " isn't";
            " able to receive things.";
        5: "You give ", (the) x1, " to ", (the) second, ".";
        6: print (The) actor, " gives ", (the) x1, " to you.^";
        7: print (The) actor, " gives ", (the) x1, " to ", (the) second, ".^";
    }
Go: switch (n) {
        1:  print "You'll have to get ";
            if (x1 has supporter) print "off "; else print "out of ";
            print_ret (the) x1, " first.";
        2:  print_ret (string) CANTGO__TX;   ! "You can't go that way."
        3:  "You are unable to climb ", (the) x1, ".";
        4:  "You are unable to descend by ", (the) x1, ".";
        5:  "You can't, since ", (the) x1, " ", (isorare) x1, " in the way.";
        6:  print "You can't, since ", (the) x1;
            if (x1 has pluralname) " lead nowhere."; else " leads nowhere.";
        7: "You'll have to say which compass direction to go in.";
        8: print (The) actor, " goes up";
        9: print (The) actor, " goes down";
        10: print (The) actor, " goes ", (name) x1;
        11: print (The) actor, " arrives from above";
        12: print (The) actor, " arrives from below";
        13: print (The) actor, " arrives from the ", (name) x1;
        14: print (The) actor, " arrives";
        15: print (The) actor, " arrives at ", (the) x1, " from above";
        16: print (The) actor, " arrives at ", (the) x1, " from below";
```

```
        17: print (The) actor, " arrives at ", (the) x1, " from the ", (name) x2;
        18: print (The) actor, " goes through ", (the) x1;
        19: print (The) actor, " arrives from ", (the) x1;
        20: print "on ", (the) x1;
        21: print "in ", (the) x1;
        22: print ", pushing ", (the) x1, " in front, and you along too";
        23: print ", pushing ", (the) x1, " in front";
        24: print ", pushing ", (the) x1, " away";
        25: print ", pushing ", (the) x1, " in";
        26: print ", taking you along";
    }
Insert: switch (n) {
        1:  "You need to be holding ", (the) x1, " before you can put ", (itorthem) x1,
            " into something else.";
        2:  print_ret (Cthatorthose) x1, " can't contain things.";
        3:  print_ret (The) x1, " ", (isorare) x1, " closed.";
        4:  "You'll need to take ", (itorthem) x1, " off first.";
        5:  "You can't put something inside itself.";
        6:  print "(first taking ", (itorthem) x1, " off)^"; say__p = 0; return;
        7:  "There is no more room in ", (the) x1, ".";
        8:  "Done.";
        9:  "You put ", (the) x1, " into ", (the) second, ".";
    10:  print (The) actor, " puts ", (the) x1, " into ", (the) second, ".^";
    }
Inv: switch (n) {
        1:  "You are carrying nothing.";
        2:  print "You are carrying";
        3:  print ":^";
        4:  print ".^";
        5: print (The) x1, " looks through ", (HisHerTheir) x1, " possessions.^";
    }
Jump:     "You jump on the spot, fruitlessly.";
Kiss:     "Keep your mind on the game.";
Listen:   "You hear nothing unexpected.";
ListMiscellany: switch (n) {
        1:  print " (providing light)";
        2:  print " (closed)";
        4:  print " (empty)";
        6:  print " (closed and empty)";
        3:  print " (closed and providing light)";
        5:  print " (empty and providing light)";
        7:  print " (closed, empty and providing light)";
        8:  print " (providing light and being worn";
        9:  print " (providing light";
        10: print " (being worn";
        11: print " (";
        12: print "open";
        13: print "open but empty";
        14: print "closed";
        15: print "closed and locked";
        16: print " and empty";
        17: print " (empty)";
        18: print " containing ";
```

```
        19: print " (on ";
        20: print ", on top of ";
        21: print " (in ";
        22: print ", inside ";
    }
LMode1:   " is now in its normal ~brief~ printing mode, which gives long descriptions
            of places never before visited and short descriptions otherwise.";
LMode2:   " is now in its ~verbose~ mode, which always gives long descriptions
            of locations (even if you've been there before).";
LMode3:   " is now in its ~superbrief~ mode, which always gives short descriptions
            of locations (even if you haven't been there before).";
Lock: switch (n) {
        1:  if (x1 has pluralname) print "They don't "; else print "That doesn't ";
            "seem to be something you can lock.";
        2:  print_ret (ctheyreorthats) x1, " locked at the moment.";
        3:  "First you'll have to close ", (the) x1, ".";
        4:  if (x1 has pluralname) print "Those don't "; else print "That doesn't ";
            "seem to fit the lock.";
        5:  "You lock ", (the) x1, ".";
        6: print (The) actor, " locks ", (the) x1, ".^";
    }
Look: switch (n) {
        1:  print " (on ", (the) x1, ")";
        2:  print " (in ", (the) x1, ")";
        3:  print " (as ", (object) x1, ")";
        4:  print "On ", (the) x1, " ";
            WriteListFrom(child(x1),
            ENGLISH_BIT+RECURSE_BIT+PARTINV_BIT+TERSE_BIT+CONCEAL_BIT+ISARE_BIT);
            ".";
        5,6:
            if (x1 ~= location) {
                if (x1 has supporter) print "On "; else print "In ";
                print (the) x1, " you";
            }
            else print "You";
            print " can ";
            if (n == 5) print "also ";
            print "see ";
            WriteListFrom(child(x1),
            ENGLISH_BIT+RECURSE_BIT+PARTINV_BIT+TERSE_BIT+CONCEAL_BIT+WORKFLAG_BIT);
            if (x1 ~= location) "."; else " here.";
        7:  "You see nothing unexpected in that direction.";
        8:  if (x1 has supporter) print " (on "; else print " (in ";
            print (the) x1, ")";
        9: print (The) actor, " looks around.^";
    }
LookUnder: switch (n) {
        1:  "But it's dark.";
        2:  "You find nothing of interest.";
        3: print (The) actor, " looks under ", (the) x1, ".^";
    }
Mild:     "Quite.";
Miscellany: switch (n) {
```

```
1:  "(considering the first sixteen objects only)^";
2:  "Nothing to do!";
3:  print " You have died ";
4:  print " You have won ";
5:  print "^Would you like to RESTART, RESTORE a saved game";
    #Ifdef DEATH_MENTION_UNDO;
    print ", UNDO your last move";
    #Endif;
    #ifdef SERIAL_COMMA;
    print ",";
    #endif;
    " or QUIT?";
6:  "[Your interpreter does not provide ~undo~.  Sorry!]";
    #Ifdef TARGET_ZCODE;
7:  "~Undo~ failed.  [Not all interpreters provide it.]";
    #Ifnot; ! TARGET_GLULX
7:  "[You cannot ~undo~ any further.]";
    #Endif; ! TARGET_
8:  "Please give one of the answers above.";
9:  "It is now pitch dark in here!";
10: "I beg your pardon?";
11: "[You can't ~undo~ what hasn't been done!]";
12: "[Can't ~undo~ twice in succession. Sorry!]";
13: "[Previous turn undone.]";
14: "Sorry, that can't be corrected.";
15: "Think nothing of it.";
16: "~Oops~ can only correct a single word.";
17: "It is pitch dark, and you can't see a thing.";
18: print "yourself";
19: "As good-looking as ever.";
20: "To repeat a command like ~frog, jump~, just say ~again~, not ~frog, again~.";
21: "You can hardly repeat that.";
22: "You can't begin with a comma.";
23: "You seem to want to talk to someone, but I can't see whom.";
24: "You can't talk to ", (the) x1, ".";
25: "To talk to someone, try ~someone, hello~ or some such.";
26: "(first taking ", (the) x1, ")";
27: "I didn't understand that sentence.";
28: print "I only understood you as far as wanting to ";
29: "I didn't understand that number.";
30: "You can't see any such thing.";
31: "You seem to have said too little!";
32: "You aren't holding that!";
33: "You can't use multiple objects with that verb.";
34: "You can only use multiple objects once on a line.";
35: "I'm not sure what ~", (address) pronoun_word, "~ refers to.";
36: "You excepted something not included anyway!";
37: "You can only do that to something animate.";
    #Ifdef DIALECT_US;
38: "That's not a verb I recognize.";
    #Ifnot;
38: "That's not a verb I recognise.";
    #Endif;
```

```
        39: "That's not something you need to refer to in the course of this game.";
        40: "You can't see ~", (address) pronoun_word, "~ (", (the) pronoun_obj,
            ") at the moment.";
        41: "I didn't understand the way that finished.";
        42: if (x1 == 0) print "None"; else print "Only ", (number) x1;
            print " of those ";
            if (x1 == 1) print "is"; else print "are";
            " available.";
        43: "Nothing to do!";
        44: "There are none at all available!";
        45: print "Who do you mean, ";
        46: print "Which do you mean, ";
        47: "Sorry, you can only have one item here. Which exactly?";
        48: print "Whom do you want";
            if (actor ~= player) print " ", (the) actor;
            print " to "; PrintCommand(); print "?^";
        49: print "What do you want";
            if (actor ~= player) print " ", (the) actor;
            print " to "; PrintCommand(); print "?^";
        50: print "Your score has just gone ";
            if (x1 > 0) print "up"; else { x1 = -x1; print "down"; }
            print " by ", (number) x1, " point";
            if (x1 > 1) print "s";
        51: "(Since something dramatic has happened, your list of commands has been cut short.)";
        52: "^Type a number from 1 to ", x1, ", 0 to redisplay or press ENTER.";
        53: "^[Please press SPACE.]";
        54: "[Comment recorded.]";
        55: "[Comment NOT recorded.]";
        56: print ".^";
        57: print "?^";
        58: print (The) actor, " ", (IsOrAre) actor, " unable to do that.^";
        59: "You must supply a noun.";
        60: "You may not supply a noun.";
        61: "You must name an object.";
        62: "You may not name an object.";
        63: "You must name a second object.";
        64: "You may not name a second object.";
        65: "You must supply a second noun.";
        66: "You may not supply a second noun.";
        67: "You must name something more substantial.";
        68: print "(", (The) actor, " first taking ", (the) x1, ")^";
        69: "(first taking ", (the) x1, ")";
        70: "The use of UNDO is forbidden in this game.";
        71: print (string) DARKNESS__TX;
        72: print (The) x1;
            if (x1 has pluralname) print " have"; else print " has";
            " better things to do.";
        73: "That noun did not make sense in this context.";
        74: print "[That command asks to do something outside of play, so it can
            only make sense from you to me. ", (The) x1, " cannot be asked to do this.]^";
    }
No,Yes:   "That was a rhetorical question.";
NotifyOff:
```

```
            "Score notification off.";
NotifyOn: "Score notification on.";
Open: switch (n) {
        1:  print_ret (ctheyreorthats) x1, " not something you can open.";
        2:  if (x1 has pluralname) print "They seem "; else print "It seems ";
            "to be locked.";
        3:  print_ret (ctheyreorthats) x1, " already open.";
        4:  print "You open ", (the) x1, ", revealing ";
            if (WriteListFrom(child(x1), ENGLISH_BIT+TERSE_BIT+CONCEAL_BIT) == 0) "nothing.";
            ".";
        5:  "You open ", (the) x1, ".";
        6: print (The) actor, " opens ", (the) x1, ".^";
        7: print (The) x1, " open";
            if (x1 hasnt pluralname) print "s";
            print ".^";
    }
Pronouns: switch (n) {
        1:  print "At the moment, ";
        2:  print "means ";
        3:  print "is unset";
        4:  "no pronouns are known to the game.";
        5:  ".";
    }
Pull,Push,Turn: switch (n) {
        1:  if (x1 has pluralname) print "Those are "; else print "It is ";
            "fixed in place.";
        2:  "You are unable to.";
        3:  "Nothing obvious happens.";
        4:  "That would be less than courteous.";
        5: print (The) actor, " pulls ", (the) x1, ".^";
        6: print (The) actor, " pushes ", (the) x1, ".^";
        7: print (The) actor, " turns ", (the) x1, ".^";
    }
! Push: see Pull
PushDir: switch (n) {
        1:  print (The) x1, " cannot be pushed from place to place.^";
        2:  "That's not a direction.";
        3:  "Not that way you can't.";
    }
PutOn: switch (n) {
        1:  "You need to be holding ", (the) x1, " before you can put ",
                (itorthem) x1, " on top of something else.";
        2:  "You can't put something on top of itself.";
        3:  "Putting things on ", (the) x1, " would achieve nothing.";
        4:  "You lack the dexterity.";
        5:  print "(first taking ", (itorthem) x1, " off)^"; say__p = 0; return;
        6:  "There is no more room on ", (the) x1, ".";
        7:  "Done.";
        8:  "You put ", (the) x1, " on ", (the) second, ".";
        9:  print (The) actor, " puts ", (the) x1, " on ", (the) second, ".^";
    }
Quit: switch (n) {
        1:  print "Please answer yes or no.";
```

```
        2:  print "Are you sure you want to quit? ";
    }
Remove: switch (n) {
        1:  if (x1 has pluralname) print "They are"; else print "It is";
            " unfortunately closed.";
        2:  if (x1 has pluralname) print "But they aren't"; else print "But it isn't";
            " there now.";
        3:  "Removed.";
    }
Restart: switch (n) {
        1:  print "Are you sure you want to restart? ";
        2:  "Failed.";
    }
Restore: switch (n) {
        1:  "Restore failed.";
        2:  "Ok.";
    }
Rub:      "You achieve nothing by this.";
Save: switch (n) {
        1:  "Save failed.";
        2:  "Ok.";
    }
Score: switch (n) {
        1:  if (deadflag) print "In that game you scored "; else print "You have so far scored ";
            print score, " out of a possible ", MAX_SCORE, ", in ", turns, " turn";
            if (turns ~= 1) print "s";
            return;
        2:  "There is no score in this story.";
        3: print ", earning you the rank of ";
    }
ScriptOff: switch (n) {
        1:  "Transcripting is already off.";
        2:  "^End of transcript.";
        3:  "Attempt to end transcript failed.";
    }
ScriptOn: switch (n) {
        1:  "Transcripting is already on.";
        2:  "Start of a transcript of";
        3:  "Attempt to begin transcript failed.";
    }
Search: switch (n) {
        1:  "But it's dark.";
        2:  "There is nothing on ", (the) x1, ".";
        3:  print "On ", (the) x1, " ";
            WriteListFrom(child(x1), ENGLISH_BIT+TERSE_BIT+CONCEAL_BIT+ISARE_BIT);
            ".";
        4:  "You find nothing of interest.";
        5:  "You can't see inside, since ", (the) x1, " ", (isorare) x1, " closed.";
        6:  print_ret (The) x1, " ", (isorare) x1, " empty.";
        7:  print "In ", (the) x1, " ";
            WriteListFrom(child(x1), ENGLISH_BIT+TERSE_BIT+CONCEAL_BIT+ISARE_BIT);
            ".";
        8: print (The) actor, " searches ", (the) x1, ".^";
```

```
    }
SetTo:    "No, you can't set ", (thatorthose) x1, " to anything.";
Show: switch (n) {
        1:  "You aren't holding ", (the) x1, ".";
        2:  print_ret (The) x1, " ", (isorare) x1, " unimpressed.";
    }
Sing:     "Your singing is abominable.";
Sleep:    "You aren't feeling especially drowsy.";
Smell:    "You smell nothing unexpected.";
            #Ifdef DIALECT_US;
Sorry:    "Oh, don't apologize.";
            #Ifnot;
Sorry:    "Oh, don't apologise.";
            #Endif;
Squeeze: switch (n) {
        1:  "Keep your hands to yourself.";
        2:  "You achieve nothing by this.";
        3: print (The) actor, " squeezes ", (the) x1, ".^";
    }
Strong:   "Real adventurers do not use such language.";
Swing:    "There's nothing sensible to swing here.";
SwitchOff: switch (n) {
        1:  print_ret (ctheyreorthats) x1, " not something you can switch.";
        2:  print_ret (ctheyreorthats) x1, " already off.";
        3:  "You switch ", (the) x1, " off.";
        4: print (The) actor, " switches ", (the) x1, " off.^";
    }
SwitchOn: switch (n) {
        1:  print_ret (ctheyreorthats) x1, " not something you can switch.";
        2:  print_ret (ctheyreorthats) x1, " already on.";
        3:  "You switch ", (the) x1, " on.";
        4: print (The) actor, " switches ", (the) x1, " on.^";
    }
Take: switch (n) {
        1:  "Taken.";
        2:  "You are always self-possessed.";
        3:  "I don't suppose ", (the) x1, " would care for that.";
        4:  print "You'd have to get ";
            if (x1 has supporter) print "off "; else print "out of ";
            print_ret (the) x1, " first.";
        5:  "You already have ", (thatorthose) x1, ".";
        6:  if (noun has pluralname) print "Those seem "; else print "That seems ";
            "to belong to ", (the) x1, ".";
        7:  if (noun has pluralname) print "Those seem "; else print "That seems ";
            "to be a part of ", (the) x1, ".";
        8:  print_ret (Cthatorthose) x1, " ", (isorare) x1,
            "n't available.";
        9:  print_ret (The) x1, " ", (isorare) x1, "n't open.";
        10: if (x1 has pluralname) print "They're "; else print "That's ";
            "hardly portable.";
        11: if (x1 has pluralname) print "They're "; else print "That's ";
            "fixed in place.";
        12: "You're carrying too many things already.";
```

```
        13: print "(putting ", (the) x1, " into ", (the) SACK_OBJECT,
            " to make room)^"; say__p = 0; return;
        14: "You can't reach into ", (the) x1, ".";
        15: "You cannot carry ", (the) x1, ".";
        16: print (The) actor, " picks up ", (the) x1, ".^";
    }
Taste:    "You taste nothing unexpected.";
Tell: switch (n) {
        1:  "You talk to yourself a while.";
        2:  "This provokes no reaction.";
    }
Think:    "What a good idea.";
ThrowAt: switch (n) {
        1:  "Futile.";
        2:  "You lack the nerve when it comes to the crucial moment.";
    }
Tie: "You would achieve nothing by this.";
Touch: switch (n) {
        1:  "Keep your hands to yourself!";
        2:  "You feel nothing unexpected.";
        3:  "If you think that'll help.";
        4: print (The) actor, " touches ", (himheritself) x1, ".^";
        5: print (The) actor, " touches you.^";
        6: print (The) actor, " touches ", (the) x1, ".^";
    }
! Turn: see Pull.
Unlock:  switch (n) {
        1:  if (x1 has pluralname) print "They don't "; else print "That doesn't ";
            "seem to be something you can unlock.";
        2:  print_ret (ctheyreorthats) x1, " unlocked at the moment.";
        3:  if (x1 has pluralname) print "Those don't "; else print "That doesn't ";
            "seem to fit the lock.";
        4:  "You unlock ", (the) x1, ".";
        5: print (The) actor, " unlocks ", (the) x1, ".^";
    }
Verify: switch (n) {
        1:  "The game file has verified as intact.";
        2:  "The game file did not verify as intact, and may be corrupt.";
    }
Wait: switch (n) {
        1:  "Time passes.";
        2: print (The) actor, " waits.^";
    }
Wake:     "The dreadful truth is, this is not a dream.";
WakeOther:"That seems unnecessary.";
Wave: switch (n) {
        1:  "But you aren't holding ", (thatorthose) x1, ".";
        2:  "You look ridiculous waving ", (the) x1, ".";
        3: print (The) actor, " waves ", (the) x1, ".^";
    }
WaveHands:"You wave, feeling foolish.";
Wear: switch (n) {
        1:  "You can't wear ", (thatorthose) x1, "!";
```

```
        2:  "You're not holding ", (thatorthose) x1, "!";
        3:  "You're already wearing ", (thatorthose) x1, "!";
        4:  "You put on ", (the) x1, ".";
        5: print (The) actor, " puts on ", (the) x1, ".^";
    }
! Yes:  see No.
];
```

§**13. Printing Mechanism.** The following routine produces a "library message" – though the library is no more, the terminology lives on.

The `L__M` routine is designed to reach up into I7 to offer it a chance to intervene, but then go back to the I6 method if it doesn't. The Standard Rules ordinarily define these three routines as stubs which always return false, so by default there's no intervention. (This is the hook for the new model of library messages which will be introduced in future builds.)

We divide into three cases because `##Miscellany` and `##ListMiscellany` are fake actions, not actions, in I6. This means it would not be type-safe to store them in I7 variables whose kind of value is "action-name", and that would make any single I7 routine handling all three kinds of message quite difficult to write.

```
[ L__M act n x1 x2 rv flag;
    @push sw__var;
    sw__var = act;
    if (n == 0) n = 1;
    @push action;
    lm_act = act;
    lm_n = n;
    lm_o = x1;
    lm_o2 = x2;
    switch (act) {
        ##Miscellany: rv = (+ whether or not intervened in miscellaneous message +);
        ##ListMiscellany: rv = (+ whether or not intervened in miscellaneous list message +);
        default: rv = (+ whether or not intervened in action message +);
    }
    action = sw__var;
    if (rv == false) rv = RunRoutines(LibraryMessages, before);
    @pull action;
    if (rv == false) LanguageLM(n, x1, x2);
    @pull sw__var;
];
```

MStack Template B/stackt

Purpose

To allocate space on the memory stack for frames of variables to be used by rulebooks, activities and actions.

---

§**1. The Memory Stack.** The M-Stack, or memory stack, is a sequence of frames, piled upwards. If we had an accessible stack in memory, we could use that, but neither the Z-machine nor Glulx has such a stack, alas, alas, alas. The following is not a very good solution, but it just about works.

```
Constant MAX_MSTACK_FRAME = 2 + {-value:max_frame_size_needed};
Constant MSTACK_CAPACITY = 20;
Constant MSTACK_SIZE = MSTACK_CAPACITY*MAX_MSTACK_FRAME;
Array MStack --> MSTACK_SIZE;
Global MStack_Top = 0; ! Topmost word currently used
```

§**2. Create Frame.** A frame is created by calling the following function with two arguments: `creator`, a function which initialises a block of variables, and an ID number identifying the owner.

The `creator` function is called with the address at which to initialise the variables as its first argument, and the value 1 as the second argument. (The idea is that the same function can be used later to deallocate the variables, and then the second argument will be -1.) The `creator` function returns the extent of the block of memory it has used, in words. Thus is required to be strictly less than `MAX_MSTACK_FRAME` minus 1.

```
[ Mstack_Create_Frame creator id extent;
    if (creator == 0) rfalse;
    extent = creator.call(MStack_Top+2, 1);
    if (extent == 0) rfalse;
    if (MStack_Top + MAX_MSTACK_FRAME >= MSTACK_SIZE + 2) {
        RunTimeProblem(RTP_MSTACKMEMORY, MSTACK_SIZE);
        Mstack_Backtrace();
        rfalse;
    }
    MStack_Top++;
    MStack-->MStack_Top = id;
    MStack_Top++;
    MStack_Top = MStack_Top + extent;
    MStack-->MStack_Top = -(extent+2);
    rtrue;
];
```

§**3. Destroy Frame.** As sketched above, the same creator function and ID number are passed to the following routine to destroy the frame again. It takes the stack down to the level of the most recently created frame with this ID number: note that each action, for instance, has its own ID number for this purpose, but can be taking place several times in a nested fashion – one taking action might have caused another taking action which caused a third, for instance, so that there are three incomplete taking actions at once. In that case, there will be three independent sets of taking action variables on the M-stack, all with the same ID number. We remove the topmost one: the implication of that is that frames must always be destroyed in reverse order of creation.

In practice, I7 uses frames such that the frame sought should always be the topmost one in any case, and so that frames are always explicitly destroyed, not wiped by being undercut when an earlier-created frame is destroyed.

```
[ Mstack_Destroy_Frame creator id pos;
    pos = Mstack_Seek_Frame(id);
    if (pos == 0) rfalse; ! Not found: do nothing
    MStack_Top = pos - 2; ! Clear mstack down to just below this frame
    if (creator) creator.call(pos, -1);
    rtrue;
];
```

§**4. Seek Frame.** We return the position on the M-stack of the most recently created frame with the given ID number (see above), or 0 if no such frame exists; the size is stored in the global variable `MStack_Frame_Extent`. (Because word 0 on the stack is used as a sentinel – all frames are placed above it – no frame can actually begin at word 0 on the stack, so 0 is safe to use as an exception.)

```
Global MStack_Frame_Extent = 0;

[ Mstack_Seek_Frame id pos;
    pos = MStack_Top;
    while (MStack-->pos ~= 0) {
        MStack_Frame_Extent = MStack-->pos;
        pos = pos + MStack_Frame_Extent;
        MStack_Frame_Extent = (-2) - MStack_Frame_Extent;
        if (MStack-->(pos+1) == id) return pos+2;
    }
    MStack_Frame_Extent = 0;
    return 0; ! Not found
];
```

§**5. Backtrace.** Purely for debugging purposes, and giving feedback if the stack runs out of memory:

```
[ Mstack_Backtrace pos k;
    print "Mstack backtrace: size ", MStack_Top+1, " words^";
    pos = MStack_Top;
    while (MStack-->pos ~= 0) {
        MStack_Frame_Extent = MStack-->pos;
        pos = pos + MStack_Frame_Extent;
        MStack_Frame_Extent = (-2) - MStack_Frame_Extent;
        print "Block at ", pos+2,
            " owner ID ", MStack-->(pos+1), " size ", MStack_Frame_Extent, "^";
        for (k=0: k<MStack_Frame_Extent: k++) print MStack-->(pos+2+k), " ";
        print "^";
    }
];
```

§**6. Access to Variables.** An M-stack variable is identified by a combination of ID number and offset: for instance ID 20007, offset 1, is the variable "room gone to" belonging to the going action. The following routine converts that into an address on the M-stack, in the topmost block with the given ID number (since "room gone to", for instance, always means its value in the most current going action of those now under way). Typechecking in the compiler should mean that it is impossible to produce either error message below: NI will only compile valid uses of `MstVO` ("M-stack variable offset") where the seek succeeds and the offset is within range.

```
[ MstVO id off pos;
    pos = Mstack_Seek_Frame(id);
    if (pos == 0) {
        print "Variable unavailable: ", id, "/", off, "^";
        rfalse;
    }
    if ((off<0) || (off >= MStack_Frame_Extent)) {
        print "Variable stack offset wrong: ", id, "/", off, " at ", pos, "^";
        rfalse;
    }
    return pos+off;
];
```

§**7. Access to Nonexistent Variables.** A long-standing point where I7 is not as strict in type-checking as it might be occurs when checking rule preambles like "Before going to a dead end...". Such a preamble must be checked whatever the current action is – in many cases, it will not be a going action at all; which means that "room gone to", a value implied by the "to" clause, will not exist. If the type-checking were stricter, it would be a nuisance for authors, and instead we relax a little by accessing such variables using a more forgiving routine. Here, if a variable does not exist, we return 0 to mean that it can be read at M-stack position 0: this is the sentinel word, which is not part of any frame, and which contains 0. Thus the variable reads as if it is 0, the default for the kind of value "object", which is the KOV for action variables such as "room gone to".

The routine may only be used where the variable is being read, and never where it is to be written, of course: that would corrupt the sentinel.

```
[ MstVON id off pos;
    pos = Mstack_Seek_Frame(id);
    if (pos == 0) {
        return 0; ! word position 0 on the M-stack
    }
    if ((off<0) || (off >= MStack_Frame_Extent)) {
        print "Variable stack offset wrong: ", id, "/", off, " at ", pos, "^";
        rfalse;
    }
    return pos+off;
];
```

§**8. Rulebook Variables.** Each rulebook has a slate of variables, usually empty, with ID number the same as the rulebook's own ID number. (Rulebook IDs number upwards from 0 in order of creation in the source text.) The associated creator functions, usually null, are stored in an array.

```
{-array:rulebook_var_creators}
[ MStack_CreateRBVars rb cr;
    cr = rulebook_var_creators-->rb;
    if (cr == 0) return;
    Mstack_Create_Frame(cr, rb);
];
[ MStack_DestroyRBVars rb cr;
    cr = rulebook_var_creators-->rb;
    if (cr == 0) return;
    Mstack_Destroy_Frame(cr, rb);
];
```

§**9. Activity Variables.** Exactly the same goes for activity variables except that here the ID number is $10000 + N$, where N is the allocation ID of the activity. (This would fail if there were more than 10,000 rulebooks, but this is very difficult to see happening.)

```
{-array:activity_var_creators}
[ MStack_CreateAVVars av cr;
    cr = activity_var_creators-->av;
    if (cr == 0) return;
    Mstack_Create_Frame(cr, av + 10000);
];
[ MStack_DestroyAVVars av cr;
    cr = activity_var_creators-->av;
    if (cr == 0) return;
    Mstack_Destroy_Frame(cr, av + 10000);
];
```

Chronology Template B/chrt

Purpose

To record information now which will be needed later, when a condition phrased in the perfect tense is tested.

---

§**1. Scheme I.** If source text contains a condition like "if the well has been dry, ...", then we need to keep a chronological record by testing at every turn whether or not the well is dry: we log whether this is true now, whether it has ever been true, for how many consecutive turns it has been true (to a maximum of 127), and how many times it has become true having just been false (again, to a maximum of 127 times). All this information is packed into a single word, arranged as a 15-bit bitmap: the least significant bit is the state of the condition now (true or false), the next 7 bits are the number of false-to-true "trips" observed over time, and the top 7 bits are the number of consecutive turns on which the condition has been true ("consecutives"). The 16th and most significant bit is unused, so that the state is always positive even in a 16-bit virtual machine – a convenience since we then don't need to worry about the effect of signed division and remainder on the bitmap.

There is no need to store a flag for "has this condition ever been true", because this is equivalent the number of trips being greater than zero. This might look wrong for a condition which is true at start of play – say, if the well was always dry – because then its state has never changed from false to true. But in fact when the VM starts up the state word is initially always 0: it's only when the startup rulebook fires the update chronological records rule (see below) that we first test whether the well is dry, and that forces a trip from false to true if the well is dry at start of play. Therefore, if T is the number of trips for the condition then $T = 0$ if and only if the condition has been false from the very start of play. If the condition is true now, then $T = 1$ if and only if it has always been so.

§**2. Present and Past.** Each individual condition has its own unique "PT number", counting upwards from 0 in order of compilation by NI, and a "chronological record" is a word array with a state word as described above for each PT number.

However, we keep not one but two chronological records: one for the situation now, called the "present chronological record", and one for the situation as it was just before interesting things most recently happened, called the "past chronological record". If one of those interesting things was that the well ran dry, then in our example the state word for the condition "if the well has been dry" would be different in the two chronological records. We keep two records in order to be able to detect conditions in four different tenses:

(1) Present tense ("if the well is dry"): none of this machinery is used, because we can just test directly instead, so a present tense condition has no PT-number.
(2) Past tense ("if the well was dry"): we look at the flag bit in the past chronological record.
(3) Perfect tense ("if the well has been dry"): we look at whether or not $T > 0$ in the present chronological record.
(4) Past perfect tense ("if the well had been dry"): we look at whether or not $T > 0$ in the past chronological record.

It's a somewhat ambiguous matter of context in English when the reference time is for past tenses. If somebody comes out into the sunshine and says, "But it was raining," when does he mean? We would reasonably guess earlier that day, and probably only an hour or two ago, but that's a contextual judgement made on the basis of our own experience of how rapidly weather changes. The context for Inform source text is always that of actions, so our convention is that the reference time for past tenses is the point just before the current action began. Such key moments – when things are just about to happen – are called "chronology points". There is a CP just before each action begins; there is a CP in the startup process; and there is a

CP at the end of each turn, for good measure. CPs happen when somebody calls `ChronologyPoint()`. The action machinery does this directly, while the startup CP and the CP at end of turn happen in the course of the update chronological records rule (below).

§**3. Chronology Point.** This is when the time of reference for past tenses is now: so it is where the past becomes the present. Soon, exciting things will happen, and the present will go on developing, while the past will remain as it was; until things calm down again and we come to another chronology point.

```
[ ChronologyPoint pt;
    for (pt=0:pt<NO_PAST_TENSE_CONDS:pt++)
        past_chronological_record-->pt = present_chronological_record-->pt;
];
```

§**4. Update Chronological Records Rule.** It might seem odd that a routine to, supposedly, update something would only call another routine called `TestSinglePastState`: but in this setting, any test updates the state, because it changes the number of times something has been found true, and so on.

```
[ UPDATE_CHRONOLOGICAL_RECORDS_R pt;
    for (pt=0: pt<NO_PAST_TENSE_CONDS: pt++) TestSinglePastState(false, pt, true, -1);
    ChronologyPoint();
    rfalse;
];
```

§**5. Test Single Past State.** `TestSinglePastState` is called with four arguments:

(a) `past_flag` is true if we want to test a condition like "if the well was dry" or "if the well had been dry", which concern only the state as it was at the last chronology point – in other words, the `past_chronological_record`, which we must not change; whereas `past_flag` is false if we want to test a present tense like "if the well is dry" or "if the well has been dry", because then we deal with the `present_chronological_record` which must be kept continuously updated.

(b) `pt` is the PT number for the condition.

(c) `turn_end` is true if we are making the test at the end of a turn, as part of the update chronological records rule, and false otherwise. (For these purposes the start of play is a turn end since the UCRR runs then, too.)

(d) `wanted` describes what information the function should return, as detailed in its code below.

```
[ TestSinglePastState past_flag pt turn_end wanted
    old new trips consecutives ct_0 ct_1;
    if (past_flag) {
        new = (past_chronological_record-->pt) & 1;
        trips = ((past_chronological_record-->pt) & $$11111110)/2;
        consecutives = ((past_chronological_record-->pt) & $$111111100000000)/256;
    } else {
        old = (present_chronological_record-->pt) & 1;
        trips = ((present_chronological_record-->pt) & $$11111110)/2;
        consecutives = ((present_chronological_record-->pt) & $$111111100000000)/256;
        switch(pt) {
! Test cases for conditions by PT number: each sets "new" to whether it is true or false now
{-call:past_tenses_i6_escape}
        }
        if (new == false) {
            consecutives = 0;
```

```
        } else {
            if (old == false) { trips++; if (trips > 127) trips = 127; }
            if (turn_end) { consecutives++; if (consecutives > 127) consecutives = 127; }
        }
        ! print pt,":o=",old," n=",new," t=",trips," c=",consecutives,"^";
        present_chronological_record-->pt = new + 2*trips + 256*consecutives;
    }
    switch(wanted) {
        0: if (new) return new;
        1: if (new) return trips;
        2: if (new) return consecutives;
        4: return new;
        5: return trips;
        6: return consecutives;
    }
    return 0;
];
```

§**6. Scheme II.** Actions discussed in the past tense – "if we have taken the ball" or, more subtly, "Instead of waiting for the third turn" (which also refers to the past) – are handled in a related but simpler way. NI counts such references, just as with other past tense conditions above, but this time stores a state in two simple word arrays: `TimesActionHasHappened` and `TurnsActionHasBeenHappening`.

It might reasonably be asked why we don't simply use all of the clever machinery above: why have an entirely different system here? One reason is that actions are events and not continuous states of being. "The well is dry" could be true for any extent of time, but "taking the ball" either happens at a given moment or doesn't: it is not continuously true. We are therefore in the business of counting events, not measuring durations. Another reason is that the point of reference for past tenses is different. It makes no sense to say "if we were looking" because that would mean at a time just before the current action, when actions were probably not happening at all; while "if we had looked" and "if we have looked" would almost always be identical in meaning for the same reason. So we need a much simpler system with just one possible past tense; we don't need to keep two different states; and we use the extra storage to enable us to count the number of times, and the number of turns, to at least 31767 instead of stopping at 127. (We also generate more legible code to test past tense actions.)

§**7. Past Action Routines.** Actions can be quite complicated to test even in the present, and this is much more conveniently done in a routine with its own NI-generated stack frame; so each past tense action has its own testing routine, with a name like `PAPR_41`, which returns true or false depending on whether the action is going on now. The word array `PastActionsI6Routines` then contains a null-terminated list of these routines.

```
{-call:past_actions_i6_routines}
```

§**8. Track Actions.** The routine `TrackActions` then updates the two arrays, and is the equivalent for past tense actions of `ChronologyPoint`. It's called twice for each action: `TrackActions(false)` just as a new action is about to begin, and then `TrackActions(true)` just after it has finished – note in particular that if action B happens in the middle of action A, for instance because of a try phrase, then the `TrackActions(true)` call for B happens when A is back as the current action. In fact, that's the point: because it tells us that B was not the main action for the turn, but only a phase which has now passed again.

For each of the action patterns we track, we need to count the number of times such an action has been tried (*not* the number of times it succeeded), and also the number of consecutive turns on which it has been "the" action. The count of the number of tries is unambiguous and easy to keep: it is incremented at the start-of-action call only, and only if the action matches. But the consecutive turn count is more problematic. The user wants to think of actions and turns as synonymous, and to write conditions like "Before going for the third turn", and we want to play along because that's a natural phrasing: but actions and turns are not synonymous at all. The rules are therefore:

(1) At the start-of-action call, provided the action is not a silent one, the consecutive turns count is either incremented or zeroed, depending on whether the action matches. Silent actions are ignored because they are almost always knock-on actions tried in the course of other actions, and are certainly not the "main" action of the turn. But the count can be incremented only once in the course of each turn, so that "Instead of taking something for the third turn: ..." cannot happen on the very first turn because TAKE ALL causes three or more take actions.
(2) At the readjustment call, provided the action is not a silent one, the consecutive turns count is zeroed if the action does not match.

Why do we zero the count in rule (2)? Well, suppose that the player types OPEN BOX, so that the opening action takes place, and that it causes a further (non-silent) action, say examining the lid: and suppose that the action pattern we track the turn count for is "examining something". Then the following calls take place:

(a) `TrackActions(false)` while the action is "opening the box". Turn count zeroed by rule (1).
(b) `TrackActions(false)` while the action is "examining the lid". Turn count incremented by rule (1).
(c) `TrackActions(true)` while the action is "opening the box". Turn count zeroed by rule (2).
(d) `TrackActions(true)` while the action is "opening the box". Turn count zeroed by rule (2).

Thus rule (2) prevents us from counting this turn as the first of a sequence of turns in which examining something was the action, because it wasn't the main action of the turn.

A further complication is that we need to record the qualification, or not, even of silent actions, because testing rules like

> Instead of taking the top hat less than three times...

works by checking that (a) we are currently taking the top hat, and (b) have done so fewer than three times before. (b) uses the turn count described above; but (a) cannot simply look to see if that count is positive, since the action might be happening silently and thus not have contributed to the count; so we also record a flag to hold whether the action seems to be happening in this turn, silent or not.

```
[ TrackActions readjust ct_0 ct_1 i;
    for (i=0: PastActionsI6Routines-->i: i++) {
        if ((PastActionsI6Routines-->i).call()) {
            ! Yes, the current action matches action pattern i:
            if (readjust) continue;
            (TimesActionHasHappened-->i)++;
            if (LastTurnActionHappenedOn-->i ~= turns + 5) {
                LastTurnActionHappenedOn-->i = turns + 5;
                ActionCurrentlyHappeningFlag->i = 1;
                if (keep_silent == false)
                    (TurnsActionHasBeenHappening-->i)++;
            }
        } else {
```

```
            ! No, the current action doesn't match action pattern i:
            if (keep_silent == false) { TurnsActionHasBeenHappening-->i = 0; }
            if (LastTurnActionHappenedOn-->i ~= turns + 5)
                ActionCurrentlyHappeningFlag->i = 0;
        }
    }
];
```

§**9. Storage.** The necessary array allocation.

```
{-call:chronology_extents_i6_escape}
Array TimesActionHasHappened-->(NO_PAST_TENSE_ACTIONS+1);
Array TurnsActionHasBeenHappening-->(NO_PAST_TENSE_ACTIONS+1);
Array LastTurnActionHappenedOn-->(NO_PAST_TENSE_ACTIONS+1);
Array ActionCurrentlyHappeningFlag->(NO_PAST_TENSE_ACTIONS+1);
Array past_chronological_record-->(NO_PAST_TENSE_CONDS+1);
Array present_chronological_record-->(NO_PAST_TENSE_CONDS+1);
```

Printing Template B/print

Purpose

To manage the line skips which space paragraphs out, and to handle the printing of names of objects, pieces of text and numbers.

---

§**1. Paragraph Control.** Ah, yes: the paragraph breaking algorithm. In *TEX: The Program*, Donald Knuth writes at §768: "It's sort of a miracle whenever `\halign` and `\valign` work, because they cut across so many of the control structures of TEX." It's sort of a miracle whenever Inform 7's paragraph breaking system works, too. Most users probably imagine that it's implemented by having I7 look at where the cursor currently is (at the start of a line or not) and whether a line has just been skipped. In fact, the virtual machines simply do not offer facilities like that, and so we have to use our own book-keeping. Given the huge number of ways in which text can be printed, this is a delicate business. For some years now, "spacing bugs" – those where a spurious extra skipped line appears in a paragraph break, or where, conversely, no line is skipped at all – have been the least welcome in the Inform bugs database.

The basic method is to set `say__p`, the paragraph flag, when we print any matter; every so often we reach a "divide paragraph" point – for instance when one rule has finished and before another is about to start – and at those positions we look for `say__p`, and print a skipped line (and clear `say__p` again) if we find it. Thus:

```
> WAIT
The clock ticks ominously. ...first rule
    ...skipped line printed at a Divide Paragraph point
Mme Tourmalet rises from her chair and slips out. second rule
    ...skipped line printed at a Divide Paragraph point
>
```

A divide paragraph point occurs between any two rules in an action rulebook, but not an activity rulebook: many activities exist to print text, such as the names of objects, and there would be wild spacing accidents if paragraphs were divided there. Inform places DPPs elsewhere, too: and the text substitution "[conditional paragraph break]" allows the user to place one anywhere.

A traditional layout convention handed down from Infocom makes an exception of the first paragraph to appear after the prompt, but only in one situation. Ordinarily, the first paragraph of any turn appears straight after the prompt:

```
> EXAMINE DOG
Mme Tourmalet's borzoi looks as if it means fashion, not business.
```

The command is echoed on screen as the player types, but this doesn't set the paragraph flag, which is still clear when the text "Mme Tourmalet's..." begins to be printed. The single exception occurs when the command calls for the player to go to a new location, when a skipped line is printed before the room description for the new room. Thus:

```
> SOUTH
    ...the "going look break"
Rocky Beach
```

(Note that this is not inherent in the looking action:

```
> LOOK
Rocky Beach
```

...which obeys the standard paragraphing conventions.)

So much for automatic paragraph breaks. However, we need a variety of different ways explicitly to control paragraphs, in order to accommodate traditional layout conventions handed down from Infocom.

The simplest exceptional kind of paragraph break is a "command clarification break", in which a single new-line is printed but there is no skipped line: as the name implies, it's traditionally used when a command such as OPEN DOOR is clarified. For example:

```
(first unlocking the oak door) ...now a command clarification break
You open the oak door.
```

This is not quite the same thing as a "run paragraph on" break, in which we also deliberately suppress the skipped line, but make an exception for the skipped line which ought to appear last before the prompt: the idea is to merge two or more paragraphs together.

```
> TAKE ALL
marmot: Taken. ...we run paragraph on here
weasel: Taken. ...and also here
     ...despite which the final skip does occur
>  ...before the next prompt
```

A more complicated case is "special look spacing", used for the break which occurs after the (boldface) short name of a room description is printed. This is tricky because it is sometimes followed directly by a long description, and we don't want a skipped line:

```
Villa Christiane ...a special look spacing break
The walled garden of a villa in Cap d'Agde.
     ...a Divide Paragraph break
Mme Tourmalet's borzoi lazes in the long grass.
```

But sometimes it is followed directly by a subsequent paragraph, and again we want no skip:

```
Villa Christiane ...a special look spacing break
Mme Tourmalet's borzoi lazes in the long grass.
```

And sometimes it is the only content of the room description and is followed only by the prompt:

```
Villa Christiane ...a special look spacing break
     ...a break inserted before the prompt
>
```

To recap, we have five kinds of paragraph break:

(a) Standard breaks at "divide paragraph points", used between rules.
(b) The "going look break", used before the room description after going to a new room.
(c) A "command clarification break", used after text clarifying a command.
(d) A "run paragraph on" break, used to merge multiple paragraphs into a single block of text.
(e) The "special look spacing" break, used after the boldface headline of a room description.

We now have to implement all of these behaviours. The code, while very simple, is highly prone to behaving unexpectedly if changes are made, simply because of the huge number of circumstances in which paragraphs are printed: so change nothing without very careful testing.

§**2. State.** The current state is stored in a combination of two global variables:

(1) `say__p`, the "say paragraph" flag, which is set if a paragraph break needs to be printed before the next text can begin;

(2) `say__pc`, originally named as the "paragraph completed" flag, but which is now a bitmap:

(2a) `PARA_COMPLETED` is set if a standard paragraph break has been made since the last time the flag was cleared;

(2b) `PARA_PROMPTSKIP` is set to indicate that the current printing position does not follow a skipped line, and that further material is expected which will run on from the previous paragraph, but that if no further material turns up then a skipped line would be needed before the next prompt;

(2c) `PARA_SUPPRESSPROMPTSKIP` is set to indicate that, despite `PARA_PROMPTSKIP` being set, no skipped line is needed before the prompt after all;

(2d) `PARA_NORULEBOOKBREAKS` suppresses divide paragraph points in between rules in rulebooks; it treats all rulebooks, and in particular action rulebooks, the way activity rulebooks are treated. (The flag is used for short periods only and never across turn boundaries, prompts and so on.)

(2e) `PARA_CONTENTEXPECTED` is set after a paragraph division as a signal that if any contents looks likely to be printed soon then `say__p` needs to be set, because a successor paragraph will then have started. This is checked by calling `ParaContent()` – while it's slow to have to call this routine so often, that's better than compiling inline code with the same effect, because minimising compiled code size is more important, and speed is never a big deal when printing.

Not all printing is to the screen: sometimes the output is to a file, or to memory, and in that case we want to start the switched output at a clear paragraphing state and then go back to the screen afterwards without any sign of change. The correct way to do this is to push the `say__p` and `say__pc` variables onto the VM stack and call `ClearParagraphing()` before starting to print to the new stream, and then pull the variables back again before resuming printing to the old stream.

In no other case should any code alter `say__pc` except via the routines below.

```
!Constant TRACE_I7_SPACING;

[ ClearParagraphing;
    say__p = 0; say__pc = 0;
];

[ DivideParagraphPoint;
    #ifdef TRACE_I7_SPACING; print "[DPP", say__p, say__pc, "]"; #endif;
    if (say__p) {
        new_line; say__p = 0; say__pc = say__pc | PARA_COMPLETED;
        if (say__pc & PARA_PROMPTSKIP) say__pc = say__pc - PARA_PROMPTSKIP;
        if (say__pc & PARA_SUPPRESSPROMPTSKIP) say__pc = say__pc - PARA_SUPPRESSPROMPTSKIP;
    }
    #ifdef TRACE_I7_SPACING; print "[-->", say__p, say__pc, "]"; #endif;
    say__pc = say__pc | PARA_CONTENTEXPECTED;
];

[ ParaContent;
    if (say__pc & PARA_CONTENTEXPECTED) {
        say__pc = say__pc - PARA_CONTENTEXPECTED;
        say__p = 1;
    }
];

[ GoingLookBreak;
    if (say__pc & PARA_COMPLETED == 0) new_line;
    ClearParagraphing();
];

[ CommandClarificationBreak;
```

```
    new_line;
    ClearParagraphing();
];
[ RunParagraphOn;
    #ifdef TRACE_I7_SPACING; print "[RPO", say__p, say__pc, "]"; #endif;
    say__p = 0;
    say__pc = say__pc | PARA_PROMPTSKIP;
    say__pc = say__pc | PARA_SUPPRESSPROMPTSKIP;
];
[ SpecialLookSpacingBreak;
    #ifdef TRACE_I7_SPACING; print "[SLS", say__p, say__pc, "]"; #endif;
    say__p = 0;
    say__pc = say__pc | PARA_PROMPTSKIP;
];
[ EnsureBreakBeforePrompt;
    if ((say__p) ||
        ((say__pc & PARA_PROMPTSKIP) && ((say__pc & PARA_SUPPRESSPROMPTSKIP)==0)))
        new_line;
    ClearParagraphing();
];
[ PrintSingleParagraph matter;
    say__p = 1;
    say__pc = say__pc | PARA_NORULEBOOKBREAKS;
    PrintText(matter);
    DivideParagraphPoint();
    say__pc = 0;
];
```

§**3. Say Number.** The global variable `say__n` is set to the numerical value of any quantity printed, and this is used for the text substitution "[s]", so that "You have been awake for [turn count] turn[s]." will expand correctly.

```
[ STextSubstitution;
    if (say__n ~= 1) print "s";
];
```

§**4. Prompt.** This is the text printed just before we wait for the player's command: it prompts him to type.

```
[ PrintPrompt i;
    style roman;
    EnsureBreakBeforePrompt();
    PrintText( (+ command prompt +) );
    ClearBoxedText();
    ClearParagraphing();
    enable_rte = true;
];
```

§**5. Boxed Quotations.** These appear once only, and happen outside of the paragraphing scheme: they are normally overlaid as windows on top of the regular text. We can request one at any time, but it will appear only at prompt time, when the screen is fairly well guaranteed not to be scrolling. (Only fairly well since it's just possible that *Border Zone*-like tricks with real-time play might be going on, but whatever happens, there is at least a human-appreciable pause in which the quotation can be read before being taken away again.)

```
Global pending_boxed_quotation; ! a routine to overlay the quotation on screen
[ DisplayBoxedQuotation Q;
    pending_boxed_quotation = Q;
];
[ ClearBoxedText i;
    if (pending_boxed_quotation) {
        for (i=0: Runtime_Quotations_Displayed-->i: i++)
            if (Runtime_Quotations_Displayed-->i == pending_boxed_quotation) {
                pending_boxed_quotation = 0;
                return;
            }
        Runtime_Quotations_Displayed-->i = pending_boxed_quotation;
        ClearParagraphing();
        pending_boxed_quotation();
        ClearParagraphing();
        pending_boxed_quotation = 0;
    }
];
```

§**6. Score Notification.** This doesn't really deserve to be at I6 level at all, but since for traditional reasons we need to use conditional compilation on `NO_SCORING`, and since we want a fancy text style for Glulx, ...

```
[ NotifyTheScore;
#Ifndef NO_SCORING;
    if (notify_mode == 1) {
        DivideParagraphPoint();
        VM_Style(NOTE_VMSTY);
        print "["; L__M(##Miscellany, 50, score-last_score); print ".]^";
        VM_Style(NORMAL_VMSTY);
    }
#Endif;
];
```

§**7. Status Line.** Status line printing happens on the upper screen window, and outside of the paragraph control system.

Support for version 6 of the Z-machine is best described as grudging. It requires a heavily rewritten `DrawStatusLine` equivalent, to be found in "ZMachine.i6t".

```
#Ifdef TARGET_ZCODE;
#Iftrue (#version_number == 6);
[ DrawStatusLine; Z6_DrawStatusLine(); ];
#Endif;
#Endif;

#Ifndef DrawStatusLine;
[ DrawStatusLine width posb;
    @push say__p; @push say__pc;
    BeginActivity(CONSTRUCTING_STATUS_LINE_ACT);
    VM_StatusLineHeight(1); VM_MoveCursorInStatusLine(1, 1);
    if (statuswin_current) {
        width = VM_ScreenWidth(); posb = width-15;
        spaces width;
        ClearParagraphing();
        if (ForActivity(CONSTRUCTING_STATUS_LINE_ACT) == false) {
            VM_MoveCursorInStatusLine(1, 2);
            switch(metaclass(left_hand_status_line)) {
                String: print (string) left_hand_status_line;
                Routine: left_hand_status_line();
            }
            VM_MoveCursorInStatusLine(1, posb);
            switch(metaclass(right_hand_status_line)) {
                String: print (string) right_hand_status_line;
                Routine: right_hand_status_line();
            }
        }
        VM_MoveCursorInStatusLine(1, 1); VM_MainWindow();
    }
    ClearParagraphing();
    EndActivity(CONSTRUCTING_STATUS_LINE_ACT);
    @pull say__pc; @pull say__p;
];
#Endif;
```

§**8. Status Line Utilities.** Two convenient routines for the default values of `right_hand_status_line` and `left_hand_status_line` respectively. `SL_Location` also implements the text substitution "[player's surroundings]".

```
[ SL_Score_Moves;
    if (not_yet_in_play) return;
    #ifdef NO_SCORING; print sline2; #ifnot; print sline1, "/", sline2; #endif;
];
[ SL_Location;
    if (not_yet_in_play) return;
    if (location == thedark) {
        BeginActivity(PRINTING_NAME_OF_DARK_ROOM_ACT);
        if (ForActivity(PRINTING_NAME_OF_DARK_ROOM_ACT) == false)
            L__M(##Miscellany, 71);
        EndActivity(PRINTING_NAME_OF_DARK_ROOM_ACT);
    } else {
        FindVisibilityLevels();
        if (visibility_ceiling == location) print (name) location;
        else print (The) visibility_ceiling;
    }
];
```

§**9. Banner.** Note that NI always compiles `Story` and `Headline` texts, but does not always compile a `Story_Author`.

```
[ Banner;
BeginActivity(PRINTING_BANNER_TEXT_ACT);
if (ForActivity(PRINTING_BANNER_TEXT_ACT) == false) {
        VM_Style(HEADER_VMSTY);
        print (string) Story;
        VM_Style(NORMAL_VMSTY);
        new_line;
        print (string) Headline;
        #ifdef Story_Author;
        print " by ", (string) Story_Author;
        #endif; ! Story_Author
        new_line;
        VM_Describe_Release();
        print " / Inform 7 build ", (string) NI_BUILD_COUNT, " ";
        print "(I6/v"; inversion;
        print " lib ", (string) LibRelease, ") ";
        #Ifdef STRICT_MODE;
        print "S";
        #Endif; ! STRICT_MODE
        #Ifdef DEBUG;
        print "D";
        #Endif; ! DEBUG
        new_line;
    }
    EndActivity(PRINTING_BANNER_TEXT_ACT);
];
```

§10. Print Decimal Number. `DecimalNumber` is a trivial function which just prints a number, in decimal digits. It is left over from the I6 library's support routines for Glulx, where it was intended as a stub to pass to the Glulx `Glulx_PrintAnything` routine (which I7 does not use). In I7, however, it's also used as the default printing routine for new kinds of value.

```
[ DecimalNumber num; print num; ];
```

§11. Print English Number. Another traditional name, this: in fact it prints the number as text in whatever is the current language of play.

```
[ EnglishNumber n; LanguageNumber(n); ];
```

§12. Print Text. The routine for printing an I7 "text" value, which might text with or without substitutions.

```
[ PrintText x;
    if (x ofclass String) print (string) x;
    if (x ofclass Routine) (x)();
];
[ I7_String x; PrintText(x); ]; ! An alternative name now used only by extensions
```

§13. Print Or Run. This utility remains from the old I6 library: it essentially treats a property as textual and prints it where possible. Where the `no_break` flag is set, we expect the text to form only a small part of a paragraph, and it's inappropriate to break here: for instance, for printing the "printed name" of an object. Where the flag is clear, however, the text is expected to form its own paragraph.

Where `PrintOrRun` is used in breaking mode, which is only for a very few properties in I7 (indeed at present only `initial` and `description`), the routine called is given the chance to decide whether to print or not. It should return `true` or `false` according to whether it did so; this allows us to divide the paragraph or not accordingly.

```
[ PrintOrRun obj prop no_break  routine_return_value;
    !print "(", obj, ".", prop, ";", say__p, say__pc, ")";
    if (prop == 0) {
        print (name) prop; routine_return_value = true;
    } else {
        switch (metaclass(obj.prop)) {
            nothing:
                routine_return_value = false;
            String:
                print (string) obj.prop; !if (no_break == false) new_line;
                routine_return_value = true;
            Routine:
                routine_return_value = RunRoutines(obj, prop);
                !print "[", routine_return_value, "]";
        }
    }
    if (routine_return_value) {
        say__p = 1;
        if (no_break == false) {
            new_line;
            !print "(DP->", say__p, say__pc, ")";
```

```
            DivideParagraphPoint();
            !print "(to", say__p, say__pc, ")";
        }
    }
    !print "(-->", say__p, say__pc, ")";
    return routine_return_value;
];
```

§**14. Short Name Storage.** None of the following functions should be called for the Z-machine if the short name exceeds the size of the following buffer: whereas the Glulx implementation of `VM_PrintToBuffer` will safely truncate overlong text, that's impossible for the Z-machine, and horrible results will follow.

`CPrintOrRun` is a variation on `PrintOrRun`, simplified by not needing to handle entire paragraphs (so, no fuss about dividing) but complicated by having to capitalise the first letter. We do this by writing to the buffer and then altering the first character.

```
Array StorageForShortName buffer 250;
[ CPrintOrRun obj prop  v length i;
    if ((obj ofclass String or Routine) || (prop == 0))
        VM_PrintToBuffer (StorageForShortName, 160, obj);
    else {
        if (obj.prop == NULL) rfalse;
        if (metaclass(obj.prop) == Routine or String)
            VM_PrintToBuffer(StorageForShortName, 160, obj, prop);
        else return RunTimeError(2, obj, prop);
    }
    length = StorageForShortName-->0;
    StorageForShortName->WORDSIZE = VM_LowerToUpperCase(StorageForShortName->WORDSIZE);
    for (i=WORDSIZE: i<length+WORDSIZE: i++) print (char) StorageForShortName->i;
    if (i>WORDSIZE) say__p = 1;
    return;
];
[ Cap str nocaps;
    if (nocaps) print (string) str;
    else CPrintOrRun(str, 0);
];
```

§**15. Object Names I.** We now begin the work of printing object names. In the lowest level of this process we print just the name itself (without articles attached), and we do it by carrying out an activity.

```
[ PSN__ o;
    if (o == 0) { print (string) NOTHING__TX; rtrue; }
    switch (metaclass(o)) {
        Routine:  print "<routine ", o, ">"; rtrue;
        String:   print "<string ~", (string) o, "~>"; rtrue;
        nothing:  print "<illegal object number ", o, ">"; rtrue;
    }
    CarryOutActivity(PRINTING_THE_NAME_ACT, o);
];
```

§**16. Standard Name Printing Rule.** In its initial state, the "printing the name of" activity has just one rule: the following "for" rule.

```
Global caps_mode = false;
[ STANDARD_NAME_PRINTING_R obj;
    obj = parameter_object;
    if (obj == 0) {
        print (string) NOTHING__TX; return;
    }
    switch (metaclass(obj)) {
        Routine:  print "<routine ", obj, ">"; return;
        String:   print "<string ~", (string) obj, "~>"; return;
        nothing:  print "<illegal object number ", obj, ">"; return;
    }
    if (obj == player) {
        if (indef_mode == NULL && caps_mode) print (string) YOU__TX;
        else print (string) YOURSELF__TX;
        return;
    }
    #Ifdef LanguagePrintShortName;
    if (LanguagePrintShortName(obj)) return;
    #Endif; ! LanguagePrintShortName
    if (indef_mode && obj.&short_name_indef ~= 0 &&
        PrintOrRun(obj, short_name_indef, true) ~= 0) return;
    if (caps_mode &&
        obj.&cap_short_name ~= 0 && PrintOrRun(obj, cap_short_name, true) ~= 0) {
        caps_mode = false;
        return;
    }
    if (obj.&short_name ~= 0 && PrintOrRun(obj, short_name, true) ~= 0) return;
    print (object) obj;
];
```

§**17. Object Names II.** The second level of the system for printing object names handles the placing of articles in front of them: *the* red herring, *an* elephant, *Some* bread. The following routine allows us to choose:

(a) `obj`, the object whose name is to be printed;
(b) `acode`, the kind of article needed: capitalised definite (0), lower case uncapitalised definite (1), or uncapitalised indefinite (2);
(c) `pluralise`, a flag forcing to a plural form (e.g., "some" being the pluralised form of an indefinite article in English);
(d) `capitalise`, a flag forcing us to capitalise the article – it's by setting this that we can achieve the fourth option missing from (b), viz., capitalised indefinite. (All of this is a legacy design from a time when the I6 library did not support capitalised indefinite articles.)

The routine then looks after issues such as which contraction form to use: for instance, in English, whether to use "a" or "an" for the indefinite singular depends on the text of the object's name.

```
Global short_name_case;
[ PrefaceByArticle obj acode pluralise capitalise  i artform findout artval;
    if (obj provides articles) {
        artval=(obj.&articles)-->(acode+short_name_case*LanguageCases);
        if (capitalise)
```

```
            print (Cap) artval, " ";
        else
            print (string) artval, " ";
        if (pluralise) return;
        print (PSN__) obj; return;
    }
    i = GetGNAOfObject(obj);
    if (pluralise) {
        if (i < 3 || (i >= 6 && i < 9)) i = i + 3;
    }
    i = LanguageGNAsToArticles-->i;
    artform = LanguageArticles
        + 3*WORDSIZE*LanguageContractionForms*(short_name_case + i*LanguageCases);
    #Iftrue (LanguageContractionForms == 2);
    if (artform-->acode ~= artform-->(acode+3)) findout = true;
    #Endif; ! LanguageContractionForms
    #Iftrue (LanguageContractionForms == 3);
    if (artform-->acode ~= artform-->(acode+3)) findout = true;
    if (artform-->(acode+3) ~= artform-->(acode+6)) findout = true;
    #Endif; ! LanguageContractionForms
    #Iftrue (LanguageContractionForms == 4);
    if (artform-->acode ~= artform-->(acode+3)) findout = true;
    if (artform-->(acode+3) ~= artform-->(acode+6)) findout = true;
    if (artform-->(acode+6) ~= artform-->(acode+9)) findout = true;
    #Endif; ! LanguageContractionForms
    #Iftrue (LanguageContractionForms > 4);
    findout = true;
    #Endif; ! LanguageContractionForms
    #Ifdef TARGET_ZCODE;
    if (standard_interpreter ~= 0 && findout) {
        StorageForShortName-->0 = 160;
        @output_stream 3 StorageForShortName;
        if (pluralise) print (number) pluralise; else print (PSN__) obj;
        @output_stream -3;
        acode = acode + 3*LanguageContraction(StorageForShortName + 2);
    }
    #Ifnot; ! TARGET_GLULX
    if (findout) {
        if (pluralise)
            Glulx_PrintAnyToArray(StorageForShortName, 160, EnglishNumber, pluralise);
        else
            Glulx_PrintAnyToArray(StorageForShortName, 160, PSN__, obj);
        acode = acode + 3*LanguageContraction(StorageForShortName);
    }
    #Endif; ! TARGET_
    Cap (artform-->acode, ~~capitalise); ! print article
    if (pluralise) return;
    print (PSN__) obj;
];
```

§**18. Object Names III.** The routines accessible from outside this segment.

```
[ IndefArt obj i;
    if (obj == 0) { print (string) NOTHING__TX; rtrue; }
    i = indef_mode; indef_mode = true;
    if (obj has proper) { indef_mode = NULL; print (PSN__) obj; indef_mode = i; return; }
    if (obj provides article) {
        PrintOrRun(obj, article, true); print " ", (PSN__) obj; indef_mode = i;
        return;
    }
    PrefaceByArticle(obj, 2); indef_mode = i;
];
[ CIndefArt obj i;
    if (obj == 0) { CPrintOrRun(NOTHING__TX, 0); rtrue; }
    i = indef_mode; indef_mode = true;
    if (obj has proper) {
        indef_mode = NULL;
        caps_mode = true;
        print (PSN__) obj;
        indef_mode = i;
        caps_mode = false;
        return;
    }
    if (obj provides article) {
        CPrintOrRun(obj, article); print " ", (PSN__) obj; indef_mode = i;
        return;
    }
    PrefaceByArticle(obj, 2, 0, 1); indef_mode = i;
];
[ DefArt obj i;
    i = indef_mode; indef_mode = false;
    if ((~~obj ofclass Object) || obj has proper) {
        indef_mode = NULL; print (PSN__) obj; indef_mode = i;
        return;
    }
    PrefaceByArticle(obj, 1); indef_mode = i;
];
[ CDefArt obj i;
    i = indef_mode; indef_mode = false;
    if ((obj ofclass Object) && (obj has proper || obj == player)) {
        indef_mode = NULL;
        caps_mode = true;
        print (PSN__) obj;
        indef_mode = i;
        caps_mode = false;
        return;
    }
    if ((~~obj ofclass Object) || obj has proper) {
        indef_mode = NULL; print (PSN__) obj; indef_mode = i;
        return;
    }
    PrefaceByArticle(obj, 0); indef_mode = i;
];
```

```
[ PrintShortName obj i;
    i = indef_mode; indef_mode = NULL;
    PSN__(obj); indef_mode = i;
];
```

§**19. Say One Of.** These routines are described in the Extensions chapter of the Inform documentation.

```
[ I7_SOO_PAR oldval count; if (count <= 1) return count; return random(count); ];
[ I7_SOO_RAN oldval count v; if (count <= 1) return count;
    v = oldval; while (v == oldval) v = random(count); return v; ];
[ I7_SOO_STI oldval count v; if (oldval) return oldval; return I7_SOO_PAR(oldval, count); ];
[ I7_SOO_CYC oldval count; oldval++; if (oldval > count) oldval = 1; return oldval; ];
[ I7_SOO_STOP oldval count; oldval++; if (oldval > count) oldval = count; return oldval; ];
[ I7_SOO_TAP oldval count tn rn c; if (count <= 1) return count; tn = count*(count+1)/2;
    rn = random(tn); for (c=1:c<=count:c++) { rn = rn - c; if (rn<=0) return (count-c+1); } ];
Array I7_SOO_SHUF->32;
[ I7_SOO_SHU oldval count sd ct v i j s ssd scope cc base;
    base = count+1;
    v = oldval%base; oldval = oldval/base; ct = oldval%base; sd = oldval/base;
    if (count > 32) return I7_SOO_PAR(oldval, count);
    if (count <= 1) v = count;
    else {
        !print "^In v=", v, " ct=", ct, " sd=", sd, "^";
        cc = base*base;
        scope = MAX_POSITIVE_NUMBER/cc - cc - base;
        if (scope%2==0) scope--;
        if (scope<0) scope = -scope;
        !print "Scope = ", scope, "^";
        if (sd == 0) { sd = random(scope); ct=0; }
        for (i=0:i<count:i++) I7_SOO_SHUF->i = i;
        ssd = sd;
        for (i=0:i<count-1:i++) {
            j = (sd)%(count-i)+i; sd = (sd*31973)+17; if (sd<0) sd=-sd;
            s = I7_SOO_SHUF->j; I7_SOO_SHUF->j = I7_SOO_SHUF->i; I7_SOO_SHUF->i = s;
        }
        !for (i=0:i<count:i++) print I7_SOO_SHUF->i, " "; print "^";
        v = (I7_SOO_SHUF->ct)+1;
        ct++; if (ct >= count) { ct = 0; ssd = 0; }
    }
    !print "Out v=", v, " ct=", ct, " ssd=", sd, "^";
    !print "Return ", v + ct*base + ssd*base*base, "^";
    return v + ct*base + ssd*base*base;
];
```

RTP Template B/rtpt

Purpose

To issue run-time problem messages, and to perform some run-time type checking which may issue such messages.

---

§**1. Reporting.** All RTPs are produced by calling the following routine, which takes one compulsory argument: `n`, the RTP number. When I7 is being used with the Inform user interface, it's important that these numbers correspond to the explanatory web pages stored within the interface application: those in turn are created by the `inrtps` utility. The interface knows to display these pages because it parses the printed output during play to look for the text layout produced by the following routine: so do not reformat RTPs without ensuring that corresponding changes have been made to the Inform user interface applications.

The arguments `par1`, `par2` and `par3` are optional parameters clarifying the message; `ln` is an optional parameter specifying a paragraph number in the original source text (though in fact I7 does not use this at present).

```
[ RunTimeProblem n par1 par2 par3 ln   i c;
    if (enable_rte == false) return;
    enable_rte = false;
    print "^*** Run-time problem P", n;
    if (ln) print " (at paragraph ", ln, " in the source text)";
    print ": ";
    switch(n) {
        RTP_BACKDROP:
            print "Tried to move ", (the) par1, " (a backdrop) to ", (the) par2,
                ", which is not a region.^";
        RTP_CANTCHANGE:
            print "Tried to change player to ", (the) par1,
                ", which is not a player-character.^";
        RTP_NOEXIT:
            print "Tried to change ", (the) par2, " exit of ", (the) par1,
                ", but it didn't seem to have such an exit to change.^";
        RTP_EXITDOOR:
            print "Tried to change ", (the) par2, " exit of ", (the) par1,
                ", but it led to a door, not a room.^";
        RTP_IMPREL:
            print "Tried to access an inappropriate relation for ", (the) par1,
                ", violating '";
            for (i=0: relation_metadata-->i ~= NULL: i=i+3) {
                c = relation_metadata-->(i+1);
                if (((c == Relation_VtoV) || (c == Relation_Sym_VtoV))
                    && (par2 == relation_metadata-->i))
                    print (string) relation_metadata-->(i+2), "'.^";
            }
        RTP_RULESTACK:
            print "Too many procedural rules acting all at once.^";
        RTP_TOOMANYRULEBOOKS:
```

```
    print "Too many rulebooks in simultaneous use.^";
RTP_TOOMANYEVENTS:
    print "Too many timed events are going on at once.^";
RTP_BADPROPERTY:
    print "Tried to access non-existent property for ", (the) par1, ".^";
RTP_UNPROVIDED:
    print "Since ", (the) par1, " is not allowed the property ~",
            (string) par2, "~, it is against the rules to try to use it.^";
RTP_UNSET:
    print "Although ", (the) par1, " is allowed to have the property ~",
        (string) par2, "~, no value was ever given, so it can't now be used.^";
RTP_TOOMANYACTS:
    print "Too many activities are going on at once.^";
RTP_CANTABANDON:
    print "Tried to abandon an activity which wasn't going on.^";
RTP_CANTEND:
    print "Tried to end an activity which wasn't going on.^";
RTP_CANTMOVENOTHING:
    print "You can't move nothing.^";
RTP_CANTREMOVENOTHING:
    print "You can't remove nothing from play.^";
RTP_DIVZERO:
    print "You can't divide by zero.^";
RTP_BADVALUEPROPERTY:
    print "Tried to access property for a value which didn't fit: ",
        "if this were a number it would be ", par1, ".^";
RTP_NOTBACKDROP:
    print "Tried to move ", (the) par1, " (not a backdrop) to ", (the) par2,
        ", which is a region.^";
RTP_TABLE_NOCOL:
    print "Attempt to look up a non-existent column in the table '",
        (PrintTableName) par1, "'.^";
RTP_TABLE_NOCORR:
    print "Attempt to look up a non-existent correspondence in the table '",
        (PrintTableName) par1, "'.^";
RTP_TABLE_NOROW:
    print "Attempt to look up a non-existent row in the table '",
        (PrintTableName) par1, "'.^";
RTP_TABLE_NOENTRY:
    print "Attempt to look up a non-existent entry at column ", par2,
        ", row ", par3, " of the table '", (PrintTableName) par1, "'.^";
RTP_TABLE_NOTABLE:
    print "Attempt to blank out a row from a non-existent table (value ",
        par1, ").^";
RTP_TABLE_NOMOREBLANKS:
    print "Attempt to choose a blank row in a table with none left: table '",
        (PrintTableName) par1, "'.^";
RTP_TABLE_NOROWS:
    print "Attempt to choose a random row in an entirely blank table: table '",
        (PrintTableName) par1, "'.^";
RTP_TABLE_CANTSORT:
    print "Attempt to sort a table whose ordering must remain fixed: table '",
        (PrintTableName) par1, "'.^";
```

```
RTP_TABLE_CANTSAVE:
    print "Attempt to save a table to a file whose data is unstable: table '",
        (PrintTableName) par1, "'.^";
RTP_TABLE_WONTFIT:
    print "File being read has too many rows or columns to fit into table: table '",
        (PrintTableName) par1, "'.^";
RTP_TABLE_BADFILE:
    print "File being read is not a previously saved table: table '",
        (PrintTableName) par1, "'.^";
RTP_NOTINAROOM:
    print "Attempt to test if the current location is '",
        (the) par1, "', which is not a room or region.^";
RTP_BADTOPIC:
    print "Attempt to see if a snippet of text matches something which
        is not a topic.^";
RTP_ROUTELESS:
    print "Attempt to find route or count steps through an implicit
        relation.^";
RTP_PROPOFNOTHING:
    print "Attempt to use a property of the 'nothing' non-object.^";
RTP_DECIDEONWRONGKIND:
    print "Attempt to 'decide on V' where V is the wrong kind of object.^";
RTP_DECIDEONNOTHING:
    print "Attempt to 'decide on nothing'.^";
RTP_LOWLEVELERROR:
    print "Low level error.^";
RTP_DONTIGNORETURNSEQUENCE:
    print "Attempt to ignore the turn sequence rules.^";
RTP_SAYINVALIDSNIPPET:
    print "Attempt to say a snippet value which is currently invalid: words ",
        par1, " to ", par2, ".^";
RTP_SPLICEINVALIDSNIPPET:
    print "Attempt to splice a snippet value which is currently invalid: words ",
        par1, " to ", par2, ".^";
RTP_INCLUDEINVALIDSNIPPET:
    print "Attempt to match a snippet value which is currently invalid: words ",
        par1, " to ", par2, ".^";
RTP_LISTWRITERMEMORY:
    print "The list-writer has run out of memory.^";
RTP_CANTREMOVEPLAYER:
    print "Attempt to remove the player from play.^";
RTP_CANTREMOVEDOORS:
    print "Attempt to remove a door from play.^";
RTP_CANTCHANGEOFFSTAGE:
    print "Attempt to change the player to a person off-stage.^";
RTP_MSTACKMEMORY:
    print "The memory stack is exhausted.^";
RTP_TYPECHECK:
    print "Phrase applied to an incompatible kind of value.^";
RTP_FILEIOERROR:
    print "Error handling external file.^";
RTP_HEAPERROR:
    print "Memory allocation proved impossible.^";
```

```
        RTP_LISTRANGEERROR:
            print "Attempt to use list item which does not exist.^";
        RTP_REGEXPSYNTAXERROR:
            print "Syntax error in regular expression.^";
        RTP_NOGLULXUNICODE:
            print "This interpreter does not support Unicode.^";
        RTP_BACKDROPONLY:
            print "Only backdrops can be moved to multiple places.^";
        RTP_NOTBACKDROP:
            print "Tried to move ", (the) par1, " (not a thing) to ", (the) par2,
                ", but only things can move around.^";
        RTP_SCENEHASNTSTARTED:
            print "The scene ", (PrintSceneName) par1,
                " hasn't started, so you can't ask when it did.^";
        RTP_SCENEHASNTENDED:
            print "The scene ", (PrintSceneName) par1,
                " hasn't ended, so you can't ask when it did.^";
        RTP_NEGATIVEROOT:
            print "You can't take the square root of a negative number.^";
        }
    print "^";
];
```

§**2. Low-Level Errors.** The following is a residue from the old I6 library, and most of its possible messages can't be seen in I7 use – for instance I7 has no timers or daemons, so error 4 is out of the question. But we retain the routine because the veneer code added by I6 requires it to be present.

```
Constant MAX_TIMERS = 0;
[ RunTimeError n p1 p2;
    #Ifdef DEBUG;
    print "** Library error ", n, " (", p1, ",", p2, ") **^** ";
    switch (n) {
    1:    print "preposition not found (this should not occur)";
    2:    print "Property value not routine or string: ~", (property) p2, "~ of ~", (name) p1,
                "~ (", p1, ")";
    3:    print "Entry in property list not routine or string: ~", (property) p2, "~ list of ~",
                (name) p1, "~ (", p1, ")";
    4:    print "Too many timers/daemons are active simultaneously.
                The limit is the library constant MAX_TIMERS (currently ",
                MAX_TIMERS, ") and should be increased";
    5:    print "Object ~", (name) p1, "~ has no ~time_left~ property";
    7:    print "The object ~", (name) p1, "~ can only be used as a player object if it has
                the ~number~ property";
    8:    print "Attempt to take random entry from an empty table array";
    9:    print p1, " is not a valid direction property number";
    10:   print "The player-object is outside the object tree";
    11:   print "The room ~", (name) p1, "~ has no ~description~ property";
    12:   print "Tried to set a non-existent pronoun using SetPronoun";
    13:   print "A 'topic' token can only be followed by a preposition";
    default: print "(unexplained)";
    }
    print " **^";
    #Ifnot;
```

```
    print "** Library error ", n, " (", p1, ",", p2, ") **^";
    #Endif; ! DEBUG
    RunTimeProblem(RTP_LOWLEVELERROR);
];
```

§**3. Argument Type Checking Failed.** This is called when run-time type checking for the argument of a phrase fails, so that no definition for the phrase can be applied.

```
[ ArgumentTypeFailed file line arg;
    RunTimeProblem(RTP_TYPECHECK, 0, 0, 0, line);
];
```

§**4. Return Type Checking Failed.** Similarly, though in a more restricted set of circumstances. NI can usually prove that the value returned by a phrase matches its supposed kind, but because of the deliberately weak type-checking within the kind of value "object" it will allow a broader kind of object to be returned when the requirement is for a narrower one. In such cases, it checks the result with the following routine. The value `V` is a valid "object", but that means it can be `nothing`, and we must check that it matches a given I7 kind `K` (where `nothing` is not valid).

```
[ CheckKindReturned V K;
    if (V ofclass K) return V;
    if (v == nothing) RunTimeProblem(RTP_DECIDEONNOTHING);
    else RunTimeProblem(RTP_DECIDEONWRONGKIND);
    return V;
];
```

§**5. Whether Provides.** This routine defines the phrase "if O provides P": there are three tests to pass, and if any of the three fail, we return `false`. (The `issue_rtp` flag, causing RTPs to be issued depending on which test fails, is never set when the routine is simply testing the condition.)

Firstly, P has to be a property known to I7. Secondly, there has to be permission either for this individual object to have it, or for its kind to have it, or its kind's kind, and so on. Thirdly, the object has to actually have the property in question at an I6 level – having permission to have it doesn't mean it actually does have.

```
[ WhetherProvides obj either_or p issue_rtp  off i textual a l;
    if (metaclass(obj) ~= Object) rfalse;
    if (p<0) p = ~p;
    if (either_or) {
        if (p < FBNA_PROP_NUMBER) off = attribute_offsets-->p;
        else off = attribute_offsets-->(50+p-FBNA_PROP_NUMBER);
    } else off = property_offsets-->p;
    if (off<0) {
        if (issue_rtp) RunTimeProblem(RTP_BADPROPERTY, obj);
        rfalse;
    }
    textual = property_metadata-->off; off++;
    if (ScanPropertyMetadata(obj, off)) jump PermissionFound;
    if (obj provides IK_0) {
        l = obj.IK_0;
        while (l > 0) {
            a = l*2;
```

```
            if (ScanPropertyMetadata(KindHierarchy-->a, off)) jump PermissionFound;
            l = KindHierarchy-->(a+1);
        }
    }
    if (issue_rtp) RunTimeProblem(RTP_UNPROVIDED, obj, textual);
    rfalse;
    .PermissionFound;
        if (either_or) rtrue;
        if (obj provides p) rtrue;
        if (issue_rtp) RunTimeProblem(RTP_UNSET, obj, textual);
        rfalse;
];
```

§**6. Scan Property Metadata.** The `property_metadata` table is a series of zero-terminated lists of objects (or class objects, representing I7 kinds). Each list corresponds to a single property; the position in the table is called the "offset" for the property. The following searches from a given offset.

```
[ ScanPropertyMetadata obj off i;
    for (i=off: property_metadata-->i >= 0: i++)
        if (obj == property_metadata-->i) rtrue;
    rfalse;
];
```

§**7. Get Either-Or Property.** If `p` represents a property, then `~p` (its bitwise negation) represents its logical negation. The bitwise negation will change the sign bit; since all properties are ordinarily positive, we can detect bitwise negation by looking for negative property numbers. (This could fail on Glulx story files above 1GB in size, but that's about 1000 times the size of the record-sized file created thus far.)

FBNA stands for First Boolean Not to be an Attribute, in the I6 sense. Some either/or (i.e., boolean) properties are stored as I6 attributes, others as I6 properties whose values are always `true` or `false`.

Note that we allow either/or properties to be *read* for any object, regardless of permissions, returning `false` if the object does not have the property. This is so that a description such as "open things" can be applied against any object without run-time errors, even though it is only normally valid for doors and containers.

```
[ GetEitherOrProperty o p;
    if (o == nothing) rfalse;
    if (p<0) p = ~p;
    if (WhetherProvides(o, true, p, false)) {
        if (p<FBNA_PROP_NUMBER) { if (o has p) rtrue; rfalse; }
        if ((o provides p) && (o.p)) rtrue;
    }
    rfalse;
];
```

§**8. Set Either-Or Property.** An attempt to *write* an either/or property which is not provided will, however, always produce a run-time problem.

```
[ SetEitherOrProperty o p negate adj;
    if (p<0) { p = ~p; negate = ~negate; }
    if (adj) {
        (adj)(o);
    } else if (WhetherProvides(o, true, p, true)) {
        if (negate) {
            if (p<FBNA_PROP_NUMBER) give o ~p; else o.p = false;
        } else {
            if (p<FBNA_PROP_NUMBER) give o p; else o.p = true;
        }
    }
];
```

§**9. Get Property For KOVs.** Similarly, but for properties belonging to values.

```
[ PropertyKOV kovp_array off inst per;
    per = kovp_array-->0;
    return kovp_array-->(1+(inst-1)*per+off);
];
```

§**10. Set Property For KOVs.** Similarly, but for properties belonging to values.

```
[ WritePropertyKOV kovp_array off inst state per;
    per = kovp_array-->0;
    kovp_array-->(1+(inst-1)*per+off) = state;
];
```

§**11. Value Property.** Some value properties belong to other values (those created in tables), and these are detected by being properties of the special `ValuePropertyHolder` pseudo-object – an I6 object which is not part of the world model, and not a valid I7 "object" value, but which is used in order that properties belonging to values are still I6 property numbers. `ValuePropertyHolder.P` is the table column address for this property; `obj` is then a value for the kind of value created by the table, so it is used as an index into the table column to get the address of the memory location storing the property value.

The `door_to` property, relevant only for doors, is called rather than read: this enables it to be an I6 routine returning the other side of the door from the one which the player is on.

```
[ ValueProperty obj pr;
    if (ValuePropertyHolder provides pr) {
        if ((obj<=0) || (obj > (ValuePropertyHolder.pr)-->0)) {
            RunTimeProblem(RTP_BADVALUEPROPERTY);
            rfalse;
        }
        return (ValuePropertyHolder.pr)-->(obj+COL_HSIZE);
    }
    if (obj == 0) { RunTimeProblem(RTP_PROPOFNOTHING, obj, pr); rfalse; }
    if ((pr == door_to) && (obj provides pr)) return obj.pr();
    if (WhetherProvides(obj, false, pr, true)) return obj.pr;
    rfalse;
];
```

§**12. Write Value Property.** This routine's name must consist of the read-value-property routine's name with the prefix `Write`, as that is how a reference to such a property is converted from an rvalue to an lvalue.

```
[ WriteValueProperty obj pr val;
    if (ValuePropertyHolder provides pr) {
        if ((obj<=0) || (obj > (ValuePropertyHolder.pr)-->0))
            return RunTimeProblem(RTP_BADVALUEPROPERTY);
        (ValuePropertyHolder.pr)-->(obj+COL_HSIZE) = val;
        return;
    }
    if (obj == 0) { RunTimeProblem(RTP_PROPOFNOTHING, obj, pr); rfalse; }
    if (WhetherProvides(obj, false, pr, true)) obj.pr = val;
];
```

Utilities Template B/utilt

Purpose

Miscellaneous utility routines for some fundamental I6 needs.

---

§**1. DigitToValue.** This takes a ZSCII or Unicode character code and returns its value as a digit, or returns -1 if it isn't a digit.

```
[ DigitToValue c n;
    n = c-'0';
    if ((n<0) || (n>9)) return -1;
    return n;
];
```

§**2. GenerateRandomNumber.** The following uses the virtual machine's RNG (via the I6 built-in function `random`) to produce a uniformly random integer in the range n to m inclusive, where n and m are allowed to be either way around; so that a random number between 17 and 4 is the same thing as a random number between 4 and 17, and there is therefore no pair of n and m corresponding to an empty range of values.

```
[ GenerateRandomNumber n m s;
    if (n==m) return n;
    if (n>m) { s = n; n = m; m = s; }
    n--;
    return random(m-n) + n;
];
```

§**3. GroupChildren.** The following runs through the child-objects of `par` in the I6 object tree, and moves those having a given property value together, to become the eldest children. It preserves the ordering in between those objects, and also in between those not having that property value. The property is required to be a common property.

We do this by temporarily moving objects into and out of `in_obj` and `out_obj`, objects which in all other circumstances never have children in the tree.

```
[ GroupChildren par prop value;
    while (child(par) ~= 0) {
        if (child(par).prop ~= value) move child(par) to out_obj;
        else move child(par) to in_obj;
    }
    while (child(in_obj) ~= 0)  move child(in_obj) to par;
    while (child(out_obj) ~= 0) move child(out_obj) to par;
    return child(par);
];
```

§**4. PrintSpaces.** Which prints a row of n spaces, for $n \geq 0$.

```
[ PrintSpaces n;
    while (n > 0) {
        print " ";
        n = n - 1;
    }
];
```

§**5. RunRoutines.** This function may not be very well-named, but the idea is to take a property of a given object and either to print it (and return `true`) if it's a string, and call it (and pass along its return value) if it's a routine. If the object does not provide the property, we act on the default value for the property if it has one, and otherwise do nothing (and return `false`).

The I6 pseudo-object `thedark` is used to give the impression that Darkness is a room in its own right, which is not really true. Note that it is not permitted to have properties other than the three named here: all other properties are redirected to the current location's object.

Properties with numbers under `INDIV_PROP_START` are "common properties". These come along with a table of default values, so that it is meaningful (in I6, anyway) to read them even when they are not provided (so that the address, returned by the `.&` operator, is zero).

```
[ RunRoutines obj prop;
    if (obj == thedark) obj = real_location;
    if ((obj.&prop == 0) && (prop >= INDIV_PROP_START)) rfalse;
    return obj.prop();
];
```

§**6. SwapWorkflags.** Recall that we have two general-purpose temporary attributes for each object: `workflag` and `workflag2`. The following swaps their values over for every object at once.

```
[ SwapWorkflags obj lst;
    objectloop (obj ofclass Object) {
        lst = false;
        if (obj has workflag2) lst = true;
        give obj ~workflag2;
        if (obj has workflag) give obj workflag2;
        give obj ~workflag;
        if (lst) give obj workflag;
    }
];
```

§**7. TestUseOption.** This routine, compiled by NI, returns `true` if the supplied argument is the number of a use option in force for the current run of NI, and `false` otherwise.

```
{-routine:TestUseOption}
```

§**8. IntegerDivide.** We can't simply use I6's `/` operator, as that translates directly into a virtual machine opcode which crashes on divide by zero.

```
[ IntegerDivide A B;
    if (B == 0) { RunTimeProblem(RTP_DIVZERO); rfalse; }
    return A/B;
];
```

§**9. IntegerRemainder.** Similarly.

```
[ IntegerRemainder A B;
    if (B == 0) { RunTimeProblem(RTP_DIVZERO); rfalse; }
    return A%B;
];
```

§**10. UnsignedCompare.** Comparison of I6 integers is normally signed, that is, treating the word as a twos-complement signed number, so that `$FFFF` is less than `0`, for instance. If we want to construe words as being unsigned integers, or as addresses, we need to compare them with the following routine, which returns 1 if $x > y$, 0 if $x = y$ and -1 if $x < y$.

```
[ UnsignedCompare x y u v;
    if (x == y) return 0;
    if (x < 0 && y >= 0) return 1;
    if (x >= 0 && y < 0) return -1;
    u = x&~WORD_HIGHBIT; v= y&~WORD_HIGHBIT;
    if (u > v) return 1;
    return -1;
];
```

§**11. ZRegion.** I7 contains many relics from I6, but here's a relic from I5: a routine which used to determine the metaclass of a value, before that concept was given a name. In I6 code, it can be implemented simply using `metaclass`, as the following shows. (The name is from "region of the Z-machine".)

```
[ ZRegion addr;
    switch (metaclass(addr)) {
        nothing: return 0;
        Object, Class: return 1;
        Routine: return 2;
        String: return 3;
    }
];
```

§**12. Library Messages.** This is another fossil, and probably soon to change.

```
[ GL__M a b c d;
    if ((actor ~= player) || (untouchable_silence)) rtrue;
    return L__M(a,b,c,d); ];
[ AGL__M a b c d;
    if (untouchable_silence) rtrue;
    return L__M(a,b,c,d); ];
```

Number Template B/numt

Purpose
Support for parsing integers.

§**1. Understanding.** In our target virtual machines, numbers are stored in twos-complement form, so that a 16-bit VM can hold the range of integers $-2^{15} = -32768$ to $2^{15} - 1 = +32767$, while a 32-bit VM can hold $-2^{31} = -2147483648$ to $2^{31} - 1 = +2147483647$: the token below accepts exactly those ranges.

```
[ DECIMAL_TOKEN wnc wna r n wa wl sign base digit digit_count original_wn group_wn;
    wnc = wn; original_wn = wn; group_wn = wn;
{-call:compile_number_grammar}
    wn = wnc;
    r = ParseTokenStopped(ELEMENTARY_TT, NUMBER_TOKEN);
    if ((r == GPR_NUMBER) && (parsed_number ~= 10000)) return r;
    wn = wnc;
    wa = WordAddress(wn);
    wl = WordLength(wn);
    sign = 1; base = 10; digit_count = 0;
    if (wa->0 ~= '-' or '$' or '0' or '1' or '2' or '3' or '4'
        or '5' or '6' or '7' or '8' or '9')
        return GPR_FAIL;
    if (wa->0 == '-') { sign = -1; wl--; wa++; }
    if (wl == 0) return GPR_FAIL;
    n = 0;
    while (wl > 0) {
        if (wa->0 >= 'a') digit = wa->0 - 'a' + 10;
        else digit = wa->0 - '0';
        digit_count++;
        switch (base) {
            2:  if (digit_count == 17) return GPR_FAIL;
            10:
                #Iftrue (WORDSIZE == 2);
                if (digit_count == 6) return GPR_FAIL;
                if (digit_count == 5) {
                    if (n > 3276) return GPR_FAIL;
                    if (n == 3276) {
                        if (sign == 1 && digit > 7) return GPR_FAIL;
                        if (sign == -1 && digit > 8) return GPR_FAIL;
                    }
                }
                #Ifnot; ! i.e., if (WORDSIZE == 4)
                if (digit_count == 11) return GPR_FAIL;
                if (digit_count == 10) {
                    if (n > 214748364) return GPR_FAIL;
                    if (n == 214748364) {
                        if (sign == 1 && digit > 7) return GPR_FAIL;
                        if (sign == -1 && digit > 8) return GPR_FAIL;
                    }
```

```
                }
                #Endif;
            16: if (digit_count == 5) return GPR_FAIL;
        }
        if (digit >= 0 && digit < base) n = base*n + digit;
        else return GPR_FAIL;
        wl--; wa++;
    }
    parsed_number = n*sign; wn++;
    return GPR_NUMBER;
];
```

§**2. Truth states.** And although truth states are not strictly speaking numbers, this seems as good a point as any to parse them:

```
[ TRUTH_STATE_TOKEN original_wn wd;
    original_wn = wn;
{-call:compile_truth_state_grammar}
    wn = original_wn;
    wd = NextWordStopped();
    if (wd == 'true') { parsed_number = 1; return GPR_NUMBER; }
    if (wd == 'false') { parsed_number = 0; return GPR_NUMBER; }
    wn = original_wn;
    return GPR_FAIL;
];
```

Time Template B/timet

Purpose
Support for parsing and printing times of day.

§**1. Rounding.** The following rounds a numerical value `t1` to the nearest unit of `t2`; for instance, if `t2` is 5 then it rounds to the nearest 5.

```
[ RoundOffTime t1 t2; return ((t1+t2/2)/t2)*t2; ];
```

§**2. Square Root.** This is an old algorithm for extracting binary square roots, taking 2 bits at a time. We start out with `one` at the highest bit which isn't the sign bit; that used to be worked out as `WORD_HIGHBIT/2`, but this caused unexpected portability problems (exposing a minor bug in Inform and also glulxe) because of differences in how C compilers handle signed division of constants in the case where the dividend is -2^{31}, the unique number which cannot be negated in 32-bit twos complement arithmetic.

```
[ SquareRoot num
    op res one;
    op = num;
    if (num < 0) { RunTimeProblem(RTP_NEGATIVEROOT); return 1; }
    ! "one" starts at the highest power of four <= the argument.
    for (one = WORD_NEXTTOHIGHBIT: one > op: one = one/4) ;
    while (one ~= 0) {
        !print "Round: op = ", op, " res = ", res, ", res**2 = ", res*res, " one = ", one, "^";
        if (op >= res + one) {
            op = op - res - one;
            res = res + one*2;
        }
        res = res/2;
        one = one/4;
    }
    !print "Res is ", res, "^";
    return res;
];
```

§**3. Cube Root.** The following is an iterative scheme for finding cube roots by Newton-Raphson approximation, not a great method but which, on the narrow ranges of integers we deal with, is good enough. The square root is used only as a sighting shot.

```
[ CubeRoot num x y n;
    if (num < 0) x = -SquareRoot(-num); else x = SquareRoot(num);
    for (n=0: (y ~= x) && (n++ < 100): y = x, x = (2*x + num/x/x)/3) ;
    return x;
];
```

§**4. Digital Printing.** For instance, “2:06 am”.

```
[ PrintTimeOfDay t h aop;
    if (t<0) { print "<no time>"; return; }
    if (t >= TWELVE_HOURS) { aop = "pm"; t = t - TWELVE_HOURS; } else aop = "am";
    h = t/ONE_HOUR; if (h==0) h=12;
    print h, ":";
    if (t%ONE_HOUR < 10) print "0"; print t%ONE_HOUR, " ", (string) aop;
];
```

§**5. Analogue Printing.** For instance, “six minutes past two”.

```
[ PrintTimeOfDayEnglish t h m dir aop;
    h = (t/ONE_HOUR) % 12; m = t%ONE_HOUR; if (h==0) h=12;
    if (m==0) { print (number) h, " o'clock"; return; }
    dir = "past";
    if (m > HALF_HOUR) { m = ONE_HOUR-m; h = (h+1)%12; if (h==0) h=12; dir = "to"; }
    switch(m) {
        QUARTER_HOUR: print "quarter"; HALF_HOUR: print "half";
        default: print (number) m;
            if (m%5 ~= 0) {
                if (m == 1) print " minute"; else print " minutes";
            }
    }
    print " ", (string) dir, " ", (number) h;
];
```

§**6. Understanding.** This I6 grammar token converts words in the player's command to a valid I7 time, and is heavily based on the one presented as a solution to an exercise in the DM4.

```
[ TIME_TOKEN first_word second_word at length flag
    illegal_char offhour hr mn i original_wn;
    original_wn = wn;
{-call:compile_time_grammar}
    wn = original_wn;
    first_word = NextWordStopped();
    switch (first_word) {
        'midnight': parsed_number = 0; return GPR_NUMBER;
        'midday', 'noon': parsed_number = TWELVE_HOURS;
        return GPR_NUMBER;
    }
    ! Next try the format 12:02
    at = WordAddress(wn-1); length = WordLength(wn-1);
    for (i=0: i<length: i++) {
        switch (at->i) {
            ':': if (flag == false && i>0 && i<length-1) flag = true;
            else illegal_char = true;
            '0', '1', '2', '3', '4', '5', '6', '7', '8', '9': ;
            default: illegal_char = true;
        }
    }
    if (length < 3 || length > 5 || illegal_char) flag = false;
```

```
    if (flag) {
        for (i=0: at->i~=':': i++, hr=hr*10) hr = hr + at->i - '0';
        hr = hr/10;
        for (i++: i<length: i++, mn=mn*10) mn = mn + at->i - '0';
        mn = mn/10;
        second_word = NextWordStopped();
        parsed_number = HoursMinsWordToTime(hr, mn, second_word);
        if (parsed_number == -1) return GPR_FAIL;
        if (second_word ~= 'pm' or 'am') wn--;
        return GPR_NUMBER;
    }
    ! Lastly the wordy format
    offhour = -1;
    if (first_word == 'half') offhour = HALF_HOUR;
    if (first_word == 'quarter') offhour = QUARTER_HOUR;
    if (offhour < 0) offhour = TryNumber(wn-1);
    if (offhour < 0 || offhour >= ONE_HOUR) return GPR_FAIL;
    second_word = NextWordStopped();
    switch (second_word) {
        ! "six o'clock", "six"
        'o^clock', 'am', 'pm', -1:
            hr = offhour; if (hr > 12) return GPR_FAIL;
        ! "quarter to six", "twenty past midnight"
        'to', 'past':
            mn = offhour; hr = TryNumber(wn);
            if (hr <= 0) {
                switch (NextWordStopped()) {
                    'noon', 'midday': hr = 12;
                    'midnight': hr = 0;
                    default: return GPR_FAIL;
                }
            }
            if (hr >= 13) return GPR_FAIL;
            if (second_word == 'to') {
                mn = ONE_HOUR-mn; hr--; if (hr<0) hr=23;
            }
            wn++; second_word = NextWordStopped();
        ! "six thirty"
        default:
            hr = offhour; mn = TryNumber(--wn);
            if (mn < 0 || mn >= ONE_HOUR) return GPR_FAIL;
            wn++; second_word = NextWordStopped();
    }
    parsed_number = HoursMinsWordToTime(hr, mn, second_word);
    if (parsed_number < 0) return GPR_FAIL;
    if (second_word ~= 'pm' or 'am' or 'o^clock') wn--;
    return GPR_NUMBER;
];

[ HoursMinsWordToTime hour minute word x;
    if (hour >= 24) return -1;
    if (minute >= ONE_HOUR) return -1;
    x = hour*ONE_HOUR + minute; if (hour >= 13) return x;
    x = x % TWELVE_HOURS; if (word == 'pm') x = x + TWELVE_HOURS;
```

```
    if (word ~= 'am' or 'pm' && hour == 12) x = x + TWELVE_HOURS;
    return x;
];
```

§**7. Relative Time Token.** "Time" is an interesting kind of value since it can hold two conceptually different ways of thinking about time: absolute times, such as "12:03 PM", and also relative times, like "ten minutes". For parsing purposes, these are completely different from each other, and the time token above handles only absolute times; we need the following for relative ones.

```
[ RELATIVE_TIME_TOKEN first_word second_word offhour mult mn original_wn;
    original_wn = wn;
    wn = original_wn;
    first_word = NextWordStopped(); wn--;
    if (first_word == 'an' or 'a//') mn=1; else mn=TryNumber(wn);
    if (mn == -1000) {
        first_word = NextWordStopped();
        if (first_word == 'half') offhour = HALF_HOUR;
        if (first_word == 'quarter') offhour = QUARTER_HOUR;
        if (offhour > 0) {
            second_word = NextWordStopped();
            if (second_word == 'of') second_word = NextWordStopped();
            if (second_word == 'an') second_word = NextWordStopped();
            if (second_word == 'hour') {
                parsed_number = offhour;
                return GPR_NUMBER;
            }
        }
        return GPR_FAIL;
    }
    wn++;
    first_word = NextWordStopped();
    switch (first_word) {
        'minutes', 'minute': mult = 1;
        'hours', 'hour': mult = 60;
        default: return GPR_FAIL;
    }
    parsed_number = mn*mult;
    if (mult == 60) {
        mn=TryNumber(wn);
        if (mn ~= -1000) {
            wn++;
            first_word = NextWordStopped();
            if (first_word == 'minutes' or 'minute')
                parsed_number = parsed_number + mn;
            else wn = wn - 2;
        }
    }
    return GPR_NUMBER;
];
```

§8. During Scene Matching.

```
[ DuringSceneMatching prop sc;
    for (sc=0: sc<NUMBER_SCENES_CREATED: sc++)
        if ((scene_status-->sc == 1) && (prop(sc+1))) rtrue;
    rfalse;
];
```

§9. Scene Questions.

```
[ SceneUtility sc task;
    if (sc <= 0) return 0;
    if (task == 1 or 2) {
        if (scene_endings-->(sc-1) == 0) return RunTimeProblem(RTP_SCENEHASNTSTARTED, sc);
    } else {
        if (scene_endings-->(sc-1) <= 1) return RunTimeProblem(RTP_SCENEHASNTENDED, sc);
    }
    switch (task) {
        1: return (the_time - scene_started-->(sc-1))%(TWENTY_FOUR_HOURS);
        2: return scene_started-->(sc-1);
        3: return (the_time - scene_ended-->(sc-1))%(TWENTY_FOUR_HOURS);
        4: return scene_ended-->(sc-1);
    }
];
```

Tables Template B/tabt

Purpose

To read, write, search and allocate rows in the Table data structure.

§**1. Format.** The I7 Table structure is not to be confused with the I6 `table` form of array: it is essentially a two-dimensional array which has some metadata at the top of each column.

The run-time representation for a Table is the address `T` of an I6 `table` array: that is, `T-->0` holds the number of columns (which is at most 99) and `T-->i` is the address of column number `i`. Columns are therefore numbered from 1 to `T-->0`, but they are also identified by an ID number of 100 or more, with each different column name having its own ID number. (This is so that multiple tables can share columns with the same name, and correlate them: it also means that NI's type-checking machinery can know the kind of value of a table entry from the name of the column alone.)

Each column `C` is also a `table` array, with `C-->1` holding the unique ID number for the column's name, `C-->2` holding the blank entry flags offset and `C-->3` up to `C-->(C-->0)` holding the entries.

`C-->1` also contains four upper bit flags. These are also defined in "Tables.w" in the NI source, and the values must agree.

```
Constant TB_COLUMN_SIGNED      $4000;
Constant TB_COLUMN_TOPIC       $2000;
Constant TB_COLUMN_DONTSORTME  $1000;
Constant TB_COLUMN_NOBLANKBITS $0800;
Constant TB_COLUMN_CANEXCHANGE $0400;
Constant TB_COLUMN_ALLOCATED   $0200;
Constant TB_COLUMN_NUMBER      $01ff; ! Mask to remove upper bit flags
Constant COL_HSIZE 2; ! Column header size: two words (ID/flags, blank bits)
```

§**2. Find Column.** Columns can be referenced either by their physical column numbers – from 1 to, potentially, 99 – or else by unique ID numbers associated with column names. For instance, if a table has a column called "liquid capacity", then all references to its "liquid capacity entry" are via the ID number associated with this column name, which will be $\geq$ 100 and on the other hand $\leq$ `TB_COLUMN_NUMBER`. At present, this is only 511, so there can be at most 411 different column names across all the tables present in the source text. (It is just about possible to imagine this being a problem on a very large work, so we will probably one day revise the above to make use of the larger word-size in Glulx and so raise this limit. But so far nobody has got even close to it.)

```
[ TableFindCol tab col f i no_cols n;
    no_cols = tab-->0;
    for (i=1: i<=no_cols: i++)
        if (col == ((tab-->i)-->1) & TB_COLUMN_NUMBER) return i;
    if (f) { RunTimeProblem(RTP_TABLE_NOCOL, tab); return 0; }
    return 0;
];
```

§**3. Number of Rows.** The columns in a table can be assumed all to have the same height (i.e., number of rows): thus the number of rows in `T` can be calculated by looking at column 1, thus...

```
[ TableRows tab; return ((tab-->1)-->0) - COL_HSIZE; ];
```

§**4. Blanks.** Each table entry is stored in a single word in memory: indeed, column `C` row `R` is at address `(T-->C)-->(R+COL_HSIZE)`.

But this is not sufficient storage in all cases, because each entry can be either a value or can be designated "blank". Since, for some columns at least, the possible values include every number, we find that we have to store $2^{16}+1$ possibilities given only a 16-bit memory word. (Well, or $2^{32}+1$ with a 32-bit word, depending on the virtual machine.) This cannot be done.

We therefore need, at least in some cases, an additional bit of storage for each table entry which indicates whether or not it is blank. If we provided such a bit for every table entry, that would be a fairly simple system to implement, but it would also be wasteful of memory, with an overhead of about 5% in practice: and memory in the virtual machine is in very short supply. The reason such a system would be wasteful is that many columns are known to hold values which are in a narrow range; for instance, a time of day cannot exceed 1440, and there will never be more than 10,000 rulebooks or scenes, and so on. For such columns it would be more efficient and indeed faster to indicate blankness by using an exceptional value in the memory cell which is such that it cannot be valid for the kind of value stored in the column. We therefore provide a "blanks bitmap" for only some columns.

This leads us to define the following dummy value, chosen so that it is both impossible for most kinds of value – which is easy to arrange – and also unlikely for even those kinds of value where it is legal. For instance, -1 would be impossible for enumerative kinds of value such as rulebooks and scenes, but it would be a poor choice for the dummy value because it occurs pretty often as an integer. Instead we use the constant `IMPROBABLE_VALUE`, whose value depends on the word size of the virtual machine, and which is declared in "Definitions.i6t".

An entry is therefore blank if and only if either

(a) its column has no blanks bitmap and the stored entry is `TABLE_NOVALUE`, or
(b) its column does have a blanks bitmap, the blanks bit for this entry is set, and the stored entry is also `TABLE_NOVALUE`.

To look up the blanks bitmap is a little slower than to access the stored entry directly. Most of the time, entries accessed will be non-blank: so it is efficient to have a system where we can quickly determine this. If we look at the entry and find that it is not `TABLE_NOVALUE`, then we know it is not a blank. If we find that it is `TABLE_NOVALUE`, on the other hand, then quite often the column has no blanks bitmap and again we have a quick answer: it's blank. Only if the column also has a blanks bitmap do we need to check that we haven't got a false negative. (The more improbable `TABLE_NOVALUE` is as a stored value, the rarer it is that we have to check the blanks bitmap for a non-blank entry.)

```
Constant TABLE_NOVALUE = IMPROBABLE_VALUE;
```

§**5. Masks.** The blanks bitmaps are stored as bytes; we therefore need a quick way to test or set whether bit number i of a byte is zero, where $0 \leq i \leq 7$. I6 provides no very useful operators here, whereas memory lookup is cheap, so we use two arrays of bitmaps:

```
Array CheckTableEntryIsBlank_LU
    -> $$00000001
        $$00000010
        $$00000100
        $$00001000
        $$00010000
        $$00100000
        $$01000000
        $$10000000;
Array CheckTableEntryIsNonBlank_LU
    -> $$11111110
        $$11111101
        $$11111011
        $$11110111
        $$11101111
        $$11011111
        $$10111111
        $$01111111;
```

§**6. Testing Blankness.** The following routine is the one which checks that there is no false negative: it should be used when we know that the table entry is `TABLE_NOVALUE` and we need to check the blank bit, if there is one, to make sure the entry is indeed blank.

The second word in the column table header, `C-->2`, holds the address of the blanks bitmap: this in turn contains one bit for each row, starting with the least significant bit of the first byte. If the table contains a number of rows which isn't a multiple of 8, the spare bits at the end of the last byte in the blanks bitmap are wasted, but this is an acceptable overhead in practice.

```
[ CheckTableEntryIsBlank tab col row i at;
    if (col >= 100) col = TableFindCol(tab, col);
    if (col == 0) rtrue;
    if ((tab-->col)-->(row+COL_HSIZE) ~= TABLE_NOVALUE) {
        print "*** CTEIB on nonblank value ", tab, " ", col, " ", row, " ***^";
    }
    if (((tab-->col)-->1) & TB_COLUMN_NOBLANKBITS) rtrue;
    row--;
    at = ((tab-->col)-->2) + (row/8);
    if ((TB_Blanks->at) & (CheckTableEntryIsBlank_LU->(row%8))) rtrue;
    rfalse;
];
```

§**7. Force Entry Blank.** We blank a table cell by storing `TABLE_NOVALUE` in its entry word and also setting the relevant bit in the blanks bitmap, if there is one.

We need to be careful if the column holds a kind of value where values are pointers to blocks of allocated memory, because if so then overwriting such a value might lead to a memory leak. So in such cases we call `BlkFree` to free the memory block. (Note that each memory block is pointed to by one and only one I7 value at any given time: we are using them as values, not pointers to values. So if this reference is deleted, it's by definition the only one.) `TABLE_NOVALUE` is chosen such that it cannot be an address of a memory block, which is convenient here. (The value 0 means "no memory block allocated yet".)

```
[ ForceTableEntryBlank tab col row i at oldv flags;
    if (col >= 100) col = TableFindCol(tab, col);
    if (col == 0) rtrue;
    flags = (tab-->col)-->1;
    oldv = (tab-->col)-->(row+COL_HSIZE);
    if ((flags & TB_COLUMN_ALLOCATED) && (oldv ~= 0 or TABLE_NOVALUE))
        BlkFree(oldv);
    (tab-->col)-->(row+COL_HSIZE) = TABLE_NOVALUE;
    if (flags & TB_COLUMN_NOBLANKBITS) return;
    row--;
    at = ((tab-->col)-->2) + (row/8);
    (TB_Blanks->at) = (TB_Blanks->at) | (CheckTableEntryIsBlank_LU->(row%8));
];
```

§**8. Force Entry Non-Blank.** To unblank a cell, we need to clear the relevant bit in the bitmap. We then go on to write a new value in to the entry – thus overwriting the `TABLE_NOVALUE` value – but that isn't done here; the expectation is that whoever calls this routine is just about to write a new entry anyway.

The exception is again for columns holding a kind of value pointing to a memory block, where we create a suitable initialised but uninteresting memory block for the KOV in question, and set the entry to that.

```
[ ForceTableEntryNonBlank tab col row i at oldv flags tc kov j;
    if (col >= 100) col=TableFindCol(tab, col);
    if (col == 0) rtrue;
    if (((tab-->col)-->1) & TB_COLUMN_NOBLANKBITS) return;
    flags = (tab-->col)-->1;
    oldv = (tab-->col)-->(row+COL_HSIZE);
    if ((flags & TB_COLUMN_ALLOCATED) &&
        (oldv == 0 or TABLE_NOVALUE)) {
        kov = UNKNOWN_TY;
        tc = ((tab-->col)-->1) & TB_COLUMN_NUMBER;
        for (j=0: TC_KOVs-->j: j=j+3)
            if (TC_KOVs-->j == tc) {
                kov = TC_KOVs-->(j+1); break;
            }
        if (kov ~= UNKNOWN_TY)
            (tab-->col)-->(row+COL_HSIZE) = BlkValueCreate(kov, 0, TC_KOVs-->(j+2));
    }
    row--;
    at = ((tab-->col)-->2) + (row/8);
    (TB_Blanks->at) = (TB_Blanks->at) & (CheckTableEntryIsNonBlank_LU->(row%8));
];
```

§**9. Swapping Blank Bits.** When sorting a table, we obviously need to swap rows from time to time; if any of its columns have blanks bitmaps, then the relevant bits in them need to be swapped to match, and the following routine performs this operation for two rows in a given column.

```
[ TableSwapBlankBits tab row1 row2 col at1 at2 bit1 bit2;
    if (col >= 100) col=TableFindCol(tab, col);
    if (col == 0) rtrue;
    if (((tab-->col)-->1) & TB_COLUMN_NOBLANKBITS) return;
    row1--;
    at1 = ((tab-->col)-->2) + (row1/8);
    row2--;
    at2 = ((tab-->col)-->2) + (row2/8);
    bit1 = ((TB_Blanks->at1) & (CheckTableEntryIsBlank_LU->(row1%8)));
    bit2 = ((TB_Blanks->at2) & (CheckTableEntryIsBlank_LU->(row2%8)));
    if (bit1) bit1 = true;
    if (bit2) bit2 = true;
    if (bit1 == bit2) return;
    if (bit1) {
        (TB_Blanks->at1)
            = (TB_Blanks->at1) & (CheckTableEntryIsNonBlank_LU->(row1%8));
        (TB_Blanks->at2)
            = (TB_Blanks->at2) | (CheckTableEntryIsBlank_LU->(row2%8));
    } else {
        (TB_Blanks->at1)
            = (TB_Blanks->at1) | (CheckTableEntryIsBlank_LU->(row1%8));
        (TB_Blanks->at2)
            = (TB_Blanks->at2) & (CheckTableEntryIsNonBlank_LU->(row2%8));
    }
];
```

§**10. Moving Blank Bits Down.** Another common table operation is to compress it by moving all the blank rows down to the bottom, so that non-blank rows occur in a contiguous block at the top: this means table sorting can be done without having to refer continually to the blanks bitmaps. The following operation is useful for keeping the blanks bitmaps up to date when blank rows are moved down.

```
[ TableMoveBlankBitsDown tab row1 row2 col at atp1 bit rx;
    if (col >= 100) col=TableFindCol(tab, col);
    if (col == 0) rtrue;
    if (((tab-->col)-->1) & TB_COLUMN_NOBLANKBITS) return;
    row1--; row2--;
    ! Read blank bit for row1:
    at = ((tab-->col)-->2) + (row1/8);
    bit = ((TB_Blanks->at) & (CheckTableEntryIsBlank_LU->(row1%8)));
    if (bit) bit = true;
    ! Loop through, setting each blank bit to the next:
    for (rx=row1:rx<row2:rx++) {
        atp1 = ((tab-->col)-->2) + ((rx+1)/8);
        at = ((tab-->col)-->2) + (rx/8);
        if ((TB_Blanks->atp1) & (CheckTableEntryIsBlank_LU->((rx+1)%8))) {
            (TB_Blanks->at)
                = (TB_Blanks->at) | (CheckTableEntryIsBlank_LU->(rx%8));
        } else {
            (TB_Blanks->at)
```

```
                = (TB_Blanks->at) & (CheckTableEntryIsNonBlank_LU->(rx%8));
        }
    }
    ! Write bit to blank bit for row2:
    at = ((tab-->col)-->2) + (row2/8);
    if (bit) {
        (TB_Blanks->at)
            = (TB_Blanks->at) | (CheckTableEntryIsBlank_LU->(row2%8));
    } else {
        (TB_Blanks->at)
            = (TB_Blanks->at) & (CheckTableEntryIsNonBlank_LU->(row2%8));
    }
];
```

§**11. Table Row Corresponding.** `TableRowCorr(T, C, V)` returns the first row on which value `V` appears in column `C` of table `T`, or prints an error if it doesn't.

`ExistsTableRowCorr(T, C, V)` returns the first row on which `V` appears in column `C` of table `T`, or 0 if `V` does not occur at all. If the column is a topic, then we match the entry as a snippet against the value as a general parsing routine.

```
[ TableRowCorr tab col lookup_value lookup_col i j f;
    if (col >= 100) col=TableFindCol(tab, col, true);
    lookup_col = tab-->col;
    j = lookup_col-->0 - COL_HSIZE;
    f=0;
    if (((tab-->col)-->1) & TB_COLUMN_ALLOCATED) f=1;
    for (i=1:i<=j:i++) {
        if ((lookup_value == TABLE_NOVALUE) &&
            (CheckTableEntryIsBlank(tab,col,i))) continue;
        if (f) {
            if (BlkValueCompare(lookup_col-->(i+COL_HSIZE), lookup_value) == 0)
                return i;
        } else {
            if (lookup_col-->(i+COL_HSIZE) == lookup_value) return i;
        }
    }
    return RunTimeProblem(RTP_TABLE_NOCORR, tab);
];
[ ExistsTableRowCorr tab col entry i k v f kov;
    if (col >= 100) col=TableFindCol(tab, col);
    if (col == 0) rfalse;
    f=0;
    if (((tab-->col)-->1) & TB_COLUMN_TOPIC) f=1;
    else if (((tab-->col)-->1) & TB_COLUMN_ALLOCATED) f=2;
    k = TableRows(tab);
    for (i=1:i<=k:i++) {
        ! print "Checking row ", i, "^";
        v = (tab-->col)-->(i+COL_HSIZE);
        if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,i))) continue;
        switch (f) {
            1: if ((v)(entry/100, entry%100) ~= GPR_FAIL) return i;
            2: if (BlkValueCompare(v, entry) == 0) return i;
```

```
            default: if (v == entry) return i;
        }
    }
    ! print "Giving up^";
    return 0;
];
```

§**12. Table Look Up Corresponding Row.** `TableLookUpCorr(T, C1, C2, V)` finds the first row on which value `V` appears in column `C2`, and returns the corresponding value in `C1`, or prints an error if the value `V` cannot be found or has no corresponding value in `C1`.

`ExistsTableLookUpCorr(T, C1, C2, V)` returns `true` if the operation `TableLookUpCorr(T, C1, C2, V)` can be done, `false` otherwise.

```
[ TableLookUpCorr tab col1 col2 lookup_value write_flag write_value cola1 cola2 i j v f;
    if (col1 >= 100) col1=TableFindCol(tab, col1, true);
    if (col2 >= 100) col2=TableFindCol(tab, col2, true);
    cola1 = tab-->col1;
    cola2 = tab-->col2;
    j = cola2-->0;
    f=0;
    if (((tab-->col2)-->1) & TB_COLUMN_ALLOCATED) f=1;
    for (i=1+COL_HSIZE:i<=j:i++) {
        if (f) {
            if (BlkValueCompare(cola2-->i, lookup_value) ~= 0) continue;
        } else {
            if (cola2-->i ~= lookup_value) continue;
        }
        if (write_flag) {
            ForceTableEntryNonBlank(tab,col1,i-COL_HSIZE);
            cola1-->i = write_value;
            rfalse;
        }
        v = cola1-->i;
        if ((v == TABLE_NOVALUE) &&
            (CheckTableEntryIsBlank(tab,col1,i-COL_HSIZE))) continue;
        return v;
    }
    return RunTimeProblem(RTP_TABLE_NOCORR, tab);
];
[ ExistsTableLookUpCorr tab col1 col2 lookup_value cola1 cola2 i j f;
    if (col1 >= 100) col1=TableFindCol(tab, col1, false);
    if (col2 >= 100) col2=TableFindCol(tab, col2, false);
    if (col1*col2 == 0) rfalse;
    cola1 = tab-->col1; cola2 = tab-->col2;
    j = cola2-->0;
    f=0;
    if (((tab-->col2)-->1) & TB_COLUMN_ALLOCATED) f=1;
    for (i=1+COL_HSIZE:i<=j:i++) {
        if (f) {
            if (BlkValueCompare(cola2-->i, lookup_value) ~= 0) continue;
        } else {
            if (cola2-->i ~= lookup_value) continue;
```

```
        }
        if ((cola1-->i == TABLE_NOVALUE) &&
            (CheckTableEntryIsBlank(tab,col1,i-COL_HSIZE))) continue;
        rtrue;
    }
    rfalse;
];
```

§13. Table Look Up Entry. `TableLookUpEntry(T, C, R)` returns the value at column `C`, row `R`, printing an error if that doesn't exist.

`ExistsTableLookUpEntry(T, C, R)` returns true if a value exists at column `C`, row `R`, false otherwise.

```
[ TableLookUpEntry tab col index write_flag write_value v;
    if (col >= 100) col=TableFindCol(tab, col, true);
    if ((index < 1) || (index > TableRows(tab)))
        return RunTimeProblem(RTP_TABLE_NOROW, tab, index);
    if (write_flag) {
        switch(write_flag) {
            1: ForceTableEntryNonBlank(tab,col,index);
                (tab-->col)-->(index+COL_HSIZE) = write_value;
            2: ForceTableEntryNonBlank(tab,col,index);
                (tab-->col)-->(index+COL_HSIZE) =
                    ((tab-->col)-->(index+COL_HSIZE)) + write_value;
            3: ForceTableEntryNonBlank(tab,col,index);
                (tab-->col)-->(index+COL_HSIZE) =
                    ((tab-->col)-->(index+COL_HSIZE)) - write_value;
            4: ForceTableEntryBlank(tab,col,index);
            5: ForceTableEntryNonBlank(tab,col,index);
                return ((tab-->col)-->(index+COL_HSIZE));
        }
        rfalse;
    }
    v = ((tab-->col)-->(index+COL_HSIZE));
    if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,index)))
        return RunTimeProblem(RTP_TABLE_NOENTRY, tab, col, index);
    return v;
];
[ ExistsTableLookUpEntry tab col index v;
    if (col >= 100) col=TableFindCol(tab, col);
    if (col == 0) rfalse;
    if ((index<1) || (index > TableRows(tab))) rfalse;
    v = ((tab-->col)-->(index+COL_HSIZE));
    if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,index)))
        rfalse;
    rtrue;
];
```

§14. Blank Rows. `TableRowIsBlank(T, R)` returns true if row `R` of table `T` is blank. (`R` must be a legal row number.)

`TableBlankOutRow(T, R)` fills row `R` of table `T` with blanks. (`R` must be a legal row number.)

`TableBlankRows(T)` returns the number of blank rows in `T`.

`TableFilledRows(T)` returns the number of non-blank rows in `T`.

`TableBlankRow(T)` finds the first blank row in `T`.

```
[ TableRowIsBlank tab j k;
    for (k=1:k<=tab-->0:k++) {
        if (((tab-->k)-->(j+COL_HSIZE)) ~= TABLE_NOVALUE) rfalse;
        if (CheckTableEntryIsBlank(tab, k, j) == false) rfalse;
    }
    rtrue;
];
[ TableBlankOutRow tab j k;
    if (tab==0) return RunTimeProblem(RTP_TABLE_NOTABLE, tab);
    for (k=1:k<=tab-->0:k++)
        ForceTableEntryBlank(tab, k, j);
];
[ TableBlankRows tab i j c;
    i = TableRows(tab); !print i, " rows^";
    for (j=1:j<=i:j++)
        if (TableRowIsBlank(tab, j)) c++;
    !print c, " blank^";
    return c;
];
[ TableFilledRows tab;
    return TableRows(tab) - TableBlankRows(tab);
];
[ TableBlankRow tab i j;
    i = TableRows(tab);
    for (j=1:j<=i:j++)
        if (TableRowIsBlank(tab, j)) return j;
    RunTimeProblem(RTP_TABLE_NOMOREBLANKS, tab);
    return i;
];
```

§15. Random Row. `TableRandomRow(T)` chooses a random non-blank row in `T`.

```
[ TableRandomRow tab i j k;
    i = TableRows(tab);
    j = TableFilledRows(tab);
    if (j==0) return RunTimeProblem(RTP_TABLE_NOROWS, tab);
    if (j>1) j = random(j);
    for (k=1:k<=i:k++) {
        if (TableRowIsBlank(tab, k) == false) j--;
        if (j==0) return k;
    }
];
```

§**16. Swap Rows.** `TableSwapRows(T, R1, R2)` exchanges rows `R1` and `R2`.

```
[ TableSwapRows tab i j k l v1 v2;
    if (i==j) return;
    l = tab-->0;
    for (k=1:k<=l:k++) {
        v1 = (tab-->k)-->(i+COL_HSIZE);
        v2 = (tab-->k)-->(j+COL_HSIZE);
        (tab-->k)-->(i+COL_HSIZE) = v2;
        (tab-->k)-->(j+COL_HSIZE) = v1;
        if ((v1 == TABLE_NOVALUE) || (v2 == TABLE_NOVALUE))
            TableSwapBlankBits(tab, i, j, k);
    }
];
```

§**17. Compare Rows.** `TableCompareRows(T, C, R1, R2, D)` returns:

(a) +1 if the entry at row R1 of column C is > the entry at row R2,
(b) 0 if they are equal, and
(c) −1 if entry at R1 < entry at R2.

When $D = +1$, a blank value is greater than all other values, so that in an ascending sort the blanks come last; when $D = -1$, a blank value is less than all others, so that once again blanks are last. Finally, a wholly blank row is always placed after a row in which the entry in C is blank but where other entries are not.

```
[ TableCompareRows tab col row1 row2 dir val1 val2 bl1 bl2;
    if (col >= 100) col=TableFindCol(tab, col, false);
    val1 = (tab-->col)-->(row1+COL_HSIZE);
    val2 = (tab-->col)-->(row2+COL_HSIZE);
    if (val1 == TABLE_NOVALUE) bl1 = CheckTableEntryIsBlank(tab,col,row1);
    if (val2 == TABLE_NOVALUE) bl2 = CheckTableEntryIsBlank(tab,col,row2);
    if ((val1 == val2) && (bl1 == bl2)) {
        if (val1 ~= TABLE_NOVALUE) return 0;
        if (bl1 == false) return 0;
        ! The two entries are both blank:
        if (TableRowIsBlank(tab, row1)) {
            if (TableRowIsBlank(tab, row2)) return 0;
            return -1*dir;
        }
        if (TableRowIsBlank(tab, row2)) return dir;
        return 0;
    }
    if (bl1) return -1*dir;
    if (bl2) return dir;
    if (((tab-->col)-->1) & TB_COLUMN_SIGNED) {
        if (val1 > val2) return 1;
        return -1;
    } else {
        if (UnsignedCompare(val1, val2) > 0) return 1;
        return -1;
    }
];
```

§18. Move Row Down.

```
[ TableMoveRowDown tab r1 r2 rx k l m v f;
    if (r1==r2) return;
    l = tab-->0;
    for (k=1:k<=l:k++) {
        f = false;
        m = (tab-->k)-->(r1+COL_HSIZE);
        if (m == TABLE_NOVALUE) f = true;
        for (rx=r1:rx<r2:rx++) {
            v = (tab-->k)-->(rx+COL_HSIZE+1);
            (tab-->k)-->(rx+COL_HSIZE) = v;
            if (v == TABLE_NOVALUE) f = true;
        }
        (tab-->k)-->(r2+COL_HSIZE) = m;
        if (f) TableMoveBlankBitsDown(tab, r1, r2, k);
    }
];
```

§19. Shuffle. `TableShuffle(T)` sorts `T` into random row order.

```
[ TableShuffle tab i j k;
    k = TableRows(tab);
    for (i=2:i<=k:i++) TableSwapRows(tab, i, random(i));
    TableMoveBlanksToBack(tab, 1, k);
];
```

§20. Next Row. `TableNextRow(T, C, R, D)` is used when scanning through a table in order of the values in column `C`: ascending order if `D = 1`, descending if `D = -1`. The current position is row `R` of column `C`, or `R = 0` if we have not yet found the first row. The return value is the row number for the next value, or 0 if we are already at the final value. Note that if there are several equal values in the column, they will be run through in turn, in order of their physical row numbers - ascending if `D = 1`, descending if `D = -1`, so that using the routine with `D = -1` always produces the exact reverse ordering from using it with `D = 1` and the same parameters. Rows with blank entries in `C` are skipped.

```
for (R=TableNextRow(T,C,0,D): R : R=TableNextRow(T,C,R,D)) ...
```

will perform a loop of valid row numbers in order of column `C`.

```
[ TableNextRow tab col row dir i k val v dv min_dv min_at signed_arithmetic f;
    if (col >= 100) col=TableFindCol(tab, col, false);
    signed_arithmetic = ((tab-->col)-->1) & TB_COLUMN_SIGNED;
    #Iftrue (WORDSIZE == 2);
    if (row == 0) {
        if (signed_arithmetic) {
            if (dir == 1) val = $8000; else val = $7fff;
        } else {
            if (dir == 1) val = 0; else val = $ffff;
        }
    } else val = (tab-->col)-->(row+COL_HSIZE);
    if (signed_arithmetic) min_dv = $7fff; else min_dv = $ffff;
    #ifnot; ! WORDSIZE == 4
    if (row == 0) {
        if (signed_arithmetic) {
```

```
            if (dir == 1) val = $80000000; else val = $7fffffff;
        } else {
            if (dir == 1) val = 0; else val = $ffffffff;
        }
    } else val = (tab-->col)-->(row+COL_HSIZE);
    if (signed_arithmetic) min_dv = $7fffffff; else min_dv = $ffffffff;
    #endif;
    k = TableRows(tab);
    if (dir == 1) {
        for (i=1:i<=k:i++) {
            v = (tab-->col)-->(i+COL_HSIZE);
            if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,i)))
                continue;
            dv = dir*v;
            if (signed_arithmetic)
            f = (((dv > dir*val) || ((v == val) && (i>row))) &&
                (dv < min_dv));
            else
            f = (((UnsignedCompare(dv, dir*val) > 0) || ((v == val) && (i>row))) &&
                (UnsignedCompare(dv, min_dv) < 0));
            if (f) { min_dv = dv; min_at = i; }
        }
    } else {
        for (i=k:i>=1:i--) {
            v = (tab-->col)-->(i+COL_HSIZE);
            if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,i)))
                continue;
            dv = dir*v;
            if (signed_arithmetic)
            f = (((dv > dir*val) || ((v == val) && (i<row))) &&
                (dv < min_dv));
            else
            f = (((UnsignedCompare(dv, dir*val) > 0) || ((v == val) && (i<row))) &&
                (UnsignedCompare(dv, min_dv) < 0));
            if (f) { min_dv = dv; min_at = i; }
        }
    }
    return min_at;
];
```

§21. Move Blanks to Back.

```
[ TableMoveBlanksToBack tab fromrow torow i fbl lnbl blc;
    if (torow < fromrow) return;
    fbl = 0; lnbl = 0;
    for (i=fromrow: i<=torow: i++)
        if (TableRowIsBlank(tab, i)) {
            if (fbl == 0) fbl = i;
            blc++;
        } else {
            lnbl = i;
        }
    if ((fbl>0) && (lnbl>0) && (fbl < lnbl)) {
        TableMoveRowDown(tab, fbl, lnbl); ! Move first blank just past last nonblank
        TableMoveBlanksToBack(tab, fbl, lnbl-1);
    }
    return torow-blc; ! Final non-blank row
];
```

§22. Sort. This is really only a front-end: it calls the sorting code at "Sort.i6t".

```
[ TableSort tab col dir test_flag algorithm i j k f;
    for (i=1:i<=tab-->0:i++) {
        j = tab-->i; ! Address of column table
        if ((j-->1) & TB_COLUMN_DONTSORTME)
            return RunTimeProblem(RTP_TABLE_CANTSORT, tab);
    }
    if (col >= 100) col=TableFindCol(tab, col, false);
    k = TableRows(tab);
    k = TableMoveBlanksToBack(tab, 1, k);
    if (test_flag) {
        print "After moving blanks to back:^"; TableColumnDebug(tab, col);
    }
    SetSortDomain(TableSwapRows, TableCompareRows);
    SortArray(tab, col, dir, k, test_flag, algorithm);
    if (test_flag) {
        print "Final state:^"; TableColumnDebug(tab, col);
    }
];
```

§23. Print Table Name. NI fills this in: it's used to say the "table" kind of value.

```
[ PrintTableName T;
    switch(T) {
{-call:compile_print_table_names}
        default: print "** No such table **";
    }
];
```

§**24. Print Table to File.** This is how we serialise a table to an external file, though the writing is done by printing characters in the standard way; it's just that the output stream will be an external file rather than the screen when this routine is called.

```
[ TablePrint tab i j k row col v tc kov;
    for (i=1:i<=tab-->0:i++) {
        j = tab-->i; ! Address of column table
        if (((j-->1) & TB_COLUMN_CANEXCHANGE) == 0)
            rtrue;
    }
    k = TableRows(tab);
    k = TableMoveBlanksToBack(tab, 1, k);
    print "! ", (PrintTableName) tab, " (", k, ")^";
    for (row=1:row<=k:row++) {
        for (col=1:col<=tab-->0:col++) {
            tc = ((tab-->col)-->1) & TB_COLUMN_NUMBER;
            kov = NUMBER_TY;
            for (i=0: TC_KOVs-->i: i=i+3)
                if (TC_KOVs-->i == tc)
                    kov = TC_KOVs-->(i+1);
            v = (tab-->col)-->(row+COL_HSIZE);
            if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,row)))
                print "-- ";
            else {
                if (BlkValueWriteToFile(v, kov) == false) print v;
                print " ";
            }
        }
        print "^";
    }
    rfalse;
];
```

§**25. Read Table from File.** And this is how we unserialise again. It makes sense only on Glulx.

```
#ifdef TARGET_GLULX;
[ TableRead tab auxf row maxrow col ch v sgn dg j tc kov;
    for (col=1:col<=tab-->0:col++) {
        j = tab-->col; ! Address of column table
        if (((j-->1) & TB_COLUMN_CANEXCHANGE) == 0)
            return RunTimeProblem(RTP_TABLE_CANTSAVE, tab);
    }
    maxrow = TableRows(tab);
    !print maxrow, " rows available.^";
    for (row=1: row<=maxrow: row++) {
        TableBlankOutRow(tab, row);
    }
    for (row=1: row<=maxrow: row++) {
        !print "Reading row ", row, "^";
        ch = FileIO_GetC(auxf);
        if (ch == '!') {
            while (ch ~= -1 or 10 or 13) ch = FileIO_GetC(auxf);
            while (ch == 10 or 13) ch = FileIO_GetC(auxf);
```

```
    }
    for (col=1: col<=tab-->0: col++) {
        if (ch == -1) { row++; jump NoMore; }
        if (ch == 10 or 13) break;
        tc = ((tab-->col)-->1) & TB_COLUMN_NUMBER;
        kov = NUMBER_TY;
        for (j=0: TC_KOVs-->j: j=j+3)
            if (TC_KOVs-->j == tc)
                kov = TC_KOVs-->(j+1);
        !print "tc = ", tc, " kov = ", kov, "^";
        sgn = 1;
        if (ch == '-') {
            ch = FileIO_GetC(auxf);
            if (ch == -1) jump NotTable;
            if (ch == '-') { ch = FileIO_GetC(auxf); jump EntryDone; }
            sgn = -1;
        }
        if (((tab-->col)-->1) & TB_COLUMN_ALLOCATED)
            ForceTableEntryNonBlank(tab, col, row);
        !print "A";
        v = BlkValueReadFromFile(0, 0, -1, kov);
        if (v) {
            if (((tab-->col)-->1) & TB_COLUMN_ALLOCATED)
                v = BlkValueReadFromFile(TableLookUpEntry(tab, col, row),
                    auxf, ch, kov);
            else
                v = BlkValueReadFromFile(0, auxf, ch, kov);
            ch = 32;
        } else {
            dg = ch - '0';
            if ((dg < 0) || (dg > 9)) jump NotTable;
            v = dg;
            for (::) {
                ch = FileIO_GetC(auxf);
                dg = ch - '0';
                if ((dg < 0) || (dg > 9)) break;
                v = 10*v + dg;
            }
            v = v*sgn;
        }
        !print "v=", v, " ";
        if (((tab-->col)-->1) & TB_COLUMN_ALLOCATED == 0)
            TableLookUpEntry(tab, col, row, true, v);
        .EntryDone;
        !print "First nd is ", ch, "^";
        while (ch == 9 or 32) ch = FileIO_GetC(auxf);
    }
    while (ch ~= -1 or 10 or 13) {
        if ((ch ~= '-') && (((ch-'0')<0) || ((ch-'0')>9))) jump NotTable;
        if (ch ~= 9 or 32) jump WontFit;
        ch = FileIO_GetC(auxf);
    }
}
```

```
    .NoMore;
    while (ch == 9 or 32 or 10 or 13) ch = FileIO_GetC(auxf);
    if (ch == -1) return;
    .WontFit;
    return RunTimeProblem(RTP_TABLE_WONTFIT, tab);
    .NotTable;
    return RunTimeProblem(RTP_TABLE_BADFILE, tab);
];
#ENDIF; ! TARGET_GLULX
```

§**26. Print Rank.** The table of scoring ranks is a residue from the ancient times of early IF: it gets a tiny amount of special treatment here, even though I7 works tend not to use these now dated conventions.

```
[ PrintRank i j v;
#ifdef RANKING_TABLE;
    L__M(##Score, 3);
    j = TableRows(RANKING_TABLE);
    for (i=j:i>=1:i--)
        if (score >= TableLookUpEntry(RANKING_TABLE, 1, i)) {
            v = TableLookUpEntry(RANKING_TABLE, 2, i);
            if (v ofclass String) print (string) v;
            else v();
            ".";
        }
#endif;
    ".";
];
```

§**27. Debugging.** A routine to print the state of a table, for debugging purposes only.

```
[ TableColumnDebug tab col k i v;
    if (col >= 100) col=TableFindCol(tab, col, false);
    k = TableRows(tab);
    print "Table col ", col, ": ";
    for (i=1:i<=k:i++) {
        v = (tab-->col)-->(i+COL_HSIZE);
        if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,i)))
            print "BLANK ";
        else print v, " ";
    }
    print "*^";
];
```

Sort Template B/sortt

Purpose

To sort arrays.

§**1. Storage.** We are required to use a stable sorting algorithm with very low, ideally zero, auxiliary storage requirement. Exchanges are generally slower than comparisons for the typical application (sorting tables, where entire rows must be exchanged whereas only entries in a single column need be compared).

In fact, we store some details in global variables for convenience and to avoid filling the stack with copies, but otherwise we will hardly need any auxiliary storage.

```
Global I7S_Tab; ! The array to be sorted, which can have almost any format
Global I7S_Col; ! The "column number" in the array, if any
Global I7S_Dir; ! The direction of sorting: ascending (1) or descending (-1)
Global I7S_Swap; ! The current routine for swapping two fields
Global I7S_Comp; ! The current routine for comparing two fields
#ifdef MEASURE_SORT_PERFORMANCE;
Global I7S_CCOUNT; Global I7S_CCOUNT2; Global I7S_XCOUNT; ! For testing only
#endif;
```

§**2. Front End.** To perform a sort, we first call `SetSortDomain` to declare the swap and compare functions to be used, and then call `SortArray` actually to sort. (It would be nice to combine these in a single call, but I6 allows a maximum of 7 call arguments for a routine, and that would make 8.) These are the only two routines which should ever be called from outside of this template segment.

The swap and compare functions are expected to take two arguments, which are the field numbers of the fields being swapped or compared, where fields are numbered from 1. Comparison is like `strcmp`: it returns 0 on equality, and then is positive or negative according to which of the fields is greater in value.

```
[ SetSortDomain swapf compf;
    I7S_Swap = swapf;
    I7S_Comp = compf;
];
[ SortArray tab col dir size test_flag algorithm;
    I7S_Tab = tab;
    I7S_Col = col;
    I7S_Dir = dir;
    #ifdef MEASURE_SORT_PERFORMANCE;
    I7S_CCOUNT = 0;
    I7S_CCOUNT2 = 0;
    I7S_XCOUNT = 0;
    #endif;
    SortRange(0, size, algorithm);
    #ifdef MEASURE_SORT_PERFORMANCE;
    if (test_flag)
        print "Sorted array of size ", size, " with ", I7S_CCOUNT2, "*10000 + ", I7S_CCOUNT,
            " comparisons and ", I7S_XCOUNT, " exchanges^";
    #endif;
];
```

§**3. Sort Range.** This routine sorts a range of fields $x \le i < y$ within the array. Fields are numbered from 0. The supplied `algorithm` is an I6 routine to implement a particular sorting algorithm; if it is not supplied, then in-place merge sort is used by default.

```
[ SortRange x y algorithm;
    if (y - x < 2) return;
    if (algorithm) {
        (algorithm)(x, y);
    } else {
        InPlaceMergeSortAlgorithm(x, y);
    }
];
```

§**4. Comparison and Exchange.** These are instrumented versions of how to swap and compare fields; note that the swap and compare functions are expected to number the fields from 1, not from 0. (This is convenient both for tables and lists, where rows and entries respectively are both numbered from 1.) The only access which the sorting algorithms have to the actual data being sorted is through these routines.

```
[ CompareFields x y;
    #ifdef MEASURE_SORT_PERFORMANCE;
    I7S_CCOUNT++;
    if (I7S_CCOUNT == 10000) { I7S_CCOUNT = 0; I7S_CCOUNT2++; }
    #endif;
    return I7S_Dir*I7S_Comp(I7S_Tab, I7S_Col, x+1, y+1, I7S_Dir);
];
[ ExchangeFields x y;
    #ifdef MEASURE_SORT_PERFORMANCE;
    I7S_XCOUNT++;
    if (I7S_XCOUNT < 0) { print "XO^"; I7S_XCOUNT = 0; }
    #endif;
    return I7S_Swap(I7S_Tab, x+1, y+1);
];
```

§**5. 4W37 Sort.** We now present three alternative sorting algorithms.

The first is the one used in builds up to and including 4W37: note that this is not quite bubble sort, and that it is unstable. It is now no longer used, but is so short that we might as well keep it in the code base in case anyone needs to resurrect a very early I7 project.

```
[ OldSortAlgorithm x y
    f i j;
    if (y - x < 2) return;
    f = true;
    while (f) {
        f = false;
        for (i=x:i<y:i++)
            for (j=i+1:j<y:j++)
                if (CompareFields(i, j) > 0) {
                    ExchangeFields(i, j); f = true; break;
                }
    }
];
```

§**6. Insertion Sort.** A stable algorithm which has $O(n^2)$ running time and therefore cannot be used with large arrays, but which has good performance on nearly sorted tables, and which has very low overhead.

```
[ InsertionSortAlgorithm from to
    i j;
    if (to > from+1) {
        for (i = from+1: i < to: i++) {
            for (j = i: j > from: j--) {
                if (CompareFields(j, j-1) < 0)
                    ExchangeFields(j, j-1);
                else break;
            }
        }
    }
];
```

§**7. In-Place Mergesort.** A stable algorithm with $O(n \log n)$ running time, at some stack cost, and which is generally good for nearly sorted tables, but which is also complex and has some overhead. The code here mostly follows Thomas Baudel's implementation, which in turn follows the `C++` STL library.

```
[ InPlaceMergeSortAlgorithm from to
    middle;
    if (to - from < 12) {
        if (to - from < 2) return;
        InsertionSortAlgorithm(from, to);
        return;
    }
    middle = (from + to)/2;
    InPlaceMergeSortAlgorithm(from, middle);
    InPlaceMergeSortAlgorithm(middle, to);
    IPMS_Merge(from, middle, to, middle-from, to - middle);
];
[ IPMS_Lower from to val
    len half mid;
    len = to - from;
    while (len > 0) {
        half = len/2;
        mid = from + half;
        if (CompareFields(mid, val) < 0) {
            from = mid + 1;
            len = len - half -1;
        } else len = half;
    }
    return from;
];
[ IPMS_Upper from to val
    len half mid;
    len = to - from;
    while (len > 0) {
        half = len/2;
        mid = from + half;
        if (CompareFields(val, mid) < 0)
```

```
            len = half;
        else {
            from = mid + 1;
            len = len - half -1;
        }
    }
    return from;
];
[ IPMS_Reverse from to;
    while (from < to) {
        ExchangeFields(from++, to--);
    }
];
[ IPMS_Rotate from mid to
    n val shift p1 p2;
    if ((from==mid) || (mid==to)) return;
    IPMS_Reverse(from, mid-1);
    IPMS_Reverse(mid, to-1);
    IPMS_Reverse(from, to-1);
];
[ IPMS_Merge from pivot to len1 len2
    first_cut second_cut len11 len22 new_mid;
    if ((len1 == 0) || (len2 == 0)) return;
    if (len1+len2 == 2) {
        if (CompareFields(pivot, from) < 0)
        ExchangeFields(pivot, from);
        return;
    }
    if (len1 > len2) {
        len11 = len1/2;
        first_cut = from + len11;
        second_cut = IPMS_Lower(pivot, to, first_cut);
        len22 = second_cut - pivot;
    } else {
        len22 = len2/2;
        second_cut = pivot + len22;
        first_cut = IPMS_Upper(from, pivot, second_cut);
        len11 = first_cut - from;
    }
    IPMS_Rotate(first_cut, pivot, second_cut);
    new_mid = first_cut + len22;
    IPMS_Merge(from, first_cut, new_mid, len11, len22);
    IPMS_Merge(new_mid, second_cut, to, len1 - len11, len2 - len22);
];
```

Relations Template B/relt

Purpose

To manage run-time storage for relations between objects, and to find routes through relations and the map.

---

§**1. One To One Relations.** We provide routines to assert a 1-to-1 relation true, or to assert it false. The relation `rel` is represented by a property number, and the property in question is used to store the fact of a relationship: $O_1 \sim O_2$ if and only if `O1.rel == O2`.

There is no routine to test a 1-to-1 relation, since the predicate calculus code in NI simplifies propositions which test these into direct looking up of the property relation.

```
[ Relation_Now1to1 obj1 relation_property obj2 ol; ! Assert 1-1 true
    if (obj2) objectloop (ol provides relation_property)
        if (ol.relation_property == obj2) ol.relation_property = nothing;
    if (obj1) obj1.relation_property = obj2;
];
[ Relation_NowN1toV obj1 relation_property obj2; ! Assert 1-1 false
    if ((obj1) && (obj1.relation_property == obj2)) obj1.relation_property = nothing;
];
```

§**2. Symmetric One To One Relations.** Here the relation is used for both objects: $O_1 \sim O_2$ if and only if both `O1.relation_property == O2` and `O2.relation_property == O1`.

```
[ Relation_NowS1to1 obj1 relation_property obj2 ol; ! Assert symmetric 1-1 true
    if ((obj1 ofclass Object) && (obj1 provides relation_property) &&
        (obj2 ofclass Object) && (obj2 provides relation_property)) {
        if (obj1.relation_property) { (obj1.relation_property).relation_property = 0; }
        if (obj2.relation_property) { (obj2.relation_property).relation_property = 0; }
        obj1.relation_property = obj2; obj2.relation_property = obj1;
    }
];
[ Relation_NowSN1to1 obj1 relation_property obj2 ol; ! Assert symmetric 1-1 false
    if ((obj1 ofclass Object) && (obj1 provides relation_property) &&
        (obj2 ofclass Object) && (obj2 provides relation_property) &&
        (obj1.relation_property == obj2)) {
        obj1.relation_property = 0; obj2.relation_property = 0;
    }
];
```

§**3. Various To Various Relations.** Here the relation is represented by an array holding its metadata. Each object in the domain of the relation provides two properties, holding its left index and its right index. The index is its position in the left or right domain. For instance, suppose we relate things to doors, and there are five things in the world, two of which are doors; then the left indexes will range from 0 to 4, while the right indexes will range from 0 to 1. It's very likely that the doors will have different left and right indexes. (If the relation relates a given kind to itself, say doors to doors, then left and right indexes will always be equal.)

It is possible for either the left or right domain set to be an enumerated kind of value, where the I6 representation of values is 1, 2, 3, ..., N, where there are N possibilities. In that case we obtain the index simply by subtracting 1 in order to begin from 0. We mark the domain set as being a KOV rather than a kind of object by storing 0 instead of a property in the relevant part of the relation metadata: note that 0 is not a valid property number.

The structure for a relation consists of eight `-->` words, followed by a bitmap in which we store 16 bits in each `-->` word. (Yes, this is wasteful in Glulx, where `-->` words store 32 bits, but memory is not in short supply in Glulx and the total cost of relations is in practice small; we prefer to keep all the code involved simple.) The structure is precompiled by the NI compiler: we do not create new ones on the fly.

In the case of a symmetric various to various relation, we could in theory save memory once again by storing only the lower triangle of the bitmap, but the time and complexity overhead are not worth it. When asserting that $O_1 \sim O_2$ for a symmetric V-to-V, we also automatically assert that $O_2 \sim O_1$, thus maintaining the bitmap as a symmetric matrix; but in reading the bitmap, we look only at the lower triangle. This costs a little time, but has the advantage of allowing the route-finding routine for V-to-V to use the same code for symmetric and asymmetric relations.

If this all seems rather suboptimally programmed in order to reduce code complexity, I can only say that careless drafts here were the source of some extremely difficult bugs to find.

```
Constant VTOVS_LEFT_INDEX_PROP = 0;
Constant VTOVS_RIGHT_INDEX_PROP = 1;
Constant VTOVS_LEFT_DOMAIN_SIZE = 2;
Constant VTOVS_RIGHT_DOMAIN_SIZE = 3;
Constant VTOVS_LEFT_PRINTING_ROUTINE = 4;
Constant VTOVS_RIGHT_PRINTING_ROUTINE = 5;
Constant VTOVS_CACHE_BROKEN = 6;
Constant VTOVS_CACHE = 7;

[ Relation_NowVtoV obj1 vtov_structure obj2 sym pr pr2 i1 i2;
    if (sym && (obj2 ~= obj1)) { Relation_NowVtoV(obj2, vtov_structure, obj1, false); }
    pr = vtov_structure-->VTOVS_LEFT_INDEX_PROP;
    pr2 = vtov_structure-->VTOVS_RIGHT_INDEX_PROP;
    vtov_structure-->VTOVS_CACHE_BROKEN = true; ! Mark any cache as broken
    if (pr) {
        if ((obj1 ofclass Object) && (obj1 provides pr)) i1 = obj1.pr;
        else return RunTimeProblem(RTP_IMPREL, obj1, vtov_structure);
    } else i1 = obj1-1;
    if (pr2) {
        if ((obj2 ofclass Object) && (obj2 provides pr2)) i2 = obj2.pr2;
        else return RunTimeProblem(RTP_IMPREL, obj2, vtov_structure);
    } else i2 = obj2-1;
    pr = i1*(vtov_structure-->VTOVS_RIGHT_DOMAIN_SIZE) + i2;
    i1 = IncreasingPowersOfTwo_TB-->(pr%16);
    pr = pr/16 + 8;
    vtov_structure-->pr = (vtov_structure-->pr) | i1;
];

[ Relation_NowNVtoV obj1 vtov_structure obj2 sym pr pr2 i1 i2;
```

```
    if (sym && (obj2 ~= obj1)) { Relation_NowNVtoV(obj2, vtov_structure, obj1, false); }
    pr = vtov_structure-->VTOVS_LEFT_INDEX_PROP;
    pr2 = vtov_structure-->VTOVS_RIGHT_INDEX_PROP;
    vtov_structure-->VTOVS_CACHE_BROKEN = true; ! Mark any cache as broken
    if (pr) {
        if ((obj1 ofclass Object) && (obj1 provides pr)) i1 = obj1.pr;
        else return RunTimeProblem(RTP_IMPREL, obj1, vtov_structure);
    } else i1 = obj1-1;
    if (pr2) {
        if ((obj2 ofclass Object) && (obj2 provides pr2)) i2 = obj2.pr2;
        else return RunTimeProblem(RTP_IMPREL, obj2, vtov_structure);
    } else i2 = obj2-1;
    pr = i1*(vtov_structure-->VTOVS_RIGHT_DOMAIN_SIZE) + i2;
    i1 = IncreasingPowersOfTwo_TB-->(pr%16);
    pr = pr/16 + 8;
    if ((vtov_structure-->pr) & i1) vtov_structure-->pr = vtov_structure-->pr - i1;
];
[ Relation_TestVtoV obj1 vtov_structure obj2 sym pr pr2 i1 i2;
    pr = vtov_structure-->VTOVS_LEFT_INDEX_PROP;
    pr2 = vtov_structure-->VTOVS_RIGHT_INDEX_PROP;
    if (sym && (obj2 > obj1)) { sym = obj1; obj1 = obj2; obj2 = sym; }
    if (pr) {
        if ((obj1 ofclass Object) && (obj1 provides pr)) i1 = obj1.pr;
        else { RunTimeProblem(RTP_IMPREL, obj1, vtov_structure); rfalse; }
    } else i1 = obj1-1;
    if (pr2) {
        if ((obj2 ofclass Object) && (obj2 provides pr2)) i2 = obj2.pr2;
        else { RunTimeProblem(RTP_IMPREL, obj2, vtov_structure); rfalse; }
    } else i2 = obj2-1;
    pr = i1*(vtov_structure-->VTOVS_RIGHT_DOMAIN_SIZE) + i2;
    i1 = IncreasingPowersOfTwo_TB-->(pr%16);
    pr = pr/16 + 8;
    if ((vtov_structure-->pr) & i1) rtrue; rfalse;
];
```

§**4. Equivalence Relations.** For every equivalence relation there is a corresponding function f such that $x \sim y$ if and only if $f(x) = f(y)$, where $f(x)$ is a number identifying the equivalence class of x. Rather than inefficiently storing a large relation bitmap (and then having a very complicated time updating it to keep the relation transitive), we store f: that is, for every object in the domain set, there is a property `prop` such that `O.prop` is the value $f(O)$.

```
[ Relation_NowEquiv obj1 relation_property obj2 big little;
    big = obj1.relation_property; little = obj2.relation_property;
    if (big == little) return;
    if (big < little) { little = obj1.relation_property; big = obj2.relation_property; }
    objectloop (obj1 provides relation_property)
        if (obj1.relation_property == big) obj1.relation_property = little;
];
[ Relation_NowNEquiv obj1 relation_property obj2 old new;
    old = obj1.relation_property; new = obj2.relation_property;
    if (old ~= new) return;
    new = 0;
```

```
    objectloop (obj2 provides relation_property)
        if (obj2.relation_property > new) new = obj2.relation_property;
    new++;
    obj1.relation_property = new;
];
```

§**5. Showing.** The rest of the code for relations has no use except for debugging: it implements the RELATIONS testing command. Speed is unimportant here.

```
[ Relation_ShowR rel_num rel_type x;
    x = rel_num*3;
    rel_type = relation_metadata-->(x+1);
    switch(rel_type) {
        Relation_Implicit: return 0;
        Relation_OtoO: Relation_RShowOtoO(relation_metadata-->x, false, x);
        Relation_OtoV: Relation_RShowOtoO(relation_metadata-->x, false, x);
        Relation_VtoO: Relation_ShowOtoO(relation_metadata-->x, false, x);
        Relation_VtoV: Relation_ShowVtoV(relation_metadata-->x, false, x);
        Relation_Sym_OtoO: Relation_ShowOtoO(relation_metadata-->x, true, x);
        Relation_Sym_VtoV: Relation_ShowVtoV(relation_metadata-->x, true, x);
        Relation_Equiv: Relation_ShowRHeader(x); Relation_ShowEquiv(relation_metadata-->x);
        Relation_ByRoutine: Relation_ShowRHeader(x);
    }
    return 1;
];
[ Relation_ShowRHeader x rel_type;
    rel_type = relation_metadata-->(x+1);
    print (string) relation_metadata-->(x+2);
    if (rel_type == Relation_ByRoutine) ".";
    print ":^";
];
```

§**6. Show Various to Various.**

```
[ Relation_ShowVtoV vtov_structure sym x obj1 obj2 pr pr2 proutine1 proutine2;
    pr = vtov_structure-->VTOVS_LEFT_INDEX_PROP;
    pr2 = vtov_structure-->VTOVS_RIGHT_INDEX_PROP;
    proutine1 = vtov_structure-->VTOVS_LEFT_PRINTING_ROUTINE;
    proutine2 = vtov_structure-->VTOVS_RIGHT_PRINTING_ROUTINE;
    if (pr && pr2) {
        objectloop (obj1 provides pr)
        objectloop (obj2 provides pr2) {
                if (sym && obj2 > obj1) continue;
                if (Relation_TestVtoV(obj1, vtov_structure, obj2)) {
                    if (x) { Relation_ShowRHeader(x); x=0; }
                    print "  ", (The) obj1;
                    if (sym) print "  <=>  "; else print "  >=>  ";
                    print (the) obj2, "^";
                }
        }
        return;
    }
```

```
    if (pr && (pr2==0)) {
        objectloop (obj1 provides pr)
        for (obj2=1:obj2<=vtov_structure-->VTOVS_RIGHT_DOMAIN_SIZE:obj2++) {
                if (Relation_TestVtoV(obj1, vtov_structure, obj2)) {
                    if (x) { Relation_ShowRHeader(x); x=0; }
                    print "  ", (The) obj1, "  >=>  ";
                    (proutine2).call(obj2);
                    print "^";
                }
        }
        return;
    }
    if ((pr==0) && (pr2)) {
        for (obj1=1:obj1<=vtov_structure-->2:obj1++)
        objectloop (obj2 provides pr2) {
                if (Relation_TestVtoV(obj1, vtov_structure, obj2)) {
                    if (x) { Relation_ShowRHeader(x); x=0; }
                    print "  ";
                    (proutine1).call(obj1);
                    print "  >=>  ", (the) obj2, "^";
                }
        }
        return;
    }
    for (obj1=1:obj1<=vtov_structure-->2:obj1++)
        for (obj2=1:obj2<=vtov_structure-->VTOVS_RIGHT_DOMAIN_SIZE:obj2++)
            if (Relation_TestVtoV(obj1, vtov_structure, obj2)) {
                if (x) { Relation_ShowRHeader(x); x=0; }
                print "  ";
                (proutine1).call(obj1);
                print "  >=>  ";
                (proutine2).call(obj2);
                print "^";
        }
];
```

§7. Show One to One.

```
[ Relation_ShowOtoO relation_property sym x obj1 obj2;
    objectloop (obj1 provides relation_property) {
            obj2 = obj1.relation_property;
            if (sym && obj2 < obj1) continue;
            if (obj2 == 0) continue;
        if (x) { Relation_ShowRHeader(x); x=0; }
        print "  ", (The) obj1;
        if (sym) print "  ==  "; else print "  >=>  ";
        print (the) obj2, "^";
    }
];
```

§**8. Show Reversed One to One.** There's no such kind of relation as this: but the same code used to show 1-to-1 relations is also used to show various-to-1 relations, since the storage is the same. To show 1-to-various relations, we need a transposed form of the same code in which left and right are exchanged: this is it.

```
[ Relation_RShowOtoO relation_property sym x obj1 obj2;
    objectloop (obj1) {
        objectloop (obj2 provides relation_property) {
            if (obj2.relation_property ~= obj1) continue;
            if (x) { Relation_ShowRHeader(x); x=0; }
            print "  ", (The) obj1;
            print "  >=>  ";
            print (the) obj2, "^";
        }
    }
];
```

§**9. Show Equivalence.**

```
[ Relation_ShowEquiv relation_property obj1 obj2 v c somegroups;
    objectloop (obj1 provides relation_property)
        obj1.relation_property = -(obj1.relation_property);
    objectloop (obj1 provides relation_property) {
        if (obj1.relation_property < 0) {
            v = obj1.relation_property; c = 0;
            objectloop (obj2 has workflag2) give obj2 ~workflag2;
            objectloop (obj2 provides relation_property) {
                if (obj2.relation_property == v) {
                    give obj2 workflag2;
                    obj2.relation_property = -v;
                    c++;
                }
            }
            if (c>1) {
                somegroups = true;
                print "  { ";
                WriteListOfMarkedObjects(ENGLISH_BIT);
                print " }^";
            } else obj1.relation_property = v;
        }
    }
    objectloop (obj2 has workflag2) give obj2 ~workflag2;
    c = 0; objectloop (obj1 provides relation_property)
        if (obj1.relation_property < 0) { c++; give obj1 workflag2; }
    if (c == 0) return;
    if (somegroups) print "  and "; else print "  ";
    if (c < 4) { WriteListOfMarkedObjects(ENGLISH_BIT); print " in"; }
    else print c;
    if (c == 1) print " a";
    print " single-member group";
    if (c > 1) print "s";
    print "^";
    objectloop (obj1 provides relation_property)
```

```
        if (obj1.relation_property < 0)
            obj1.relation_property = -(obj1.relation_property);
];
```

§**10. Map Route-Finding.** The general problem we have to solve here is: given $x, y \in R$, where R is the set of rooms and we write $x \sim y$ if there is a map connection from x to y,

(i) find the smallest m such that there exist $x = r_1 \sim r_2 \sim ... \sim r_m = y \in R$, or determine that no such m exists, and

(ii) find d, the first direction to take from x to lead to r_2, or set $d = 0$ if no such path exists or if $m = 1$ so that $x = y$.

Thus a typical outcome might be either "a shortest path from the Town Square to the Hilltop takes 11 moves, starting by going northeast from the Town Square", or alternatively "there's no path from the Town Square to the Hilltop at all". Note that the length of the shortest path is unambiguous, but that there might be many alternative paths of this minimum length: we deliberately do not specify which path is chosen if so, and the two algorithms used below do not necessarily choose the same one.

Route-finding is not an easy operation in computation terms: the various algorithms available have theoretical running times which are easy (if sobering) to compute, but which are not in practice typical of what will happen, because they are quite sensitive to the map in question. Are all the rooms laid out in a long line? Are there clusters of connected rooms like islands? Are there dense clumps of interconnecting rooms? Are there huge but possibly time-saving loops? And so on. Overhead is also important. We present a choice of two algorithms: the "fast" one has a theoretical running time of $O(n^3)$, where n is the number of rooms, whereas the "slow" one runs in $O(n^2)$, yet in practice the fast one easily outperforms the slow on typical heavy-use cases with large maps.

The other issue is memory usage: we essentially have to strike a bargain between speed and memory overhead. Our "slow" algorithm needs only $O(n)$ storage, whereas our "fast" algorithm needs $O(n^2)$, and this is very significant in the Z-machine where array space is in desperately short supply and where, if $n > 50$ or so, the user is already likely to be fighting for the last few bytes in readable memory.

The user is therefore offered the choice, by selecting the use options "Use fast route-finding" and "Use slow route-finding": and the defaults, if neither option is explicitly set, are fast on Glulx and slow on the Z-machine. If both use options are explicitly set – which might happen due to a disagreement between extensions – "fast" wins.

```
#ifndef FAST_ROUTE_FINDING;
#ifndef SLOW_ROUTE_FINDING;
#ifdef TARGET_GLULX;
Constant FAST_ROUTE_FINDING;
#ifnot;
Constant SLOW_ROUTE_FINDING;
#endif;
#endif;
#endif;
```

§**11. Cache Control.** We provide code to enable our route-finding algorithms to cache their partial results from one usage to the next (though at present only the "fast" algorithm does this). The difficulty here is that the result of a route search depends on three things, any of which may change:

(a) which subset of rooms we are route-finding through;
(b) which subset of doors we are allowing ourselves to use; and
(c) the current map connections between rooms.

We keep track of (c) by watching for calls to `SignalMapChange()` from the routines in "WorldModel.i6t" which alter the map. (a) and (b), however, require tracking from call to call what the current subset of rooms and doors is. (It is not sufficient to remember the criteria used last time and this time, because circumstances could have changed such that the criteria produce a different outcome. For instance, searching through lighted rooms and using unlocked doors will produce a different result if a door has been locked or unlocked since last time, or if a room has become lighted or not.) We store the set of applicable rooms and doors by enumerating them in the property `room_index` and by the flags in the `DoorRoutingViable` array respectively.

```
Constant NUM_DOORS = {-value:instance_count_for_kind(kind_door)};
Constant NUM_ROOMS = {-value:instance_count_for_kind(kind_room)};

Array DoorRoutingViable -> NUM_DOORS+1;

Global map_has_changed = true;
Global last_filter; Global last_use_doors;

[ SignalMapChange; map_has_changed = true; ];

[ MapRouteTo from to filter use_doors count  oy oyi ds;
    if (from == nothing) return nothing;
    if (to == nothing) return nothing;
    if (from == to) return nothing;
    if ((filter) && (filter(from) == 0)) return nothing;
    if ((filter) && (filter(to) == 0)) return nothing;
    if ((last_filter ~= filter) || (last_use_doors ~= use_doors)) map_has_changed = true;
    oyi = 0;
    objectloop (oy has mark_as_room) {
        if ((filter == 0) || (filter(oy))) {
            if (oy.room_index == -1) map_has_changed = true;
            oy.room_index = oyi++;
        } else {
            if (oy.room_index >= 0) map_has_changed = true;
            oy.room_index = -1;
        }
    }
    oyi = 0;
    objectloop (oy ofclass K4_door) {
        ds = false;
        if ((use_doors & 2) ||
            (oy has open) || ((oy has openable) && (oy hasnt locked))) ds = true;
        if (DoorRoutingViable->oyi ~= ds) map_has_changed = true;
        DoorRoutingViable->oyi = ds;
        oyi++;
    }
    if (map_has_changed) {
        #ifdef FAST_ROUTE_FINDING; ComputeFWMatrix(filter, use_doors); #endif;
        map_has_changed = false; last_filter = filter; last_use_doors = use_doors;
    }
    #ifdef FAST_ROUTE_FINDING;
    if (count) return FastCountRouteTo(from, to, filter, use_doors);
```

```
    return FastRouteTo(from, to, filter, use_doors);
    #ifnot;
    if (count) return SlowCountRouteTo(from, to, filter, use_doors);
    return SlowRouteTo(from, to, filter, use_doors);
    #endif;
];
```

§**12. Fast Route-Finding.** The following is a form of Floyd's adaptation of Warshall's algorithm for finding the transitive closure of a directed graph.

We need to store a matrix which for each pair of rooms R_i and R_j records a_{ij}, the shortest path length from R_i to R_j or 0 if no path exists, and also d_{ij}, the first direction to take on leaving R_i along a shortest path to R_j, or 0 if no path exists. For the sake of economy we represent the directions as their instance counts (numbered from 0 in order of creation), not as their direction object values, and then store a single word for each pair (i, j): we store $d_{ij} + Da_{ij}$. This restricts us on a signed 16-bit virtual machine, and with the conventional set of $D = 12$ directions, to the range $0 \le a_{ij} \le 5461$, that is, to path lengths of 5461 steps or fewer. A work of IF with 5461 rooms will not fit in the Z-machine anyway: such a work would be on Glulx, which is 32-bit, and where $0 \le a_{ij} \le 357,913,941$.

We begin with $a_{ij} = 0$ for all pairs except where there is a viable map connection between R_i and R_j: for those we set $a_{ij} = 1$ and d_{ij} equal to the direction of that map connection.

Following Floyd and Warshall we test if each known shortest path R_x to R_y can be used to shorten the best known path from R_x to anywhere else: that is, we look for cases where $a_{xy} + a_{yj} < a_{xj}$, since those show that going from R_x to R_j via R_y takes fewer steps than going directly. See for instance Robert Sedgewick, *Algorithms* (1988), chapter 32.

The trouble with the Floyd-Warshall algorithm is not so much that it takes in principle $O(n^3)$ time to construct the matrix: it does, but the coefficient is low, and in the early stages of the outer loop the fact that the vertex degree is at most D and usually much lower helps to reduce the work further. The trouble is that there is no way to compute only the part of the matrix we want: we have to have the entire thing, and that means storing n^2 words of data, by which point we have computed not only the fastest route from R_x to R_y but also the fastest route from anywhere to anywhere else. Even when the original map is sparse, the Floyd-Warshall matrix is not, and it is difficult to store in any very compressed way without greatly increasing the complexity of the code. This is why we cache the results: we might as well, since we had to build the entire memory structure anyway, and it means the time expense is only paid once (or once for every time the state of doors and map connections changes), and the cache is useful for all future routes whatever their endpoints.

```
#ifdef FAST_ROUTE_FINDING;
Array FWMatrix --> NUM_ROOMS*NUM_ROOMS;
[ FastRouteTo from to filter use_doors diri i dir oy;
    diri = (FWMatrix-->(from.room_index*NUM_ROOMS + to.room_index))%No_Directions;
    objectloop (dir ofclass K3_direction) {
        if (i == diri) return dir;
        i++;
    }
    return nothing;
];
[ FastCountRouteTo from to filter use_doors oy;
    return (FWMatrix-->(from.room_index*NUM_ROOMS + to.room_index))/No_Directions;
];
```

§**13.** And here is the FW-matrix:

```
[ ComputeFWMatrix filter use_doors  oy ox oj axy ayj axj dir diri nd row;
    objectloop (oy has mark_as_room) if (oy.room_index >= 0)
        objectloop (ox has mark_as_room) if (ox.room_index >= 0)
            FWMatrix-->(oy.room_index*NUM_ROOMS + ox.room_index) = 0;
    objectloop (oy has mark_as_room) if (oy.room_index >= 0) {
        row = (oy.IK_1)*No_Directions;
        for (diri=0: diri<No_Directions: diri++) {
            ox = Map_Storage-->(row+diri);
            if ((ox) && (ox has mark_as_room) && (ox.room_index >= 0)) {
                FWMatrix-->(oy.room_index*NUM_ROOMS + ox.room_index) = No_Directions + diri;
                continue;
            }
            if ((ox ofclass K4_door) && (DoorRoutingViable->(ox.IK_4))) {
                @push location; location = oy;
                ox = ox.door_to();
                @pull location;
                if ((ox) && (ox has mark_as_room) && (ox.room_index >= 0)) {
                    FWMatrix-->(oy.room_index*NUM_ROOMS + ox.room_index) = No_Directions + diri;
                    continue;
                }
            }
        }
    }
    objectloop (oy has mark_as_room) if (oy.room_index >= 0)
        objectloop (ox has mark_as_room) if (ox.room_index >= 0) {
            axy = (FWMatrix-->(ox.room_index*NUM_ROOMS + oy.room_index))/No_Directions;
            if (axy > 0)
                objectloop (oj has mark_as_room) if (oj.room_index >= 0) {
                    ayj = (FWMatrix-->(oy.room_index*NUM_ROOMS + oj.room_index))/No_Directions;
                    if (ayj > 0) {
                        !print "Is it faster to go from ", (name) ox, " to ",
                        !   (name) oj, " via ", (name) oy, "?^";
                        axj = (FWMatrix-->(ox.room_index*NUM_ROOMS + oj.room_index))/
                            No_Directions;
                        if ((axj == 0) || (axy + ayj < axj)) {
                            !print "Yes^";
                            FWMatrix-->(ox.room_index*NUM_ROOMS + oj.room_index) =
                                (axy + ayj)*No_Directions +
                                (FWMatrix-->(ox.room_index*NUM_ROOMS + oy.room_index))%
                                    No_Directions;
                        }
                    }
                }
        }
];
#ENDIF;
```

§**14. Slow Route-Finding.** The alternative algorithm, used when only $O(n)$ memory is available, computes only some of the shortest paths leading to R_y, and is not cached – both because the storage is likely to be reused often by other searches and because there is little gain from doing so, given that a subsequent search with different endpoints will not benefit from the results of this one. On the other hand, to call it "slow" is a little unfair. It is somewhat like Prim's algorithm for finding a minimum spanning tree, rooted at R_y, and grows the tree outward from R_y until either R_x is reached – in which case we stop immediately – or the (directed) component containing R_y has been exhausted – in which case R_x, which must lie outside this, can have no path to R_y. In principle, the running time is $O(dn^2)$, where $d \leq D$ is the maximum vertex degree and n is the number of rooms in the component containing R_y: in practice the degree is often much less than 12, while the algorithm finishes quickly in cases where R_y is relatively isolated and inaccessible or where a shortish route does exist, and those are very common cases in typical usage. There will be circumstances where, because few routes need to be found and because of the shape of the map, the "slow" algorithm will outperform the "fast" one: this is why the user is allowed to control which algorithm is used.

For each room R_z, the property `vector` stores the direction object of the way to go to its parent room in the tree rooted at R_y. Thus if the algorithm succeeds in finding a route from R_x to R_y then we generate the route by starting at R_x and repeatedly going in the `vector` direction from where we currently stand until we reach R_y. Since every room needs a `vector` value, this requires n words of storage. (The `vector` values store only enough of the minimal spanning tree to go upwards through the tree, but that's the only way we need to traverse it.)

The method can be summed up thus:

(a) Begin with every vector blank except that of R_y, the destination.
(b) Repeatedly: For every room in the domain set, try each direction: if this leads to a room whose vector was determined on the last round (*not* on this one, as that may be a suboptimal route), set the vector to point to that room.
(c) Stop as soon as the vector from the origin is set, or when a round happens in which no further vectors are found: in which case, we have completely explored the component of the map from which the destination can be reached, and the origin isn't in it, so we can return "no".

To prove the correctness of this, we show inductively that after round n we have set the `vector` for every room having a shortest path to R_y of length n, and that every `vector` points to a room having a `vector` in the direction of the shortest path from there to R_y.

```
#ifndef FAST_ROUTE_FINDING;
[ SlowRouteTo from to filter use_doors  obj dir in_direction progressed sl through_door;
    if (from == nothing) return nothing;
    if (to == nothing) return nothing;
    if (from == to) return nothing;
    objectloop (obj has mark_as_room) obj.vector = 0;
    to.vector = 1;
    !print "Routing from ", (the) from, " to ", (the) to, "^";
    while (true) {
        progressed = false;
        !print "Pass begins^";
        objectloop (obj has mark_as_room)
            if ((filter == 0) || (filter(obj)))
                if (obj.vector == 0)
                    objectloop (dir ofclass K3_direction) {
                        in_direction = Map_Storage-->((obj.IK_1)*No_Directions + dir.IK_3);
                        if (in_direction == nothing) continue;
                        !print (the) obj, " > ", (the) dir, " > ", (the) in_direction, "^";
                        if ((in_direction)
                            && (in_direction has mark_as_room)
                            && (in_direction.vector > 0)
```

```
                            && ((filter == 0) || (filter(in_direction)))) {
                            obj.vector = dir | WORD_HIGHBIT;
                            !print "* ", (the) obj, " vector is ", (the) dir, "^";
                            progressed = true;
                            continue;
                        }
                        if (use_doors && (in_direction ofclass K4_door) &&
                            ((use_doors & 2) ||
                            (in_direction has open) ||
                            ((in_direction has openable) && (in_direction hasnt locked)))) {
                            sl = location; location = obj;
                            through_door = in_direction.door_to();
                            location = sl;
                            !print "Through door is ", (the) through_door, "^";
                            if ((through_door)
                                && (through_door has mark_as_room)
                                && (through_door.vector > 0)
                                && ((filter == 0) || (filter(through_door)))) {
                                obj.vector = dir | WORD_HIGHBIT;
                                !print "* ", (the) obj, " vector is ", (the) dir, "^";
                                progressed = true;
                                continue;
                            }
                        }
                    }
        objectloop (obj has mark_as_room) obj.vector = obj.vector &~ WORD_HIGHBIT;
        if (from.vector) return from.vector;
        if (progressed == false) return from.vector;
    }
];
[ SlowCountRouteTo from to filter use_doors obj i;
    if (from == nothing) return -1;
    if (to == nothing) return -1;
    if (from == to) return 0;
    if (from has mark_as_room && to has mark_as_room) {
        obj = MapRouteTo(from,to,filter,use_doors);
        if (obj == nothing) return -1;
        i = 0; obj = from;
        while ((obj ~= to) && (i<NUM_ROOMS)) { i++; obj = MapConnection(obj,obj.vector); }
        return i;
    }
    return -1;
];
#ENDIF;
```

§**15. Relation Route-Finding.** The general problem we have to solve here is: given $x, y \in D$, where $\sim$ is a relation on a domain set D of objects,

(i) find the smallest n such that there exist $x = r_1 \sim r_2 \sim ... \sim r_n = y \in D$ such that $r_i \sim r_{i+1}$, or determine that no such n exists, and if so

(ii) find a value of r_2 in such a "route" between x and y, or set $r_2 = 0$ if $x = y$ so that $n = 1$.

While in general a relation can have different left and right domains (a relation between doors and rooms, say), route-finding on those relations is unlikely to be very useful, so is discouraged. (In the case of doors and rooms, a route could never be longer than 1 step, since no object is both a door and a room, for instance.) The "fast" V-to-V algorithm requires D to have the same left and right domains; NI compiles the memory caches for V-to-V relations to force any cases with different domains into using the "slow" algorithm.

`MAX_ROUTE_LENGTH` is used simply as a sanity check to prevent hangs if something should go wrong, for instance if the property of a 1-to-V relation has been modified by some third-party code in such a way that it loses its defining invariant.

```
Constant MAX_ROUTE_LENGTH = {-value:NUMBER_CREATED(world_object)} + 32;

[ RelationRouteTo relation from to count  relation_data relation_kind rv obj i;
    if (count) {
        if (from == nothing) return -1;
        if (to == nothing) return -1;
        if (relation == 0) return -1;
    } else {
        if (from == nothing) return nothing;
        if (to == nothing) return nothing;
        if (relation == 0) return nothing;
    }
    if (from == to) return nothing;
    relation_data = relation_metadata-->(3*relation);
    relation_kind = relation_metadata-->(3*relation+1);
    if (relation_kind == Relation_ByRoutine or Relation_Implicit) {
        RunTimeProblem(RTP_ROUTELESS);
        return nothing;
    }
    if (relation_data == 0) return nothing;
    switch(relation_kind) {
        Relation_OtoO: rv = OtoVRelRouteTo(relation_data, from, to);
        Relation_VtoO: rv = VtoORelRouteTo(relation_data, from, to);
        Relation_OtoV: rv = OtoVRelRouteTo(relation_data, from, to);
        default: return VtoVRelRouteTo(relation_data, from, to, count);
    }
    if (count) {
        if (rv == nothing) return -1;
        i = 0; obj = from;
        while ((obj ~= to) && (i<=MAX_ROUTE_LENGTH)) { i++; obj = obj.vector; }
        return i;
    }
    return rv;
];
```

§**16. One To Various Route-Finding.** Here we can immediately determine, given y, the unique y' such that $y' \sim y$, so finding a path from x to y is a matter of following the only path leading to y and seeing if it ever passed through x; thus the running time is $O(n)$, where n is the size of the domain. It would be pointless to cache this.

Note that we can assume here that $x \neq y$, or rather, that `from ~= to`, because that case has already been taken care of.

```
[ OtoVRelRouteTo relation_property from to previous;
    while ((to) && (to provides relation_property) && (to.relation_property)) {
        previous = to.relation_property;
        previous.vector = to;
        if (previous == from) return to;
        to = previous;
    }
    return nothing;
];
```

§**17. Various To One Route-Finding.** This time the simplifying assumption is that, given x, we can immediately determine the unique x' such that $x \sim x'$, so it suffices to follow the only path forwards from x and see if it ever reaches y. The routine is not quite a mirror image of the one above, because both have the same return requirements: we have to ensure that the `vector` properties lay out the path, and also return the next step after x.

```
[ VtoORelRouteTo relation_property from to next  start;
    start = from;
    while ((from) && (from provides relation_property) && (from.relation_property)) {
        next = from.relation_property;
        from.vector = next;
        if (from == to) return start.vector;
        from = next;
    }
    return nothing;
];
```

§**18. Slow Various To Various Route-Finding.** Now there are no simplifying assumptions and the problem is essentially the same as the one solved for route-finding in the map, above. Once again we present two different algorithms: first, a form of Prim's algorithm for minimal spanning trees. Note that, whereas this algorithm was not always so "slow" for the map – because of the fairly low vertex degrees involved, i.e., because most rooms had few connections to other rooms – here the relation might well be almost complete, with almost all the objects related to each other, and then the algorithm will indeed be "slow". So it is likely that the "fast" algorithm will always be better, if the memory can be spared for it.

We use the fast algorithm for a given relation if and only if the NI compiler has allocated the necessary cache memory; the two use options above, for map route-finding, don't control this.

```
[ VtoVRelRouteTo vtov_structure from to count obj obj2 related progressed left_ix pr2 i;
    if (vtov_structure-->VTOVS_CACHE)
        return FastVtoVRelRouteTo(vtov_structure, from, to, count);
    left_ix = vtov_structure-->VTOVS_LEFT_INDEX_PROP;
    pr2 = vtov_structure-->VTOVS_RIGHT_INDEX_PROP;
    objectloop (obj ofclass Object && obj provides vector) obj.vector = 0;
    to.vector = 1;
    while (true) {
```

```
        progressed = false;
        objectloop (obj ofclass Object && obj provides left_ix)
            if (obj.vector == 0) {
                objectloop (obj2 ofclass Object && obj2 provides pr2 && obj2.vector > 0) {
                    if (Relation_TestVtoV(obj, vtov_structure, obj2)) {
                        obj.vector = obj2 | WORD_HIGHBIT;
                        progressed = true;
                        continue;
                    }
                }
            }
        objectloop (obj ofclass Object && obj provides left_ix)
            obj.vector = obj.vector &~ WORD_HIGHBIT;
        if (from.vector) break;
        if (progressed == false) break;
    }
    if (count) {
        if (from.vector == nothing) return -1;
        i = 0; obj = from;
        while ((obj ~= to) && (i<=MAX_ROUTE_LENGTH)) { i++; obj = obj.vector; }
        return i;
    }
    return from.vector;
];
```

§**19. Fast Various To Various Route-Finding.** Now, as above, a form of the Floyd-Warshall algorithm. The matrix is here stored in the cache of memory pointed to in the V-to-V relation structure. We are unable to combine a_{ij} and d_{ij} into a single cell of memory, so in fact we store two separate matrices: one for a_{ij} (this is `cache` below), the other for n_{ij}, where n_{ij} is the next object in the shortest path from O_i to O_j (this is `cache2` below).

Where $n < 256$ a shortest path must be such that $a_{ij} \leq 255$, so can be stored in a single byte, and we similarly store n_{ij} as the index of the object rather than the object value itself: the index ranges from 0 to $n-1$, so that $0 \leq n_{ij} < 255$ and we can use $n_{ij} = 255$ as a sentinel value meaning "no path". Although the reconversion of n_{ij} back into a valid object value takes a little time, it is only $O(n)$, and of course we know n is relatively small; and in this way we reduce the storage overhead to only n^2 bytes.

Where $n \geq 256$, we resign ourselves to storing two words for each pair (i, j), making $2n^2$ bytes of storage on the Z-machine and $4n^2$ bytes of storage on Glulx, but lookup of a cached result is slightly faster.

```
[ FastVtoVRelRouteTo vtov_structure from to count
    domainsize cache cache2 left_ix ox oy oj offset axy axj ayj;
    domainsize = vtov_structure-->2; ! Number of left instances
    left_ix = vtov_structure-->VTOVS_LEFT_INDEX_PROP;
    if ((from provides left_ix) && (to provides left_ix)) {
        if (domainsize < 256) {
            cache = vtov_structure-->VTOVS_CACHE;
            cache2 = cache + domainsize*domainsize;
            if (vtov_structure-->VTOVS_CACHE_BROKEN == true) {
                vtov_structure-->VTOVS_CACHE_BROKEN = false;
                objectloop (oy provides left_ix)
                    objectloop (ox provides left_ix)
                        if (Relation_TestVtoV(oy, vtov_structure, ox)) {
                            offset = ((oy.left_ix)*domainsize + (ox.left_ix));
```

```
                    cache->offset = 1;
                    cache2->offset = ox.left_ix;
                } else {
                    offset = ((oy.left_ix)*domainsize + (ox.left_ix));
                    cache->offset = 0;
                    cache2->offset = 255;
                }
            for (oy=0: oy<domainsize: oy++)
                for (ox=0: ox<domainsize: ox++) {
                    axy = cache->(ox*domainsize + oy);
                    if (axy > 0)
                        for (oj=0: oj<domainsize: oj++) {
                            ayj = cache->(oy*domainsize + oj);
                            if (ayj > 0) {
                                offset = ox*domainsize + oj;
                                axj = cache->offset;
                                if ((axj == 0) || (axy + ayj < axj)) {
                                    cache->offset = (axy + ayj);
                                    cache2->offset = cache2->(ox*domainsize + oy);
                                }
                            }
                        }
                }
        }
        if (count) {
            count = cache->((from.left_ix)*domainsize + (to.left_ix));
            if (count == 0) return -1;
            return count;
        }
        oy = cache2->((from.left_ix)*domainsize + (to.left_ix));
        if (oy < 255)
            objectloop (ox provides left_ix)
                if (ox.left_ix == oy) return oy;
        return nothing;
    } else {
        cache = vtov_structure-->VTOVS_CACHE;
        cache2 = cache + WORDSIZE*domainsize*domainsize;
        if (vtov_structure-->VTOVS_CACHE_BROKEN == true) {
            vtov_structure-->VTOVS_CACHE_BROKEN = false;
            objectloop (oy provides left_ix)
                objectloop (ox provides left_ix)
                    if (Relation_TestVtoV(oy, vtov_structure, ox)) {
                        offset = ((oy.left_ix)*domainsize + (ox.left_ix));
                        cache-->offset = 1;
                        cache2-->offset = ox;
                    } else {
                        offset = ((oy.left_ix)*domainsize + (ox.left_ix));
                        cache-->offset = 0;
                        cache2-->offset = nothing;
                    }
            for (oy=0: oy<domainsize: oy++)
                for (ox=0: ox<domainsize: ox++) {
                    axy = cache-->(ox*domainsize + oy);
```

```
                        if (axy > 0)
                            for (oj=0: oj<domainsize: oj++) {
                                ayj = cache-->(oy*domainsize + oj);
                                if (ayj > 0) {
                                    offset = ox*domainsize + oj;
                                    axj = cache-->offset;
                                    if ((axj == 0) || (axy + ayj < axj)) {
                                        cache-->offset = (axy + ayj);
                                        cache2-->offset = cache2-->(ox*domainsize + oy);
                                    }
                                }
                            }
                    }
            }
            if (count) {
                count = cache-->((from.left_ix)*domainsize + (to.left_ix));
                if (count == 0) return -1;
                return count;
            }
            return cache2-->((from.left_ix)*domainsize + (to.left_ix));
        }
    }
    if (count) return -1;
    return nothing;
];
```

Figures Template B/figst

Purpose

To display figures and play sound effects.

§**1. Resource Usage.** We record whether pictures or sounds have been used before by storing single byte flags in the following array. (The extra 5 values allow for the fact that it can be legal to use low undeclared sound effect resource numbers in the Z-machine for short beeps, though this is deprecated in I7.)

Pictures and sounds are identified within blorb archives by resource ID numbers which count upwards from 1 in order of creation, but can mix pictures and sounds freely. (For instance, 1 might be a picture, 2 and 3 sound effects, then 4 a picture again, etc.) ID number 1 is in fact always a picture: it means the cover art, and is the I6 representation of the value "figure of cover".

```
Array ResourceUsageFlags ->
    ({-value:NUMBER_CREATED(blorb_figure)}+{-value:NUMBER_CREATED(blorb_sound)}+5);
```

§**2. Figures.**

```
[ DisplayFigure resource_ID one_time;
    if ((one_time) && (ResourceUsageFlags->resource_ID)) return;
    ResourceUsageFlags->resource_ID = true;
    print "^"; VM_Picture(resource_ID); print "^";
];
```

§**3. Sound Effects.**

```
[ PlaySound resource_ID one_time;
    if (resource_ID == 0) return; ! The "silence" non-sound effect
    if ((one_time) && (ResourceUsageFlags->resource_ID)) return;
    ResourceUsageFlags->resource_ID = true;
    VM_SoundEffect(resource_ID);
];
```

IndexedText Template B/inxt

Purpose

Code to support the indexed text kind of value.

---

§**1. Head.** As ever: if there is no heap, there are no indexed texts.

```
#IFDEF MEMORY_HEAP_SIZE; ! Will exist if any use is made of indexed texts
```

§**2. Character Set.** On the Z-machine, we use the 8-bit ZSCII character set, stored in bytes; on Glulx, we use the opening 16-bit subset of Unicode (which though only a subset covers almost all letter forms used on Earth), stored in two-byte half-words.

The Z-machine does have very partial Unicode support, but not in a way that can help us here. It is capable of printing a wide range of Unicode characters, and on a good interpreter with a good font (such as Zoom for Mac OS X, using the Lucida Grande font) can produce many thousands of glyphs. But it is not capable of printing those characters into memory rather than the screen, an essential technique for indexed texts: it can only write each character to a single byte, and it does so in ZSCII. That forces our hand when it comes to choosing the indexed-text character set.

```
#IFDEF TARGET_ZCODE;
Constant IT_Storage_Flags = BLK_FLAG_MULTIPLE;
Constant ZSCII_Tables;
#IFNOT;
Constant IT_Storage_Flags = BLK_FLAG_MULTIPLE + BLK_FLAG_16_BIT;
Constant Large_Unicode_Tables;
#ENDIF;
{-segment:UnicodeData.i6t}
{-segment:Char.i6t}
```

§**3. KOV Support.** See the "BlockValues.i6t" segment for the specification of the following routines.

```
[ INDEXED_TEXT_TY_Support task arg1 arg2 arg3;
    switch(task) {
        CREATE_KOVS:     return INDEXED_TEXT_TY_Create(arg1);
        CAST_KOVS:       return INDEXED_TEXT_TY_Cast(arg1, arg2, arg3);
        DESTROY_KOVS:    rfalse;
        PRECOPY_KOVS:    rfalse;
        COPY_KOVS:       rfalse;
        COMPARE_KOVS:    return INDEXED_TEXT_TY_Compare(arg1, arg2);
        READ_FILE_KOVS:  if (arg3 == -1) rtrue;
                         return INDEXED_TEXT_TY_ReadFile(arg1, arg2, arg3);
        WRITE_FILE_KOVS: return INDEXED_TEXT_TY_WriteFile(arg1);
    }
];
```

§**4. Creation.** Indexed texts are are simply "C strings", that is, the array entries in the block are a sequence of character codes terminated by the character code 0, which is free for this purpose in both ZSCII and Unicode. Since none of the data in an indexed-text is a pointer back onto the heap, it can all freely be bitwise copied or forgotten, which is why we need do nothing special to copy or destroy an indexed text.

Note that a freshly allocated block contains 0s in its data section, so its array entries already form a null-terminated empty text.

```
[ INDEXED_TEXT_TY_Create opcast x;
    x =  BlkAllocate(32, INDEXED_TEXT_TY, IT_Storage_Flags);
    if (opcast) INDEXED_TEXT_TY_Cast(opcast, TEXT_TY, x);
    return x;
];
```

§**5. Casting.** In general computing, "casting" is the process of translating data in one type into semantically equivalent data in another: for instance, translating the integer 1 into the floating point number 1.0, which will have an entirely different binary representation but has roughly the same meaning.

Here we are given a snippet – a word-selection of the player's command – or an ordinary text, and must cast it into an indexed text. In each case, what we do is simply to print out the value we have, but with the output stream set to memory rather than the screen. That gives us the character by character version, neatly laid out in an array, and all we have to do is to copy it into the indexed text and add a null termination byte.

What complicates things is that the two virtual machines handle printing to memory quite differently, and that the original text has unpredictable length. We are going to try printing it into the array `IT_MemoryBuffer`, but what if the text is too big? Disastrously, the Z-machine simply writes on in memory, corrupting all subsequent arrays and almost certainly causing the story file to crash soon after. There is nothing we can do to predict or avoid this, or to repair the damage: this is why the Inform documentation warns users to be wary of using indexed text with large strings in the Z-machine, and advises the use of Glulx instead. Glulx does handle overruns safely, and indeed allows us to dynamically allocate memory as necessary so that we can always avoid overruns entirely.

In either case, though, it's useful to have `IT_MemoryBufferSize`, the size of the temporary buffer, large enough that it will never be overrun in ordinary use. This is controllable with the use option "maximum indexed text length".

The following routine is not as messy as it looks: it is complicated by the fact that the Z-machine and Glulx (a) use different formats when printing text to memory, and (b) handle overruns differently, as explained above.

```
#ifndef IT_MemoryBufferSize;
Constant IT_MemoryBufferSize = 512;
#endif;

Constant IT_Memory_NoBuffers = 2;

#ifndef IT_Memory_NoBuffers;
Constant IT_Memory_NoBuffers = 1;
#endif;

#ifdef TARGET_ZCODE;
Array IT_MemoryBuffer -> IT_MemoryBufferSize*IT_Memory_NoBuffers; ! Where characters are bytes
#ifnot;
Array IT_MemoryBuffer --> (IT_MemoryBufferSize+2)*IT_Memory_NoBuffers; ! Where characters are words
#endif;

Global RawBufferAddress = IT_MemoryBuffer;
Global RawBufferSize = IT_MemoryBufferSize;
```

```
Global IT_cast_nesting;
[ INDEXED_TEXT_TY_Cast tx fromkov indt
    len i str oldstr offs realloc news buff buffx freebuff results;
    #ifdef TARGET_ZCODE;
    buffx = IT_MemoryBufferSize;
    #ifnot;
    buffx = (IT_MemoryBufferSize + 2)*WORDSIZE;
    #endif;
    buff = RawBufferAddress + IT_cast_nesting*buffx;
    IT_cast_nesting++;
    if (IT_cast_nesting > IT_Memory_NoBuffers) {
        buff = VM_AllocateMemory(buffx); freebuff = buff;
        if (buff == 0) {
            BlkAllocationError("ran out with too many simultaneous indexed text conversions");
            return;
        }
    }
    .RetryWithLargerBuffer;
    if (tx == 0) {
        #ifdef TARGET_ZCODE;
        buff-->0 = 1;
        buff->2 = 0;
        #ifnot;
        buff-->0 = 0;
        #endif;
        len = 1;
    } else {
        #ifdef TARGET_ZCODE;
        @output_stream 3 buff;
        #ifnot;
        if (unicode_gestalt_ok == false) { RunTimeProblem(RTP_NOGLULXUNICODE); jump Failed; }
        oldstr = glk_stream_get_current();
        str = glk_stream_open_memory_uni(buff, RawBufferSize, filemode_Write, 0);
        glk_stream_set_current(str);
        #endif;
        @push say__p; @push say__pc;
        ClearParagraphing();
        if (fromkov == SNIPPET_TY) print (PrintSnippet) tx;
        else {
            if (tx ofclass String) print (string) tx;
            if (tx ofclass Routine) (tx)();
        }
        @pull say__pc; @pull say__p;
        #ifdef TARGET_ZCODE;
        @output_stream -3;
        len = buff-->0;
        if (len > RawBufferSize-1) len = RawBufferSize-1;
        offs = 2;
        buff->(len+2) = 0;
        #ifnot; ! i.e. GLULX
        results = buff + buffx - 2*WORDSIZE;
```

```
        glk_stream_close(str, results);
        if (oldstr) glk_stream_set_current(oldstr);
        len = results-->1;
        if (len > RawBufferSize-1) {
            ! Glulx had to truncate text output because the buffer ran out:
            ! len is the number of characters which it tried to print
            news = RawBufferSize;
            while (news < len) news=news*2;
            news = news*4; ! Bytes rather than words
            i = VM_AllocateMemory(news);
            if (i ~= 0) {
                if (freebuff) VM_FreeMemory(freebuff);
                freebuff = i;
                buff = i;
                RawBufferSize = news/4;
                jump RetryWithLargerBuffer;
            }
            ! Memory allocation refused: all we can do is to truncate the text
            len = RawBufferSize-1;
        }
        offs = 0;
        buff-->(len) = 0;
        #endif;
        len++;
    }
    IT_cast_nesting--;
    if (indt == 0) {
        indt = BlkAllocate(len+1, INDEXED_TEXT_TY, IT_Storage_Flags);
        if (indt == 0) jump Failed;
    } else {
        if (BlkValueSetExtent(indt, len+1, 1) == false) { indt = 0; jump Failed; }
    }
    #ifdef TARGET_ZCODE;
    for (i=0:i<=len:i++) BlkValueWrite(indt, i, buff->(i+offs));
    #ifnot;
    for (i=0:i<=len:i++) BlkValueWrite(indt, i, buff-->(i+offs));
    #endif;
    .Failed;
    if (freebuff) VM_FreeMemory(freebuff);
    return indt;
];
```

§**6. Comparison.** This is more or less `strcmp`, the traditional C library routine for comparing strings.

```
[ INDEXED_TEXT_TY_Compare indtleft indtright pos ch1 ch2 dsizeleft dsizeright;
    dsizeleft = BlkValueExtent(indtleft);
    dsizeright = BlkValueExtent(indtright);
    for (pos=0:(pos<dsizeleft) && (pos<dsizeright):pos++) {
        ch1 = BlkValueRead(indtleft, pos);
        ch2 = BlkValueRead(indtright, pos);
        if (ch1 ~= ch2) return ch2-ch1;
        if (ch1 == 0) return 0;
    }
    if (pos == dsizeleft) return 1;
    return -1;
];
[ INDEXED_TEXT_TY_Distinguish indtleft indtright;
    if (INDEXED_TEXT_TY_Compare(indtleft, indtright) == 0) rfalse;
    rtrue;
];
```

§**7. Printing.** Unicode is not the native character set on Glulx: it came along as a late addition to Glulx's specification. The deal is that we have to explicitly tell the Glk interface layer to perform certain operations in a Unicode way; if we simply perform `print (char) ch;` then the character `ch` will be printed in ZSCII rather than Unicode.

```
[ INDEXED_TEXT_TY_Say indt  ch i dsize;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return;
    dsize = BlkValueExtent(indt);
    for (i=0:i<dsize:i++) {
        ch = BlkValueRead(indt, i);
        if (ch == 0) break;
        #ifdef TARGET_ZCODE;
        print (char) ch;
        #ifnot; ! TARGET_ZCODE
        glk_put_char_uni(ch);
        #endif;
    }
];
```

§**8. Serialisation.** Here we print a serialised form of an indexed text which can later be used to reconstruct the original text. The printing is apparently to the screen, but in fact always takes place when the output stream is a file.

The format chosen is a letter "S" for string, then a comma-separated list of decimal character codes, ending with the null terminator, and followed by a semicolon: thus `S65,66,67,0;` is the serialised form of the text "ABC".

```
[ INDEXED_TEXT_TY_WriteFile txb len pos ch;
    len = BlkValueExtent(txb);
    print "S";
    for (pos=0: pos<=len: pos++) {
        if (pos == len) ch = 0; else ch = BlkValueRead(txb, pos);
        if (ch == 0) {
            print "0;"; break;
        } else {
            print ch, ",";
        }
    }
];
```

§**9. Unserialisation.** If that's the word: the reverse process, in which we read a stream of characters from a file and reconstruct the indexed text which gave rise to them.

```
[ INDEXED_TEXT_TY_ReadFile indt auxf ch i v dg pos tsize;
    tsize = BlkValueExtent(indt);
    while (ch ~= 32 or 9 or 10 or 13 or 0 or -1) {
        ch = FileIO_GetC(auxf);
        if (ch == ',' or ';') {
            if (pos+1 >= tsize) {
                if (BlkValueSetExtent(indt, 2*pos, 20) == false) break;
                tsize = BlkValueExtent(indt);
            }
            BlkValueWrite(indt, pos++, v);
            v = 0;
            if (ch == ';') break;
        } else {
            dg = ch - '0';
            v = v*10 + dg;
        }
    }
    BlkValueWrite(indt, pos, 0);
    return indt;
];
```

§**10. Recognition-only-GPR.** An I6 general parsing routine to look at words from the position marker `wn` in the player's command to see if they match the contents of the indexed text `indt`, returning either `GPR_PREPOSITION` or `GPR_FAIL` according to whether a match could be made. This is used when the an object's name is set to include one of its properties, and the property in question is an indexed text: "A flowerpot is a kind of thing. A flowerpot has an indexed text called pattern. Understand the pattern property as describing a flowerpot." When the player types EXAMINE STRIPED FLOWERPOT, and there is a flowerpot in scope, the following routine is called to test whether its pattern property – an indexed text – matches any words at the position STRIPED FLOWERPOT. Assuming a pot does indeed have the pattern "striped", the routine advances `wn` by 1 and returns `GPR_PREPOSITION` to indicate a match.

This kind of GPR is called a "recognition-only-GPR", because it only recognises an existing value: it doesn't parse a new one.

```
[ INDEXED_TEXT_TY_ROGPR indt
    pos len wa wl wpos bdm ch own;
    if (indt == 0) return GPR_FAIL;
    bdm = true; own = wn;
    len = BlkValueExtent(indt);
    for (pos=0: pos<=len: pos++) {
        if (pos == len) ch = 0; else ch = BlkValueRead(indt, pos);
        if (ch == 32 or 9 or 10 or 0) {
            if (bdm) continue;
            bdm = true;
            if (wpos ~= wl) return GPR_FAIL;
            if (ch == 0) break;
        } else {
            if (bdm) {
                bdm = false;
                if (NextWordStopped() == -1) return GPR_FAIL;
                wa = WordAddress(wn-1);
                wl = WordLength(wn-1);
                wpos = 0;
            }
            if (wa->wpos ~= ch or IT_RevCase(ch)) return GPR_FAIL;
            wpos++;
        }
    }
    if (wn == own) return GPR_FAIL; ! Progress must be made to avoid looping
    return GPR_PREPOSITION;
];
```

§11. Blobs. That completes the compulsory services required for this KOV to function: from here on, the remaining routines provide definitions of text-related phrases in the Standard Rules.

What are the basic operations of text-handling? Clearly we want to be able to search, and replace, but that is left for the segment "RegExp.i6t" to handle. More basically we would like to be able to read and write characters from the text. But texts in I7 tend to be of natural language, rather than containing arbitrary material – that's indeed why we call them texts rather than strings. This means they are likely to be punctuated sequences of words, divided up perhaps into sentences and even paragraphs.

So we provide facilities which regard a text as being an array of "blobs", where a "blob" is a unit of text. The user can choose whether to see it as an array of characters, or words (of three different sorts: see the Inform documentation for details), or paragraphs, or lines.

```
Constant CHR_BLOB = 1; ! Construe as an array of characters
Constant WORD_BLOB = 2; ! Of words
Constant PWORD_BLOB = 3; ! Of punctuated words
Constant UWORD_BLOB = 4; ! Of unpunctuated words
Constant PARA_BLOB = 5; ! Of paragraphs
Constant LINE_BLOB = 6; ! Of lines
Constant REGEXP_BLOB = 7; ! Not a blob type as such, but needed as a distinct value
```

§12. Blob Access. The following routine runs a small finite-state-machine to count the number of blobs in an indexed text, using any of the above blob types (except `REGEXP_BLOB`, which is used for other purposes). If the optional arguments `cindt` and `wanted` are supplied, it also copies the text of blob number `wanted` (counting upwards from 1 at the start of the text) into the indexed text `cindt`. If the further optional argument `rindt` is supplied, then `cindt` is instead written with the original text `indt` as it would read if the blob in question were replaced with the indexed text in `rindt`.

```
Constant WS_BRM = 1;
Constant SKIPPED_BRM = 2;
Constant ACCEPTED_BRM = 3;
Constant ACCEPTEDP_BRM = 4;
Constant ACCEPTEDN_BRM = 5;
Constant ACCEPTEDPN_BRM = 6;
[ IT_BlobAccess indt blobtype cindt wanted rindt
    brm oldbrm ch i dsize csize blobcount gp cl j;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return 0;
    if (blobtype == CHR_BLOB) return IT_CharacterLength(indt);
    dsize = BlkValueExtent(indt);
    if (cindt) csize = BlkValueExtent(cindt);
    else if (rindt) "*** rindt without cindt ***";
    brm = WS_BRM;
    for (i=0:i<dsize:i++) {
        ch = BlkValueRead(indt, i);
        if (ch == 0) break;
        oldbrm = brm;
        if (ch == 10 or 13 or 32 or 9) {
            if (oldbrm ~= WS_BRM) {
                gp = 0;
                for (j=i:j<dsize:j++) {
                    ch = BlkValueRead(indt, j);
                    if (ch == 0) { brm = WS_BRM; break; }
                    if (ch == 10 or 13) { gp++; continue; }
                    if (ch ~= 32 or 9) break;
```

```
        }
        ch = BlkValueRead(indt, i);
        if (j == dsize) brm = WS_BRM;
        switch (blobtype) {
            PARA_BLOB: if (gp >= 2) brm = WS_BRM;
            LINE_BLOB: if (gp >= 1) brm = WS_BRM;
            default: brm = WS_BRM;
        }
    }
} else {
    gp = false;
    if ((blobtype == WORD_BLOB or PWORD_BLOB or UWORD_BLOB) &&
        (ch == '.' or ',' or '!' or '?'
                or '-' or '/' or '"' or ':' or ';'
                or '(' or ')' or '[' or ']' or '{' or '}'))
        gp = true;
    switch (oldbrm) {
        WS_BRM:
            brm = ACCEPTED_BRM;
            if (blobtype == WORD_BLOB) {
                if (gp) brm = SKIPPED_BRM;
            }
            if (blobtype == PWORD_BLOB) {
                if (gp) brm = ACCEPTEDP_BRM;
            }
        SKIPPED_BRM:
            if (blobtype == WORD_BLOB) {
                if (gp == false) brm = ACCEPTED_BRM;
            }
        ACCEPTED_BRM:
            if (blobtype == WORD_BLOB) {
                if (gp) brm = SKIPPED_BRM;
            }
            if (blobtype == PWORD_BLOB) {
                if (gp) brm = ACCEPTEDP_BRM;
            }
        ACCEPTEDP_BRM:
            if (blobtype == PWORD_BLOB) {
                if (gp == false) brm = ACCEPTED_BRM;
                else {
                    if ((ch == BlkValueRead(indt, i-1)) &&
                        (ch == '-' or '.')) blobcount--;
                    blobcount++;
                }
            }
        ACCEPTEDN_BRM:
            if (blobtype == WORD_BLOB) {
                if (gp) brm = SKIPPED_BRM;
            }
            if (blobtype == PWORD_BLOB) {
                if (gp) brm = ACCEPTEDP_BRM;
            }
        ACCEPTEDPN_BRM:
```

```
                    if (blobtype == PWORD_BLOB) {
                        if (gp == false) brm = ACCEPTED_BRM;
                        else {
                            if ((ch == BlkValueRead(indt, i-1)) &&
                                (ch == '-' or '.')) blobcount--;
                            blobcount++;
                        }
                    }
                }
            }
            if (brm == ACCEPTED_BRM or ACCEPTEDP_BRM) {
                if (oldbrm ~= brm) blobcount++;
                if ((cindt) && (blobcount == wanted)) {
                    if (rindt) {
                        BlkValueWrite(cindt, cl, 0);
                        IT_Concatenate(cindt, rindt, CHR_BLOB);
                        csize = BlkValueExtent(cindt);
                        cl = IT_CharacterLength(cindt);
                        if (brm == ACCEPTED_BRM) brm = ACCEPTEDN_BRM;
                        if (brm == ACCEPTEDP_BRM) brm = ACCEPTEDPN_BRM;
                    } else {
                        if (cl+1 >= csize) {
                            if (BlkValueSetExtent(cindt, 2*cl, 2) == false) break;
                            csize = BlkValueExtent(cindt);
                        }
                        BlkValueWrite(cindt, cl++, ch);
                    }
                } else {
                    if (rindt) {
                        if (cl+1 >= csize) {
                            if (BlkValueSetExtent(cindt, 2*cl, 3) == false) break;
                            csize = BlkValueExtent(cindt);
                        }
                        BlkValueWrite(cindt, cl++, ch);
                    }
                }
            } else {
                if ((rindt) && (brm ~= ACCEPTEDN_BRM or ACCEPTEDPN_BRM)) {
                    if (cl+1 >= csize) {
                        if (BlkValueSetExtent(cindt, 2*cl, 4) == false) break;
                        csize = BlkValueExtent(cindt);
                    }
                    BlkValueWrite(cindt, cl++, ch);
                }
            }
        }
        if (cindt) BlkValueWrite(cindt, cl++, 0);
        return blobcount;
];
```

§**13. Get Blob.** The front end which uses the above routine to read a blob. (Note that, for efficiency's sake, we read characters more directly.)

```
[ IT_GetBlob cindt indt wanted blobtype;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return;
    if (blobtype == CHR_BLOB) return IT_GetCharacter(cindt, indt, wanted);
    IT_BlobAccess(indt, blobtype, cindt, wanted);
    return cindt;
];
```

§**14. Replace Blob.** The front end which uses the above routine to replace a blob. (Once again, characters are handled directly to avoid incurring all that overhead.)

```
[ IT_ReplaceBlob blobtype indt wanted rindt cindt ilen rlen i;
    if (blobtype == CHR_BLOB) {
        ilen = IT_CharacterLength(indt);
        rlen = IT_CharacterLength(rindt);
        wanted--;
        if ((wanted >= 0) && (wanted<ilen)) {
            if (rlen == 1) {
                BlkValueWrite(indt, wanted, BlkValueRead(rindt, 0));
            } else {
                cindt = BlkValueCreate(INDEXED_TEXT_TY);
                if (BlkValueSetExtent(cindt, ilen+rlen+1, 5)) {
                    for (i=0:i<wanted:i++)
                        BlkValueWrite(cindt, i, BlkValueRead(indt, i));
                    for (i=0:i<rlen:i++)
                        BlkValueWrite(cindt, wanted+i, BlkValueRead(rindt, i));
                    for (i=wanted+1:i<ilen:i++)
                        BlkValueWrite(cindt, rlen+i-1, BlkValueRead(indt, i));
                    BlkValueWrite(cindt, rlen+ilen, 0);
                    BlkValueCopy(indt, cindt);
                }
                BlkFree(cindt);
            }
        }
    } else {
        cindt = BlkValueCreate(INDEXED_TEXT_TY);
        IT_BlobAccess(indt, blobtype, cindt, wanted, rindt);
        BlkValueCopy(indt, cindt);
        BlkFree(cindt);
    }
];
```

§**15. Replace Text.** This is the general routine which searches for any instance of `findt`, as a blob, in `indt`, and replaces it with the text `rindt`. It works on any of the above blob-types, but two cases are special: first, if the blob-type is `CHR_BLOB`, then it can do more than search and replace for any instance of a single character: it can search and replace any instance of a substring, so that `findt` is not required to be only a single character. Second, if the blob-type is the special value `REGEXP_BLOB` then `findt` is interpreted as a regular expression rather than something literal to find: see "RegExp.i6t" for what happens next.

```
[ IT_ReplaceText blobtype indt findt rindt
    cindt csize ilen flen i cl mpos ch chm whitespace punctuation;
    if (blobtype == REGEXP_BLOB or CHR_BLOB)
        return IT_Replace_RE(blobtype, indt, findt, rindt);
    ilen = IT_CharacterLength(indt);
    flen = IT_CharacterLength(findt);
    cindt = BlkValueCreate(INDEXED_TEXT_TY);
    csize = BlkValueExtent(cindt);
    mpos = 0;
    whitespace = true; punctuation = false;
    for (i=0:i<=ilen:i++) {
        ch = BlkValueRead(indt, i);
        .MoreMatching;
        chm = BlkValueRead(findt, mpos++);
        if (mpos == 1) {
            switch (blobtype) {
                WORD_BLOB:
                    if ((whitespace == false) && (punctuation == false)) chm = -1;
            }
        }
        whitespace = false;
        if (ch == 10 or 13 or 32 or 9) whitespace = true;
        punctuation = false;
        if (ch == '.' or ',' or '!' or '?'
            or '-' or '/' or '"' or ':' or ';'
            or '(' or ')' or '[' or ']' or '{' or '}') {
            if (blobtype == WORD_BLOB) chm = -1;
            punctuation = true;
        }
        if (ch == chm) {
            if (mpos == flen) {
                if (i == ilen) chm = 0;
                else chm = BlkValueRead(indt, i+1);
                if ((blobtype == CHR_BLOB) ||
                    (chm == 0 or 10 or 13 or 32 or 9) ||
                    (chm == '.' or ',' or '!' or '?'
                        or '-' or '/' or '"' or ':' or ';'
                        or '(' or ')' or '[' or ']' or '{' or '}')) {
                    mpos = 0;
                    cl = cl - (flen-1);
                    BlkValueWrite(cindt, cl, 0);
                    IT_Concatenate(cindt, rindt, CHR_BLOB);
                    csize = BlkValueExtent(cindt);
                    cl = IT_CharacterLength(cindt);
                    continue;
                }
```

```
            }
        } else {
            mpos = 0;
        }
        if (cl+1 >= csize) {
            if (BlkValueSetExtent(cindt, 2*cl, 9) == false) break;
            csize = BlkValueExtent(cindt);
        }
        BlkValueWrite(cindt, cl++, ch);
    }
    BlkValueCopy(indt, cindt);
    BlkFree(cindt);
];
```

§**16. Character Length.** When accessing at the character-by-character level, things are much easier and we needn't go through any finite state machine palaver.

```
[ IT_CharacterLength indt ch i dsize;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return 0;
    dsize = BlkValueExtent(indt);
    for (i=0:i<dsize:i++) {
        ch = BlkValueRead(indt, i);
        if (ch == 0) return i;
    }
    return dsize;
];
[ INDEXED_TEXT_TY_Empty indt;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) rfalse;
    if (IT_CharacterLength(indt) == 0) rtrue;
    rfalse;
];
```

§**17. Get Character.** Characters in a text are numbered upwards from 1 by the users of this routine: which is why we subtract 1 when reading the array in the block-value, which counts from 0.

```
[ IT_GetCharacter cindt indt i ch;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return;
    if ((i<=0) || (i>IT_CharacterLength(indt))) ch = 0;
    else ch = BlkValueRead(indt, i-1);
    BlkValueWrite(cindt, 0, ch);
    BlkValueWrite(cindt, 1, 0);
    return cindt;
];
```

§**18. Casing.** In many programming languages, characters are a distinct data type from strings, but not in I7. To I7, a character is simply an indexed text which happens to have length 1 – this has its inefficiencies, but is conceptually easy for the user.

`IT_CharactersOfCase(indt, case)` determines whether all the characters in `indt` are letters of the given casing: 0 for lower case, 1 for upper case. In the case of ZSCII, this is done correctly handling all of the European accented letters; in the case of Unicode, it follows the Unicode standard.

Note that there is no requirement for `indt` to be only a single character long.

```
[ IT_CharactersOfCase indt case i ch len;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) rfalse;
    len = IT_CharacterLength(indt);
    for (i=0:i<len:i++) {
        ch = BlkValueRead(indt, i);
        if ((ch) && (CharIsOfCase(ch, case) == false)) rfalse;
    }
    rtrue;
];
```

§**19. Change Case.** We set `cindt` to the text in `indt`, except that all the letters are converted to the `case` given (0 for lower, 1 for upper). The definition of what is a "letter", what case it has and what the other-case form is are as specified in the ZSCII and Unicode standards.

```
[ IT_CharactersToCase cindt indt case i ch len bnd;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return;
    len = IT_CharacterLength(indt);
    if (BlkValueSetExtent(cindt, len+1, 11) == false) return cindt;
    bnd = 1;
    for (i=0:i<len:i++) {
        ch = BlkValueRead(indt, i);
        if (case < 2) {
            BlkValueWrite(cindt, i, CharToCase(ch, case));
        } else {
            BlkValueWrite(cindt, i, CharToCase(ch, bnd));
            if (case == 2) {
                bnd = 0;
                if (ch == 0 or 10 or 13 or 32 or 9
                    or '.' or ',' or '!' or '?'
                    or '-' or '/' or '"' or ':' or ';'
                    or '(' or ')' or '[' or ']' or '{' or '}') bnd = 1;
            }
            if (case == 3) {
                if (ch ~= 0 or 10 or 13 or 32 or 9) {
                    if (bnd == 1) bnd = 0;
                    else {
                        if (ch == '.' or '!' or '?') bnd = 1;
                    }
                }
            }
        }
    }
    BlkValueWrite(cindt, len, 0);
    return cindt;
];
```

§**20. Concatenation.** To concatenate two indexed texts is to place one after the other: thus "green" concatenated with "horn" makes "greenhorn". In this routine, `indt_from` would be "horn", and is added at the end of `indt_to`, which is returned in its expanded state.

When the blob type is `REGEXP_BLOB`, the routine is used not for simple concatenation but to handle the concatenations occurring when a regular expression search-and-replace is going on: see "RegExp.i6t".

```
[ IT_Concatenate indt_to indt_from blobtype indt_ref
    pos len ch i tosize x y case;
    if ((indt_to==0) || (BlkType(indt_to) ~= INDEXED_TEXT_TY)) rfalse;
    if ((indt_from==0) || (BlkType(indt_from) ~= INDEXED_TEXT_TY)) return indt_to;
    switch(blobtype) {
        CHR_BLOB, 0:
            pos = IT_CharacterLength(indt_to);
            len = IT_CharacterLength(indt_from);
            if (BlkValueSetExtent(indt_to, pos+len+1, 10) == false) return indt_to;
            for (i=0:i<len:i++) {
                ch = BlkValueRead(indt_from, i);
                BlkValueWrite(indt_to, i+pos, ch);
            }
            BlkValueWrite(indt_to, len+pos, 0);
            return indt_to;
        REGEXP_BLOB:
            return IT_RE_Concatenate(indt_to, indt_from, blobtype, indt_ref);
        default:
            print "*** IT_Concatenate used on impossible blob type ***^";
            rfalse;
    }
];
```

§**21. Setting the Player's Command.** In effect, the text typed most recently by the player is a sort of indexed text already, though it isn't in indexed text format, and doesn't live on the heap. (We can't simply make it an indexed text, as tidy as that would seem, because then no I7 work could ever compile without a heap to use, and that would severely affect many works which have to fit in the Z-machine and can't afford the storage for a heap.)

```
[ SetPlayersCommand indt_from i len at;
    len = IT_CharacterLength(indt_from);
    if (len > 118) len = 118;
    #ifdef TARGET_ZCODE;
    buffer->1 = len; at = 2;
    #ifnot;
    buffer-->0 = len; at = 4;
    #endif;
    for (i=0:i<len:i++) buffer->(i+at) = CharToCase(BlkValueRead(indt_from, i), 0);
    for (:at+i<120:i++) buffer->(at+i) = ' ';
    VM_Tokenise(buffer, parse);
    players_command = 100 + WordCount(); ! The snippet variable ''player's command''
];
```

§**22. Stubs.** And the usual meaningless versions to ensure that function-names exist if there is no heap, and there are no indexed texts anyway.

```
#IFNOT; ! IFDEF MEMORY_HEAP_SIZE
[ INDEXED_TEXT_TY_Support t a b c; rfalse; ];
[ INDEXED_TEXT_TY_Say indt; ];
[ SetPlayersCommand indt_from; ];
[ INDEXED_TEXT_TY_Create; ];
[ INDEXED_TEXT_TY_Cast a b c; ];
[ INDEXED_TEXT_TY_Empty t; rfalse; ];
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```

RegExp Template B/regxt

Purpose

Code to match and replace on regular expressions against indexed text strings.

---

§**1. Head.** As ever: if there is no heap, there are no indexed texts, and if there are no indexed texts then there is no point in compiling any of this code.

```
#IFDEF MEMORY_HEAP_SIZE; ! Will exist if any use is made of indexed texts
Global IT_RE_Trace = false; ! Change to true for (a lot of) debugging data in use
[ IT_RE_SetTrace F; IT_RE_Trace = F; ];
```

§**2. Algorithm.** Once Inform 7 supported indexed text, regular-expression matching became an obvious goal: regexp-based features offer more or less the gold standard in text search and replace facilities, and I7 is so concerned with text that we shouldn't make do with less. But the best and most portable implementation of regular expression matching, PCRE by Philip Hazel, is about a hundred times larger than the code in this section, and also had unacceptable memory needs: there was no practicable way to make it small enough to do useful work on the Z-machine. Nor could an I6 regexp-matcher compile just-in-time code, or translate the expression into a suitable deterministic finite state machine. One day, though, I read one of the papers which Brian Kernighan writes every few years to the effect that regular-expression matching is much easier than you think. Kernighan is right: writing a regexp matcher is indeed easier than you think (one day's worth of cheerful hacking), but debugging one until it passes the trickiest hundred of Perl's 645 test cases is another matter (and it took a whole week more). Still, the result seems to be robust. The main compromise made is that backtracking is not always comprehensive with regexps like `^(a\1?){4}$`, because we do not allocate individual storage to backtrack individually through all possibilities of each of the four uses of the bracketed subexpression – which means we miss some cases, since the subexpression contains a reference to itself, so that its content can vary in the four uses. PCRE's approach here is to expand the expression as if it were a sequence of four bracketed expressions, thus removing the awkward quantifier `{4}`, but that costs memory: indeed this is why PCRE cannot solve all of Perl's test cases without its default memory allocation being raised. In other respects, the algorithm below appears to be accurate if not very fast.

§**3. Class Codes.** While in principle we could keep the match expression in textual form, in practice the syntax of regular expressions is complex enough that this would tricky and rather slow: we would be parsing the same notations over and over again. So we begin by compiling it to a simple tree structure. The tree is made up of nodes, and each node has a "class code": these are identified by the `*_RE_CC` constants below. Note that the class codes below are all negative: this is so that they are distinct from all valid ZSCII or Unicode characters. (ZSCII is used only on the Z-machine, which has a 16-bit word but an 8-bit character set, so that all character values are positive; similarly, Unicode is (for our purposes) a 16-bit character set on a 32-bit virtual machine.)

```
! Character classes
Constant NEWLINE_RE_CC = -1;
Constant TAB_RE_CC = -2;
Constant DIGIT_RE_CC = -3;
Constant NONDIGIT_RE_CC = -4;
Constant WHITESPACE_RE_CC = -5;
Constant NONWHITESPACE_RE_CC = -6;
Constant PUNCTUATION_RE_CC = -7;
Constant NONPUNCTUATION_RE_CC = -8;
Constant WORD_RE_CC = -9;
Constant NONWORD_RE_CC = -10;
Constant ANYTHING_RE_CC = -11;
Constant NOTHING_RE_CC = -12;
Constant RANGE_RE_CC = -13;
Constant LCASE_RE_CC = -14;
Constant NONLCASE_RE_CC = -15;
Constant UCASE_RE_CC = -16;
Constant NONUCASE_RE_CC = -17;
! Control structures
Constant SUBEXP_RE_CC = -20;
Constant DISJUNCTION_RE_CC = -21;
Constant CHOICE_RE_CC = -22;
Constant QUANTIFIER_RE_CC = -23;
Constant IF_RE_CC = -24;
Constant CONDITION_RE_CC = -25;
Constant THEN_RE_CC = -26;
Constant ELSE_RE_CC = -27;
! Substring matchers
Constant VARIABLE_RE_CC = -30;
Constant LITERAL_RE_CC = -31;
! Positional matchers
Constant START_RE_CC = -40;
Constant END_RE_CC = -41;
Constant BOUNDARY_RE_CC = -42;
Constant NONBOUNDARY_RE_CC = -43;
Constant ALWAYS_RE_CC = -44;
Constant NEVER_RE_CC = -45;
! Mode switches
Constant SENSITIVITY_RE_CC = -50;
```

§**4. Packets.** The nodes of the compiled expression tree are stored in "packets", which are segments of a fixed array. A regexp complicated enough that it cannot be stored in `RE_MAX_PACKETS` packets will be rejected with an error: it looks like a rather low limit, but in fact suffices to handle all of Perl's test cases, some of which are works of diabolism.

A packet is then a record containing 14 fields, with offsets defined by the constants defined below. These fields combine the compilation of the corresponding fragment of the regexp with both the tree structure holding these packets together and also the current state of the temporary variables recording how far we have progressed in trying all of the possible ways to match the packet.

```
Constant RE_MAX_PACKETS = 32;
Constant RE_PACKET_SIZE = 14; ! Words of memory used per packet
Constant RE_PACKET_SIZE_IN_BYTES = WORDSIZE*RE_PACKET_SIZE; ! Bytes used per packet
Array RE_PACKET_space --> RE_MAX_PACKETS*RE_PACKET_SIZE;
Constant RE_CCLASS = 0;      ! One of the class codes defined above
Constant RE_PAR1 = 1;        ! Three parameters whose meaning depends on class code
Constant RE_PAR2 = 2;
Constant RE_PAR3 = 3;
Constant RE_NEXT = 4;        ! Younger sibling in the compiled tree
Constant RE_PREVIOUS = 5;    ! Elder sibling
Constant RE_DOWN = 6;        ! Child
Constant RE_UP = 7;          ! Parent
Constant RE_DATA1 = 8;       ! Backtracking data
Constant RE_DATA2 = 9;
Constant RE_CONSTRAINT = 10;
Constant RE_CACHE1 = 11;
Constant RE_CACHE2 = 12;
Constant RE_MODES = 13;
```

§**5. Nodes.** The routine to create a node, something which happens only during the compilation phase, and also the routine which returns the address of a given node. Nodes are numbered from 0 up to $M-1$, where M is the constant `RE_MAX_PACKETS`.

```
[ IT_RE_Node n cc par1 par2 par3  offset;
    if ((n<0) || (n >= RE_MAX_PACKETS)) rfalse;
    offset = RE_PACKET_space + n*RE_PACKET_SIZE_IN_BYTES;
    offset-->RE_CCLASS = cc;
    offset-->RE_PAR1 = par1;
    offset-->RE_PAR2 = par2;
    offset-->RE_PAR3 = par3;
    offset-->RE_NEXT = NULL;
    offset-->RE_PREVIOUS = NULL;
    offset-->RE_DOWN = NULL;
    offset-->RE_UP = NULL;
    offset-->RE_DATA1 = -1; ! Match start
    offset-->RE_DATA2 = -1; ! Match end
    offset-->RE_CONSTRAINT = -1; ! Rewind edge
    return offset;
];
[ IT_RE_NodeAddress n;
    if ((n<0) || (n >= RE_MAX_PACKETS)) return -1;
    return RE_PACKET_space + n*RE_PACKET_SIZE_IN_BYTES;
];
```

§**6. Match Variables.** A bracketed subexpression can be used as a variable: we support `\1`, ..., `\9` to mean "the value of subexpression 1 to 9", and `\0` to mean "the whole text matched", as if the entire regexp were bracketed. (PCRE and Perl also allow `\10`, `\11`, ..., but we don't, because it complicates parsing and memory is too short.)

`RE_Subexpressions-->10` stores the number of subexpressions in use, not counting `\0`. During the compiling stage, `RE_Subexpressions-->N` is set to point to the node representating `\N`, where `N` varies from 1 to 9. When matching is complete, and assuming we care about the contents of these variables – which we might not, and if not we certainly don't want to waste time and memory – we call `IT_RE_CreateMatchVars` to allocate indexed text variables and fill them in as appropriate, memory permitting.

`IT_RE_EmptyMatchVars` empties any such variables which may survive from a previous match, setting them to the empty text.

```
Array RE_Subexpressions --> 11; ! Address of node for this subexpression
Array Allocated_Match_Vars --> 10; ! Indexed text to hold values of the variables

[ IT_RE_DebugMatchVars indt
    offset n i;
    print RE_Subexpressions-->10, " collecting subexps^";
    for (n=0:(n<RE_Subexpressions-->10) && (n<10): n++) {
        offset = RE_Subexpressions-->n;
        print "Subexp ", offset-->RE_PAR1,
            " = [", offset-->RE_DATA1, ",", offset-->RE_DATA2, "] = ";
        for (i=offset-->RE_DATA1:i<offset-->RE_DATA2:i++)
            print (char) BlkValueRead(indt, i);
        print "^";
    }
];

[ IT_RE_CreateMatchVars indt
    offset n i ch cindt cl csize;
    for (n=0:(n<RE_Subexpressions-->10) && (n<10): n++) {
        offset = RE_Subexpressions-->n;
        if (Allocated_Match_Vars-->n == 0)
            Allocated_Match_Vars-->n = INDEXED_TEXT_TY_Create(); ! Permanently
        cindt = Allocated_Match_Vars-->n;
        csize = BlkValueExtent(cindt);
        cl = 0;
        for (i=offset-->RE_DATA1:i<offset-->RE_DATA2:i++) {
            ch = BlkValueRead(indt, i);
            if (cl+1 >= csize) {
                if (BlkValueSetExtent(cindt, 2*cl, 6) == false) break;
                csize = BlkValueExtent(cindt);
            }
            BlkValueWrite(cindt, cl++, ch);
        }
        BlkValueWrite(cindt, cl, 0);
    }
];

[ IT_RE_EmptyMatchVars indt
    n;
    for (n=0:(n<RE_Subexpressions-->10) && (n<10): n++)
        if (Allocated_Match_Vars-->n ~= 0)
            BlkValueWrite(Allocated_Match_Vars-->n, 0, 0);
];
```

```
[ IT_RE_GetMatchVar indt vn
    offset;
    if ((vn<0) || (vn>=10) || (vn >= RE_Subexpressions-->10)) jump Nope;
    offset = RE_Subexpressions-->vn;
    if (offset == 0) jump Nope;
    if (offset-->RE_DATA1 < 0) jump Nope;
    if (Allocated_Match_Vars-->vn == 0) {
        print "*** ", vn, " unallocated! ***^";
        jump Nope;
    }
    BlkValueCopy(indt, Allocated_Match_Vars-->vn);
    return indt;
    .Nope;
    BlkValueWrite(indt, 0, 0);
    return indt;
];
```

§**7. Markers.** At each node, the `-->RE_DATA1` and `-->RE_DATA2` fields represent the character positions of the start and end of the text matched by the node and its subtree (if any). These are called markers.

Thus `IT_MV_End(N, 0)` returns the start of `\N` and `IT_MV_End(N, 1)` the end of `\N`, according to the current match of subexpression `N`.

```
[ IT_MV_End n end
    offset;
    offset = RE_Subexpressions-->n;
    if (end==0) return offset-->RE_DATA1;
    return offset-->RE_DATA2;
];
[ IT_RE_Clear_Markers token;
    for (: token ~= NULL: token = token-->RE_NEXT) {
        if (token-->RE_DOWN ~= NULL) IT_RE_Clear_Markers(token-->RE_DOWN);
        token-->RE_DATA1 = -1;
        token-->RE_DATA2 = -1;
        token-->RE_CONSTRAINT = -1;
    }
];
```

§**8. Debugging.** Code in this paragraph simply prints a convenient screen representation of the compiled regexp, together with the current values of its markers. It is invaluable for debugging purposes and, touch wood, may not be needed again, but it is relatively compact and we keep it just in case.

```
[ IT_RE_DebugTree findt detail;
    print "Pattern: ", (INDEXED_TEXT_TY_Say) findt, "^";
    IT_RE_DebugSubtree(findt, 1, RE_PACKET_space, detail);
];
[ IT_RE_DebugSubtree findt depth offset detail
    cup;
    if (offset ~= NULL) {
        cup = offset-->RE_UP;
        if (offset-->RE_PREVIOUS ~= NULL) print "*** broken initial previous ***^";
    }
    while (offset ~= NULL) {
        if (offset-->RE_UP ~= cup) print "*** broken up matching ***^";
        spaces(depth*2);
        IT_RE_DebugNode(offset, findt, detail);
        if (offset-->RE_DOWN ~= NULL) {
            if ((offset-->RE_DOWN)-->RE_UP ~= offset)
                print "*** broken down/up ***^";
            IT_RE_DebugSubtree(findt, depth+1, offset-->RE_DOWN, detail);
        }
        if (offset-->RE_NEXT ~= NULL) {
            if ((offset-->RE_NEXT)-->RE_PREVIOUS ~= offset)
                print "*** broken next/previous ***^";
        }
        offset = offset-->RE_NEXT;
    }
];
[ IT_RE_DebugNode offset findt detail
    i par1 par2 par3;
    if (offset == NULL) "[NULL]";
    print "[", (offset-RE_PACKET_space)/(RE_PACKET_SIZE_IN_BYTES), "] ";
    ! for (i=0:i<RE_PACKET_SIZE:i++) print offset-->i, " ";
    par1 = offset-->RE_PAR1;
    par2 = offset-->RE_PAR2;
    par3 = offset-->RE_PAR3;
    switch (offset-->RE_CCLASS) {
        DIGIT_RE_CC: print "DIGIT";
        NONDIGIT_RE_CC: print "NONDIGIT";
        UCASE_RE_CC: print "UCASE";
        NONUCASE_RE_CC: print "NONUCASE";
        LCASE_RE_CC: print "LCASE";
        NONLCASE_RE_CC: print "NONLCASE";
        WHITESPACE_RE_CC: print "WHITESPACE";
        NONWHITESPACE_RE_CC: print "NONWHITESPACE";
        PUNCTUATION_RE_CC: print "PUNCTUATION";
        NONPUNCTUATION_RE_CC: print "NONPUNCTUATION";
        WORD_RE_CC: print "WORD";
        NONWORD_RE_CC: print "NONWORD";
        ALWAYS_RE_CC: print "ALWAYS";
        NEVER_RE_CC: print "NEVER";
```

```
        START_RE_CC: print "START";
        END_RE_CC: print "END";
        BOUNDARY_RE_CC: print "BOUNDARY";
        NONBOUNDARY_RE_CC: print "NONBOUNDARY";
        ANYTHING_RE_CC: print "ANYTHING";
        NOTHING_RE_CC: print "NOTHING";
        RANGE_RE_CC: print "RANGE"; if (par3 == true) print " (negated)";
            print " ";
            for (i=par1:i<par2:i++) print (char) BlkValueRead(findt, i);
        VARIABLE_RE_CC: print "VARIABLE ", par1;
        SUBEXP_RE_CC:
            if (par1 == 0) print "EXP";
            else print "SUBEXP ";
            if (par1 >= 0) print "= V", par1;
            if (par2 == 1) {
                if (par3 == 0) print " (?=...) lookahead";
                else print " (?<=...) lookbehind of width ", par3;
            }
            if (par2 == 2) {
                if (par3 == 0) print " (?!...) negated lookahead";
                else print " (?<!...) negated lookbehind of width ", par3;
            }
            if (par2 == 3) print " uncollecting";
            if (par2 == 0 or 3) {
                if (par3 == 1) print " forcing case sensitivity";
                if (par3 == 2) print " forcing case insensitivity";
            }
            if (par2 == 4) print " (?>...) possessive";
        NEWLINE_RE_CC: print "NEWLINE";
        TAB_RE_CC: print "TAB";
        QUANTIFIER_RE_CC: print "QUANTIFIER min=", par1, " max=", par2;
            if (par3) print " (lazy)"; else print " (greedy)";
        LITERAL_RE_CC: print "LITERAL";
            print " ";
            for (i=par1:i<par2:i++) print (char) BlkValueRead(findt, i);
        DISJUNCTION_RE_CC: print "DISJUNCTION of ", par1, " choices";
        CHOICE_RE_CC: print "CHOICE no ", par1;
        SENSITIVITY_RE_CC: print "SENSITIVITY";
            if (par1) print " off"; else print " on";
        IF_RE_CC: print "IF"; if (par1 >= 1) print " = V", par1;
        CONDITION_RE_CC: print "CONDITION"; if (par1 >= 1) print " = V", par1;
        THEN_RE_CC: print "THEN";
        ELSE_RE_CC: print "ELSE";
    }
    if (detail)
        print ": ", offset-->RE_DATA1, ", ", offset-->RE_DATA2, ", ", offset-->RE_CONSTRAINT;
    print "^";
];
```

§**9. Compiling Tree For Substring Search.** When we search for a literal substring, say looking for "per" in "Supernumerary", we will in fact use the same apparatus as when searching for a regular expression: we compile a very simple node tree in which `\0` as the root contains just one child node, a `LITERAL_RE_CC` matching exactly the text "per". We return 2 since that's the number of nodes in the tree.

```
[ IT_CHR_CompileTree findt exactly
    root literal fto no_packets token attach_to;
    fto = IT_CharacterLength(findt);
    root = IT_RE_Node(0, SUBEXP_RE_CC, 0, 0, 0);
    literal = IT_RE_Node(1, LITERAL_RE_CC, 0, fto, 0);
    root-->RE_DOWN = literal;
    literal-->RE_UP = root;
    if (exactly) {
        no_packets = 2;
        if (no_packets+3 > RE_MAX_PACKETS) return "regexp too complex";
        exactly = RE_PACKET_space-->RE_DOWN;
        token = IT_RE_Node(no_packets++, START_RE_CC, 0, 0, 0);
        RE_PACKET_space-->RE_DOWN = token; token-->RE_UP = RE_PACKET_space;
        attach_to = IT_RE_Node(no_packets++, SUBEXP_RE_CC, -1, 3, 0);
        token-->RE_NEXT = attach_to; attach_to-->RE_PREVIOUS = token;
        attach_to-->RE_UP = RE_PACKET_space;
        attach_to-->RE_NEXT = IT_RE_Node(no_packets++, END_RE_CC, 0, 0, 0);
        (attach_to-->RE_NEXT)-->RE_PREVIOUS = attach_to;
        (attach_to-->RE_NEXT)-->RE_UP = RE_PACKET_space;
        attach_to-->RE_DOWN = exactly;
        while (exactly ~= NULL) {
            exactly-->RE_UP = attach_to; exactly = exactly-->RE_NEXT;
        }
    }
    no_packets = IT_RE_ExpandChoices(RE_PACKET_space, no_packets);
];
```

§**10. Compiling Tree For Regexp Search.** But in general we need to compile a regular expression string into a tree of the kind described above, and here is the routine which does that, returning the number of nodes used to build the tree. The syntax it accepts is very fully documented in *Writing with Inform*, so no details are given here.

```
Array Subexp_Posns --> 20;
[ IT_RE_CompileTree findt exactly
    no_packets ffrom fto cc par1 par2 par3
    quantifiable token attach_to no_subs blevel bits;
    fto = IT_CharacterLength(findt);
    if (fto == 0) {
        IT_RE_Node(no_packets++, NEVER_RE_CC, 0, 0, 0); ! Empty regexp never matches
        return 1;
    }
    attach_to = IT_RE_Node(no_packets++, SUBEXP_RE_CC, 0, 0, 0);
    RE_Subexpressions-->0 = attach_to; RE_Subexpressions-->10 = 1; no_subs = 1;
    quantifiable = false; blevel = 0;
    for (ffrom = 0: ffrom < fto: ) {
        cc = BlkValueRead(findt, ffrom++); par1 = 0; par2 = 0; par3 = 0;
```

```
if (cc == '\') {
    if (ffrom == fto) return "Search pattern not terminated";
    cc = BlkValueRead(findt, ffrom++);
    switch (cc) {
        'b': cc = BOUNDARY_RE_CC;
        'B': cc = NONBOUNDARY_RE_CC;
        'd': cc = DIGIT_RE_CC;
        'D': cc = NONDIGIT_RE_CC;
        'l': cc = LCASE_RE_CC;
        'L': cc = NONLCASE_RE_CC;
        'n': cc = NEWLINE_RE_CC;
        'p': cc = PUNCTUATION_RE_CC;
        'P': cc = NONPUNCTUATION_RE_CC;
        's': cc = WHITESPACE_RE_CC;
        'S': cc = NONWHITESPACE_RE_CC;
        't': cc = TAB_RE_CC;
        'u': cc = UCASE_RE_CC;
        'U': cc = NONUCASE_RE_CC;
        'w': cc = WORD_RE_CC;
        'W': cc = NONWORD_RE_CC;
        default:
            if ((cc >= '1') && (cc <= '9')) {
                par1 = cc-'0';
                cc = VARIABLE_RE_CC;
            } else {
                if (((cc >= 'a') && (cc <= 'z')) ||
                    ((cc >= 'A') && (cc <= 'Z'))) return "unknown escape";
                cc = LITERAL_RE_CC;
                par1 = ffrom-1; par2 = ffrom;
            }
    }
    quantifiable = true;
} else {
    switch (cc) {
        '(': par2 = 0;
            !if (BlkValueRead(findt, ffrom) == ')') return "empty subexpression";
            if (BlkValueRead(findt, ffrom) == '?') {
                ffrom++;
                bits = true;
                if (BlkValueRead(findt, ffrom) == '-') { ffrom++; bits = false; }
                else if (BlkValueRead(findt, ffrom) == '<') { ffrom++; bits = false; }
                switch (cc = BlkValueRead(findt, ffrom++)) {
                    '#': while (BlkValueRead(findt, ffrom++) ~= 0 or ')') ;
                        if (BlkValueRead(findt, ffrom-1) == 0)
                            return "comment never ends";
                        continue;
                    '(': cc = BlkValueRead(findt, ffrom);
                        if ((cc == '1' or '2' or '3' or '4' or
                            '5' or '6' or '7' or '8' or '9') &&
                            (BlkValueRead(findt, ffrom+1) ==')')) {
                            ffrom = ffrom + 2;
                            par1 = cc - '0';
                        } else ffrom--;
```

```
                cc = IF_RE_CC; ! (?(...)...) conditional
                quantifiable = false;
                if (blevel == 20) return "subexpressions too deep";
                Subexp_Posns-->(blevel++) = IT_RE_NodeAddress(no_packets);
                jump CClassKnown;
            '=': par2 = 1; ! (?=...) lookahead/behind
                par3 = 0; if (bits == false) par3 = -1;
            '!': par2 = 2; ! (?!...) negated lookahead/behind
                par3 = 0; if (bits == false) par3 = -1;
            ':': par2 = 3; ! (?:...) uncollecting subexpression
            '>': par2 = 4; ! (?>...) possessive
            default:
                if (BlkValueRead(findt, ffrom) == ')') {
                    if (cc == 'i') {
                        cc = SENSITIVITY_RE_CC; par1 = bits; ffrom++;
                        jump CClassKnown;
                    }
                }
                if (BlkValueRead(findt, ffrom) == ':') {
                    if (cc == 'i') {
                        par1 = bits; par2 = 3; par3 = bits+1; ffrom++;
                        jump AllowForm;
                    }
                }
                return "unknown (?...) form";
        }
    }
    .AllowForm;
    if (par2 == 0) par1 = no_subs++; else par1 = -1;
    cc = SUBEXP_RE_CC;
    quantifiable = false;
    if (blevel == 20) return "subexpressions too deep";
    Subexp_Posns-->(blevel++) = IT_RE_NodeAddress(no_packets);
')': if (blevel == 0) return "subexpression bracket mismatch";
    blevel--;
    attach_to = Subexp_Posns-->blevel;
    if (attach_to-->RE_DOWN == NULL) {
        if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
        attach_to-->RE_DOWN =
            IT_RE_Node(no_packets++, ALWAYS_RE_CC, 0, 0, 0);
        (attach_to-->RE_DOWN)-->RE_UP = attach_to;
    }
    quantifiable = true;
    continue;
'.': cc = ANYTHING_RE_CC; quantifiable = true;
'|': cc = CHOICE_RE_CC; quantifiable = false;
'^': cc = START_RE_CC; quantifiable = false;
'$': cc = END_RE_CC; quantifiable = false;
'{': if (quantifiable == false) return "quantifier misplaced";
    par1 = 0; par2 = -1; bits = 1;
    while ((cc=BlkValueRead(findt, ffrom++)) ~= 0 or '}') {
        if (cc == ',') {
            bits++;
```

```
            if (bits >= 3) return "too many colons in ?{...}";
            continue;
        }
        if ((cc >= '0') || (cc <= '9')) {
            if (bits == 1) {
                if (par1 < 0) par1 = 0;
                par1 = par1*10 + (cc-'0');
            } else {
                if (par2 < 0) par2 = 0;
                par2 = par2*10 + (cc-'0');
            }
        } else return "non-digit in ?{...}";
    }
    if (cc ~= '}') return "{x,y} quantifier never ends";
    cc = QUANTIFIER_RE_CC;
    if (par2 == -1) {
        if (bits == 2) par2 = 30000;
        else par2 = par1;
    }
    if (par1 > par2) return "{x,y} with x greater than y";
    if (BlkValueRead(findt, ffrom) == '?') { ffrom++; par3 = true; }
    quantifiable = false;
'<', '[': par3 = false; if (cc == '<') bits = '>'; else bits = ']';
    if (BlkValueRead(findt, ffrom) == '^') { ffrom++; par3 = true; }
    par1 = ffrom;
    if (BlkValueRead(findt, ffrom) == bits) { ffrom++; }
    while (cc ~= bits or 0) {
        cc = BlkValueRead(findt, ffrom++);
        if (cc == '\') {
            cc = BlkValueRead(findt, ffrom++);
            if (cc ~= 0) cc = BlkValueRead(findt, ffrom++);
        }
    }
    if (cc == 0) return "Character range never ends";
    par2 = ffrom-1;
    if ((par2 > par1 + 1) &&
        (BlkValueRead(findt, par1) == ':') &&
        (BlkValueRead(findt, par2-1) == ':') &&
        (BlkValueRead(findt, par2-2) ~= '\'))
        return "POSIX named character classes unsupported";
    bits = IT_RE_RangeSyntaxCorrect(findt, par1, par2);
    if (bits) return bits;
    if (par1 < par2) cc = RANGE_RE_CC;
    else cc = NOTHING_RE_CC;
    quantifiable = true;
'*': if (quantifiable == false) return "quantifier misplaced";
    cc = QUANTIFIER_RE_CC;
    par1 = 0; par2 = 30000;
    if (BlkValueRead(findt, ffrom) == '?') { ffrom++; par3 = true; }
    quantifiable = false;
'+': if (quantifiable == false) return "quantifier misplaced";
    cc = QUANTIFIER_RE_CC;
    par1 = 1; par2 = 30000;
```

```
            if (BlkValueRead(findt, ffrom) == '?') { ffrom++; par3 = true; }
            quantifiable = false;
        '?': if (quantifiable == false) return "quantifier misplaced";
            cc = QUANTIFIER_RE_CC;
            par1 = 0; par2 = 1;
            if (BlkValueRead(findt, ffrom) == '?') { ffrom++; par3 = true; }
            quantifiable = false;
    }
}
.CClassKnown;
if (cc >= 0) {
    quantifiable = true;
    if ((attach_to-->RE_CCLASS == LITERAL_RE_CC) &&
        (BlkValueRead(findt, ffrom) ~= '*' or '+' or '?' or '{')) {
        (attach_to-->RE_PAR2)++;
        if (IT_RE_Trace == 2) {
            print "Extending literal by ", cc, "=", (char) cc, "^";
        }
        continue;
    }
    cc = LITERAL_RE_CC; par1 = ffrom-1; par2 = ffrom;
}
if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
if (IT_RE_Trace == 2) {
    print "Attaching packet ", no_packets+1, " to ";
    IT_RE_DebugNode(attach_to, findt);
    IT_RE_DebugTree(findt);
}
token = IT_RE_Node(no_packets++, cc, par1, par2, par3);
if ((token-->RE_CCLASS == SUBEXP_RE_CC) && (token-->RE_PAR2 == 0)) {
    RE_Subexpressions-->(token-->RE_PAR1) = token;
    (RE_Subexpressions-->10)++;
}
if ((attach_to-->RE_CCLASS == SUBEXP_RE_CC or CHOICE_RE_CC or IF_RE_CC) &&
    (attach_to-->RE_DOWN == NULL)) {
    attach_to-->RE_DOWN = token; token-->RE_UP = attach_to;
} else {
    if ((token-->RE_CCLASS == CHOICE_RE_CC) &&
        ((attach_to-->RE_UP)-->RE_CCLASS == CHOICE_RE_CC)) {
        no_packets--; token = attach_to-->RE_UP;
    } else {
        if (token-->RE_CCLASS == CHOICE_RE_CC) {
            while (attach_to-->RE_PREVIOUS ~= NULL)
                attach_to = attach_to-->RE_PREVIOUS;
        }
        if (token-->RE_CCLASS == QUANTIFIER_RE_CC or CHOICE_RE_CC) {
            token-->RE_PREVIOUS = attach_to-->RE_PREVIOUS;
            token-->RE_UP = attach_to-->RE_UP;
            if ((attach_to-->RE_UP ~= NULL) && (attach_to-->RE_PREVIOUS == NULL))
                (attach_to-->RE_UP)-->RE_DOWN = token;
            token-->RE_DOWN = attach_to;
            bits = attach_to;
```

```
                    while (bits ~= NULL) {
                        bits-->RE_UP = token;
                        bits = bits-->RE_NEXT;
                    }
                    attach_to-->RE_PREVIOUS = NULL;
                    if (token-->RE_PREVIOUS ~= NULL)
                        (token-->RE_PREVIOUS)-->RE_NEXT = token;
                } else {
                    attach_to-->RE_NEXT = token; token-->RE_PREVIOUS = attach_to;
                    token-->RE_UP = attach_to-->RE_UP;
                }
            }
        }
        if (token-->RE_CCLASS == CHOICE_RE_CC) {
            if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
            token-->RE_NEXT = IT_RE_Node(no_packets++, CHOICE_RE_CC, 0, 0, 0);
            (token-->RE_NEXT)-->RE_PREVIOUS = token;
            (token-->RE_NEXT)-->RE_UP = token-->RE_UP;
            token = token-->RE_NEXT;
        }
        attach_to = token;
        if (IT_RE_Trace == 2) {
            print "Result:^";
            IT_RE_DebugTree(findt);
        }
    }
    if (blevel ~= 0) return "subexpression bracket mismatch";
    if (exactly) {
        if (no_packets+3 > RE_MAX_PACKETS) return "regexp too complex";
        exactly = RE_PACKET_space-->RE_DOWN;
        token = IT_RE_Node(no_packets++, START_RE_CC, 0, 0, 0);
        RE_PACKET_space-->RE_DOWN = token; token-->RE_UP = RE_PACKET_space;
        attach_to = IT_RE_Node(no_packets++, SUBEXP_RE_CC, -1, 3, 0);
        token-->RE_NEXT = attach_to; attach_to-->RE_PREVIOUS = token;
        attach_to-->RE_UP = RE_PACKET_space;
        attach_to-->RE_NEXT = IT_RE_Node(no_packets++, END_RE_CC, 0, 0, 0);
        (attach_to-->RE_NEXT)-->RE_PREVIOUS = attach_to;
        (attach_to-->RE_NEXT)-->RE_UP = RE_PACKET_space;
        attach_to-->RE_DOWN = exactly;
        while (exactly ~= NULL) {
            exactly-->RE_UP = attach_to; exactly = exactly-->RE_NEXT;
        }
    }
    no_packets = IT_RE_ExpandChoices(RE_PACKET_space, no_packets);
    if (IT_RE_Trace) {
        print "Compiled pattern:^";
        IT_RE_DebugTree(findt);
    }
    bits = IT_RE_CheckTree(RE_PACKET_space, no_subs); if (bits) return bits;
    return no_packets;
];
```

```
[ IT_RE_RangeSyntaxCorrect findt rf rt
    i chm;
    for (i=rf: i<rt: i++) {
        chm = BlkValueRead(findt, i);
        if ((chm == '\') && (i+1<rt)) {
            chm = BlkValueRead(findt, ++i);
            if (((chm >= 'a') && (chm <= 'z')) ||
                ((chm >= 'A') && (chm <= 'Z'))) {
                if (chm ~= 's' or 'S' or 'p' or 'P' or 'w' or 'W' or 'd'
                    or 'D' or 'n' or 't' or 'l' or 'L' or 'u' or 'U')
                    return "Invalid escape in {} range";
            }
        }
        if ((i+2<rt) && (BlkValueRead(findt, i+1) == '-')) {
            if (chm > BlkValueRead(findt, i+2)) return "Invalid {} range";
            i=i+2;
        }
    }
    rfalse;
];
[ IT_RE_ExpandChoices token no_packets
    rv prev nex holder new ct n cond_node then_node else_node;
    while (token ~= NULL) {
        if (token-->RE_CCLASS == IF_RE_CC) {
            if ((token-->RE_DOWN)-->RE_CCLASS == CHOICE_RE_CC) {
                for (nex=token-->RE_DOWN, n=0: nex~=NULL: nex=nex-->RE_NEXT) n++;
                if (n~=2) return "conditional has too many clauses";
                if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
                cond_node = IT_RE_Node(no_packets++, CONDITION_RE_CC, 0, 0, 0);
                if (token-->RE_PAR1 >= 1) {
                    cond_node-->RE_PAR1 = token-->RE_PAR1;
                }
                then_node = token-->RE_DOWN;
                then_node-->RE_CCLASS = THEN_RE_CC;
                else_node = then_node-->RE_NEXT;
                else_node-->RE_CCLASS = ELSE_RE_CC;
                if (cond_node-->RE_PAR1 < 1) {
                    cond_node-->RE_DOWN = then_node-->RE_DOWN;
                    then_node-->RE_DOWN = (then_node-->RE_DOWN)-->RE_NEXT;
                    if (then_node-->RE_DOWN ~= NULL)
                        (then_node-->RE_DOWN)-->RE_PREVIOUS = NULL;
                    (cond_node-->RE_DOWN)-->RE_NEXT = NULL;
                    (cond_node-->RE_DOWN)-->RE_UP = cond_node;
                }
                token-->RE_DOWN = cond_node; cond_node-->RE_UP = token;
                cond_node-->RE_NEXT = then_node; then_node-->RE_PREVIOUS = cond_node;
            } else {
                if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
                cond_node = IT_RE_Node(no_packets++, CONDITION_RE_CC, 0, 0, 0);
                if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
                then_node = IT_RE_Node(no_packets++, THEN_RE_CC, 0, 0, 0);
                if (token-->RE_PAR1 >= 1) {
                    cond_node-->RE_PAR1 = token-->RE_PAR1;
```

```
            then_node-->RE_DOWN = token-->RE_DOWN;
        } else {
            cond_node-->RE_DOWN = token-->RE_DOWN;
            then_node-->RE_DOWN = (token-->RE_DOWN)-->RE_NEXT;
            (cond_node-->RE_DOWN)-->RE_NEXT = NULL;
            (cond_node-->RE_DOWN)-->RE_UP = cond_node;
        }
        token-->RE_DOWN = cond_node;
        cond_node-->RE_UP = token; cond_node-->RE_NEXT = then_node;
        then_node-->RE_PREVIOUS = cond_node; then_node-->RE_UP = token;
        then_node-->RE_NEXT = NULL;
        if (then_node-->RE_DOWN ~= NULL)
            (then_node-->RE_DOWN)-->RE_PREVIOUS = NULL;
        for (nex = then_node-->RE_DOWN: nex ~= NULL: nex = nex-->RE_NEXT) {
            nex-->RE_UP = then_node;
        }
    }
    if (cond_node-->RE_DOWN ~= NULL) {
        nex = cond_node-->RE_DOWN;
        if ((nex-->RE_CCLASS ~= SUBEXP_RE_CC) ||
            (nex-->RE_NEXT ~= NULL) ||
            (nex-->RE_PAR2 ~= 1 or 2)) {
            !IT_RE_DebugSubtree(0, 0, nex, true);
            return "condition not lookahead/behind";
        }
    }
}
if ((token-->RE_CCLASS == CHOICE_RE_CC) && (token-->RE_PAR1 < 1)) {
    prev = token-->RE_PREVIOUS;
    nex = token-->RE_NEXT;
    while ((nex ~= NULL) && (nex-->RE_CCLASS == CHOICE_RE_CC))
        nex = nex-->RE_NEXT;
    holder = token-->RE_UP; if (holder == NULL) return "bang";
    if (no_packets >= RE_MAX_PACKETS) return "regexp too complex";
    new = IT_RE_Node(no_packets++, DISJUNCTION_RE_CC, 0, 0, 0);
    holder-->RE_DOWN = new; new-->RE_UP = holder;
    if (prev ~= NULL) {
        prev-->RE_NEXT = new; new-->RE_PREVIOUS = prev;
    }
    if (nex ~= NULL) {
        nex-->RE_PREVIOUS = new; new-->RE_NEXT = nex;
    }
    new-->RE_DOWN = token;
    token-->RE_PREVIOUS = NULL;
    ct = 1;
    while (token ~= NULL) {
        token-->RE_PAR1 = ct++;
        token-->RE_UP = new;
        if ((token-->RE_NEXT ~= NULL) &&
            ((token-->RE_NEXT)-->RE_CCLASS ~= CHOICE_RE_CC))
            token-->RE_NEXT = NULL;
        token = token-->RE_NEXT;
    }
```

```
            new-->RE_PAR1 = ct-1;
            if (token ~= NULL) token-->RE_NEXT = NULL;
            token = new; continue;
        }
        if (token-->RE_DOWN ~= NULL) {
            no_packets = IT_RE_ExpandChoices(token-->RE_DOWN, no_packets);
            if ((no_packets<0) || (no_packets >= RE_MAX_PACKETS)) break;
        }
        token = token-->RE_NEXT;
    }
    return no_packets;
];
[ IT_RE_CheckTree token no_subs
    rv;
    while (token ~= NULL) {
        if (token-->RE_CCLASS == VARIABLE_RE_CC) {
            if (token-->RE_PAR1 >= no_subs) return "reference to nonexistent group";
        }
        if ((token-->RE_CCLASS == SUBEXP_RE_CC) &&
            (token-->RE_PAR2 == 1 or 2) &&
            (token-->RE_PAR3 == -1)) {
            token-->RE_PAR3 = IT_RE_Width(token-->RE_DOWN);
            if (token-->RE_PAR3 == -1) return "variable length lookbehind not implemented";
        }
        if (token-->RE_DOWN ~= NULL) {
            rv = IT_RE_CheckTree(token-->RE_DOWN, no_subs);
            if (rv) return rv;
        }
        token = token-->RE_NEXT;
    }
    rfalse;
];
[ IT_RE_Width token downwards
    w rv aw choice;
    while (token ~= NULL) {
        switch (token-->RE_CCLASS) {
            DIGIT_RE_CC, NONDIGIT_RE_CC, WHITESPACE_RE_CC, NONWHITESPACE_RE_CC,
            PUNCTUATION_RE_CC, NONPUNCTUATION_RE_CC, WORD_RE_CC, NONWORD_RE_CC,
            ANYTHING_RE_CC, NOTHING_RE_CC, RANGE_RE_CC, NEWLINE_RE_CC, TAB_RE_CC,
            UCASE_RE_CC, NONUCASE_RE_CC, LCASE_RE_CC, NONLCASE_RE_CC:
                w++;
            START_RE_CC, END_RE_CC, BOUNDARY_RE_CC, NONBOUNDARY_RE_CC, ALWAYS_RE_CC:
                ;
            LITERAL_RE_CC:
                w = w + token-->RE_PAR2 - token-->RE_PAR1;
            VARIABLE_RE_CC:
                return -1;
            IF_RE_CC:
                rv = IT_RE_Width((token-->RE_DOWN)-->RE_NEXT);
                if (rv == -1) return -1;
                if (rv ~= IT_RE_Width(((token-->RE_DOWN)-->RE_NEXT)-->RE_NEXT))
                    return -1;
                w = w + rv;
```

```
            SUBEXP_RE_CC:
                if (token-->RE_PAR2 == 1 or 2) rv = 0;
                else {
                    rv = IT_RE_Width(token-->RE_DOWN);
                    if (rv == -1) return -1;
                }
                w = w + rv;
            QUANTIFIER_RE_CC:
                if (token-->RE_PAR1 ~= token-->RE_PAR2) return -1;
                rv = IT_RE_Width(token-->RE_DOWN);
                if (rv == -1) return -1;
                w = w + rv*(token-->RE_PAR1);
            DISJUNCTION_RE_CC:
                aw = -1;
                for (choice = token-->RE_DOWN: choice ~= NULL: choice = choice-->RE_NEXT) {
                    rv = IT_RE_Width(choice-->RE_DOWN);
                    !print "Option found ", rv, "^";
                    if (rv == -1) return -1;
                    if ((aw >= 0) && (aw ~= rv)) return -1;
                    aw = rv;
                }
                w = w + aw;
            SENSITIVITY_RE_CC:
                ;
        }
        if (downwards) return w;
        if (token ~= NULL) token = token-->RE_NEXT;
    }
    return w;
];
```

§**11. Parser.** The virtue of all of that tree compilation is that the code which actually does the work – which parses the source text to see if the regular expression matches it – is much shorter and quicker: indeed, it takes up fewer lines than the compiler part, which goes to show that decoding regular expression syntax is a more complex task than acting upon it. This would have surprised the pioneers of regexp, but the syntax has become much more complicated over the decades because of a steady increase in the number of extended notations. The process shows no sign of stopping, with Python and PCRE continuing to push boundaries beyond Perl, which was once thought the superest, duperest regexp syntax there could be. However: to work.

The main matcher simply starts a recursive subroutine to perform the match. However, the subroutine tests for a match at a particular position in the source text; so the main routine tries the subroutine everywhere convenient in the source text, from left to right, until a match is made – unless the regexp is constrained by a `^` glyph to begin matching at the start of the source text, which will cause a `START_RE_CC` node to be the eldest child of the `\0` root.

```
Global IT_RE_RewindCount;
[ IT_RE_PrintNoRewinds; print IT_RE_RewindCount; ];

Constant CIS_MFLAG = 1;
Constant ACCUM_MFLAG = 2;

[ IT_RE_Parse findt indt ipos insens
    ilen rv root i initial_mode;

    ilen = IT_CharacterLength(indt);
```

```
    if ((ipos<0) || (ipos>ilen)) return -1;
    root = RE_PACKET_space;
    initial_mode = 0; if (insens) initial_mode = CIS_MFLAG;
    IT_RE_Clear_Markers(RE_PACKET_space);
    for (:ipos<=ilen:ipos++) {
        if ((RE_PACKET_space-->RE_DOWN ~= NULL) &&
            ((RE_PACKET_space-->RE_DOWN)-->RE_CCLASS == START_RE_CC) &&
            (ipos>0)) { rv = -1; break; }
        if (ipos > 0) IT_RE_EraseConstraints(RE_PACKET_space, initial_mode);
        IT_RE_RewindCount = 0;
        rv = IT_RE_ParseAtPosition(findt, indt, ipos, ilen, RE_PACKET_space, initial_mode);
        if (rv >= 0) break;
    }
    if (rv == -1) {
        root-->RE_DATA1 = -1;
        root-->RE_DATA2 = -1;
    } else {
        root-->RE_DATA1 = ipos;
        root-->RE_DATA2 = ipos+rv;
    }
    return rv;
];
```

§**12. Parse At Position.** `IT_RE_ParseAtPosition(findt, indt, ifrom, ito)` attempts to match text beginning at position `ifrom` in the indexed text `indt` and extending for any length up to position `ito`: it returns the number of characters which were matched (which can legitimately be 0), or -1 if no match could be made. `findt` is the original text of the regular expression in its precompiled form, which we need partly to print good debugging information, but mostly in order to match against a `LITERAL_RE_CC` node.

```
[ IT_RE_ParseAtPosition findt indt ifrom ito token mode_flags
    outcome ipos npos rv i ch edge rewind_this;
    if (ifrom > ito) return -1;
    ipos = ifrom;
    .Rewind;
    while (token ~= NULL) {
        outcome = false;
                if (IT_RE_Trace) {
                    print "Matching at ", ipos, ": ";
                    IT_RE_DebugNode(token, findt, true);
                }
        if (ipos<ito) ch = BlkValueRead(indt, ipos); else ch = 0;
        token-->RE_MODES = mode_flags; ! Save in case of backtrack
        switch (token-->RE_CCLASS) {
            ! Should never happen
            CHOICE_RE_CC: return "internal error";
            ! Mode switches
            SENSITIVITY_RE_CC:
                if (token-->RE_PAR1) mode_flags = mode_flags | CIS_MFLAG;
                else mode_flags = mode_flags & (~CIS_MFLAG);
```

```
    outcome = true;
! Zero-length positional markers
ALWAYS_RE_CC:
    outcome = true;
NEVER_RE_CC:
START_RE_CC:
    if (ipos == 0) outcome = true;
END_RE_CC:
    if (BlkValueRead(indt, ipos) == 0) outcome = true;
BOUNDARY_RE_CC:
    rv = 0;
    if (BlkValueRead(indt, ipos) == 0 or 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') rv++;
    if (ipos == 0) ch = 0;
    else ch = BlkValueRead(indt, ipos-1);
    if (ch == 0 or 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') rv++;
    if (rv == 1) outcome = true;
NONBOUNDARY_RE_CC:
    rv = 0;
    if (BlkValueRead(indt, ipos) == 0 or 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') rv++;
    if (ipos == 0) ch = 0;
    else ch = BlkValueRead(indt, ipos-1);
    if (ch == 0 or 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') rv++;
    if (rv ~= 1) outcome = true;
! Control constructs
IF_RE_CC:
    i = token-->RE_PAR1; ch = false;
    if (IT_RE_Trace) {
        print "Trying conditional from ", ipos, ": ";
        IT_RE_DebugNode(token, findt, true);
    }
    if (i >= 1) {
        if ((i<RE_Subexpressions-->10) &&
            ((RE_Subexpressions-->i)-->RE_DATA1 >= 0)) ch = true;
    } else {
        rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito,
            (token-->RE_DOWN)-->RE_DOWN, mode_flags);
        if (rv >= 0) ch = true;
    }
    if (IT_RE_Trace) {
        print "Condition found to be ", ch, "^";
```

```
    }
    if (ch) {
        rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito,
            ((token-->RE_DOWN)-->RE_NEXT)-->RE_DOWN, mode_flags);
        !print "Then clause returned ", rv, "^";
    } else {
        if ((((token-->RE_DOWN)-->RE_NEXT)-->RE_NEXT) == NULL)
            rv = 0; ! The empty else clause matches
        else rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito,
            (((token-->RE_DOWN)-->RE_NEXT)-->RE_NEXT)-->RE_DOWN, mode_flags);
        !print "Else clause returned ", rv, "^";
    }
    if (rv >= 0) {
        outcome = true;
        ipos = ipos + rv;
    }
DISJUNCTION_RE_CC:
    if (IT_RE_Trace) {
        print "Trying disjunction from ", ipos, ": ";
        IT_RE_DebugNode(token, findt, true);
    }
    for (ch = token-->RE_DOWN: ch ~= NULL: ch = ch-->RE_NEXT) {
        if (ch-->RE_PAR1 <= token-->RE_CONSTRAINT) continue;
        if (IT_RE_Trace) {
            print "Trying choice at ", ipos, ": ";
            IT_RE_DebugNode(ch, findt, true);
        }
        rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito,
            ch-->RE_DOWN, mode_flags);
        if (rv >= 0) {
            token-->RE_DATA1 = ipos; ! Where match was made
            token-->RE_DATA2 = ch-->RE_PAR1; ! Option taken
            ipos = ipos + rv;
            outcome = true;
            if (IT_RE_Trace) {
                print "Choice worked with width ", rv, ": ";
                IT_RE_DebugNode(ch, findt, true);
            }
            break;
        } else {
            if (mode_flags & ACCUM_MFLAG == false)
                IT_RE_FailSubexpressions(ch-->RE_DOWN);
        }
    }
    if (outcome == false) {
        if (IT_RE_Trace) {
            print "Failed disjunction from ", ipos, ": ";
            IT_RE_DebugNode(token, findt, true);
        }
        token-->RE_DATA1 = ipos; ! Where match was tried
        token-->RE_DATA2 = -1; ! No option was taken
    }
SUBEXP_RE_CC:
```

```
    if (token-->RE_PAR2 == 1 or 2) {
        npos = ipos - token-->RE_PAR3;
        if (npos<0) rv = -1; ! Lookbehind fails: nothing behind
        else rv = IT_RE_ParseAtPosition(findt, indt, npos, ito, token-->RE_DOWN,
            mode_flags);
    } else {
        switch (token-->RE_PAR3) {
            0: rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito, token-->RE_DOWN,
                mode_flags);
            1: rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito, token-->RE_DOWN,
                mode_flags & (~CIS_MFLAG));
            2: rv = IT_RE_ParseAtPosition(findt, indt, ipos, ito, token-->RE_DOWN,
                mode_flags | CIS_MFLAG);
        }
    }
    npos = ipos;
    if (rv >= 0) npos = ipos + rv;
    switch (token-->RE_PAR2) {
        1: if (rv >= 0) rv = 0;
        2: if (rv >= 0) rv = -1; else rv = 0;
    }
    if (rv >= 0) {
        token-->RE_DATA1 = ipos;
        ipos = ipos + rv;
        token-->RE_DATA2 = npos;
        outcome = true;
    } else {
        if (mode_flags & ACCUM_MFLAG == false) {
            token-->RE_DATA1 = -1;
            token-->RE_DATA2 = -1;
        }
    }
    if (token-->RE_PAR2 == 2) IT_RE_FailSubexpressions(token, true);
QUANTIFIER_RE_CC:
    token-->RE_DATA1 = ipos;
    if ((token-->RE_DOWN)-->RE_CCLASS == SUBEXP_RE_CC) {
        (token-->RE_DOWN)-->RE_CACHE1 = -1;
        (token-->RE_DOWN)-->RE_CACHE2 = -1;
    }
    if (IT_RE_Trace) {
        print "Trying quantifier from ", ipos, ": ";
        IT_RE_DebugNode(token, findt, true);
    }
    if (token-->RE_PAR3 == false) { ! Greedy quantifier
        !edge = ito; if (token-->RE_CONSTRAINT >= 0) edge = token-->RE_CONSTRAINT;
        edge = token-->RE_PAR2;
        if (token-->RE_CONSTRAINT >= 0) edge = token-->RE_CONSTRAINT;
        rv = -1;
        for (i=0, npos=ipos: i<edge: i++) {
            if (IT_RE_Trace) {
                print "Trying quant rep ", i+1, " at ", npos, ": ";
                IT_RE_DebugNode(token, findt, true);
            }
```

```
            rv = IT_RE_ParseAtPosition(findt, indt, npos, ito, token-->RE_DOWN,
                mode_flags | ACCUM_MFLAG);
            if (rv < 0) break;
            if ((token-->RE_DOWN)-->RE_CCLASS == SUBEXP_RE_CC) {
                (token-->RE_DOWN)-->RE_CACHE1 = (token-->RE_DOWN)-->RE_DATA1;
                (token-->RE_DOWN)-->RE_CACHE2 = (token-->RE_DOWN)-->RE_DATA2;
            }
            if ((rv == 0) && (token-->RE_PAR2 == 30000) && (i>=1)) { i++; break; }
            npos = npos + rv;
        }
        if ((i >= token-->RE_PAR1) && (i <= token-->RE_PAR2))
            outcome = true;
    } else { ! Lazy quantifier
        edge = token-->RE_PAR1;
        if (token-->RE_CONSTRAINT > edge) edge = token-->RE_CONSTRAINT;
        for (i=0, npos=ipos: (npos<ito) && (i < token-->RE_PAR2): i++) {
            if (i >= edge) break;
            if (IT_RE_Trace) {
                print "Trying quant rep ", i+1, " at ", npos, ": ";
                IT_RE_DebugNode(token, findt, true);
            }
            rv = IT_RE_ParseAtPosition(findt, indt, npos, ito, token-->RE_DOWN,
                mode_flags | ACCUM_MFLAG);
            if (rv < 0) break;
            if ((token-->RE_DOWN)-->RE_CCLASS == SUBEXP_RE_CC) {
                (token-->RE_DOWN)-->RE_CACHE1 = (token-->RE_DOWN)-->RE_DATA1;
                (token-->RE_DOWN)-->RE_CACHE2 = (token-->RE_DOWN)-->RE_DATA2;
            }
            if ((rv == 0) && (token-->RE_PAR2 == 30000) && (i>=1)) { i++; break; }
            npos = npos + rv;
        }
        if ((i >= edge) && (i <= token-->RE_PAR2))
            outcome = true;
    }
    if (outcome) {
        if (token-->RE_PAR3 == false) { ! Greedy quantifier
            if (i > token-->RE_PAR1) { ! I.e., if we have been greedy
                token-->RE_DATA2 = i-1; ! And its edge limitation
            } else {
                token-->RE_DATA2 = -1;
            }
        } else { ! Lazy quantifier
            if (i < token-->RE_PAR2) { ! I.e., if we have been lazy
                token-->RE_DATA2 = i+1; ! And its edge limitation
            } else {
                token-->RE_DATA2 = -1;
            }
        }
        ipos = npos;
        if ((i == 0) && (mode_flags & ACCUM_MFLAG == false))
            IT_RE_FailSubexpressions(token-->RE_DOWN);
        if ((token-->RE_DOWN)-->RE_CCLASS == SUBEXP_RE_CC) {
            (token-->RE_DOWN)-->RE_DATA1 = (token-->RE_DOWN)-->RE_CACHE1;
```

```
            (token-->RE_DOWN)-->RE_DATA2 = (token-->RE_DOWN)-->RE_CACHE2;
        }
        if (IT_RE_Trace) {
            print "Successful quant reps ", i, ": ";
            IT_RE_DebugNode(token, findt, true);
        }
    } else {
        !token-->RE_DATA2 = -1;
        if (mode_flags & ACCUM_MFLAG == false)
            IT_RE_FailSubexpressions(token-->RE_DOWN);
        if (IT_RE_Trace) {
            print "Failed quant reps ", i, ": ";
            IT_RE_DebugNode(token, findt, true);
        }
    }
! Character classes
NOTHING_RE_CC: ;
ANYTHING_RE_CC: if (ch) outcome = true; ipos++;
WHITESPACE_RE_CC:
    if (ch == 10 or 13 or 32 or 9) { outcome = true; ipos++; }
NONWHITESPACE_RE_CC:
    if ((ch) && (ch ~= 10 or 13 or 32 or 9)) { outcome = true; ipos++; }
PUNCTUATION_RE_CC:
    if (ch == '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') { outcome = true; ipos++; }
NONPUNCTUATION_RE_CC:
    if ((ch) && (ch ~= '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '})) { outcome = true; ipos++; }
WORD_RE_CC:
    if ((ch) && (ch ~= 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '})) { outcome = true; ipos++; }
NONWORD_RE_CC:
    if (ch == 10 or 13 or 32 or 9
        or '.' or ',' or '!' or '?'
        or '-' or '/' or '"' or ':' or ';'
        or '(' or ')' or '[' or ']' or '{' or '}') { outcome = true; ipos++; }
DIGIT_RE_CC:
    if (ch == '0' or '1' or '2' or '3' or '4'
        or '5' or '6' or '7' or '8' or '9') { outcome = true; ipos++; }
NONDIGIT_RE_CC:
    if ((ch) && (ch ~= '0' or '1' or '2' or '3' or '4'
        or '5' or '6' or '7' or '8' or '9')) { outcome = true; ipos++; }
LCASE_RE_CC:
    if (CharIsOfCase(ch, 0)) { outcome = true; ipos++; }
NONLCASE_RE_CC:
    if ((ch) && (CharIsOfCase(ch, 0) == false)) { outcome = true; ipos++; }
UCASE_RE_CC:
    if (CharIsOfCase(ch, 1)) { outcome = true; ipos++; }
NONUCASE_RE_CC:
```

```
        if ((ch) && (CharIsOfCase(ch, 1) == false)) { outcome = true; ipos++; }
    NEWLINE_RE_CC: if (ch == 10) { outcome = true; ipos++; }
    TAB_RE_CC: if (ch == 9) { outcome = true; ipos++; }
    RANGE_RE_CC:
        if (IT_RE_Range(ch, findt,
            token-->RE_PAR1, token-->RE_PAR2, token-->RE_PAR3, mode_flags & CIS_MFLAG))
            { outcome = true; ipos++; }
    ! Substring matches
    LITERAL_RE_CC:
        rv = IT_RE_MatchSubstring(indt, ipos,
            findt, token-->RE_PAR1, token-->RE_PAR2, mode_flags & CIS_MFLAG);
        if (rv >= 0) { ipos = ipos + rv; outcome = true; }
    VARIABLE_RE_CC:
        i = token-->RE_PAR1;
        if ((RE_Subexpressions-->i)-->RE_DATA1 >= 0) {
            rv = IT_RE_MatchSubstring(indt, ipos,
                indt, (RE_Subexpressions-->i)-->RE_DATA1,
                (RE_Subexpressions-->i)-->RE_DATA2, mode_flags & CIS_MFLAG);
            if (rv >= 0) { ipos = ipos + rv; outcome = true; }
        }
        .NeverMatchIncompleteVar;
}
if (outcome == false) {
    if (IT_RE_RewindCount++ >= 10000) {
        if (IT_RE_RewindCount == 10001) {
            style bold; print "OVERFLOW^"; style roman;
        }
        return -1;
    }
    if (IT_RE_Trace) {
        print "Rewind sought from failure at pos ", ipos, " with: ";
            IT_RE_DebugNode(token, findt, true);
    }
    if ((token-->RE_CCLASS == QUANTIFIER_RE_CC) &&
        (IT_RE_SeekBacktrack(token-->RE_DOWN, findt, false, ito, false)))
        jump RewindFound;
    if (mode_flags & ACCUM_MFLAG == false) IT_RE_FailSubexpressions(token);
    token = token-->RE_PREVIOUS;
    while (token ~= NULL) {
        if (IT_RE_SeekBacktrack(token, findt, true, ito, false)) {
            .RewindFound;
            ipos = token-->RE_DATA1;
            mode_flags = token-->RE_MODES;
            if (mode_flags & ACCUM_MFLAG == false)
                IT_RE_FailSubexpressions(token, true);
            if (ipos == -1)
                IT_RE_DebugTree(findt, true);
            if (IT_RE_Trace) {
                print "^[", ifrom, ",", ito, "] rewinding to ", ipos, " at ";
                IT_RE_DebugNode(token, findt, true);
            }
            jump Rewind;
        }
```

```
                token = token-->RE_PREVIOUS;
            }
            if (IT_RE_Trace)
                print "^Rewind impossible^";
            return -1;
        }
        token = token-->RE_NEXT;
    }
    return ipos - ifrom;
];
```

§**13. Backtracking.** It would be very straightforward to match regular expressions with the above recursive code if, for every node, there were a fixed number of characters (depending on the node) such that there would either be a match eating that many characters, or else no match at all. If that were true, we could simply march through the text matching until we could match no more, and although some nodes might have ambiguous readings, we could always match the first possibility which worked. There would never be any need to retreat.

Well, in fact that happy state does apply to a surprising number of nodes, and some quite complicated regular expressions can be made which use only them: `<abc>{2}\d\d\1`, for instance, matches a sequence of exactly 6 characters or else fails to match altogether, and there is never any need to backtrack. One reason why backtracking is a fairly good algorithm in practice is that these "good" cases occur fairly often, in subexpressions if not in the entire expression, and the simple method above disposes of them efficiently.

But in an expression like `ab+bb`, there is no alternative to backtracking if we are going to try to match the nodes from left to right: we match the "a", then we match as many "b"s as we can – but then we find that we have to match "bb", and this is necessarily impossible because we have just eaten all of the "b"s available. We therefore backtrack one node to the `b+` and try again. We obviously can't literally try again because that would give the same result: instead we impose a constraint. Suppose it previously matched a row of 23 letter "b"s, so that the quantifier `+` resulted in a multiplicity of 23. We then constrain the node and in effect consider it to be `b{1,22}`, that is, to match at least 1 and at most 22 letter "b"s. That still won't work, as it happens, so we backtrack again with a constraint tightened to make it `b{1,21}`, and now the match occurs as we would hope. When the expression becomes more complex, backtracking becomes a longer-distance, recursive procedure – we have to exhaust all possibilities of a more recent node before tracking back to one from longer ago. (This is why the worst test cases are those which entice us into a long, long series of matches only to find that a guess made right back at the start was ill-fated.)

Rather than describing `IT_RE_SeekBacktrack` in detail here, it is probably more useful to suggest that the reader observe it in action by setting `IT_RE_Trace` and trying a few regular expressions.

```
[ IT_RE_SeekBacktrack token findt downwards ito report_only
    untried;
    for (: token ~= NULL: token = token-->RE_NEXT) {
        if ((IT_RE_Trace) && (report_only == false)) {
            print "Scan for rewind: ";
            IT_RE_DebugNode(token, findt, true);
        }
        if ((token-->RE_CCLASS == SUBEXP_RE_CC) &&
            (token-->RE_PAR2 == 1 or 2 or 4)) {
            if (downwards) rfalse;
            continue;
        }
        if (token-->RE_DOWN ~= NULL) {
            if ((IT_RE_Trace) && (report_only == false)) print "Descend^";
            if (IT_RE_SeekBacktrack(token-->RE_DOWN, findt, false, ito, report_only)) rtrue;
```

```
        }
        untried = false;
        switch (token-->RE_CCLASS) {
            DISJUNCTION_RE_CC:
                if ((token-->RE_DATA2 >= 1) &&
                    (token-->RE_DATA2 < token-->RE_PAR1) &&
                    (token-->RE_CONSTRAINT < token-->RE_PAR1)) { ! Matched but earlier than last
                    if (report_only) rtrue;
                    if (token-->RE_CONSTRAINT == -1)
                        token-->RE_CONSTRAINT = 1;
                    else
                        (token-->RE_CONSTRAINT)++;
                    untried = true;
                }
            QUANTIFIER_RE_CC:
                if (token-->RE_CONSTRAINT ~= -2) {
                    if ((IT_RE_Trace) && (report_only == false)) {
                        print "Quant with cons not -2: ";
                        IT_RE_DebugNode(token, findt, true);
                    }
                    if (token-->RE_DATA2 >= 0) {
                        if (report_only) rtrue;
                        token-->RE_CONSTRAINT = token-->RE_DATA2;
                        untried = true;
                    }
                }
        }
        if (untried) {
            if (IT_RE_Trace) {
                print "Grounds for rewind at: ";
                IT_RE_DebugNode(token, findt, true);
            }
            IT_RE_EraseConstraints(token-->RE_NEXT);
            IT_RE_EraseConstraints(token-->RE_DOWN);
            rtrue;
        }
        if (downwards) rfalse;
    }
    rfalse;
];
```

§**14. Fail Subexpressions.** Here, an attempt to make a complicated match against the node in `token` has failed: that means that any subexpressions which were matched in the course of the attempt must also in retrospect be considered unmatched. So we work down through the subtree at `token` and empty any markers for subexpressions, which in effect clears their backslash variables – this is important as, otherwise, the contents left over could cause the alternative reading of the `token` to be misparsed if it refers to the backslash variables in question. (If you think nobody would ever be crazy enough to write a regular expression like that, you haven't see Perl's test suite.)

If the `downwards` flag is clear, it not only invalidates subexpression matches below the node but also to the right of the node – this is useful for a backtrack which runs back quite some distance.

```
[ IT_RE_FailSubexpressions token downwards;
    for (: token ~= NULL: token = token-->RE_NEXT) {
        if (token-->RE_DOWN ~= NULL) IT_RE_FailSubexpressions(token-->RE_DOWN);
        if (token-->RE_CCLASS == SUBEXP_RE_CC) {
            token-->RE_DATA1 = -1;
            token-->RE_DATA2 = -1;
        }
        if (downwards) break;
    }
];
```

§**15. Erasing Constraints.** As explained above, temporary constraints are placed on some nodes when we are backtracking to test possible cases. When we do backtrack, though, it's important to lift any constraints left over from the previous attempt to parse material which is part of or subsequent to the token whose match attempt has been abandoned.

```
[ IT_RE_EraseConstraints token;
    while (token ~= NULL) {
        switch (token-->RE_CCLASS) {
            DISJUNCTION_RE_CC: token-->RE_CONSTRAINT = -1;
            QUANTIFIER_RE_CC: token-->RE_CONSTRAINT = -1;
        }
        if (token-->RE_DOWN) IT_RE_EraseConstraints(token-->RE_DOWN);
        token = token-->RE_NEXT;
    }
];
```

§**16. Matching Literal Text.** Here we attempt to make a match of the substring of the indexed text `mindt` which runs from character `mfrom` to character `mto`, looking for it at the given position `ipos` in the source text `indt`.

```
[ IT_RE_MatchSubstring indt ipos mindt mfrom mto insens
    i ch;
    if (mfrom < 0) return 0;
    if (insens)
        for (i=mfrom:i<mto:i++) {
            ch = BlkValueRead(mindt, i);
            if (BlkValueRead(indt, ipos++) ~= ch or IT_RevCase(ch))
                return -1;
        }
    else
        for (i=mfrom:i<mto:i++)
            if (BlkValueRead(indt, ipos++) ~= BlkValueRead(mindt, i))
                return -1;
    return mto-mfrom;
];
```

§**17. Matching Character Range.** Suppose that a character range is stored in `findt` between the character positions `rf` and `rt`. Then `IT_RE_Range(ch, findt, rf, rt, negate, insens)` tests whether a given character `ch` lies within that character range, negating the outcome if `negate` is set, and performing comparisons case insensitively if `insens` is set.

```
[ IT_RE_Range ch findt rf rt negate insens
    i chm upper crev;
    if (ch == 0) rfalse;
    if (negate == true) {
        if (IT_RE_Range(ch, findt, rf, rt, false, insens)) rfalse;
        rtrue;
    }
    for (i=rf: i<rt: i++) {
        chm = BlkValueRead(findt, i);
        if ((chm == '\') && (i+1<rt)) {
            chm = BlkValueRead(findt, ++i);
            switch (chm) {
                's':
                    if (ch == 10 or 13 or 32 or 9) rtrue;
                'S':
                    if ((ch) && (ch ~= 10 or 13 or 32 or 9)) rtrue;
                'p':
                    if (ch == '.' or ',' or '!' or '?'
                        or '-' or '/' or '"' or ':' or ';'
                        or '(' or ')' or '[' or ']' or '{' or '}') rtrue;
                'P':
                    if ((ch) && (ch ~= '.' or ',' or '!' or '?'
                        or '-' or '/' or '"' or ':' or ';'
                        or '(' or ')' or '[' or ']' or '{' or '})) rtrue;
                'w':
                    if ((ch) && (ch ~= 10 or 13 or 32 or 9
                        or '.' or ',' or '!' or '?'
                        or '-' or '/' or '"' or ':' or ';'
```

```
                    or '(' or ')' or '[' or ']' or '{' or '}')) rtrue;
            'W':
                if (ch == 10 or 13 or 32 or 9
                    or '.' or ',' or '!' or '?'
                    or '-' or '/' or '"' or ':' or ';'
                    or '(' or ')' or '[' or ']' or '{' or '}') rtrue;
            'd':
                if (ch == '0' or '1' or '2' or '3' or '4'
                    or '5' or '6' or '7' or '8' or '9') rtrue;
            'D':
                if ((ch) && (ch ~= '0' or '1' or '2' or '3' or '4'
                    or '5' or '6' or '7' or '8' or '9')) rtrue;
            'l': if (CharIsOfCase(ch, 0)) rtrue;
            'L': if (CharIsOfCase(ch, 0) == false) rtrue;
            'u': if (CharIsOfCase(ch, 1)) rtrue;
            'U': if (CharIsOfCase(ch, 1) == false) rtrue;
            'n': if (ch == 10) rtrue;
            't': if (ch == 9) rtrue;
        }
    }
    if ((i+2<rt) && (BlkValueRead(findt, i+1) == '-')) {
        upper = BlkValueRead(findt, i+2);
        if ((ch >= chm) && (ch <= upper)) rtrue;
        if (insens) {
            crev = IT_RevCase(ch);
            if ((crev >= chm) && (crev <= upper)) rtrue;
        }
        i=i+2;
    } else {
        if (chm == ch) rtrue;
        if ((insens) && (chm == IT_RevCase(ch))) rtrue;
    }
  }
  rfalse;
];
```

§**18. Search And Replace.** And finally, last but not least: the routine which searches an indexed text `indt` trying to match it against `findt`. If `findtype` is set to `REGEXP_BLOB` then `findt` is expected to be a regular expression such as `ab+(c*de)?`, whereas if `findtype` is `CHR_BLOB` then it is expected only to be a simple string of characters taken literally, such as `frog`.

Each match found is replaced with the indexed text in `rindt`, except that if the blob type is `REGEXP_BLOB` then we recognise a few syntaxes as special: for instance, `\2` expands to the value of subexpression 2 as it was matched – see *Writing with Inform* for details.

The optional argument `insens` is a flag which, if set, causes the matching to be done case insensitively; the optional argument `exactly`, if set, causes the matching to work only if the entire `indt` is matched. (This is not especially useful with regular expressions, because the effect can equally be achieved by turning `ab+c`, say, into `^ab+c$`, but it is indeed useful where the blob type is `CHR_BLOB`.)

For an explanation of the use of the word "blob", see "IndexedText.i6t".

```
[ IT_Replace_RE findtype indt findt rindt insens exactly
    cindt csize ilen i cl mpos cpos ch chm;
    ilen = IT_CharacterLength(indt);
```

```
IT_RE_Err = 0;
switch (findtype) {
    REGEXP_BLOB: i = IT_RE_CompileTree(findt, exactly);
    CHR_BLOB: i = IT_CHR_CompileTree(findt, exactly);
    default: "*** bad findtype ***";
}
if ((i<0) || (i>RE_MAX_PACKETS)) {
    IT_RE_Err = i;
    print "*** Regular expression error: ", (string) IT_RE_Err, " ***^";
    RunTimeProblem(RTP_REGEXPSYNTAXERROR);
    return 0;
}
if (IT_RE_Trace) {
    IT_RE_DebugTree(findt);
    print "(compiled to ", i, " packets)^";
}
if (findtype == REGEXP_BLOB) IT_RE_EmptyMatchVars();
mpos = 0; chm = 0; cpos = 0;
while (IT_RE_Parse(findt, indt, mpos, insens) >= 0) {
    chm++;
    if (IT_RE_Trace) {
        print "^*** Match ", chm, " found (", RE_PACKET_space-->RE_DATA1, ",",
            RE_PACKET_space-->RE_DATA2, "): ";
        if (RE_PACKET_space-->RE_DATA1 == RE_PACKET_space-->RE_DATA2) {
            print "<empty>";
        }
        for (i=RE_PACKET_space-->RE_DATA1:i<RE_PACKET_space-->RE_DATA2:i++) {
            print (char) BlkValueRead(indt, i);
        }
        print " ***^";
    }
    if (rindt == 0) break; ! Accept only one match, replace nothing
    if (rindt ~= 0 or 1) {
        if (chm == 1) {
            cindt = BlkValueCreate(INDEXED_TEXT_TY);
            csize = BlkValueExtent(cindt);
        }
        for (i=cpos:i<RE_PACKET_space-->RE_DATA1:i++) {
            ch = BlkValueRead(indt, i);
            if (cl+1 >= csize) {
                if (BlkValueSetExtent(cindt, 2*cl, 7) == false) break;
                csize = BlkValueExtent(cindt);
            }
            BlkValueWrite(cindt, cl++, ch);
        }
        BlkValueWrite(cindt, cl, 0);
        IT_Concatenate(cindt, rindt, findtype, indt);
        csize = BlkValueExtent(cindt);
        cl = IT_CharacterLength(cindt);
    }
    mpos = RE_PACKET_space-->RE_DATA2; cpos = mpos;
```

```
        if (RE_PACKET_space-->RE_DATA1 == RE_PACKET_space-->RE_DATA2)
            mpos++;
        if (IT_RE_Trace) {
            if (chm == 100) { ! Purely to keep the output from being excessive
                print "(Stopping after 100 matches.)^"; break;
            }
        }
    }
    if (chm > 0) {
        if (rindt ~= 0 or 1) {
            for (i=cpos:i<ilen:i++) {
                ch = BlkValueRead(indt, i);
                if (cl+1 >= csize) {
                    if (BlkValueSetExtent(cindt, 2*cl, 8) == false) break;
                    csize = BlkValueExtent(cindt);
                }
                BlkValueWrite(cindt, cl++, ch);
            }
        }
        if (findtype == REGEXP_BLOB) {
            IT_RE_CreateMatchVars(indt);
            if (IT_RE_Trace)
                IT_RE_DebugMatchVars(indt);
        }
        if (rindt ~= 0 or 1) {
            BlkValueWrite(cindt, cl, 0);
            BlkValueCopy(indt, cindt);
            BlkFree(cindt);
        }
    }
    return chm;
];
```

§**19. Concatenation.** See the corresponding routine in "IndexedText.i6t": this is a variation which handles the special syntaxes used in search-and-replace.

```
[ IT_RE_Concatenate indt_to indt_from blobtype indt_ref
    pos len ch i tosize x y case;
    if ((indt_to==0) || (BlkType(indt_to) ~= INDEXED_TEXT_TY)) rfalse;
    if ((indt_from==0) || (BlkType(indt_from) ~= INDEXED_TEXT_TY)) return indt_to;
    pos = IT_CharacterLength(indt_to);
    tosize = BlkValueExtent(indt_to);
    len = IT_CharacterLength(indt_from);
    for (i=0:i<len:i++) {
        ch = BlkValueRead(indt_from, i);
        if ((ch == '\') && (i < len-1)) {
            ch = BlkValueRead(indt_from, ++i);
            if (ch == 'n') ch = 10;
            if (ch == 't') ch = 9;
            case = -1;
            if (ch == 'l') case = 0;
            if (ch == 'u') case = 1;
```

```
            if (case >= 0) ch = BlkValueRead(indt_from, ++i);
            if ((ch >= '0') && (ch <= '9')) {
                ch = ch - '0';
                if (ch < RE_Subexpressions-->10) {
                    x = (RE_Subexpressions-->ch)-->RE_DATA1;
                    y = (RE_Subexpressions-->ch)-->RE_DATA2;
                    if (x >= 0) {
                        for (:x<y:x++) {
                            ch = BlkValueRead(indt_ref, x);
                            if (pos+1 >= tosize) {
                                if (BlkValueSetExtent(indt_to, 2*tosize, 11) == false) break;
                                tosize = BlkValueExtent(indt_to);
                            }
                            if (case >= 0)
                                BlkValueWrite(indt_to, pos++, CharToCase(ch, case));
                            else
                                BlkValueWrite(indt_to, pos++, ch);
                        }
                    }
                }
                continue;
            }
        }
        if (pos+1 >= tosize) {
            if (BlkValueSetExtent(indt_to, 2*tosize, 12) == false) break;
            tosize = BlkValueExtent(indt_to);
        }
        BlkValueWrite(indt_to, pos++, ch);
    }
    BlkValueWrite(indt_to, pos, 0);
    return indt_to;
];
```

§**20. Stubs.** This time, there are no stubs: if there are no indexed texts, none of these routines is ever referred to.

```
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```

Char Template B/chart

Purpose

To decide whether letters are upper or lower case, and convert between the two.

---

§**1. Char Is Of Case.** The following decides whether a character `c` belongs to case `case`, where 0 means lower case and 1 means upper. `c` is interpreted according to the character casing chart in "UnicodeData.i6t", which means, it will be ZSCII for the Z-machine and Unicode for Glulx.

```
[ CharIsOfCase c case
    i tab min max len par;
    if (c<'A') rfalse;
    if (case == 0) {
        if ((c >= 'a') && (c <= 'z')) rtrue;
        tab = CharCasingChart0;
    } else {
        if ((c >= 'A') && (c <= 'Z')) rtrue;
        tab = CharCasingChart1;
    }
    if (c<128) rfalse;
    while (tab-->i) {
        min = tab-->i; i++;
        len = tab-->i; i++;
        i++;
        par = 0;
        if (len<0) { par = 1; len = -len; }
        if (c < min) rfalse;
        if (c < min+len) {
            if (par) { if ((c-min) % 2 == 0) rtrue; }
            else { rtrue; }
        }
    }
    rfalse;
];
```

§**2. Char To Case.** And the following converts character `c` to the desired case, or returns it unchanged if it is not a letter with variant casings.

```
[ CharToCase c case
    i tab min max len par del f;
    if (c<'A') return c;
    if (case == 1) {
        if ((c >= 'a') && (c <= 'z')) return c-32;
        tab = CharCasingChart0;
    } else {
        if ((c >= 'A') && (c <= 'Z')) return c+32;
        tab = CharCasingChart1;
    }
    if (c<128) return c;
    while (tab-->i) {
        min = tab-->i; i++;
        len = tab-->i; i++;
        del = tab-->i; i++;
        par = 0;
        if (len<0) { par = 1; len = -len; }
        if (c < min) return c;
        if (c < min+len) {
            f = false;
            if (par) { if ((c-min) % 2 == 0) f = true; }
            else { f = true; }
            if (f) {
                if (del == UNIC_NCT) return c;
                return c+del;
            }
        }
    }
    return c;
];
```

§**3. Reversing Case.** It's convenient to provide this relatively fast routine to reverse the case of a letter since this is an operation used frequently in regular expression matching (see "RegExp.i6t").

```
#IFDEF TARGET_ZCODE;
[ IT_RevCase ch;
    if (ch<'A') return ch;
    if ((ch >= 'a') && (ch <= 'z')) return ch-'a'+'A';
    if ((ch >= 'A') && (ch <= 'Z')) return ch-'A'+'a';
    if (ch<128) return ch;
    if ((ch >= 155) && (ch <= 157)) return ch+3; ! a, o, u umlaut in ZSCII
    if ((ch >= 158) && (ch <= 160)) return ch-3; ! A, O, U umlaut
    if ((ch >= 164) && (ch <= 165)) return ch+3; ! e, i umlaut
    if ((ch >= 167) && (ch <= 168)) return ch-3; ! E, I umlaut
    if ((ch >= 169) && (ch <= 174)) return ch+6; ! a, e, i, o, u, y acute
    if ((ch >= 175) && (ch <= 180)) return ch-6; ! A, E, I, O, U, Y acute
    if ((ch >= 181) && (ch <= 185)) return ch+5; ! a, e, i, o, u grave
    if ((ch >= 186) && (ch <= 190)) return ch-5; ! A, E, I, O, U grave
    if ((ch >= 191) && (ch <= 195)) return ch+5; ! a, e, i, o, u circumflex
    if ((ch >= 196) && (ch <= 200)) return ch-5; ! A, E, I, O, U circumflex
```

```
    if (ch == 201) return 202; ! a circle
    if (ch == 202) return 201; ! A circle
    if (ch == 203) return 204; ! o slash
    if (ch == 204) return 203; ! O slash
    if ((ch >= 205) && (ch <= 207)) return ch+3; ! a, n, o tilde
    if ((ch >= 208) && (ch <= 210)) return ch-3; ! A, N, O tilde
    if (ch == 211) return 212; ! ae ligature
    if (ch == 212) return 211; ! AE ligature
    if (ch == 213) return 214; ! c cedilla
    if (ch == 214) return 213; ! C cedilla
    if (ch == 215 or 216) return ch+2; ! thorn, eth
    if (ch == 217 or 218) return ch-2; ! Thorn, Eth
    if (ch == 220) return 221; ! oe ligature
    if (ch == 221) return 220; ! OE ligature
    return ch;
];
#IFNOT;
[ IT_RevCase ch;
    if (ch<'A') return ch;
    if ((ch >= 'a') && (ch <= 'z')) return ch-'a'+'A';
    if ((ch >= 'A') && (ch <= 'Z')) return ch-'A'+'a';
    if (ch<128) return ch;
    if (CharIsOfCase(ch, 0)) return CharToCase(ch, 1);
    if (CharIsOfCase(ch, 1)) return CharToCase(ch, 0);
    return ch;
];
#ENDIF;
```

§**4. Testing.** Not actually used: simply for testing the tables.

```
[ CharTestCases case i j;
    for (i=32: i<$E0; i++) {
        if ((i>=127) && (i<155)) continue;
        print i, " - ", (char) i, " -";
        if (CharIsOfCase(i, 0)) print "  lower";
        if (CharIsOfCase(i, 1)) print "  upper";
        j = CharToCase(i, 0); if (j ~= i) print "  tolower: ", (char) j;
        j = CharToCase(i, 1); if (j ~= i) print "  toupper: ", (char) j;
        print "^";
    }
];
```

UnicodeData Template B/unict

Purpose

To tabulate casings in the character set.

§**1. Source.** When this section is included, exactly one of the following constants is defined:

(a) `ZSCII_Tables`, meaning that we will use ZSCII as the character set for characters in indexed text strings.
(b) `Small_Unicode_Tables`, meaning that we will use Unicode but store only single-byte characters, so that only the codes 0 to 255 are valid: in effect ISO Latin-1.
(c) `Large_Unicode_Tables`, meaning that we will use Unicode and store two-byte characters, so that all of Unicode in the range 0 to 65535 are valid.

Whichever is defined, we must create two arrays:

(i) `CharCasingChart0`, a table indicating lower-case letters with transitions to convert them to upper case;
(ii) `CharCasingChart1`, vice versa.

Each array is a sequence of three-word records, consisting of the start of a character range, the size of the range (the number of characters in it), and the numerical offset to convert to the opposite case. For instance, the sequence $(97, 26, -32)$ means the 26 lower-case letters "a" to "z", and marks them as convertible to upper case by subtracting 32 from the character code (so "a", 97, becomes "A", 65). If the size of the range is negative, this indicates that only every alternate code is valid. (This makes for efficient storage since there are large parts of the Unicode number-space in which upper and lower case letters alternate.)

An offset of `UNIC_NCT` means no case change is possible; and any character not included in the ranges below is not a letter.

```
Constant UNIC_NCT = 10000; ! Safe as highest case-change delta is 8383
```

§**2. ZSCII Casing Tables.**

```
#IFDEF ZSCII_Tables;
Array CharCasingChart0 -->
    $0061 (  26) (     -32) $009b (   3) (       3) $00a1 (   1) (UNIC_NCT)
    $00a4 (   2) (       3) $00a6 (   1) (UNIC_NCT) $00a9 (   6) (       6)
    $00b5 (   5) (       5) $00bf (   5) (       5) $00c9 (  -3) (       1)
    $00cd (   3) (       3) $00d3 (  -3) (       1) $00d7 (   2) (       2)
    $00dc (   1) (       1) $0000
;
Array CharCasingChart1 -->
    $0041 (  26) (      32) $009e (   3) (      -3) $00a7 (   2) (      -3)
    $00af (   6) (      -6) $00ba (   5) (      -5) $00c4 (   5) (      -5)
    $00ca (  -3) (      -1) $00d0 (   3) (      -3) $00d4 (  -3) (      -1)
    $00d9 (   2) (      -2) $00dd (   1) (      -1) $0000
;
#ENDIF; ! ZSCII_Tables
```

§3. Small Unicode Casing Tables.

```
#IFDEF Small_Unicode_Tables;
Array CharCasingChart0 -->
    $0061 (  26) (     -32) $00aa (   1) (UNIC_NCT) $00b5 (   1) (UNIC_NCT) $00ba (   1) (UNIC_NCT)
    $00df (   1) (UNIC_NCT) $00e0 (  23) (     -32) $00f8 (   7) (     -32) $00ff (   1) (UNIC_NCT)
    $0000
;
Array CharCasingChart1 -->
    $0041 (  26) (      32) $00c0 (  23) (      32) $00d8 (   7) (      32) $0000
;
#ENDIF; ! Small_Unicode_Tables
```

§4. Large Unicode Casing Tables.

```
#IFDEF Large_Unicode_Tables;
Array CharCasingChart0 -->
    $0061 (  26) (     -32) $00aa (   1) (UNIC_NCT) $00b5 (   1) (     743) $00ba (   1) (UNIC_NCT)
    $00df (   1) (UNIC_NCT) $00e0 (  23) (     -32) $00f8 (   7) (     -32) $00ff (   1) (     121)
    $0101 ( -47) (      -1) $0131 (   1) (    -232) $0133 (  -5) (      -1) $0138 (   1) (UNIC_NCT)
    $013a ( -15) (      -1) $0149 (   1) (UNIC_NCT) $014b ( -45) (      -1) $017a (  -5) (      -1)
    $017f (   1) (    -300) $0180 (   1) (UNIC_NCT) $0183 (  -3) (      -1) $0188 (   1) (      -1)
    $018c (   1) (      -1) $018d (   1) (UNIC_NCT) $0192 (   1) (      -1) $0195 (   1) (      97)
    $0199 (   1) (      -1) $019a (   2) (UNIC_NCT) $019e (   1) (     130) $01a1 (  -5) (      -1)
    $01a8 (   1) (      -1) $01aa (   2) (UNIC_NCT) $01ad (   1) (      -1) $01b0 (   1) (      -1)
    $01b4 (  -3) (      -1) $01b9 (   1) (      -1) $01ba (   1) (UNIC_NCT) $01bd (   1) (      -1)
    $01be (   1) (UNIC_NCT) $01bf (   1) (      56) $01c6 (   1) (      -2) $01c9 (   1) (      -2)
    $01cc (   1) (      -2) $01ce ( -15) (      -1) $01dd (   1) (     -79) $01df ( -17) (      -1)
    $01f0 (   1) (UNIC_NCT) $01f3 (   1) (      -2) $01f5 (   1) (      -1) $01f9 ( -39) (      -1)
    $0221 (   1) (UNIC_NCT) $0223 ( -17) (      -1) $0234 (   3) (UNIC_NCT) $0250 (   3) (UNIC_NCT)
    $0253 (   1) (    -210) $0254 (   1) (    -206) $0255 (   1) (UNIC_NCT) $0256 (   2) (    -205)
    $0258 (   1) (UNIC_NCT) $0259 (   1) (    -202) $025a (   1) (UNIC_NCT) $025b (   1) (    -203)
    $025c (   4) (UNIC_NCT) $0260 (   1) (    -205) $0261 (   2) (UNIC_NCT) $0263 (   1) (    -207)
    $0264 (   4) (UNIC_NCT) $0268 (   1) (    -209) $0269 (   1) (    -211) $026a (   5) (UNIC_NCT)
    $026f (   1) (    -211) $0270 (   2) (UNIC_NCT) $0272 (   1) (    -213) $0273 (   2) (UNIC_NCT)
    $0275 (   1) (    -214) $0276 (  10) (UNIC_NCT) $0280 (   1) (    -218) $0281 (   2) (UNIC_NCT)
    $0283 (   1) (    -218) $0284 (   4) (UNIC_NCT) $0288 (   1) (    -218) $0289 (   1) (UNIC_NCT)
    $028a (   2) (    -217) $028c (   6) (UNIC_NCT) $0292 (   1) (    -219) $0293 (  29) (UNIC_NCT)
    $0390 (   1) (UNIC_NCT) $03ac (   1) (     -38) $03ad (   3) (     -37) $03b0 (   1) (UNIC_NCT)
    $03b1 (  17) (     -32) $03c2 (   1) (     -31) $03c3 (   9) (     -32) $03cc (   1) (     -64)
    $03cd (   2) (     -63) $03d0 (   1) (     -62) $03d1 (   1) (     -57) $03d5 (   1) (     -47)
    $03d6 (   1) (     -54) $03d7 (   1) (UNIC_NCT) $03d9 ( -23) (      -1) $03f0 (   1) (     -86)
    $03f1 (   1) (     -80) $03f2 (   1) (       7) $03f3 (   1) (UNIC_NCT) $03f5 (   1) (     -96)
    $03f8 (   1) (      -1) $03fb (   1) (      -1) $0430 (  32) (     -32) $0450 (  16) (     -80)
    $0461 ( -33) (      -1) $048b ( -53) (      -1) $04c2 ( -13) (      -1) $04d1 ( -37) (      -1)
    $04f9 (   1) (      -1) $0501 ( -15) (      -1) $0561 (  38) (     -48) $0587 (   1) (UNIC_NCT)
    $1d00 (  44) (UNIC_NCT) $1d62 (  10) (UNIC_NCT) $1e01 (-149) (      -1) $1e96 (   5) (UNIC_NCT)
    $1e9b (   1) (     -59) $1ea1 ( -89) (      -1) $1f00 (   8) (       8) $1f10 (   6) (       8)
    $1f20 (   8) (       8) $1f30 (   8) (       8) $1f40 (   6) (       8) $1f50 (   1) (UNIC_NCT)
    $1f51 (   1) (       8) $1f52 (   1) (UNIC_NCT) $1f53 (   1) (       8) $1f54 (   1) (UNIC_NCT)
    $1f55 (   1) (       8) $1f56 (   1) (UNIC_NCT) $1f57 (   1) (       8) $1f60 (   8) (       8)
    $1f70 (   2) (      74) $1f72 (   4) (      86) $1f76 (   2) (     100) $1f78 (   2) (     128)
```

```
    $1f7a (   2) (     112) $1f7c (   2) (     126) $1f80 (   8) (       8) $1f90 (   8) (       8)
    $1fa0 (   8) (       8) $1fb0 (   2) (       8) $1fb2 (   1) (UNIC_NCT) $1fb3 (   1) (       9)
    $1fb4 (  -3) (UNIC_NCT) $1fb7 (   1) (UNIC_NCT) $1fbe (   1) (   -7205) $1fc2 (   1) (UNIC_NCT)
    $1fc3 (   1) (       9) $1fc4 (  -3) (UNIC_NCT) $1fc7 (   1) (UNIC_NCT) $1fd0 (   2) (       8)
    $1fd2 (   2) (UNIC_NCT) $1fd6 (   2) (UNIC_NCT) $1fe0 (   2) (       8) $1fe2 (   3) (UNIC_NCT)
    $1fe5 (   1) (       7) $1fe6 (   2) (UNIC_NCT) $1ff2 (   1) (UNIC_NCT) $1ff3 (   1) (       9)
    $1ff4 (  -3) (UNIC_NCT) $1ff7 (   1) (UNIC_NCT) $2071 (   1) (UNIC_NCT) $207f (   1) (UNIC_NCT)
    $210a (   1) (UNIC_NCT) $210e (   2) (UNIC_NCT) $2113 (   1) (UNIC_NCT) $212f (   1) (UNIC_NCT)
    $2134 (   1) (UNIC_NCT) $2139 (   1) (UNIC_NCT) $213d (   1) (UNIC_NCT) $2146 (   4) (UNIC_NCT)
    $fb00 (   7) (UNIC_NCT) $fb13 (   5) (UNIC_NCT) $ff41 (  26) (     -32) $0000
;
Array CharCasingChart1 -->
    $0041 (  26) (      32) $00c0 (  23) (      32) $00d8 (   7) (      32) $0100 ( -47) (       1)
    $0130 (   1) (    -199) $0132 (  -5) (       1) $0139 ( -15) (       1) $014a ( -45) (       1)
    $0178 (   1) (    -121) $0179 (  -5) (       1) $0181 (   1) (     210) $0182 (  -3) (       1)
    $0186 (   1) (     206) $0187 (   1) (       1) $0189 (   2) (     205) $018b (   1) (       1)
    $018e (   1) (      79) $018f (   1) (     202) $0190 (   1) (     203) $0191 (   1) (       1)
    $0193 (   1) (     205) $0194 (   1) (     207) $0196 (   1) (     211) $0197 (   1) (     209)
    $0198 (   1) (       1) $019c (   1) (     211) $019d (   1) (     213) $019f (   1) (     214)
    $01a0 (  -5) (       1) $01a6 (   1) (     218) $01a7 (   1) (       1) $01a9 (   1) (     218)
    $01ac (   1) (       1) $01ae (   1) (     218) $01af (   1) (       1) $01b1 (   2) (     217)
    $01b3 (  -3) (       1) $01b7 (   1) (     219) $01b8 (   1) (       1) $01bc (   1) (       1)
    $01c4 (   1) (       2) $01c7 (   1) (       2) $01ca (   1) (       2) $01cd ( -15) (       1)
    $01de ( -17) (       1) $01f1 (   1) (       2) $01f4 (   1) (       1) $01f6 (   1) (     -97)
    $01f7 (   1) (     -56) $01f8 ( -39) (       1) $0220 (   1) (    -130) $0222 ( -17) (       1)
    $0386 (   1) (      38) $0388 (   3) (      37) $038c (   1) (      64) $038e (   2) (      63)
    $0391 (  17) (      32) $03a3 (   9) (      32) $03d2 (   3) (UNIC_NCT) $03d8 ( -23) (       1)
    $03f4 (   1) (     -60) $03f7 (   1) (       1) $03f9 (   1) (      -7) $03fa (   1) (       1)
    $0400 (  16) (      80) $0410 (  32) (      32) $0460 ( -33) (       1) $048a ( -53) (       1)
    $04c0 (   1) (UNIC_NCT) $04c1 ( -13) (       1) $04d0 ( -37) (       1) $04f8 (   1) (       1)
    $0500 ( -15) (       1) $0531 (  38) (      48) $10a0 (  38) (UNIC_NCT) $1e00 (-149) (       1)
    $1ea0 ( -89) (       1) $1f08 (   8) (      -8) $1f18 (   6) (      -8) $1f28 (   8) (      -8)
    $1f38 (   8) (      -8) $1f48 (   6) (      -8) $1f59 (  -7) (      -8) $1f68 (   8) (      -8)
    $1fb8 (   2) (      -8) $1fba (   2) (     -74) $1fc8 (   4) (     -86) $1fd8 (   2) (      -8)
    $1fda (   2) (    -100) $1fe8 (   2) (      -8) $1fea (   2) (    -112) $1fec (   1) (      -7)
    $1ff8 (   2) (    -128) $1ffa (   2) (    -126) $2102 (   1) (UNIC_NCT) $2107 (   1) (UNIC_NCT)
    $210b (   3) (UNIC_NCT) $2110 (   3) (UNIC_NCT) $2115 (   1) (UNIC_NCT) $2119 (   5) (UNIC_NCT)
    $2124 (   1) (UNIC_NCT) $2126 (   1) (   -7517) $2128 (   1) (UNIC_NCT) $212a (   1) (   -8383)
    $212b (   1) (   -8262) $212c (   2) (UNIC_NCT) $2130 (   2) (UNIC_NCT) $2133 (   1) (UNIC_NCT)
    $213e (   2) (UNIC_NCT) $2145 (   1) (UNIC_NCT) $ff21 (  26) (      32) $0000
;
#ENDIF; ! Large_Unicode_Tables
```

StoredAction Template B/stact

Purpose
Code to support the stored action kind of value.

§**1. Head.** As ever: if there is no heap, there are no stored actions.

```
#IFDEF MEMORY_HEAP_SIZE; ! Will exist if any use is made of indexed texts
```

§**2. KOV Support.** See the "BlockValues.i6t" segment for the specification of the following routines.

```
[ STORED_ACTION_TY_Support task arg1 arg2 arg3;
    switch(task) {
        CREATE_KOVS:     return STORED_ACTION_TY_Create();
        CAST_KOVS:       rfalse;
        DESTROY_KOVS:    return STORED_ACTION_TY_Destroy(arg1);
        PRECOPY_KOVS:    rfalse;
        COPY_KOVS:       return STORED_ACTION_TY_Copy(arg1, arg2);
        COMPARE_KOVS:    return STORED_ACTION_TY_Compare(arg1, arg2);
        READ_FILE_KOVS:  rfalse;
        WRITE_FILE_KOVS: rfalse;
    }
];
```

§**3. Creation.** A stored action block has fixed size, so this is a single-block KOV: its data consists of six words, laid out as shown in the following routine. Note that it initialises to the default value for this KOV, an action in which the player waits.

An action which involves a topic – such as the one produced by the command LOOK UP JIM MCDIVITT IN ENCYCLOPAEDIA – cannot be tried without the text of that topic (JIM MCDIVITT) being available. That's no problem if the action is tried in the same turn in which it is generated, because the text will still be in the command buffer. But once we store actions up for future use it becomes an issue. So when we store an action involving a topic, we record the actual text typed at the time when it is stored, and this goes into array entry 5 of the block. Because that in turn is indexed text, and therefore a block value on the heap in its own right, we have to be a little more careful about destroying and copying stored actions than we otherwise would be.

Note that entries 1 and 2 are values whose kind depends on the action in entry 0: but they are never block values, because actions are not allowed to apply to block values. This simplifies matters considerably.

```
[ STORED_ACTION_TY_Create stora;
    stora = BlkAllocate(6*WORDSIZE, STORED_ACTION_TY, BLK_FLAG_WORD);
    BlkValueWrite(stora, 0, ##Wait); ! action
    BlkValueWrite(stora, 1, 0); ! noun
    BlkValueWrite(stora, 2, 0); ! second
    BlkValueWrite(stora, 3, player); ! actor
    BlkValueWrite(stora, 4, false); ! whether a request
    BlkValueWrite(stora, 5, 0); ! indexed text of command if necessary, 0 if not
    return stora;
];
```

§**4. Setting Up.** In practice it's convenient for NI to have a routine which creates a stored action with a given slate of action variables, rather than have to set them all one at a time, so the following is provided as a shorthand form. (For other kinds of value, we might do this by casting, but we can't regard a quintuplet of variables as a single value to cast from, so we do it this way instead.)

```
[ STORED_ACTION_TY_New a n s ac req  stora;
    if (stora == 0) stora = STORED_ACTION_TY_Create();
    BlkValueWrite(stora, 0, a);
    BlkValueWrite(stora, 1, n);
    BlkValueWrite(stora, 2, s);
    BlkValueWrite(stora, 3, ac);
    BlkValueWrite(stora, 4, req);
    BlkValueWrite(stora, 5, 0);
    return stora;
];
```

§**5. Destruction.** Entries 0 to 4 are forgettable non-block values: only the optional indexed text requires destruction.

```
[ STORED_ACTION_TY_Destroy stora toc;
    toc = BlkValueRead(stora, 5);
    if (toc) BlkFree(toc);
];
```

§**6. Copying.** The only entry needing attention is, again, entry 5: if this is non-zero in the source, then we need to create a new indexed text block to hold a duplicate copy of the text.

```
[ STORED_ACTION_TY_Copy storato storafrom tocfrom tocto;
    tocfrom = BlkValueRead(storafrom, 5);
    if (tocfrom == 0) return;
    tocto = INDEXED_TEXT_TY_Support(CREATE_KOVS);
    BlkValueCopy(tocto, tocfrom);
    BlkValueWrite(storato, 5, tocto);
];
```

§**7. Comparison.** There is no very convincing ordering on stored actions, but we need to devise a comparison which will exhaustively determine whether two actions are or are not different.

```
[ STORED_ACTION_TY_Compare storaleft storaright delta itleft itright;
    delta = BlkValueRead(storaleft, 0) - BlkValueRead(storaright, 0);
    if (delta) return delta;
    delta = BlkValueRead(storaleft, 1) - BlkValueRead(storaright, 1);
    if (delta) return delta;
    delta = BlkValueRead(storaleft, 2) - BlkValueRead(storaright, 2);
    if (delta) return delta;
    delta = BlkValueRead(storaleft, 3) - BlkValueRead(storaright, 3);
    if (delta) return delta;
    delta = BlkValueRead(storaleft, 4) - BlkValueRead(storaright, 4);
    if (delta) return delta;
    itleft = BlkValueRead(storaleft, 5);
    itright = BlkValueRead(storaright, 5);
    if ((itleft ~= 0) && (itright ~= 0))
        return INDEXED_TEXT_TY_Support(COMPARE_KOVS, itleft, itright);
    return itleft - itright;
];
[ STORED_ACTION_TY_Distinguish stora1 stora2;
    if (STORED_ACTION_TY_Compare(stora1, stora2) == 0) rfalse;
    rtrue;
];
```

§**8. Printing.** We share some code here with the routines originally written for the ACTIONS testing command. (The `DB` in `DB_Action` stands for "debugging".) When printing a topic, it prints the relevant words from the player's command: so if our stored action is one which contains an entry 5, then we have to temporarily adopt this as the player's command, and restore the old player's command once printing is done. To do this, we need to save the old player's command, and we do that by creating an indexed text for the duration.

```
[ STORED_ACTION_TY_Say stora text_of_command saved_command;
    if ((stora==0) || (BlkType(stora) ~= STORED_ACTION_TY)) return;
    text_of_command = BlkValueRead(stora, 5);
    if (text_of_command) {
        saved_command = INDEXED_TEXT_TY_Support(CREATE_KOVS);
        INDEXED_TEXT_TY_Support(CAST_KOVS, players_command, SNIPPET_TY, saved_command);
        SetPlayersCommand(text_of_command);
    }
    DB_Action(BlkValueRead(stora, 3), BlkValueRead(stora, 4), BlkValueRead(stora, 0),
        BlkValueRead(stora, 1), BlkValueRead(stora, 2), true);
    if (text_of_command) {
        SetPlayersCommand(saved_command);
        BlkFree(saved_command);
    }
];
```

§**9. Involvement.** That completes the compulsory services required for this KOV to function: from here on, the remaining routines provide definitions of stored action-related phrases in the Standard Rules.

An action "involves" an object if it appears as either the actor or the first or second noun.

```
[ STORED_ACTION_TY_Involves stora item at;
    at = FindAction(BlkValueRead(stora, 0));
    if (at) {
        if ((ActionData-->(at+AD_NOUN_KOV) == OBJECT_TY) &&
            (BlkValueRead(stora, 1) == item)) rtrue;
        if ((ActionData-->(at+AD_SECOND_KOV) == OBJECT_TY) &&
            (BlkValueRead(stora, 2) == item)) rtrue;
    }
    if (BlkValueRead(stora, 3) == item) rtrue;
    rfalse;
];
```

§**10. Nouns.** Extracting the noun or second noun from an action is a delicate business because simply returning the values in entries 1 and 2 would not be type-safe; it would fail to be an object if the stored action did not apply to objects. So the following returns `nothing` if requested to produce noun or second noun for such an action.

```
[ STORED_ACTION_TY_Part stora ind at ado;
    if (ind == 1 or 2) {
        if (ind == 1) ado = AD_NOUN_KOV; else ado = AD_SECOND_KOV;
        at = FindAction(BlkValueRead(stora, 0));
        if ((at) && (ActionData-->(at+ado) == OBJECT_TY)) return BlkValueRead(stora, ind);
        return nothing;
    }
    return BlkValueRead(stora, ind);
];
```

§**11. Pattern Matching.** In order to apply an action pattern such as "doing something with the kazoo" to a stored action, it needs to be the current action, because the code which compiles conditions like this looks at the `action`, `noun`, ..., variables. We don't want to do anything as disruptive as temporarily starting the stored action and then halting it again, so instead we simply "adopt" it, saving the slate of action variables and setting them from the stored action: almost immediately after – the moment the condition has been tested – we "unadopt" it again, restoring the stored values. Since the action pattern cannot itself refer to a stored action, the following code won't be nested, and we don't need to worry about stacking up saved copies of the action variables.

`SAT_Tmp-->0` stores the outcome of the condition, and is set in code compiled by NI.

```
Array SAT_Tmp-->6;
[ STORED_ACTION_TY_Adopt stora;
    SAT_Tmp-->1 = action;
    SAT_Tmp-->2 = noun;
    SAT_Tmp-->3 = second;
    SAT_Tmp-->4 = actor;
    SAT_Tmp-->5 = act_requester;
    action = BlkValueRead(stora, 0);
    noun = BlkValueRead(stora, 1);
    second = BlkValueRead(stora, 2);
    actor = BlkValueRead(stora, 3);
```

```
    if (BlkValueRead(stora, 4)) act_requester = player; else act_requester = nothing;
];
[ STORED_ACTION_TY_Unadopt;
    action = SAT_Tmp-->1;
    noun = SAT_Tmp-->2;
    second = SAT_Tmp-->3;
    actor = SAT_Tmp-->4;
    act_requester = SAT_Tmp-->5;
    return SAT_Tmp-->0;
];
```

§**12. Current Action.** Although we never cast other values to stored actions, because none of them really imply an action (not even an action-name, since that gives no help as to what the nouns might be), there is of course one action almost always present within a story file at run-time, even if it is not a single value as such: the action which is currently running. The following routine translates that into a stored action – thus allowing us to store it.

This is the place where we look to see if the action applies to a topic as either its noun or second noun, and if it does, we copy the player's command into an indexed text block-value in entry 5.

```
[ STORED_ACTION_TY_Current stora at text_of_command;
    if ((stora==0) || (BlkType(stora) ~= STORED_ACTION_TY)) return 0;
    BlkValueWrite(stora, 0, action);
    BlkValueWrite(stora, 1, noun);
    BlkValueWrite(stora, 2, second);
    BlkValueWrite(stora, 3, actor);
    if (act_requester) BlkValueWrite(stora, 4, true); else BlkValueWrite(stora, 4, false);
    at = FindAction(-1);
    if ((at) && ((ActionData-->(at+AD_NOUN_KOV) == UNDERSTANDING_TY) ||
             (ActionData-->(at+AD_SECOND_KOV) == UNDERSTANDING_TY))) {
        text_of_command = BlkValueRead(stora, 5);
        if (text_of_command == 0) {
            text_of_command = INDEXED_TEXT_TY_Support(CREATE_KOVS);
            BlkValueWrite(stora, 5, text_of_command);
        }
        INDEXED_TEXT_TY_Support(CAST_KOVS, players_command, SNIPPET_TY, text_of_command);
    } else BlkValueWrite(stora, 5, 0);
    return stora;
];
```

§**13. Trying.** Finally: having stored an action for perhaps many turns, we now let it happen, either silently or not.

```
[ STORED_ACTION_TY_Try stora ks  text_of_command saved_command;
    if ((stora==0) || (BlkType(stora) ~= STORED_ACTION_TY)) return;
    if (ks) { @push keep_silent; keep_silent=1; }
    text_of_command = BlkValueRead(stora, 5);
    if (text_of_command) {
        saved_command = INDEXED_TEXT_TY_Support(CREATE_KOVS);
        INDEXED_TEXT_TY_Support(CAST_KOVS, players_command, SNIPPET_TY, saved_command);
        SetPlayersCommand(text_of_command);
    }
    TryAction(BlkValueRead(stora, 4), BlkValueRead(stora, 3),
        BlkValueRead(stora, 0), BlkValueRead(stora, 1), BlkValueRead(stora, 2));
    if (text_of_command) {
        SetPlayersCommand(saved_command);
        BlkFree(saved_command);
    }
    if (ks) { @pull keep_silent; }
];
```

§**14. Stubs.** And the usual meaningless versions to ensure that function-names exist if there is no heap, and there are no stored actions anyway.

```
#IFNOT; ! IFDEF MEMORY_HEAP_SIZE
[ STORED_ACTION_TY_Say stora; ];
[ STORED_ACTION_TY_New a n s ac req stora; return false; ];
[ STORED_ACTION_TY_Support t a b c; rfalse; ];
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```

Lists Template B/listt

Purpose

Code to support the list of... kind of value constructor.

§**1. Head.** As ever: if there is no heap, there are no lists (in this sense).

```
#IFDEF MEMORY_HEAP_SIZE; ! Will exist if any use is made of heap
```

§**2. KOV Support.** See the "BlockValues.i6t" segment for the specification of the following routines.

```
[ LIST_OF_TY_Support task arg1 arg2 arg3;
    switch(task) {
        CREATE_KOVS:     arg3 = LIST_OF_TY_Create(arg2);
                         if (arg1) LIST_OF_TY_CopyRawArray(arg3, arg1, 2);
                         return arg3;
        CAST_KOVS:       rfalse;
        DESTROY_KOVS:    return LIST_OF_TY_Destroy(arg1);
        PRECOPY_KOVS:    return LIST_OF_TY_PreCopy(arg1, arg2);
        COPY_KOVS:       return LIST_OF_TY_Copy(arg1, arg2);
        COMPARE_KOVS:    return LIST_OF_TY_Compare(arg1, arg2);
        READ_FILE_KOVS:  rfalse;
        WRITE_FILE_KOVS: rfalse;
    }
];
```

§**3. Creation.** A list is a multiple-block value with word-sized entries: the first few entries of the block are used for details about the list; the items in the the list then follow. Thus, to convert an item index to an array entry index, add `LIST_ITEM_BASE`.

Lists are by default created empty but in a block-value with enough capacity to hold 25 items, this being what's left in a 32-word block once all overheads are taken care of: 4 words are consumed by the header, then 3 more by the list metadata entries below.

```
Constant LIST_ITEM_KOV_F = 0; ! Entry 0: base KOV stored within
Constant LIST_LENGTH_F = 1; ! Entry 1: length, i.e., number of items
Constant LIST_ITEM_COMPARE_F = 2; ! Entry 2: comparison function for items
Constant LIST_ITEM_BASE = 3; ! List items begin at this entry

[ LIST_OF_TY_Create skov list;
    list = BlkAllocate(28*WORDSIZE, LIST_OF_TY, BLK_FLAG_MULTIPLE + BLK_FLAG_WORD);
    BlkValueWrite(list, LIST_ITEM_KOV_F, skov);
    BlkValueWrite(list, LIST_LENGTH_F, 0);
    BlkValueWrite(list, LIST_ITEM_COMPARE_F, KOVComparisonFunction(skov));
    return list;
];
```

§**4. Setting Up.** NI needs to compile code which will create constant lists such as `{1, 4, 9}` at run-time: the following routine is convenient for that. A "raw array" in this routine's sense is an array `raw` such that `raw-->2` contains the number of items, `raw-->1` the kind of value, and `raw-->3` onwards are the items themselves.

```
[ LIST_OF_TY_CopyRawArray list arr rea len i ex bk v w;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    ex = BlkValueExtent(list);
    len = arr-->2;
    if ((len+LIST_ITEM_BASE > ex) &&
        (BlkValueSetExtent(list, len+LIST_ITEM_BASE) == false)) return 0;
    BlkValueWrite(list, LIST_LENGTH_F, len);
    if (rea == 2) bk = BlkValueRead(list, LIST_ITEM_KOV_F);
    else {
        bk = arr-->1;
        BlkValueWrite(list, LIST_ITEM_KOV_F, bk);
    }
    BlkValueWrite(list, LIST_ITEM_COMPARE_F, KOVComparisonFunction(arr-->1));
    for (i=0:i<len:i++) {
        v = arr-->(i+3);
        if (bk == LIST_OF_TY) {
            w = LIST_OF_TY_Create(v-->1);
            LIST_OF_TY_CopyRawArray(w, v);
            BlkValueWrite(list, i+LIST_ITEM_BASE, w);
        } else {
            if (KOVIsBlockValue(bk)) v = BlkValueCreate(bk, v);
            BlkValueWrite(list, i+LIST_ITEM_BASE, v);
        }
    }
    #ifdef SHOW_ALLOCATIONS;
    print "Copied raw array to list: "; LIST_OF_TY_Say(list, 1); print "^";
    #endif;
    return list;
];
```

§**5. Destruction.** If the list items are themselves block-values, they must all be freed before the list itself can be freed.

```
[ LIST_OF_TY_Destroy list no_items i;
    if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F))) {
        no_items = BlkValueRead(list, LIST_LENGTH_F);
        for (i=0; i<no_items; i++) BlkFree(BlkValueRead(list, i+LIST_ITEM_BASE));
    }
    return list;
];
```

§**6. Copying.** Again, if the list contains block-values then they must be duplicated rather than bitwise copied as pointers.

Note that we use the pre-copy stage to remember the kind of value stored in the list. Type-checking ordinarily means that one list can only be copied into another if they have the same kind of contents: but there is one exception, which is when the list being copied is the empty list.

```
Global precopied_list_kov;

[ LIST_OF_TY_PreCopy lto lfrom list no_items i nv bk;
    precopied_list_kov = BlkValueRead(lto, LIST_ITEM_KOV_F);
];

[ LIST_OF_TY_Copy lto lfrom list no_items i nv bk;
    no_items = BlkValueRead(lfrom, LIST_LENGTH_F);
    bk = BlkValueRead(lfrom, LIST_ITEM_KOV_F);
    BlkValueWrite(lto, LIST_ITEM_KOV_F, precopied_list_kov);
    if ((precopied_list_kov == INDEXED_TEXT_TY) && (bk == TEXT_TY)) {
        for (i=0; i<no_items; i++) {
            nv = BlkValueCreate(INDEXED_TEXT_TY);
            INDEXED_TEXT_TY_Cast(BlkValueRead(lfrom, i+LIST_ITEM_BASE), TEXT_TY, nv);
            BlkValueWrite(lto, i+LIST_ITEM_BASE, nv);
        }
    } else {
        if (KOVIsBlockValue(BlkValueRead(lfrom, LIST_ITEM_KOV_F))) {
            for (i=0; i<no_items; i++) {
                nv = BlkValueCreate(bk);
                BlkValueCopy(nv, BlkValueRead(lfrom, i+LIST_ITEM_BASE));
                BlkValueWrite(lto, i+LIST_ITEM_BASE, nv);
            }
        }
    }
];
```

§**7. Comparison.** Lists of a given kind of value are always grouped together, in this comparison: but the effect of that is unlikely to be noticed since NI's type-checker will probably prevent comparisons of lists of differing items in any case. The next criterion is length: a short list precedes a long one. Beyond that, we use the list's own preferred comparison function to judge the items in turn, stopping as soon as a pair of corresponding items differs: thus we sort lists of equal size in lexicographic order.

Since the comparison function depends only on the KOV, it may seem wasteful of a word of memory to store it in the list, given that we are already storing the KOV in any case. But we do this because comparisons have to be fast: we don't want to incur the overhead of translating KOV to comparison function.

```
[ LIST_OF_TY_Compare listleft listright delta no_items i cf;
    delta = BlkValueRead(listleft, LIST_ITEM_KOV_F) - BlkValueRead(listright, LIST_ITEM_KOV_F);
    if (delta) return delta;
    delta = BlkValueRead(listleft, LIST_LENGTH_F) - BlkValueRead(listright, LIST_LENGTH_F);
    if (delta) return delta;
    no_items = BlkValueRead(listleft, LIST_LENGTH_F);
    cf = BlkValueRead(listleft, LIST_ITEM_COMPARE_F);
    if (cf == 0 or UnsignedCompare) {
        for (i=0; i<no_items; i++) {
            delta = BlkValueRead(listleft, i+LIST_ITEM_BASE) -
                BlkValueRead(listright, i+LIST_ITEM_BASE);
            if (delta) return delta;
```

```
        }
    } else {
        for (i=0; i<no_items; i++) {
            delta = cf(BlkValueRead(listleft, i+LIST_ITEM_BASE),
                BlkValueRead(listright, i+LIST_ITEM_BASE));
            if (delta) return delta;
        }
    }
    return 0;
];
[ LIST_OF_TY_Distinguish txb1 txb2;
    if (LIST_OF_TY_Compare(txb1, txb2) == 0) rfalse;
    rtrue;
];
```

§**8. Printing.** Unusually, this function can print the value in one of several formats: 0 for a comma-separated list; 1 for a braced, set-notation list; 2 for a comma-separated list with definite articles, which only makes sense if the list contains objects; 3 for a comma-separated list with indefinite articles. Note that a list in this sense is *not* printed using the "ListWriter.i6t" code for elaborate lists of objects, and it doesn't use the "listing contents of..." activity in any circumstances.

```
[ LIST_OF_TY_Say list format no_items v i bk;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    bk = BlkValueRead(list, LIST_ITEM_KOV_F);
    ! print "kov=", bk, ":";
    if (format == 1) print "{";
    for (i=0:i<no_items:i++) {
        v = BlkValueRead(list, i+LIST_ITEM_BASE);
        switch (format) {
            2: print (the) v;
            3: print (a) v;
            default:
                if (bk == LIST_OF_TY) LIST_OF_TY_Say(v, 1);
                else if ((bk == TEXT_TY or INDEXED_TEXT_TY) && (format == 1)) {
                    print "~"; PrintKindValuePair(bk, v); print "~";
                }
                else PrintKindValuePair(bk, v);
        }
        if (i<no_items-2) print ", ";
        if (i==no_items-2) {
            if (format == 1) print ", "; else {
                #ifdef SERIAL_COMMA; print ","; #endif;
                print (string) LISTAND2__TX;
            }
        }
    }
    if (format == 1) print "}";
];
```

§**9. List From Description.** That completes the compulsory services required for this KOV to function: from here on, the remaining routines provide definitions of stored action-related phrases in the Standard Rules.

Given a description `D` which applies to some objects and not others – say, "lighted rooms adjacent to the Transport Hub" – we can cast this into a list of all objects satisfying `D` with the following routine. Slightly wastefully of time, we have to iterate through the objects twice in order first to work out the length of list we will need, and then to transcribe them.

```
[ LIST_OF_TY_Desc list desc kov obj no_items ex len i;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    ex = BlkValueExtent(list);
    len = desc(-3);
! for (len=0, obj=desc(-2, nothing, len): obj: len++, obj=desc(-2, obj, len)) ;
! len++;
    if (len+LIST_ITEM_BASE > ex) {
        if (BlkValueSetExtent(list, len+LIST_ITEM_BASE) == false)
            return 0;
    }
    if (kov) BlkValueWrite(list, LIST_ITEM_KOV_F, kov);
    else BlkValueWrite(list, LIST_ITEM_KOV_F, OBJECT_TY);
    BlkValueWrite(list, LIST_LENGTH_F, len);
    obj = 0;
    for (i=0: i<len: i++) {
        obj = desc(-2, obj, i);
        ! print "i = ", i, " and obj = ", obj, "^";
        BlkValueWrite(list, i+LIST_ITEM_BASE, obj);
    }
    return list;
];
```

§**10. Find Item.** We test whether a list `list` includes a value equal to `v` or not. Equality here is in the sense of the list's comparison function: thus for indexed texts or other lists, say, deep comparisons rather than simple pointer comparisons are performed. In other words, one copy of "Alert" is equal to another.

```
[ LIST_OF_TY_FindItem list v i no_items cf;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) rfalse;
    cf = BlkValueRead(list, LIST_ITEM_COMPARE_F);
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (cf == 0 or UnsignedCompare) {
        for (i=0; i<no_items; i++)
            if (v == BlkValueRead(list, i+LIST_ITEM_BASE)) rtrue;
    } else {
        for (i=0; i<no_items; i++)
            if (cf(v, BlkValueRead(list, i+LIST_ITEM_BASE)) == 0) rtrue;
    }
    rfalse;
];
```

§**11. Insert Item.** The following routine inserts an item into the list. If this would break the size of the current block-value, then we extend by at least enough room to hold at least another 16 entries.

In the call `LIST_OF_TY_InsertItem(list, v, posnflag, posn, nodups)`, only the first two arguments are compulsory.

(a) If `nodups` is set, and an item equal to `v` is already present in the list, we return and do nothing. (`nodups` means "no duplicates".)
(b) Otherwise, if `posnflag` is `false`, we append a new entry `v` at the back of the given `list`.
(c) Otherwise, when `posnflag` is `true`, `posn` indicates the insertion position, from 1 (before the current first item) to $N + 1$ (after the last), where N is the number of items in the list at present.

```
[ LIST_OF_TY_InsertItem list v posnflag posn nodups i no_items ex nv;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    if (nodups && (LIST_OF_TY_FindItem(list, v))) return list;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if ((posnflag) && ((posn<1) || (posn > no_items+1))) {
        print "*** Couldn't add at entry ", posn, " in the list ";
        LIST_OF_TY_Say(list, true);
        print ", which has entries in the range 1 to ", no_items, " ***^";
        RunTimeProblem(RTP_LISTRANGEERROR);
        rfalse;
    }
    ex = BlkValueExtent(list);
    if (no_items+LIST_ITEM_BASE+1 > ex) {
        if (BlkValueSetExtent(list, ex+16) == false) return 0;
    }
    if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F))) {
        nv = BlkValueCreate(BlkValueRead(list, LIST_ITEM_KOV_F));
        BlkValueCopy(nv, v);
        v = nv;
    }
    if (posnflag) {
        posn--;
        for (i=no_items:i>posn:i--) {
            BlkValueWrite(list, i+LIST_ITEM_BASE,
                BlkValueRead(list, i-1+LIST_ITEM_BASE));
        }
        BlkValueWrite(list, posn+LIST_ITEM_BASE, v);
    } else {
        BlkValueWrite(list, no_items+LIST_ITEM_BASE, v);
    }
    BlkValueWrite(list, LIST_LENGTH_F, no_items+1);
    return list;
];
```

§**12. Append List.** Instead of adjoining a single value, we adjoin an entire second list, which must be of a compatible kind of value (something which NI's type-checking machinery polices for us). Except that we have a list `more` rather than a value `v` to insert, the specification is the same as for `LIST_OF_TY_InsertItem`.

```
[ LIST_OF_TY_AppendList list more posnflag posn nodups v i j no_items msize ex nv;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    if ((more==0) || (BlkType(more) ~= LIST_OF_TY)) return list;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if ((posnflag) && ((posn<1) || (posn > no_items+1))) {
        print "*** Couldn't add at entry ", posn, " in the list ";
        LIST_OF_TY_Say(list, true);
        print ", which has entries in the range 1 to ", no_items, " ***^";
        RunTimeProblem(RTP_LISTRANGEERROR);
        rfalse;
    }
    msize = BlkValueRead(more, LIST_LENGTH_F);
    ex = BlkValueExtent(list);
    if (no_items+msize+LIST_ITEM_BASE > ex) {
        if (BlkValueSetExtent(list, no_items+msize+LIST_ITEM_BASE+8) == false)
            return 0;
    }
    if (posnflag) {
        posn--;
        for (i=no_items+msize:i>=posn+msize:i--) {
            BlkValueWrite(list, i+LIST_ITEM_BASE,
                BlkValueRead(list, i-msize+LIST_ITEM_BASE));
        }
        BlkValueWrite(list, posn, v);
        for (j=0: j<msize: j++) {
            v = BlkValueRead(more, j+LIST_ITEM_BASE);
            if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F))) {
                nv = BlkValueCreate(BlkValueRead(list, LIST_ITEM_KOV_F));
                BlkValueCopy(nv, v);
                v = nv;
            }
            BlkValueWrite(list, posn+j+LIST_ITEM_BASE, v);
        }
    } else {
        for (i=0, j=0: i<msize: i++) {
            v = BlkValueRead(more, i+LIST_ITEM_BASE);
            if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F))) {
                nv = BlkValueCreate(BlkValueRead(list, LIST_ITEM_KOV_F));
                BlkValueCopy(nv, v);
                v = nv;
            }
            if ((nodups == 0) || (LIST_OF_TY_FindItem(list, v) == false)) {
                BlkValueWrite(list, no_items+j+LIST_ITEM_BASE, v);
                j++;
            }
        }
    }
    BlkValueWrite(list, LIST_LENGTH_F, no_items+j);
    return list;
```

```
];
```

§**13. Remove Value.** We remove every instance of the value `v` from the given `list`. If the optional flag `forgive` is set, then we make no complaint if no value of `v` was present in the first place: otherwise, we issue a run-time problem.

Note that if the list contains block-values then the value must be properly destroyed with `BlkFree` before being overwritten as the items shuffle down.

```
[ LIST_OF_TY_RemoveValue list v forgive i j no_items odsize f cf delendum;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) rfalse;
    cf = BlkValueRead(list, LIST_ITEM_COMPARE_F);
    no_items = BlkValueRead(list, LIST_LENGTH_F); odsize = no_items;
    for (i=0; i<no_items; i++) {
        delendum = BlkValueRead(list, i+LIST_ITEM_BASE);
        if (cf == 0 or UnsignedCompare)
            f = (v == delendum);
        else
            f = (cf(v, delendum) == 0);
        if (f) {
            if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F)))
                BlkFree(delendum);
            for (j=i+1; j<no_items; j++)
                BlkValueWrite(list, j-1+LIST_ITEM_BASE,
                    BlkValueRead(list, j+LIST_ITEM_BASE));
            no_items--; i--;
            BlkValueWrite(list, LIST_LENGTH_F, no_items);
        }
    }
    if (odsize ~= no_items) rfalse;
    if (forgive) rfalse;
    print "*** Couldn't remove: the value ";
    PrintKindValuePair(BlkValueRead(list, LIST_ITEM_KOV_F), v);
    print " was not present in the list ";
    LIST_OF_TY_Say(list, true);
    print " ***^";
    RunTimeProblem(RTP_LISTRANGEERROR);
];
```

§**14. Remove Item Range.** We excise items `from` to `to` from the given `list`, which numbers its items upwards from 1. If the optional flag `forgive` is set, then we truncate a range overspilling the actual list, and we make no complaint if it turns out that there is then nothing to be done: otherwise, in either event, we issue a run-time problem.

Once again, we destroy any block-values whose pointers will be overwritten as the list shuffles down to fill the void.

```
[ LIST_OF_TY_RemoveItemRange list from to forgive i d no_items;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) rfalse;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if ((from > to) || (from <= 0) || (to > no_items)) {
        if (forgive) {
            if (from <= 0) from = 1;
            if (to >= no_items) to = no_items;
            if (from > to) return list;
        } else {
            print "*** Couldn't remove entries ", from, " to ", to, " from the list ";
            LIST_OF_TY_Say(list, true);
            print ", which has entries in the range 1 to ", no_items, " ***^";
            RunTimeProblem(RTP_LISTRANGEERROR);
            rfalse;
        }
    }
    to--; from--;
    d = to-from+1;
    if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F)))
        for (i=0; i<d; i++)
            BlkFree(BlkValueRead(list, from+i+LIST_ITEM_BASE));
    for (i=from: i<no_items-d: i++)
        BlkValueWrite(list, i+LIST_ITEM_BASE,
            BlkValueRead(list, i+d+LIST_ITEM_BASE));
    BlkValueWrite(list, LIST_LENGTH_F, no_items-d);
    return list;
];
```

§**15. Remove List.** We excise all values from the removal list `rlist`, wherever they occur in `list`. Inevitably, given that we haven't sorted these lists and can spare neither time nor storage to do so, this is an expensive process with a running time proportional to the product of the two list sizes: we accept that as an overhead because in practice the `rlist` is almost always small in real-world use.

If the initial lists were disjoint, so that no removals occur, we always forgive the user: the request was not necessarily a foolish one, it only happened in this situation to be unhelpful.

```
[ LIST_OF_TY_Remove_List list rlist i j k v w no_items odsize rsize cf f;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) rfalse;
    no_items = BlkValueRead(list, LIST_LENGTH_F); odsize = no_items;
    rsize = BlkValueRead(rlist, LIST_LENGTH_F);
    cf = BlkValueRead(list, LIST_ITEM_COMPARE_F);
    for (i=0: i<no_items: i++) {
        v = BlkValueRead(list, i+LIST_ITEM_BASE);
        for (k=0: k<rsize: k++) {
            w = BlkValueRead(rlist, k+LIST_ITEM_BASE);
            if (cf == 0 or UnsignedCompare)
```

```
                f = (v == w);
            else
                f = (cf(v, w) == 0);
            if (f) {
                if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F)))
                    BlkFree(v);
                for (j=i+1: j<no_items: j++)
                    BlkValueWrite(list, j+LIST_ITEM_BASE-1,
                        BlkValueRead(list, j+LIST_ITEM_BASE));
                no_items--; i--;
                BlkValueWrite(list, LIST_LENGTH_F, no_items);
                break;
            }
        }
    }
    rfalse;
];
```

§16. Get Length.

```
[ LIST_OF_TY_GetLength list;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    return BlkValueRead(list, LIST_LENGTH_F);
];
[ LIST_OF_TY_Empty list;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) rfalse;
    if (BlkValueRead(list, LIST_LENGTH_F) == 0) rtrue;
    rfalse;
];
```

§17. Set Length. This is rather harder: it might lengthen the list, in which case we have to pad out with the default value for the kind of value stored – padding a list of numbers with 0s, a list of texts with copies of the empty text, and so on – creating such block-values as might be needed; or else it might shorten the list, in which case we must cut items, destroying them properly if they were block-values.

`LIST_OF_TY_SetLength(list, newsize, this_way_only, truncation_end)` alters the length of the given `list` to `newsize`. If `this_way_only` is 1, the list is only allowed to grow, and nothing happens if we have asked to shrink it; if it is -1, the list is only allowed to shrink; if it is 0, the list is allowed either to grow or shrink. In the event that the list does have to shrink, entries must be removed, and we remove from the end if `truncation_end` is 1, or from the start if it is -1.

```
[ LIST_OF_TY_SetLength list newsize this_way_only truncation_end no_items ex i dv;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    if (newsize < 0) "*** Cannot resize a list to negative length ***";
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (no_items < newsize) {
        if (this_way_only == -1) return list;
        ex = BlkValueExtent(list);
        if (newsize+LIST_ITEM_BASE > ex) {
            if (BlkValueSetExtent(list, newsize+LIST_ITEM_BASE) == false)
                return 0;
        }
        dv = DefaultValueOfKOV(BlkValueRead(list, LIST_ITEM_KOV_F));
```

```
        for (i=no_items: i<newsize: i++)
            BlkValueWrite(list, LIST_ITEM_BASE+i, dv);
        BlkValueWrite(list, LIST_LENGTH_F, newsize);
    }
    if (no_items > newsize) {
        if (this_way_only == 1) return list;
        if (truncation_end == -1) {
            if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F)))
                for (i=0: i<no_items-newsize: i++)
                    BlkFree(BlkValueRead(list, LIST_ITEM_BASE+i));
            for (i=0: i<newsize: i++)
                BlkValueWrite(list, LIST_ITEM_BASE+i,
                    BlkValueRead(list, LIST_ITEM_BASE+no_items-newsize+i));
        } else {
            if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F)))
                for (i=newsize: i<no_items: i++)
                    BlkFree(BlkValueRead(list, LIST_ITEM_BASE+i));
        }
        BlkValueWrite(list, LIST_LENGTH_F, newsize);
    }
    return list;
];
```

§18. Get Item.

```
[ LIST_OF_TY_GetItem list i forgive no_items;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if ((i<=0) || (i>no_items)) {
        if (forgive) return false;
        print "*** Couldn't read from list entry ", i, ": the list ";
        LIST_OF_TY_Say(list, true);
        switch (no_items) {
            0: print " is empty ***^";
            1: print " has only one entry, numbered 1 ***^";
            default: print " has entries numbered from 1 to ", no_items, " ***^";
        }
        RunTimeProblem(RTP_LISTRANGEERROR);
        if (no_items >= 1) i = 1;
        else return false;
    }
    return BlkValueRead(list, LIST_ITEM_BASE+i-1);
];
```

§**19. Write Item.** The slightly odd name for this function comes about because our usual way to convert an rvalue such as `LIST_OF_TY_GetItem(L, 4)` is to prefix `Write`, so that it becomes `WriteLIST_OF_TY_GetItem(L, 4)`.

```
[ WriteLIST_OF_TY_GetItem list i val no_items;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if ((i<=0) || (i>no_items)) {
        print "*** Couldn't write to list entry ", i, ": the list ";
        LIST_OF_TY_Say(list, true);
        switch (no_items) {
            0: print " is empty ***^";
            1: print " has only one entry, numbered 1 ***^";
            default: print " has entries numbered from 1 to ", no_items, " ***^";
        }
        return RunTimeProblem(RTP_LISTRANGEERROR);
    }
    BlkValueWrite(list, LIST_ITEM_BASE+i-1, val);
];
```

§**20. Put Item.** Higher-level code should not use `Write_LIST_OF_TY_GetItem`, because it does not properly keep track of block-value copying: the following should be used instead.

```
[ LIST_OF_TY_PutItem list i v  no_items nv;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return false;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (KOVIsBlockValue(BlkValueRead(list, LIST_ITEM_KOV_F))) {
        nv = BlkValueCreate(BlkValueRead(list, LIST_ITEM_KOV_F));
        BlkValueCopy(nv, v);
        v = nv;
    }
    if ((i<=0) || (i>no_items)) return false;
    BlkValueWrite(list, LIST_ITEM_BASE+i-1, v);
];
```

§**21. Multiple Object List.** The parser uses one data structure which is really a list: but which can't be represented as such because the heap might not exist. This is the multiple object list, which is used to handle commands like TAKE ALL by firing off a sequence of actions with one of the objects taken from entries in turn of the list. The following converts it to a list structure.

```
[ LIST_OF_TY_Mol list len i;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    len = multiple_object-->0;
    LIST_OF_TY_SetLength(list, len);
    for (i=1: i<=len: i++)
        LIST_OF_TY_PutItem(list, i, multiple_object-->i);
    return list;
];
[ LIST_OF_TY_Set_Mol list len i;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    len = BlkValueRead(list, LIST_LENGTH_F);
    if (len > 63) len = 63;
```

```
    multiple_object-->0 = len;
    for (i=1: i<=len: i++)
        multiple_object-->i = BlkValueRead(list, LIST_ITEM_BASE+i-1);
];
```

§**22. Reversing.** Reversing a list is, happily, a very efficient operation when the list contains block-values: because the pointers are rearranged but none is duplicated or destroyed, we can for once ignore the fact that they are pointers to block-values and simply move them around like any other data.

```
[ LIST_OF_TY_Reverse list no_items i v;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (no_items < 2) return list;
    for (i=0;i*2<no_items;i++) {
        v = BlkValueRead(list, LIST_ITEM_BASE+i);
        BlkValueWrite(list, LIST_ITEM_BASE+i,
            BlkValueRead(list, LIST_ITEM_BASE+no_items-1-i));
        BlkValueWrite(list, LIST_ITEM_BASE+no_items-1-i, v);
    }
    return list;
];
```

§**23. Rotation.** The same is true of rotation. Here, "forwards" rotation means towards the end of the list, "backwards" means towards the start.

```
[ LIST_OF_TY_Rotate list backwards  no_items i v;
    if ((list==0) || (BlkType(list) ~= LIST_OF_TY)) return 0;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (no_items < 2) return list;
    if (backwards) {
        v = BlkValueRead(list, LIST_ITEM_BASE);
        for (i=0:i<no_items-1:i++)
            BlkValueWrite(list, LIST_ITEM_BASE+i,
                BlkValueRead(list, LIST_ITEM_BASE+i+1));
        BlkValueWrite(list, no_items-1+LIST_ITEM_BASE, v);
    } else {
        v = BlkValueRead(list, no_items-1+LIST_ITEM_BASE);
        for (i=no_items-1:i>0:i--)
            BlkValueWrite(list, LIST_ITEM_BASE+i,
                BlkValueRead(list, LIST_ITEM_BASE+i-1));
        BlkValueWrite(list, LIST_ITEM_BASE, v);
    }
    return list;
];
```

§**24. Sorting.** And the same, again, is true of sorting: but we do have to take note of block values when it comes to performing comparisons, because we can only compare items in the list by looking at their contents, not the pointers to their contents.

`LIST_OF_TY_Sort(list, dir, prop)` sorts the given `list` in ascending order if `dir` is 1, in descending order if `dir` is −1, or in random order if `dir` is 2. The comparison used is the one for the kind of value stored in the list, unless the optional argument `prop` is supplied, in which case we sort based not on the item values but on their values for the property `prop`. (This only makes sense if the list contains objects.)

```
[ LIST_OF_TY_Sort list dir prop  i j no_items v;
    no_items = BlkValueRead(list, LIST_LENGTH_F);
    if (dir == 2) {
        if (no_items < 2) return;
        for (i=1:i<no_items:i++) {
            j = random(i+1) - 1;
            v = BlkValueRead(list, LIST_ITEM_BASE+i);
            BlkValueWrite(list, LIST_ITEM_BASE+i, BlkValueRead(list, LIST_ITEM_BASE+j));
            BlkValueWrite(list, LIST_ITEM_BASE+j, v);
        }
        return;
    }
    SetSortDomain(ListSwapEntries, ListCompareEntries);
    SortArray(list, prop, dir, no_items, false, 0);
];
[ ListSwapEntries list i j v;
    if (i==j) return;
    v = BlkValueRead(list, LIST_ITEM_BASE+i-1);
    BlkValueWrite(list, LIST_ITEM_BASE+i-1, BlkValueRead(list, LIST_ITEM_BASE+j-1));
    BlkValueWrite(list, LIST_ITEM_BASE+j-1, v);
];
[ ListCompareEntries list col i j d cf;
    if (i==j) return 0;
    if (I7S_Col) {
        i = BlkValueRead(list, LIST_ITEM_BASE+i-1);
        j = BlkValueRead(list, LIST_ITEM_BASE+j-1);
        if (i provides I7S_Col) i=i.I7S_Col; else i=0;
        if (j provides I7S_Col) j=j.I7S_Col; else j=0;
        return i - j;
    }
    cf = BlkValueRead(list, LIST_ITEM_COMPARE_F);
    if (cf == 0)
        return BlkValueRead(list, LIST_ITEM_BASE+i-1) -
            BlkValueRead(list, LIST_ITEM_BASE+j-1);
    else
        return cf(BlkValueRead(list, LIST_ITEM_BASE+i-1),
            BlkValueRead(list, LIST_ITEM_BASE+j-1));
];
#IFNOT; ! IFDEF MEMORY_HEAP_SIZE
[ LIST_OF_TY_Support t a b c; rfalse; ];
[ LIST_OF_TY_Say list; ];
[ LIST_OF_TY_FindItem list v; rfalse; ];
[ LIST_OF_TY_Empty list; rfalse; ];
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```

BlockValues Template B/blkvt

Purpose

Routines for copying, comparing, creating and destroying block values, and for reading and writing them as if they were arrays.

---

§**1. Head.** As with Flex.i6t, none of this material is worth compiling unless there is a heap on which the block values can be stored.

```
#IFDEF MEMORY_HEAP_SIZE;
```

§**2. Data Section.** In this segment, we provide a layer which comes between "Flex.i6t" and its eventual users. These are routines which handle the data section for the allocated blocks, and which isolate the end user from having to know how the blocks are divided. As far as the user is concerned, each block value has an "extent" E, and consists of an array of values indexed from 0 to $E-1$. These are bytes unless `BLK_FLAG_WORD` is set in the header, in which case they are words; unless `BLK_FLAG_16_BIT` is set, in which case they are two-byte quantities. (For instance, indexed text is stored using bytes on the Z-machine but 16-bit quantities on Glulx, since the ZSCII character set can fit into the range 0 to 255 but Unicode cannot.)

`BlkValueExtent(B)` returns the value E for the block `B`.

`BlkValueSetExtent(B, N)` resizes the block `B` so that value E is at least `N`. If this is a reduction, entries are lost from the end, i.e., from the highest-indexed entries. If it is an expansion, entries are added at the end and the existing entries are preserved.

`BlkValueRead(B, I)` returns the value of array entry `I` in `B`.

`BlkValueWrite(B, I, V)` puts the value `V` into array entry `I` in `B`.

```
[ BlkValueExtent block  tsize flags;
    if (block == 0) return 0;
    flags = block->BLK_HEADER_FLAGS;
    if (flags & BLK_FLAG_MULTIPLE == 0)
        tsize = BlkSize(block) - BLK_DATA_OFFSET;
    else
        for (:block~=NULL:block=block-->BLK_NEXT)
            tsize = tsize + BlkSize(block) - BLK_DATA_MULTI_OFFSET;
    if (flags & BLK_FLAG_16_BIT) return tsize/2;
    if (flags & BLK_FLAG_WORD) return tsize/WORDSIZE;
    return tsize;
];
[ BlkValueSetExtent block tsize flags wsize;
    if (block == 0) return 0;
    flags = block->BLK_HEADER_FLAGS; wsize = 1;
    if (flags & BLK_FLAG_WORD) wsize = WORDSIZE;
    if (flags & BLK_FLAG_16_BIT) wsize = 2;
    return BlkResize(block, (tsize)*wsize);
];
[ BlkValueRead block pos dsize hsize flags wsize ot op;
    if (block==0) rfalse;
```

```
    flags = block->BLK_HEADER_FLAGS; wsize = 1;
    if (flags & BLK_FLAG_WORD) wsize = WORDSIZE;
    if (flags & BLK_FLAG_16_BIT) wsize = 2;
    ot = block; op = pos;
    pos = pos*wsize;
    if (flags & BLK_FLAG_MULTIPLE) hsize = BLK_DATA_MULTI_OFFSET;
    else hsize = BLK_DATA_OFFSET;
    for (:block~=NULL:block=block-->BLK_NEXT) {
        dsize = BlkSize(block) - hsize;
        if ((pos >= 0) && (pos<dsize)) {
            block = block + hsize + pos;
            switch(wsize) {
                1: return block->0;
                2: #Iftrue (WORDSIZE == 2); return block-->0;
                    #ifnot; return (block->0)*256 + (block->1);
                    #endif;
                4: return block-->0;
            }
        }
        pos = pos - dsize;
    }
    "*** BlkValueRead: reading from index out of range: ", op, " in ", ot, " ***";
];
[ BlkValueWrite block pos val dsize hsize flags wsize ot op;
    if (block==0) rfalse;
    flags = block->BLK_HEADER_FLAGS; wsize = 1;
    if (flags & BLK_FLAG_WORD) wsize = WORDSIZE;
    if (flags & BLK_FLAG_16_BIT) wsize = 2;
    ot = block; op = pos;
    pos = pos*wsize;
    if (flags & BLK_FLAG_MULTIPLE) hsize = BLK_DATA_MULTI_OFFSET;
    else hsize = BLK_DATA_OFFSET;
    for (:block~=NULL:block=block-->BLK_NEXT) {
        dsize = BlkSize(block) - hsize;
        if ((pos >= 0) && (pos<dsize)) {
            block = block + hsize + pos;
            switch(wsize) {
                1: block->0 = val;
                2: #Iftrue (WORDSIZE == 2); block-->0 = val;
                    #ifnot; block->0 = (val/256)%256; block->1 = val%256;
                    #endif;
                4: block-->0 = val;
            }
            return;
        }
        pos = pos - dsize;
    }
    "*** BlkValueWrite: writing to index out of range: ", op, " in ", ot, " ***";
];
```

§**3. KOV Support.** To carry out the four fundamental operations on block values – creating, destroying, copying and comparing – we need to take account of what data they hold. Letting an indexed text go matters little: it contains only character codes. But letting a list of lists of numbers disappear is risky, because we might leave allocated blocks all over the heap for those individual lists of numbers which are now no longer in their parent list. Every KOV has different needs, because every KOV uses its data differently.

Each block KOV, then, provides a single "KOV support" function. These are named systematically by suffixing `_Support`: that is, the support function for `INDEXED_TEXT_TY` is called `INDEXED_TEXT_TY_Support` and so on. NI compiles a function called `KOVSupportFunction` which takes a KOV number as its argument and returns the relevant function.

The support function can be called with any of the following task constants as its first argument: it then has a further one to three arguments depending on the task in hand.

```
Constant CREATE_KOVS     = 1;
Constant CAST_KOVS       = 2;
Constant DESTROY_KOVS    = 3;
Constant PRECOPY_KOVS    = 4;
Constant COPY_KOVS       = 5;
Constant COMPARE_KOVS    = 6;
Constant READ_FILE_KOVS  = 7;
Constant WRITE_FILE_KOVS = 8;
```

§**4. Creation.** To create a new value, we can't simply allocate a block and then fill it in some way appropriate to the kind of value needed: because we don't know what size of block would be a good idea, and even that much depends on the KOV. So the following routine delegates the whole process to the relevant KOV's support routines.

Depending on the KOV, we can also create it with a value cast from some existing value: for instance, an indexed text can be created with a value cast from the ordinary text "Marshmellow".

`K_Support(CREATE_KOVS, cast_from, skov)` – where `cast_from` and `skov` are optional arguments – is called to allocate a block on the heap and, if `cast_from` is non-zero, cast that data into the new block value. Of course `cast_from` itself is a value, and we don't specify its KOV: the support function is expected to know what it will get (if anything). `skov` stands for "secondary kind of value" and is only applicable if `K` is a KOV constructor rather than a base KOV: for instance, if `K` is "list of...", then to create a "list of numbers" we would have `skov` equal to `NUMBER_TY`, whereas to create a "list of lists of texts" `skov` would be `LIST_OF_TY`.

```
Global block_value_tally;
[ BlkValueCreate kov cast_from skov  block sf;
    sf = KOVSupportFunction(kov);
    if (sf) block = sf(CREATE_KOVS, cast_from, skov);
    else { print "*** Impossible runtime creation ***^"; rfalse; }
#ifdef SHOW_ALLOCATIONS;
    print "[created ", kov, " at ", block, ": ", block_value_tally++, "]^";
#endif;
    return block;
];
```

§**5. Casting.** Just as we can create a value with a cast, we can also perform an assignment to an already-created block value in the form of a cast: well, for some kinds of value we can.

`K_Support(CAST_KOVS, fromval, fromkov, block)` casts from a value `fromval` with a specified KOV `fromkov` into the existing block value `block`.

```
[ BlkValueCast block tokov fromkov fromval  sf;
    sf = KOVSupportFunction(tokov);
    if (sf) return sf(CAST_KOVS, fromval, fromkov, block);
    else { print "*** Impossible runtime cast ***^"; rfalse; }
];
```

§**6. Destruction.** This must be called whenever the current contents of a block value is about to be lost, either because of overwriting or because the block value is in a variable which is going out of scope. What must be done depends on the kind of value: for instance, for indexed text nothing need be done, because the array entries only contains character values. But for a list of lists of numbers, for instance, array entries are pointers to lists of numbers. Simply overwriting or forgetting them means that the lists to which they point will be stuck on the heap forever, and will never be deallocated. So value destruction for a list involves destroying each individual entry and then deallocating it: and this may of course recurse.

`K_Support(DESTROY_KOVS, block)` takes whatever action is necessary to ensure that the values remaining in `block` can be discarded without leaving unreferenced data left on the heap.

```
[ BlkValueDestroy block  k rv sf;
    if (block == 0) return;
    k = block-->BLK_HEADER_KOV;
    sf = KOVSupportFunction(k);
    if (sf) return sf(DESTROY_KOVS, block);
    else { print "*** Impossible runtime deallocation ***^"; rfalse; }
];
```

§**7. Deep Copy.** A deep copy transfers the data contents – that is, the arrays described above – from one block to another; the old contents must be destroyed first. Note that we first perform a simplistic copy of the bits over, but that for some block values that won't be good enough. If the two blocks contain lists of lists, then each entry is itself a list, and a deep copy of each entry must also be performed.

`K_Support(COPY_KOVS, blockto, blockfrom)` takes whatever additional action might be necessary to ensure that pointers to heap data are not duplicated; it runs after the raw data has been copied bitwise and may only have to correct a few entries, or even none at all.

```
[ BlkValueCopy blockto blockfrom dsize i sf;
    if (blockto == 0) { print "*** Deep copy failed: destination empty ***^"; rfalse; }
    if (blockfrom == 0) { print "*** Deep copy failed: source empty ***^"; rfalse; }
    if (blockfrom->BLK_HEADER_N == 0) {
        ! A hack to handle precompiled array constants: N=0 blocks otherwise don't exist
        LIST_OF_TY_CopyRawArray(blockto, blockfrom, 1);
        return blockto;
    }
    if (blockfrom-->BLK_HEADER_KOV ~= blockto-->BLK_HEADER_KOV) {
        print "*** Deep copy failed: types mismatch ***^"; rfalse;
    }
    BlkValueDestroy(blockto);
    dsize = BlkValueExtent(blockfrom);
```

```
    if (((blockfrom->BLK_HEADER_FLAGS) & BLK_FLAG_MULTIPLE) &&
        (BlkValueSetExtent(blockto, dsize, -1) == false)) {
        print "*** Deep copy failed: resizing failed ***^"; rfalse;
    }
    sf = KOVSupportFunction(blockfrom-->BLK_HEADER_KOV);
    if (sf) sf(PRECOPY_KOVS, blockto, blockfrom);
    for (i=0:i<dsize:i++) BlkValueWrite(blockto, i, BlkValueRead(blockfrom, i));
    if (sf) sf(COPY_KOVS, blockto, blockfrom);
    else { print "*** Impossible runtime copy ***^"; rfalse; }
    return blockto;
];
```

§**8. Comparison.** And it's a similar story with comparison.

`K_Support(COMPARE_KOVS, blockleft, blockright)` looks at the data in the two blocks and returns 0 if they are equal, a positive number if `blockright` is "greater than" `blockleft`, and a negative number if not. The interpretation of "greater than" depends on the KOV: it should be something which the user would find natural.

```
[ BlkValueCompare blockleft blockright  kov sf;
    if ((blockleft == 0) && (blockright == 0)) return 0;
    if (blockleft == 0) return 1;
    if (blockright == 0) return -1;
    if (blockleft-->BLK_HEADER_KOV ~= blockright-->BLK_HEADER_KOV)
        return blockleft-->BLK_HEADER_KOV - blockright-->BLK_HEADER_KOV;
    kov = blockleft-->BLK_HEADER_KOV;
    sf = KOVSupportFunction(kov);
    if (sf) return sf(COMPARE_KOVS, blockleft, blockright);
    else { print "*** Impossible runtime comparison ***^"; rfalse; }
];
```

§**9. Creating Constants.** If the NI compiler reads, say, the string literal "Sea Devils" in the source text, and it is expecting to find indexed text – for instance if it's in a column of a table marked as containing indexed text – then how is this to find its way into memory as a block value? The heaps starts entirely empty, and NI cannot precompile parts of it: nor would we really want that, since it would hardwire assumptions about the heap's format into NI itself.

The answer is that at start-up time we perform creations of all of the constants we will need. Because they are constants, they will never be deallocated and never change in their contents – so they need not live on the heap at all, and in fact we store them elsewhere in memory. NI simply creates a blank array of the right 2^n size. Suppose that `X` is the address of this array. Then during the start-up rules, we at some point do this:

```
    BlkValueInitialCopy(X, BlkValueCreate(INDEXED_TEXT_TY, "Sea Devils"))
```

The effect is to create a suitable structure on the heap to hold this as indexed text, and then to copy it bitwise into `X`. A bitwise copy is ordinarily against the rules because it duplicates pointers, but of course the created block has been created for this purpose only: nothing points to it, and it is about to vanish – indeed the address of the block in question exists only briefly on the VM stack. (It is because of this that we don't perform a deep copy using the routine above.)

```
[ BlkValueInitialCopy blockto blockfrom dsize i;
    if (blockto == 0) { print "*** Initial copy failed: destination empty ***^"; rfalse; }
    if (blockfrom == 0) { print "*** Initial copy failed: source empty ***^"; rfalse; }
```

```
    dsize = 1; for (i=1: i<=blockfrom->BLK_HEADER_N: i++) dsize=dsize*2;
    for (i=0:i<dsize:i++) blockto->i = blockfrom->i;
    return blockto;
];
```

§**10. Serialisation.** Some block values can be written to external files (on Glulx): others cannot. The following routines abstract that.

If `ch` is −1, then `K_Support(READ_FILE_KOVS, block, auxf, ch)` returns `true` or `false` according to whether it is possible to read data from an auxiliary file `auxf` into the block value `block`. If `ch` is any other value, then the routine should do exactly that, taking `ch` to be the first character of the text read from the file which makes up the serialised form of the data.

`K_Support(WRITE_FILE_KOVS, block)` is simpler because, strictly speaking, it doesn't write to a file at all: it simply prints a serialised form of the data in `block` to the output stream. Since it is called only when that output stream has been redirected to an auxiliary file, and since the serialised form would often be illegible on screen, it seems reasonable to call it a file input-output function just the same.

```
[ BlkValueReadFromFile block auxf ch kov  sf;
    sf = KOVSupportFunction(kov);
    if (sf) return sf(READ_FILE_KOVS, block, auxf, ch);
    rfalse;
];
[ BlkValueWriteToFile block kov  sf;
    sf = KOVSupportFunction(kov);
    if (sf) return sf(WRITE_FILE_KOVS, block);
    rfalse;
];
```

§**11. Stubs.** To ensure that the template code will still compile if `MEMORY_HEAP_SIZE` is undefined and there's no heap: none of these routines do anything in such a situation, but nor are they ever called – it's just that I6 source code may refer to them anyway, so they need to exist as routine names.

```
#IFNOT; ! IFDEF MEMORY_HEAP_SIZE
[ BlkValueReadFromFile; rfalse; ];
[ BlkValueWriteToFile; rfalse; ];
[ BlkValueCreate; ];
[ BlkValueCompare x y; ];
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```

Flex Template B/flext

Purpose

To allocate flexible-sized blocks of memory as needed to hold arbitrary-length strings of text, stored actions or other block values.

---

§**1. Overview.** Each I7 value is represented at run-time by an I6 word: on the Z-machine, a 16-bit number, and on Glulx, a 32-bit number. The correspondence between these numbers and the original values depends on the kind of value: "number" comes out as a signed twos-complement number, but "time" as an integer number of minutes since midnight, "rulebook" as the index of the rulebook in order of creation, and so on.

Even if a 32-bit number is available, this is not enough to represent the full range of values we might want: consider all the possible hundred-word essays of text, for instance. In some cases, then, the value is (either directly or indirectly) a pointer, telling the run-time code that the data is not in the value itself but can be found at a given location in memory. For instance, a "text" value is a (packed) address of either some encoded text or a routine to print text. When NI compiles references to a text value, it also creates the data to which this address belongs.

This works well if the data need not change in size, and if we never need to create or throw away values. But if we want a variable which can hold an arbitrary string of text which we will compose for ourselves, say, then we need at run-time to find space somewhere in memory to hold that data, and we need to be able to cope if that space runs out, and lastly we need a way to reclaim the memory again when the text's usefulness has finished. For instance, if a rule uses a temporary "let" variable which will hold indexed text then the block of indexed text data must be allocated; the variable must then be set to a pointer to this data; the rule must then run, and lastly the block of data deallocated again.

Values of this kind are called "block values" since they are pointers to blocks of memory. We have to be careful when making assignments to variables having these kinds of value. In I7, text like "change the motto text to the player's command" must not be compiled to I6 code such as `MT = PC;`, because that simply sets `MT` to have the same address as `PC`. There are now two independent pointers to the same piece of text, so that changing either one would change both, while the previous contents of `MT` are un-pointed to. The latter problem is almost as bad as the former, because it means that memory has been wasted forever – it would never be reclaimed. Repeat the process often enough and all the memory will be lost in this way: this is called a "leak".

I7 treats block values exactly like other values, as far as the writer is concerned. There is no concept of "a pointer to..." in I7 source text. Instead:

(1) For every allocated block of data, there is exactly one I7 value – stored in an I6 local or global variable, in a property, in a table entry, or even as part of another allocated block of data – which points to it.
(2) When assigning `Y` to `X`, we always perform a "deep copy" of the contents of the block pointed to by `Y` into that pointed to by `X`: we never copy pointers, which would be a "shallow copy".
(3) If `X` is the I6 representation of a block kind of value, then `X` is either 0 – meaning "not allocated yet" – or a valid pointer to a block of data. This is purely for efficiency's sake, making table columns of indexed text (for instance) a sparse representation when, as often happens, not many are filled in. When assigning to 0, we must first allocate a block, then change the pointer from 0 to point to it, and assign to the newly-allocated block. The user is oblivious to all of this.
(4) Any value which will be lost – for instance, a value in a variable which goes out of scope – must have its block deallocated first.
(5) Any value which is passed as an argument to a function must be copied first, as otherwise there are two pointers to the same block of data, one in the calling stack frame and the other in the one called;

in other words, we call by value, never by reference. (As we shall see, I7 phrases can indeed by defined which work by reference rather than by value, but those are defined using inline I6, not as function calls.)

As simple as this scheme looks – there is no need for garbage collection or reference counting – it is not entirely easy to get right. There are many complications, but the basic slogan is one pointer, one block.

§**2. Head.** On the Z-machine, though not always on Glulx, the entire heap has to be allocated at compile-time, and we tend to want to make it reasonably large to cover most eventualities (though the heap size is controllable with use options, so the user does have control over this).

Many small works of IF never have need of the heap at all, and at the same time can't spare the memory for it. NI only creates the constant `MEMORY_HEAP_SIZE`, the number of bytes initially given over to the heap, if need arises. So the code and arrays in this segment will never be compiled unless needed (but see also the "Stubs" paragraph below).

```
#IFDEF MEMORY_HEAP_SIZE;
! Constant SHOW_ALLOCATIONS = 1; ! Uncomment this for debugging purposes
```

§**3. Blocks.** A "block" is a continuous range of 2^n bytes of memory, where $n \geq 3$ for a 16-bit VM (i.e., for the Z-machine) and $n \geq 4$ for a 32-bit VM (i.e., on Glulx). Internally, a block is divided into a header followed by a data section.

The header consists of 4, 8 or 16 bytes, depending on the word size and the kind of block (see below). It always begins with a byte specifying n, the binary logarithm of its size: thus the largest block representable is 2^{255} bytes long, but somehow I think we can live with that. The second byte contains a bitmap of (at present) three flags, whose meanings will be explained below. The second *word* of the block, which might be at byte offset 2 or 4 from the start of the block depending on the word-size of the VM, is a number specifying the kind of value (KOV) which the block contains data of.

It might be objected that KOVs are not reducible to simple numbers. For instance, for any KOV K there is another KOV "list of K", so there is an infinite range of possibilities. "List of..." is what, in other languages, would be called a type constructor; whereas a KOV like "indexed text" is what would be called a base type, since it is not the result of any type constructor. In I7, there is a finite range of base types and type constructors, and these have distinct ID numbers: that is what is stored in the `BLK_HEADER_KOV` field. A block which has KOV "list of indexed texts" will have the same value here as a block which has KOV "list of numbers": it will store the I6 constant `LIST_OF_TY`. (To find out whether such a list does indeed contain numbers or texts – and it is essential to be able to do this – one must look at the data section of the block. See "Lists.i6t".)

The data section of a block begins at the byte offset `BLK_DATA_OFFSET` from the address of the block: but see below for how multiple-blocks behave differently.

```
Constant BLK_HEADER_N = 0;
Constant BLK_HEADER_FLAGS = 1;
Constant BLK_FLAG_MULTIPLE = $$00000001;
Constant BLK_FLAG_16_BIT   = $$00000010;
Constant BLK_FLAG_WORD     = $$00000100;
Constant BLK_HEADER_KOV = 1;
Constant BLK_DATA_OFFSET = 2*WORDSIZE;
```

§**4. Multiple Blocks.** There are two kinds of block values: those which can always be stored in a single block (for instance, a floating-point number stored in exactly 8 bytes of data would be suitable for this), and those which can change unpredictably in size and might at any point overflow their current storage, so that they may need to occupy multiple blocks (for instance, an indexed text). In such a multiple-block KOV, the data is stored in a doubly linked list of blocks, and the I6 value for the result is the pointer to the block which heads the linked list. For instance, the indexed text

```
"But now I worship a celestiall Sunne"
```

might be represented by an I6 value `BN` which points to a list of blocks like so:

```
NULL <-- BN: "But now I wor" <--> BN2: "ship a celestiall Sunne" --> NULL
```

Note that the unique pointer to `BN2` is the one in the header of the `BN` block. When we need to grow such a text, we add additional blocks; if the text should shrink, blocks at the end can at our discretion be deallocated. If the entire text should be deallocated, then all of the blocks used for it are deallocated, starting at the back and working towards the front.

A multiple-block is one whose flags byte contains the `BLK_FLAG_MULTIPLE`. This information is redundant since it could in principle be deduced from the kind of value stored in the block, which is recorded in the `-->BLK_HEADER_KOV` word, but that would be too slow. `BLK_FLAG_MULTIPLE` can never change for a currently allocated block, just as it can never change its KOV.

A multiple-block header is longer than that of an ordinary block, because it contains two extra words: `-->BLK_NEXT` is the next block in the doubly-linked list of blocks representing the current value, or `NULL` if this is the end; `-->BLK_PREV` is the previous block, or `NULL` if this is the beginning. The need to fit these two extra words in means that the data section is deferred, and so for a multiple-block data begins at the byte offset `BLK_DATA_MULTI_OFFSET` rather than `BLK_DATA_OFFSET`.

```
Constant BLK_DATA_MULTI_OFFSET = 4*WORDSIZE;
Constant BLK_NEXT 2;
Constant BLK_PREV 3;
```

§**5. Block Routines.**

```
[ BlkType txb;
    return txb-->BLK_HEADER_KOV;
];
[ BlkSize txb bsize n; ! Size of an individual block, including header
    if (txb == 0) return 0;
    for (bsize=1: n<txb->BLK_HEADER_N: bsize=bsize*2) n++;
    return bsize;
];
[ BlkTotalSize txb tsize; ! Combined size of multiple-blocks for a value
    if (txb == 0) return 0;
    if ((txb->BLK_HEADER_FLAGS) & BLK_FLAG_MULTIPLE == 0)
        return BlkSize(txb);
    for (:txb~=NULL:txb=txb-->BLK_NEXT) {
        tsize = tsize + BlkSize(txb);
    }
    return tsize;
];
```

§**6. Debugging Routines.** These two routines are purely for testing the code.

```
[ BlkDebug txb n k i bsize tot dtot kov;
    if (txb == 0) "Block never created.";
    kov = txb-->BLK_HEADER_KOV;
    print "Block ", txb, " (kov ", kov, "): ";
    for (:txb~=NULL:txb = txb-->BLK_NEXT) {
        if (k++ == 100) " ... and so on.";
        if (txb-->BLK_HEADER_KOV ~= kov)
            print "*Wrong kov=", txb-->BLK_HEADER_KOV, "* ";
        n = txb->BLK_HEADER_N;
        for (bsize=1:n>0:n--) bsize=bsize*2;
        i = bsize - BLK_DATA_OFFSET;
        dtot = dtot+i;
        tot = tot+bsize;
        print txb, "(", bsize, ") > ";
    }
    print dtot, " data in ", tot, " bytes^";
];
[ BlkDebugDecomposition from to txb pf;
    if (to==0) to = NULL;
    for (txb=from:(txb~=to) && (txb~=NULL):txb=txb-->BLK_NEXT) {
        if (pf) print "+";
        print BlkSize(txb);
        pf = true;
    }
    print "^";
];
```

§**7. The Heap.** Properly speaking, a "heap" is a specific kind of structure often used for managing uneven-sized or unpredictably changing data. We use "heap" here in the looser sense of being an amorphous-sized collection of blocks of memory, some free, others allocated; our actual representation of free space on the heap is not a heap structure in computer science terms. (Though this segment could easily be rewritten to make it so, or to adopt any other scheme which might be faster, without modifying the rest of the template or NI itself.) The heap begins as a contiguous region of memory, but it need not remain so: on Glulx we use dynamic memory allocation to extend it.

For I7 purposes we don't need a way to represent allocated memory, only the free memory. A block is free if and only if it has `-->BLK_HEADER_KOV` equal to 0, which is never a valid kind of value, and also has the multiple flag set. We do that because we construct the whole collection of free blocks, at any given time, as a single, multiple-block "value": a doubly linked list joined by the `-->BLK_NEXT` and `<--BLK_PREV`.

A single block, at the bottom of memory and never moving, never allocated to anyone, is preserved in order to be the head of this linked list of free blocks. This is a 16-byte (i.e., $n = 4$) block, which we format when the heap is initialised in `HeapInitialise()`. Thus the heap is full if and only if the `-->BLK_NEXT` of the head-free-block is `NULL`.

So far we have described a somewhat lax regime. After many allocations and deallocations one could imagine the list of free blocks becoming a very long list of individually small blocks, which would both make it difficult to allocate large blocks, and also slow to look through the list. To ameliorate matters, we maintain the following invariants:

(a) In the free blocks list, `B-->BLK_NEXT` is always an address after `B`;
(b) For any contiguous run of free space blocks in memory (excluding the head-free-block), taking up a total of T bytes, the last block in the run has size 2^n where n is the largest integer such that $2^n \leq T$.

For instance, there can never be two consecutive free blocks of size 128: they would form a "run" in the sense of rule (b) of size $T = 256$, and when T is a power of two the run must contain a single block. In general, it's easy to prove that the number of blocks in the run is exactly the number of 1s when T is written out as a binary number, and that the blocks are ordered in memory from small to large (the reverse of the direction of reading, i.e., rightmost 1 digit first). Maintaining (b) is a matter of being careful to fragment blocks only from the front when smaller blocks are needed, and to rejoin from the back when blocks are freed and added to the free space object.

```
Array Blk_Heap -> MEMORY_HEAP_SIZE + 16; ! Plus 16 to allow room for head-free-block
```

§**8. Initialisation.** To recap: the constant `MEMORY_HEAP_SIZE` has been predefined by the NI compiler, and is always itself a power of 2, say 2^n. We therefore have $2^n + 2^4$ bytes available to us, and we format these as a free space list of two blocks: the 2^4-sized "head-free-block" described above followed by a 2^n-sized block exactly containing the whole of the rest of the heap.

```
[ HeapInitialise n bsize blk2;
    blk2 = Blk_Heap + 16;
    Blk_Heap->BLK_HEADER_N = 4;
    Blk_Heap-->BLK_HEADER_KOV = 0;
    Blk_Heap->BLK_HEADER_FLAGS = BLK_FLAG_MULTIPLE;
    Blk_Heap-->BLK_NEXT = blk2;
    Blk_Heap-->BLK_PREV = NULL;
    for (bsize=1: bsize < MEMORY_HEAP_SIZE: bsize=bsize*2) n++;
    blk2->BLK_HEADER_N = n;
    blk2-->BLK_HEADER_KOV = 0;
    blk2->BLK_HEADER_FLAGS = BLK_FLAG_MULTIPLE;
    blk2-->BLK_NEXT = NULL;
    blk2-->BLK_PREV = Blk_Heap;
];
```

§**9. Net Free Space.** "Net" in the sense of "after deductions for the headers": this is the actual number of free bytes left on the heap which could be used for data. Note that it is used to predict whether it is possible to fit something further in: so there are two answers, depending on whether the something is multiple-block data (with a larger header and therefore less room for data) or single-block data (smaller header, more room).

```
[ HeapNetFreeSpace multiple txb asize;
    for (txb=Blk_Heap-->BLK_NEXT: txb~=NULL: txb=txb-->BLK_NEXT) {
        asize = asize + BlkSize(txb);
        if (multiple) asize = asize - BLK_DATA_MULTI_OFFSET;
        else asize = asize - BLK_DATA_OFFSET;
    }
    return asize;
];
```

§**10. Make Space.** The following routine determines if there is enough free space to accommodate another `size` bytes of data, given that it has to be multiple-block data if the `multiple` flag is set. If the answer turns out to be "no", we see if the host virtual machine is able to allocate more for us: if it is, then we ask for 2^m further bytes, where 2^m is at least `size` plus the worst-case header storage requirement (16 bytes), and in addition is large enough to make it worth while allocating. We don't want to bother the VM by asking for trivial amounts of memory.

This looks to be more memory than is needed, since after all we've asked for enough that the new data can fit entirely into the new block allocated, and we might have been able to squeeze some of it into the existing free space. But it ensures that heap invariant (b) above is preserved, and besides, running out of memory tends to be something you don't do only once.

(The code below is a refinement on the original, suggested by Jesse McGrew, which handles non-multiple blocks better.)

```
Constant SMALLEST_BLK_WORTH_ALLOCATING = 12; ! i.e. 2^12 = 4096 bytes

[ HeapMakeSpace size multiple  newblocksize newblock B n;
    for (::) {
        if (multiple) {
            if (HeapNetFreeSpace(multiple) >= size) rtrue;
        } else {
            if (HeapLargestFreeBlock(0) >= size) rtrue;
        }
        newblocksize = 1;
        for (n=0: (n<SMALLEST_BLK_WORTH_ALLOCATING) || (newblocksize<size): n++)
            newblocksize = newblocksize*2;
        while (newblocksize < size+16) newblocksize = newblocksize*2;
        newblock = VM_AllocateMemory(newblocksize);
        if (newblock == 0) rfalse;
        newblock->BLK_HEADER_N = n;
        newblock-->BLK_HEADER_KOV = 0;
        newblock->BLK_HEADER_FLAGS = BLK_FLAG_MULTIPLE;
        newblock-->BLK_NEXT = NULL;
        newblock-->BLK_PREV = NULL;
        for (B = Blk_Heap-->BLK_NEXT:B ~= NULL:B = B-->BLK_NEXT)
            if (B-->BLK_NEXT == NULL) {
                B-->BLK_NEXT = newblock;
                newblock-->BLK_PREV = B;
                jump Linked;
            }
        Blk_Heap-->BLK_NEXT = newblock;
        newblock-->BLK_PREV = Blk_Heap;
        .Linked; ;
        #ifdef SHOW_ALLOCATIONS;
        print "Increasing heap to free space map: "; BlkDebugDecomposition(Blk_Heap, 0);
        #endif;
    }
    rtrue;
];

[ HeapLargestFreeBlock multiple txb asize best;
    best = 0;
    for (txb=Blk_Heap-->BLK_NEXT: txb~=NULL: txb=txb-->BLK_NEXT) {
        asize = BlkSize(txb);
        if (multiple) asize = asize - BLK_DATA_MULTI_OFFSET;
```

```
        else asize = asize - BLK_DATA_OFFSET;
        if (asize > best) best = asize;
    }
    return best;
];
```

§**11. Block Allocation.** The routine `BlkAllocate(N, K, F)` allocates a block with room for `size` net bytes of data, which will have kind of value `K` and with flags `F`. If the flags include `BLK_FLAG_MULTIPLE`, this may be either a list of blocks or a single block. It returns either the address of the block or else throws run-time problem message and returns 0.

In allocation, we try to find a block which is as close as possible to the right size, and we may have to subdivide blocks: see case II below. For instance, if a block of size 2^n is available and we only need a block of size 2^k where $k < n$ then we break it up in memory as a sequence of blocks of size $2^k, 2^k, 2^{k+1}, 2^{k+2}, ..., 2^{n-1}$: note that the sum of these sizes is the 2^n we started with. We then use the first block of size 2^k. To continue the comparison with binary arithmetic, this is like a subtraction with repeated carries:

$$10000000_2 - 00001000_2 = 01111000_2$$

```
[ BlkAllocate size kov flags
    dsize n m free_block min_m max_m smallest_oversized_block secondhalf i hsize head tail;
    if (HeapMakeSpace(size, flags & BLK_FLAG_MULTIPLE) == false)
        return BlkAllocationError("ran out");
    ! Calculate the header size for a block of this KOV
    if (flags & BLK_FLAG_MULTIPLE) hsize = BLK_DATA_MULTI_OFFSET;
    else hsize = BLK_DATA_OFFSET;
    ! Calculate the data size
    n=0; for (dsize=1: dsize < hsize+size: dsize=dsize*2) n++;
    ! Seek a free block closest to the correct size, but starting from the
    ! block after the fixed head-free-block, which we can't touch
    min_m = 10000; max_m = 0;
    for (free_block = Blk_Heap-->BLK_NEXT:
        free_block ~= NULL:
        free_block = free_block-->BLK_NEXT) {
        m = free_block->BLK_HEADER_N;
        ! Current block the ideal size
        if (m == n) jump CorrectSizeFound;
        ! Current block too large: find the smallest which is larger than needed
        if (m > n) {
            if (min_m > m) {
                min_m = m;
                smallest_oversized_block = free_block;
            }
        }
        ! Current block too small: find the largest which is smaller than needed
        if (m < n) {
            if (max_m < m) {
                max_m = m;
            }
        }
    }
    if (min_m == 10000) {
```

```
    ! Case I: No block is large enough to hold the entire size
    if (flags & BLK_FLAG_MULTIPLE == 0) return BlkAllocationError("too fragmented");
    ! Set dsize to the size in bytes if the largest block available
    for (dsize=1: max_m > 0: dsize=dsize*2) max_m--;
    ! Split as a head (dsize-hsize), which we can be sure fits into one block,
    ! plus a tail (size-(dsize-hsize), which might be a list of blocks
    head = BlkAllocate(dsize-hsize, kov, flags);
    if (head == 0) return BlkAllocationError("head block not available");
    tail = BlkAllocate(size-(dsize-hsize), kov, flags);
    if (tail == 0) return BlkAllocationError("tail block not available");
    head-->BLK_NEXT = tail;
    tail-->BLK_PREV = head;
    return head;
}
! Case II: No block is the right size, but some exist which are too big
! Set dsize to the size in bytes of the smallest oversized block
for (dsize=1,m=1: m<=min_m: dsize=dsize*2) m++;
free_block = smallest_oversized_block;
while (min_m > n) {
    ! Repeatedly halve free_block at the front until the two smallest
    ! fragments left are the correct size: then take the frontmost
    dsize = dsize/2;
    secondhalf = free_block + dsize;
    secondhalf-->BLK_NEXT = free_block-->BLK_NEXT;
    if (secondhalf-->BLK_NEXT ~= NULL)
        (secondhalf-->BLK_NEXT)-->BLK_PREV = secondhalf;
    secondhalf-->BLK_PREV = free_block;
    free_block-->BLK_NEXT = secondhalf;
    free_block->BLK_HEADER_N = (free_block->BLK_HEADER_N) - 1;
    secondhalf->BLK_HEADER_N = free_block->BLK_HEADER_N;
    secondhalf-->BLK_HEADER_KOV = free_block-->BLK_HEADER_KOV;
    secondhalf->BLK_HEADER_FLAGS = free_block->BLK_HEADER_FLAGS;
    min_m--;
}
! Once that is done, free_block points to a block which is exactly the
! right size, so we can fall into...
! Case III: There is a free block which has the correct size.
.CorrectSizeFound;
! Delete the free block from the double linked list of free blocks: note
! that it cannot be the head of this list, which is fixed
if (free_block-->BLK_NEXT == NULL) {
    ! We remove final block, so previous is now final
    (free_block-->BLK_PREV)-->BLK_NEXT = NULL;
} else {
    ! We remove a middle block, so join previous to next
    (free_block-->BLK_PREV)-->BLK_NEXT = free_block-->BLK_NEXT;
    (free_block-->BLK_NEXT)-->BLK_PREV = free_block-->BLK_PREV;
}
free_block-->BLK_HEADER_KOV = kov;
free_block->BLK_HEADER_FLAGS = flags;
if (flags & BLK_FLAG_MULTIPLE) {
    free_block-->BLK_NEXT = NULL;
    free_block-->BLK_PREV = NULL;
```

```
    }
    ! Zero out the data bytes in the memory allocated
    for (i=hsize:i<dsize:i++) free_block->i=0;
    return free_block;
];
[ BlkAllocationError reason;
    print "*** Memory ", (string) reason, " ***^";
    RunTimeProblem(RTP_HEAPERROR);
    rfalse;
];
```

§**12. Merging.** Given a free block `block`, find the maximal contiguous run of free blocks which contains it, and then call `BlkRecut` to recut it to conform to invariant (b) above.

```
[ BlkMerge block first last pv nx;
    first = block; last = block;
    while (last-->BLK_NEXT == last+BlkSize(last))
        last = last-->BLK_NEXT;
    while ((first-->BLK_PREV + BlkSize(first-->BLK_PREV) == first) &&
        (first-->BLK_PREV ~= Blk_Heap))
        first = first-->BLK_PREV;
    pv = first-->BLK_PREV;
    nx = last-->BLK_NEXT;
    #ifdef SHOW_ALLOCATIONS;
    print "Merging: "; BlkDebugDecomposition(pv-->BLK_NEXT, nx); print "^";
    #endif;
    if (BlkRecut(first, last)) {
        #ifdef SHOW_ALLOCATIONS;
        print " --> "; BlkDebugDecomposition(pv-->BLK_NEXT, nx); print "^";
        #endif;
    }
];
```

§**13. Recutting.** Given a segment of the free block list, containing blocks known to be contiguous in memory, we recut into a sequence of blocks satisfying invariant (b): we repeatedly cut the largest 2^m-sized chunk off the back end until it is all used up.

```
[ BlkRecut first last tsize backsize mfrom mto bnext backend n dsize fine_so_far;
    if (first == last) rfalse;
    mfrom = first; mto = last + BlkSize(last);
    bnext = last-->BLK_NEXT;
    fine_so_far = true;
    for (:mto>mfrom: mto = mto - backsize) {
        for (n=0, backsize=1: backsize*2 <= mto-mfrom: n++) backsize=backsize*2;
        if ((fine_so_far) && (backsize == BlkSize(last))) {
            bnext = last; last = last-->BLK_PREV;
            bnext-->BLK_PREV = last;
            last-->BLK_NEXT = bnext;
            continue;
        }
        fine_so_far = false; ! From this point, "last" is meaningless
        backend = mto - backsize;
```

```
        backend->BLK_HEADER_N = n;
        backend-->BLK_HEADER_KOV = 0;
        backend->BLK_HEADER_FLAGS = BLK_FLAG_MULTIPLE;
        backend-->BLK_NEXT = bnext;
        if (bnext ~= NULL) {
            bnext-->BLK_PREV = backend;
            bnext = backend;
        }
    }
    if (fine_so_far) rfalse;
    rtrue;
];
```

§**14. Deallocation.** There are two complications: first, when we free a multiple block we need to free all of the blocks in the list, starting from the back end and working forwards to the front – this is the job of `BlkFree`. Second, when any given block is freed it has to be put into the free block list at the correct position to preserve invariant (a): it might either come after all of the currently free blocks in memory, and have to be added to the end of the list, or in between two, and have to be inserted mid-list, but it can't be before all of them because the head-free-block is kept lowest in memory of all possible blocks. (Note that Glulx can't allocate memory dynamically which undercuts the ordinary array space created by I6: I6 arrays fill up memory from the bottom.)

```
[ BlkFree block fromtxb ptxb;
    if (block == 0) return;
    BlkValueDestroy(block);
    if ((block->BLK_HEADER_FLAGS) & BLK_FLAG_MULTIPLE) {
        if (block-->BLK_PREV ~= NULL) (block-->BLK_PREV)-->BLK_NEXT = NULL;
        fromtxb = block;
        for (:(block-->BLK_NEXT)~=NULL:block = block-->BLK_NEXT) ;
        while (block ~= fromtxb) {
            ptxb = block-->BLK_PREV; BlkFreeSingleBlock(block); block = ptxb;
        }
    }
    BlkFreeSingleBlock(block);
];
[ BlkFreeSingleBlock block free nx;
    block-->BLK_HEADER_KOV = 0;
    block->BLK_HEADER_FLAGS = BLK_FLAG_MULTIPLE;
    for (free = Blk_Heap:free ~= NULL:free = free-->BLK_NEXT) {
        nx = free-->BLK_NEXT;
        if (nx == NULL) {
            free-->BLK_NEXT = block;
            block-->BLK_PREV = free;
            block-->BLK_NEXT = NULL;
            BlkMerge(block);
            return;
        }
        if (UnsignedCompare(nx, block) == 1) {
            free-->BLK_NEXT = block;
            block-->BLK_PREV = free;
            block-->BLK_NEXT = nx;
            nx-->BLK_PREV = block;
```

```
            BlkMerge(block);
            return;
        }
    }
];
```

§**15. Resizing.** When the content of a value stretches or shrinks, we will sometimes need to change the size of the block(s) containing the data – though not always: we might sometimes need to resize a 1052-byte text to a 1204-byte text and find that we are sitting in a 2048-byte block in any case. We either shed blocks from the end of the chain, or add new blocks at the end, that being the simplest thing to do. Sometimes it might mean preserving a not very efficient block division, but it minimises the churn of blocks being allocated and freed, which is probably good.

```
[ BlkResize block req newsize dsize newblk kov n i otxb flags;
    if (block == 0) "*** Cannot resize null block ***";
    kov = block-->BLK_HEADER_KOV;
    flags = block->BLK_HEADER_FLAGS;
    if (flags & BLK_FLAG_MULTIPLE == 0) "*** Cannot resize inextensible block ***";
    otxb = block;
    newsize = req;
    for (:: block = block-->BLK_NEXT) {
        n = block->BLK_HEADER_N;
        for (dsize=1: n>0: n--) dsize = dsize*2;
        i = dsize - BLK_DATA_MULTI_OFFSET;
        newsize = newsize - i;
        if (newsize > 0) {
            if (block-->BLK_NEXT ~= NULL) continue;
            newblk = BlkAllocate(newsize, kov, flags);
            if (newblk == 0) rfalse;
            block-->BLK_NEXT = newblk;
            newblk-->BLK_PREV = block;
            rtrue;
        }
        if (block-->BLK_NEXT ~= NULL) {
            BlkFree(block-->BLK_NEXT);
            block-->BLK_NEXT = NULL;
        }
        rtrue;
    }
];
[ DebugHeap;
    print "Managing a heap of initially ", MEMORY_HEAP_SIZE+16, " bytes.^";
    print HeapNetFreeSpace(false), " bytes currently free.^";
    print "Free space decomposition: "; BlkDebugDecomposition(Blk_Heap);
    print "Free space map: "; BlkDebug(Blk_Heap);
];
```

§**16. Stubs.** To ensure that the template code will still compile if `MEMORY_HEAP_SIZE` is undefined and there's no heap: none of these routines do anything in such a situation, but nor are they ever called – it's just that I6 source code may refer to them anyway, so they need to exist as routine names.

```
#IFNOT; ! IFDEF MEMORY_HEAP_SIZE
[ HeapInitialise; ];
[ BlkFree; ];
[ DebugHeap;
    "This story file does not use a heap of managed memory.";
];
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```

ZMachine Template B/zmt

Purpose

To provide routines handling low-level Z-machine facilities.

---

§**1. Summary.** This segment closely parallels "Glulx.i6t", which provides exactly equivalent functionality (indeed, usually the same-named functions and in the same order) for the Glulx VM. This is intended to make the rest of the template code independent of the choice of VM, although that is more of an ideal than a reality, because there are so many fiddly differences in some of the grammar and dictionary tables that it is not really practical for the parser (for instance) to call VM-neutral routines to get the data it wants out of these arrays.

§2. Variables and Arrays.

```
Global top_object; ! largest valid number of any tree object
Global xcommsdir; ! true if command recording is on
Global transcript_mode; ! true if game scripting is on

Constant INPUT_BUFFER_LEN = 120; ! Length of buffer array

Array  buffer    -> 123;            ! Buffer for parsing main line of input
Array  buffer2   -> 123;            ! Buffers for supplementary questions
Array  buffer3   -> 123;            ! Buffer retaining input for "again"
Array  parse     buffer 63;         ! Parse table mirroring it
Array  parse2    buffer 63;         !

Global dict_start;
Global dict_entry_size;
Global dict_end;
```

§**3. Starting Up.** `VM_Initialise()` is almost the first routine called, except that the "starting the virtual machine" activity is allowed to go first.

```
[ VM_Initialise;
    standard_interpreter = HDR_TERPSTANDARD-->0;
    transcript_mode = ((HDR_GAMEFLAGS-->0) & 1);

    dict_start = HDR_DICTIONARY-->0;
    dict_entry_size = dict_start->(dict_start->0 + 1);
    dict_start = dict_start + dict_start->0 + 4;
    dict_end = dict_start + ((dict_start - 2)-->0) * dict_entry_size;

    buffer->0  = INPUT_BUFFER_LEN;
    buffer2->0 = INPUT_BUFFER_LEN;
    buffer3->0 = INPUT_BUFFER_LEN;
    parse->0   = 15;
    parse2->0  = 15;

    top_object = #largest_object-255;

    #ifdef FIX_RNG;
    @random 10000 -> i;
    i = -i-2000;
    print "[Random number generator seed is ", i, "]^";
    @random i -> i;
    #endif; ! FIX_RNG
];
```

§**4. Enable Acceleration.** This rule enables use of March 2009 extension to Glulx which optimises the speed of Inform-compiled story files, so for the Z-machine it has no effect.

```
[ ENABLE_GLULX_ACCEL_R;
    rfalse;
];
```

§**5. Release Number.** Like all software, IF story files have release numbers to mark revised versions being circulated: unlike most software, and partly for traditional reasons, the version number is recorded not in some print statement or variable but is branded on, so to speak, in a specific memory location of the story file header.

`VM_Describe_Release()` describes the release and is used as part of the "banner", IF's equivalent to a title page.

```
[ VM_Describe_Release i;
    print "Release ", (HDR_GAMERELEASE-->0) & $03ff, " / Serial number ";
    for (i=0 : i<6 : i++) print (char) HDR_GAMESERIAL->i;
];
```

§**6. Keyboard Input.** The VM must provide three routines for keyboard input:

(a) `VM_KeyChar()` waits for a key to be pressed and then returns the character chosen as a ZSCII character.
(b) `VM_KeyDelay(N)` waits up to $N/10$ seconds for a key to be pressed, returning the ZSCII character if so, or 0 if not.
(c) `VM_ReadKeyboard(b, t)` reads a whole newline-terminated command into the buffer `b`, then parses it into a word stream in the table `t`.

There are elaborations to due with mouse clicks, but this isn't the place to document all of that.

```
[ VM_KeyChar win  key;
    if (win) @set_window win;
    @read_char 1 -> key;
    return key;
];
[ VM_KeyDelay tenths  key;
    @read_char 1 tenths VM_KeyDelay_Interrupt -> key;
    return key;
];
[ VM_KeyDelay_Interrupt; rtrue; ];
[ VM_ReadKeyboard  a_buffer a_table i;
    read a_buffer a_table;
    #ifdef ECHO_COMMANDS;
    print "** ";
    for (i=2: i<=(a_buffer->1)+1: i++) print (char) a_buffer->i;
    print "^";
    #ifnot;
    i=0;  ! suppress compiler warning
    #endif;
    #Iftrue (#version_number == 6);
    @output_stream -1;
    @loadb a_buffer 1 -> sp;
    @add a_buffer 2 -> sp;
    @print_table sp sp;
    new_line;
    @output_stream 1;
    #Endif;
];
```

§**7. Buffer Functions.** A "buffer", in this sense, is an array containing a stream of characters typed from the keyboard; a "parse buffer" is an array which resolves this into individual words, pointing to the relevant entries in the dictionary structure. Because each VM has its own format for each of these arrays (not to mention the dictionary), we have to provide some standard operations needed by the rest of the template as routines for each VM.

The Z-machine buffer and parse buffer formats are documented in the DM4.

`VM_CopyBuffer(to, from)` copies one buffer into another.

`VM_Tokenise(buff, parse_buff)` takes the text in the buffer `buff` and produces the corresponding data in the parse buffer `parse_buff` – this is called tokenisation since the characters are divided into words: in traditional computing jargon, such clumps of characters treated syntactically as units are called tokens.

`LTI_Insert` is documented in the DM4 and the `LTI` prefix stands for "Language To Informese": it's used only by translations into non-English languages of play, and is not called in the template.

```
[ VM_CopyBuffer bto bfrom i;
    for (i=0: i<INPUT_BUFFER_LEN: i++) bto->i = bfrom->i;
];
[ VM_PrintToBuffer buf len a b c;
    @output_stream 3 buf;
    switch (metaclass(a)) {
        String: print (string) a;
        Routine: a(b, c);
        Object, Class: if (b) PrintOrRun(a, b, true); else print (name) a;
    }
    @output_stream -3;
    if (buf-->0 > len) print "Error: Overflow in VM_PrintToBuffer.^";
    return buf-->0;
];
[ VM_Tokenise b p; b->(2 + b->1) = 0; @tokenise b p; ];
[ LTI_Insert i ch  b y;
    ! Protect us from strict mode, as this isn't an array in quite the
    ! sense it expects
    b = buffer;
    ! Insert character ch into buffer at point i.
    ! Being careful not to let the buffer possibly overflow:
    y = b->1;
    if (y > b->0) y = b->0;
    ! Move the subsequent text along one character:
    for (y=y+2 : y>i : y--) b->y = b->(y-1);
    b->i = ch;
    ! And the text is now one character longer:
    if (b->1 < b->0) (b->1)++;
];
```

§**8. Dictionary Functions.** Again, the dictionary structure is differently arranged on the different VMs. This is a data structure containing, in compressed form, the text of all the words to be recognised by tokenisation (above). In I6 for Z, a dictionary word value is represented at run-time by its record number in the dictionary, 0, 1, 2, ..., in alphabetical order.

`VM_InvalidDictionaryAddress(A)` tests whether `A` is a valid record address in the dictionary data structure.

`VM_DictionaryAddressToNumber(A)` and `VM_NumberToDictionaryAddress(N)` convert between record numbers and dictionary addresses.

```
[ VM_InvalidDictionaryAddress addr;
    if ((UnsignedCompare(addr, dict_start) < 0) ||
        (UnsignedCompare(addr, dict_end) >= 0) ||
        ((addr - dict_start) % dict_entry_size ~= 0)) rtrue;
    rfalse;
];
[ VM_DictionaryAddressToNumber w; return (w-(HDR_DICTIONARY-->0 + 7))/9; ];
[ VM_NumberToDictionaryAddress n; return HDR_DICTIONARY-->0 + 7 + 9*n; ];
```

§**9. Command Tables.** The VM is also generated containing a data structure for the grammar produced by I6's `Verb` and `Extend` directives: this is essentially a list of command verbs such as DROP or PUSH, together with a list of synonyms, and then the grammar for the subsequent commands to be recognised by the parser.

```
[ VM_CommandTableAddress i;
    return (HDR_STATICMEMORY-->0)-->i;
];
[ VM_PrintCommandWords i da j;
    da = HDR_DICTIONARY-->0;
    for (j=0 : j<(da+5)-->0 : j++)
        if (da->(j*9 + 14) == $ff-i)
            print "'", (address) VM_NumberToDictionaryAddress(j), "' ";
];
```

§**10. SHOWVERB support.** Further VM-specific tables cover actions and attributes, and these are used by the SHOWVERB testing command.

```
#Ifdef DEBUG;
[ DebugAction a anames;
    if (a >= 4096) { print "<fake action ", a-4096, ">"; return; }
    anames = #identifiers_table;
    anames = anames + 2*(anames-->0) + 2*48;
    print (string) anames-->a;
];
[ DebugAttribute a anames;
    if (a < 0 || a >= 48) print "<invalid attribute ", a, ">";
    else {
        anames = #identifiers_table; anames = anames + 2*(anames-->0);
        print (string) anames-->a;
    }
];
#Endif;
```

§**11. RNG.** No routine is needed for extracting a random number, since I6's built-in `random` function does that, but it's useful to abstract the process of seeding the RNG so that it produces a repeatable sequence of "random" numbers from here on: the necessary opcodes are different for the two VMs.

```
[ VM_Seed_RNG n;
    if (n > 0) n = -n;
    @random n -> n;
];
```

§**12. Memory Allocation.** This is dynamic memory allocation: something which is never practicable in the Z-machine, because the whole address range is already claimed, but which is viable on recent revisions of Glulx.

```
[ VM_AllocateMemory amount;
    return 0;
];
[ VM_FreeMemory address;
];
```

§**13. Audiovisual Resources.** The Z-machine only barely supports figures and sound effects. I7 only allows us to use them for Version 6 of the Z-machine, even though sound effects have a longer pedigree and Infocom used them on some version 5 and even some version 3 works: really, though, from an I7 point of view we would prefer that anyone needing figures and sounds use Glulx instead.

```
[ VM_Picture resource_ID;
    #IFTRUE #version_number == 6; ! Z-machine version 6
    @draw_picture resource_ID;
    #ENDIF;
];
[ VM_SoundEffect resource_ID;
    #IFTRUE #version_number == 6; ! Z-machine version 6
    @sound_effect resource_ID;
    #ENDIF;
];
```

§**14. Typography.** Relatively few typographic effects are available on the Z-machine, so that many of the semantic markups for text which would be distinguishable on Glulx are indistinguishable here.

```
[ VM_Style sty;
    switch (sty) {
        NORMAL_VMSTY, NOTE_VMSTY: style roman;
        HEADER_VMSTY, SUBHEADER_VMSTY, ALERT_VMSTY: style bold;
    }
];
```

§**15. Character Casing.** The following are the equivalent of `tolower` and `toupper`, the traditional C library functions for forcing letters into lower and upper case form, for the ZSCII character set.

```
[ VM_UpperToLowerCase c;
switch (c) {
    'A' to 'Z': c = c + 32;
    202, 204, 212, 214, 221: c--;
    217, 218: c = c - 2;
    158 to 160, 167 to 169, 208 to 210: c = c - 3;
    186 to 190, 196 to 200: c = c - 5 ;
    175 to 180: c = c - 6;
}
return c;
];
[ VM_LowerToUpperCase c;
switch (c) {
    'a' to 'z': c = c - 32;
    201, 203, 211, 213, 220: c++;
    215, 216: c = c + 2;
    155 to 157, 164 to 166, 205 to 207: c = c + 3;
    181 to 185, 191 to 195: c = c + 5 ;
    169 to 174: c = c + 6;
}
return c;
];
```

§**16. The Screen.** Our generic screen model is that the screen is made up of windows: we tend to refer only to two of these, the main window and the status line, but others may also exist from time to time. Windows have unique ID numbers: the special window ID -1 means "all windows" or "the entire screen", which usually amounts to the same thing.

Screen height and width are measured in characters, with respect to the fixed-pitch font used for the status line. The main window normally contains variable-pitch text which may even have been kerned, and character dimensions make little sense there.

Clearing all windows (`WIN_ALL` here) has the side-effect of collapsing the status line, so we need to ensure that `statuswin_cursize` is reduced to 0, in order to keep it accurate.

```
[ VM_ClearScreen window;
    switch (window) {
        WIN_ALL:    @erase_window -1; statuswin_cursize = 0;
        WIN_STATUS: @erase_window 1;
        WIN_MAIN:   @erase_window 0;
    }
];
#Iftrue (#version_number == 6);
[ VM_ScreenWidth  width charw;
    @get_wind_prop 1 3 -> width;
    @get_wind_prop 1 13 -> charw;
    charw = charw & $FF;
    return (width+charw-1) / charw;
];
#Ifnot;
[ VM_ScreenWidth; return (HDR_SCREENWCHARS->0); ];
```

```
#Endif;
[ VM_ScreenHeight; return (HDR_SCREENHLINES->0); ];
```

§**17. Window Colours.** Each window can have its own foreground and background colours.

The colour of individual letters or words of type is not controllable in Glulx, to the frustration of many, and so the template layer of I7 has no framework for handling this (even though it is controllable on the Z-machine, which is greatly superior in this respect).

```
[ VM_SetWindowColours f b window;
    if (clr_on && f && b) {
        if (window == 0) {  ! if setting both together, set reverse
            clr_fgstatus = b;
            clr_bgstatus = f;
            }
        if (window == 1) {
            clr_fgstatus = f;
            clr_bgstatus = b;
        }
        if (window == 0 or 2) {
            clr_fg = f;
            clr_bg = b;
        }
        if (statuswin_current)
            @set_colour clr_fgstatus clr_bgstatus;
        else
            @set_colour clr_fg clr_bg;
    }
];
[ VM_RestoreWindowColours; ! compare I6 library patch L61007
    if (clr_on) { ! check colour has been used
        VM_SetWindowColours(clr_fg, clr_bg, 2); ! make sure both sets of variables are restored
        VM_SetWindowColours(clr_fgstatus, clr_bgstatus, 1, true);
        VM_ClearScreen();
    }
    #Iftrue (#version_number == 6); ! request screen update
    (0-->8) = (0-->8) | $$00000100;
    #Endif;
];
```

§**18. Main Window.** The part of the screen on which commands and responses are printed, which ordinarily occupies almost all of the screen area.

`VM_MainWindow()` switches printing back from another window, usually the status line, to the main window. Note that the Z-machine implementation emulates the Glulx model of window rather than text colours.

```
[ VM_MainWindow;
    if (statuswin_current) {
        if (clr_on && clr_bgstatus > 1) @set_colour clr_fg clr_bg;
        else style roman;
        @set_window 0;
    }
    statuswin_current = false;
];
```

§**19. Status Line.** Despite the name, the status line need not be a single line at the top of the screen: that's only the conventional default arrangement. It can expand to become the equivalent of an old-fashioned VT220 terminal, with menus and grids and mazes displayed lovingly in character graphics, or it can close up to invisibility.

`VM_StatusLineHeight(n)` sets the status line to have a height of `n` lines of type. (The width of the status line is always the width of the whole screen, and the position is always at the top, so the height is the only controllable aspect.) The $n = 0$ case makes the status line disappear.

`VM_MoveCursorInStatusLine(x, y)` switches printing to the status line, positioning the "cursor" – the position at which printing will begin – at the given character grid position (x, y). Line 1 represents the top line; line 2 is underneath, and so on; columns are similarly numbered from 1 at the left.

```
[ VM_MoveCursorInStatusLine line column; ! 1-based position on text grid
    if (~~statuswin_current) {
        @set_window 1;
        if (clr_on && clr_bgstatus > 1) @set_colour clr_fgstatus clr_bgstatus;
        else                            style reverse;
    }
    if (line == 0) {
        line = 1;
        column = 1;
    }
    #Iftrue (#version_number == 6);
    Z6_MoveCursor(line, column);
    #Ifnot;
    @set_cursor line column;
    #Endif;
    statuswin_current = true;
];
#Iftrue (#version_number == 6);
[ Z6_MoveCursor line column  charw charh; ! 1-based position on text grid
    @get_wind_prop 1 13 -> charw; ! font size
    @log_shift charw $FFF8 -> charh;
    charw = charw / $100;
    line = 1 + charh*(line-1);
    column = 1 + charw*(column-1);
    @set_cursor line column;
];
#Endif;
```

```
#Iftrue (#version_number == 6);
[ VM_StatusLineHeight height  wx wy x y charh;
    ! Split the window. Standard 1.0 interpreters should keep the window 0
    ! cursor in the same absolute position, but older interpreters,
    ! including Infocom's don't - they keep the window 0 cursor in the
    ! same position relative to its origin. We therefore compensate
    ! manually.
    @get_wind_prop 0 0 -> wy; @get_wind_prop 0 1 -> wx;
    @get_wind_prop 0 13 -> charh; @log_shift charh $FFF8 -> charh;
    @get_wind_prop 0 4 -> y; @get_wind_prop 0 5 -> x;
    height = height * charh;
    @split_window height;
    y = y - height + wy - 1;
    if (y < 1) y = 1;
    x = x + wx - 1;
    @set_cursor y x 0;
    statuswin_cursize = height;
];
#Ifnot;
[ VM_StatusLineHeight height;
    if (statuswin_cursize ~= height)
        @split_window height;
    statuswin_cursize = height;
];
#Endif;

#Iftrue (#version_number == 6);
[ Z6_DrawStatusLine width x charw scw;
    (0-->8) = (0-->8) &~ $$00000100;
    @push say__p; @push say__pc;
    BeginActivity(CONSTRUCTING_STATUS_LINE_ACT);
    VM_StatusLineHeight(statuswin_size);
    ! Now clear the window. This isn't totally trivial. Our approach is to select the
    ! fixed space font, measure its width, and print an appropriate
    ! number of spaces. We round up if the screen isn't a whole number
    ! of characters wide, and rely on window 1 being set to clip by default.
    VM_MoveCursorInStatusLine(1, 1);
    @set_font 4 -> x;
    width = VM_ScreenWidth();
    spaces width;
    ClearParagraphing();
    if (ForActivity(CONSTRUCTING_STATUS_LINE_ACT) == false) {
        ! Back to standard font for the display. We use output_stream 3 to
        ! measure the space required, the aim being to get 50 characters
        ! worth of space for the location name.
        VM_MoveCursorInStatusLine(1, 2);
        @set_font 1 -> x;
        switch (metaclass(left_hand_status_line)) {
            String: print (string) left_hand_status_line;
            Routine: left_hand_status_line();
        }
        @get_wind_prop 1 3 -> width;
        @get_wind_prop 1 13 -> charw;
        charw = charw & $FF;
```

```
        @output_stream 3 StorageForShortName;
        print (PrintText) right_hand_status_line;
        @output_stream -3; scw = HDR_PIXELSTO3-->0 + charw;
        x = 1+width-scw;
        @set_cursor 1 x; print (PrintText) right_hand_status_line;
    }
    ! Reselect roman, as Infocom's interpreters go funny if reverse is selected twice.
    VM_MainWindow();
    ClearParagraphing();
    EndActivity(CONSTRUCTING_STATUS_LINE_ACT);
    @pull say__pc; @pull say__p;
];
#Endif;
```

§**20. Quotation Boxes.** No routine is needed to produce quotation boxes: the I6 `box` statement generates the necessary Z-machine opcodes all by itself.

§**21. Undo.** These simply wrap the relevant opcodes.

```
[ VM_Undo result_code;
    @restore_undo result_code;
    return result_code;
];
[ VM_Save_Undo result_code;
    @save_undo result_code;
    return result_code;
];
```

§**22. Quit The Game Rule.**

```
[ QUIT_THE_GAME_R; if (actor ~= player) rfalse;
    GL__M(##Quit,2); if (YesOrNo()~=0) quit; ];
```

§**23. Restart The Game Rule.**

```
[ RESTART_THE_GAME_R;
    if (actor ~= player) rfalse;
    GL__M(##Restart,1);
    if (YesOrNo()~=0) { @restart; GL__M(##Restart,2); }
];
```

§**24. Restore The Game Rule.**

```
[ RESTORE_THE_GAME_R;
    if (actor ~= player) rfalse;
    restore Rmaybe;
    return GL__M(##Restore,1);
    .RMaybe; GL__M(##Restore,2);
];
```

§25. Save The Game Rule.

```
[ SAVE_THE_GAME_R flag;
    if (actor ~= player) rfalse;
    #IFV5;
    @save -> flag;
    switch (flag) {
        0: GL__M(##Save,1);
        1: GL__M(##Save,2);
        2: GL__M(##Restore,2);
    }
    #IFNOT;
    save Smaybe;
    return GL__M(##Save,1);
    .SMaybe; GL__M(##Save,2);
    #ENDIF;
];
```

§26. Verify The Story File Rule. This is a fossil now, really, but in the days of Infocom, the 110K story file occupying an entire disc was a huge data set: floppy discs were by no means a reliable medium, and cheap hardware often used hit-and-miss components, as on the notorious Commodore 64 disc controller. If somebody experienced an apparent bug in play, it could easily be that he had a corrupt disc or was unable to read data of that density. So the VERIFY command, which took up to ten minutes on some early computers, would chug through the entire story file and compute a checksum, compare it against a known result in the header, and determine that the story file could or could not properly be read. The Z-machine provided this service as an opcode, and so Glulx followed suit.

```
[ VERIFY_THE_STORY_FILE_R;
    if (actor ~= player) rfalse;
    @verify ?Vmaybe;
    jump Vwrong;
    .Vmaybe; return GL__M(##Verify,1);
    .Vwrong;
    GL__M(##Verify,2);
];
```

§27. Switch Transcript On Rule.

```
[ SWITCH_TRANSCRIPT_ON_R;
    if (actor ~= player) rfalse;
    transcript_mode = ((0-->8) & 1);
    if (transcript_mode) return GL__M(##ScriptOn,1);
    @output_stream 2;
    if (((0-->8) & 1) == 0) return GL__M(##ScriptOn,3);
    GL__M(##ScriptOn,2); VersionSub();
    transcript_mode = true;
];
```

§28. Switch Transcript Off Rule.

```
[ SWITCH_TRANSCRIPT_OFF_R;
    if (actor ~= player) rfalse;
    transcript_mode = ((0-->8) & 1);
    if (transcript_mode == false) return GL__M(##ScriptOff,1);
    GL__M(##ScriptOff,2);
    @output_stream -2;
    if ((0-->8) & 1) return GL__M(##ScriptOff,3);
    transcript_mode = false;
];
```

§29. Announce Story File Version Rule.

```
[ ANNOUNCE_STORY_FILE_VERSION_R ix;
    if (actor ~= player) rfalse;
    Banner();
    print "Identification number: ";
    for (ix=6: ix <= UUID_ARRAY->0: ix++) print (char) UUID_ARRAY->ix;
    print "^";
    ix = 0; ! shut up compiler warning
    if (standard_interpreter > 0) {
        print "Standard interpreter ",
            standard_interpreter/256, ".", standard_interpreter%256,
            " (", HDR_TERPNUMBER->0;
        #Iftrue (#version_number == 6);
        print (char) '.', HDR_TERPVERSION->0;
        #Ifnot;
        print (char) HDR_TERPVERSION->0;
        #Endif;
        print ") / ";
    } else {
        print "Interpreter ", HDR_TERPNUMBER->0, " Version ";
        #Iftrue (#version_number == 6);
        print HDR_TERPVERSION->0;
        #Ifnot;
        print (char) HDR_TERPVERSION->0;
        #Endif;
        print " / ";
    }
    print "Library serial number ", (string) LibSerial, "^";
    #Ifdef LanguageVersion;
    print (string) LanguageVersion, "^";
    #Endif; ! LanguageVersion
    #ifdef ShowExtensionVersions;
    ShowExtensionVersions();
    #endif;
    say__p = 1;
];
```

§**30. Descend To Specific Action Rule.** There are 100 or so actions, typically, and this rule is for efficiency's sake: rather than perform 100 or so comparisons to see which routine to call, we indirect through a jump table. The routines called are the `-Sub` routines: thus, for instance, if `action` is `##Wait` then `WaitSub` is called. It is essential that this routine not be called for fake actions: in I7 use this is guaranteed, since fake actions are not allowed into the action machinery at all.

Strangely, Glulx's action routines table is numbered in an off-by-one way compared to the Z-machine's: hence the `+1`.

```
[ DESCEND_TO_SPECIFIC_ACTION_R;
    indirect(#actions_table-->action);
    rtrue;
];
```

§**31. Veneer.**

```
[ OhLookItsReal; ];
[ OhLookItsRoom; ];
[ OhLookItsThing; ];
[ OC__Cl obj cla j a n objflag;
! if (cla > 4) OhLookItsReal();
! if (cla == K1_room) OhLookItsRoom();
! if (cla == K2_thing) OhLookItsThing();
    @jl obj 1 ?NotObj;
    @jg obj max_z_object ?NotObj;
    @inc objflag;
    @je cla K1_room ?~NotRoom;
    @test_attr obj mark_as_room ?rtrue;
    @rfalse;
    .NotRoom;
    @je cla K2_thing ?~NotObj;
    @test_attr obj mark_as_thing ?rtrue;
    @rfalse;
    .NotObj;
    @je cla Object Class ?ObjOrClass;
    @je cla Routine String ?RoutOrStr;
    @jin cla 1 ?~Mistake;
    @jz objflag ?rfalse;
    @get_prop_addr obj 2 -> a;
    @jz a ?rfalse;
    @get_prop_len a -> n;
    @div n 2 -> n;
    .Loop;
    @loadw a j -> sp;
    @je sp cla ?rtrue;
    @inc j;
    @jl j n ?Loop;
    @rfalse;
    .ObjOrClass;
    @jz objflag ?rfalse;
    @je cla Object ?JustObj;
```

```
    ! So now cla is Class
    @jg obj String ?~rtrue;
    @jin obj Class ?rtrue;
    @rfalse;
    .JustObj;
    ! So now cla is Object
    @jg obj String ?~rfalse;
    @jin obj Class ?rfalse;
    @rtrue;
    .RoutOrStr;
    @jz objflag ?~rfalse;
    @call_2s Z__Region obj -> sp;
    @inc sp;
    @je sp cla ?rtrue;
    @rfalse;
    .Mistake;
    RT__Err("apply 'ofclass' for", cla, -1);
    rfalse;
];
[ Unsigned__Compare x y u v;
    @je x y ?rfalse; ! i.e., return 0
    @jl x 0 ?XNegative;
    ! So here x >= 0 and x ~= y
    @jl y 0 ?XPosYNeg;
    ! Here x >=0, y >= 0, x ~= y
    @jg x y ?rtrue; ! i.e., return 1
    @ret -1;
    .XPosYNeg;
    ! Here x >= 0, y < 0, x ~= y
    @ret -1;
    .XNegative;
    @jl y 0 ?~rtrue; ! if x < 0, y >= 0, return 1
    ! Here x < 0, y < 0, x ~= y
    @jg x y ?rtrue;
    @ret -1;
];
[ RT__ChLDW base offset;
    @loadw base offset -> sp;
    @ret sp;
];
```

Purpose

To start up the Glk interface for the Glulx virtual machine, and provide Glulx-specific printing functions.

§**1. Summary.** This segment closely parallels "ZMachine.i6t", which provides exactly equivalent functionality (indeed, usually the same-named functions and in the same order) for the Z-machine VM. This is intended to make the rest of the template code independent of the choice of VM, although that is more of an ideal than a reality, because there are so many fiddly differences in some of the grammar and dictionary tables that it is not really practical for the parser (for instance) to call VM-neutral routines to get the data it wants out of these arrays.

§2. Variables and Arrays.

```
Array gg_event --> 4;
Array gg_arguments buffer 28;
Global gg_mainwin = 0;
Global gg_statuswin = 0;
Global gg_quotewin = 0;
Global gg_scriptfref = 0;
Global gg_scriptstr = 0;
Global gg_savestr = 0;
Global gg_commandstr = 0;
Global gg_command_reading = 0;      ! true if gg_commandstr is being replayed
Global gg_foregroundchan = 0;
Global gg_backgroundchan = 0;

Constant INPUT_BUFFER_LEN = 260;    ! No extra byte necessary
Constant MAX_BUFFER_WORDS = 20;
Constant PARSE_BUFFER_LEN = 61;

Array  buffer    buffer INPUT_BUFFER_LEN;
Array  buffer2   buffer INPUT_BUFFER_LEN;
Array  buffer3   buffer INPUT_BUFFER_LEN;
Array  parse     --> PARSE_BUFFER_LEN;
Array  parse2    --> PARSE_BUFFER_LEN;
```

§**3. Infglk.** This section is a verbatim copy of John Cater's invaluable I6 header file `infglk.h`, kindly donated by the author. The routines are convenient to have on hand, and also provide a canonical set of I6 names for the many gestalt and other codes.

```
#IFNDEF infglk_h; ! Standard Glulx definitions contributed by John Cater
Constant infglk_h;
!-----------------------------------------------------------------------------
!  infglk.h - an Inform library to allow easy access to glk functions
!    under glulx
!  Dynamically created by glk2inf.pl on 08/31/2006 at 19:20:21.
!  Send comments or suggestions to: katre@ruf.rice.edu
!-----------------------------------------------------------------------------
#Ifdef infglk_h;  ! remove "Constant declared but not used" warnings
#Endif;
Constant GLK_NULL 0;
! Constant definitions from glk.h
Constant gestalt_Version 0;
Constant gestalt_CharInput 1;
Constant gestalt_LineInput 2;
Constant gestalt_CharOutput 3;
Constant gestalt_CharOutput_CannotPrint 0;
Constant gestalt_CharOutput_ApproxPrint 1;
Constant gestalt_CharOutput_ExactPrint 2;
Constant gestalt_MouseInput 4;
Constant gestalt_Timer 5;
Constant gestalt_Graphics 6;
Constant gestalt_DrawImage 7;
Constant gestalt_Sound 8;
Constant gestalt_SoundVolume 9;
Constant gestalt_SoundNotify 10;
Constant gestalt_Hyperlinks 11;
Constant gestalt_HyperlinkInput 12;
Constant gestalt_SoundMusic 13;
Constant gestalt_GraphicsTransparency 14;
Constant gestalt_Unicode 15;
Constant evtype_None 0;
Constant evtype_Timer 1;
Constant evtype_CharInput 2;
Constant evtype_LineInput 3;
Constant evtype_MouseInput 4;
Constant evtype_Arrange 5;
Constant evtype_Redraw 6;
Constant evtype_SoundNotify 7;
Constant evtype_Hyperlink 8;
Constant keycode_Unknown $ffffffff;
Constant keycode_Left $fffffffe;
Constant keycode_Right $fffffffd;
Constant keycode_Up $fffffffc;
Constant keycode_Down $fffffffb;
Constant keycode_Return $fffffffa;
Constant keycode_Delete $fffffff9;
Constant keycode_Escape $fffffff8;
Constant keycode_Tab $fffffff7;
```

```
Constant keycode_PageUp $fffffff6;
Constant keycode_PageDown $fffffff5;
Constant keycode_Home $fffffff4;
Constant keycode_End $fffffff3;
Constant keycode_Func1 $ffffffef;
Constant keycode_Func2 $ffffffee;
Constant keycode_Func3 $ffffffed;
Constant keycode_Func4 $ffffffec;
Constant keycode_Func5 $ffffffeb;
Constant keycode_Func6 $ffffffea;
Constant keycode_Func7 $ffffffe9;
Constant keycode_Func8 $ffffffe8;
Constant keycode_Func9 $ffffffe7;
Constant keycode_Func10 $ffffffe6;
Constant keycode_Func11 $ffffffe5;
Constant keycode_Func12 $ffffffe4;
Constant keycode_MAXVAL 28;
Constant style_Normal 0;
Constant style_Emphasized 1;
Constant style_Preformatted 2;
Constant style_Header 3;
Constant style_Subheader 4;
Constant style_Alert 5;
Constant style_Note 6;
Constant style_BlockQuote 7;
Constant style_Input 8;
Constant style_User1 9;
Constant style_User2 10;
Constant style_NUMSTYLES 11;
Constant wintype_AllTypes 0;
Constant wintype_Pair 1;
Constant wintype_Blank 2;
Constant wintype_TextBuffer 3;
Constant wintype_TextGrid 4;
Constant wintype_Graphics 5;
Constant winmethod_Left $00;
Constant winmethod_Right $01;
Constant winmethod_Above $02;
Constant winmethod_Below $03;
Constant winmethod_DirMask $0f;
Constant winmethod_Fixed $10;
Constant winmethod_Proportional $20;
Constant winmethod_DivisionMask $f0;
Constant fileusage_Data $00;
Constant fileusage_SavedGame $01;
Constant fileusage_Transcript $02;
Constant fileusage_InputRecord $03;
Constant fileusage_TypeMask $0f;
Constant fileusage_TextMode $100;
Constant fileusage_BinaryMode $000;
Constant filemode_Write $01;
Constant filemode_Read $02;
Constant filemode_ReadWrite $03;
```

```
Constant filemode_WriteAppend $05;
Constant seekmode_Start 0;
Constant seekmode_Current 1;
Constant seekmode_End 2;
Constant stylehint_Indentation 0;
Constant stylehint_ParaIndentation 1;
Constant stylehint_Justification 2;
Constant stylehint_Size 3;
Constant stylehint_Weight 4;
Constant stylehint_Oblique 5;
Constant stylehint_Proportional 6;
Constant stylehint_TextColor 7;
Constant stylehint_BackColor 8;
Constant stylehint_ReverseColor 9;
Constant stylehint_NUMHINTS 10;
Constant stylehint_just_LeftFlush 0;
Constant stylehint_just_LeftRight 1;
Constant stylehint_just_Centered 2;
Constant stylehint_just_RightFlush 3;
Constant imagealign_InlineUp $01;
Constant imagealign_InlineDown $02;
Constant imagealign_InlineCenter $03;
Constant imagealign_MarginLeft $04;
Constant imagealign_MarginRight $05;
! The actual glk functions.
[ glk_exit _vararg_count ret;
! glk_exit ()
! And now the @glk call
@glk 1 _vararg_count ret;
return ret;
];
[ glk_set_interrupt_handler _vararg_count ret;
! glk_set_interrupt_handler (func)
! And now the @glk call
@glk 2 _vararg_count ret;
return ret;
];
[ glk_tick _vararg_count ret;
! glk_tick ()
! And now the @glk call
@glk 3 _vararg_count ret;
return ret;
];
[ glk_gestalt _vararg_count ret;
! glk_gestalt (sel val)
! And now the @glk call
@glk 4 _vararg_count ret;
return ret;
];
[ glk_gestalt_ext _vararg_count ret;
! glk_gestalt_ext (sel val arr arrlen)
! And now the @glk call
```

```
@glk 5 _vararg_count ret;
return ret;
];
[ glk_char_to_lower _vararg_count ret;
! glk_char_to_lower (ch)
! And now the @glk call
@glk 160 _vararg_count ret;
return ret;
];
[ glk_char_to_upper _vararg_count ret;
! glk_char_to_upper (ch)
! And now the @glk call
@glk 161 _vararg_count ret;
return ret;
];
[ glk_window_get_root _vararg_count ret;
! glk_window_get_root ()
! And now the @glk call
@glk 34 _vararg_count ret;
return ret;
];
[ glk_window_open _vararg_count ret;
! glk_window_open (split method size wintype rock)
! And now the @glk call
@glk 35 _vararg_count ret;
return ret;
];
[ glk_window_close _vararg_count ret;
! glk_window_close (win result)
! And now the @glk call
@glk 36 _vararg_count ret;
return ret;
];
[ glk_window_get_size _vararg_count ret;
! glk_window_get_size (win widthptr heightptr)
! And now the @glk call
@glk 37 _vararg_count ret;
return ret;
];
[ glk_window_set_arrangement _vararg_count ret;
! glk_window_set_arrangement (win method size keywin)
! And now the @glk call
@glk 38 _vararg_count ret;
return ret;
];
[ glk_window_get_arrangement _vararg_count ret;
! glk_window_get_arrangement (win methodptr sizeptr keywinptr)
! And now the @glk call
@glk 39 _vararg_count ret;
return ret;
];
```

```
[ glk_window_iterate _vararg_count ret;
! glk_window_iterate (win rockptr)
! And now the @glk call
@glk 32 _vararg_count ret;
return ret;
];
[ glk_window_get_rock _vararg_count ret;
! glk_window_get_rock (win)
! And now the @glk call
@glk 33 _vararg_count ret;
return ret;
];
[ glk_window_get_type _vararg_count ret;
! glk_window_get_type (win)
! And now the @glk call
@glk 40 _vararg_count ret;
return ret;
];
[ glk_window_get_parent _vararg_count ret;
! glk_window_get_parent (win)
! And now the @glk call
@glk 41 _vararg_count ret;
return ret;
];
[ glk_window_get_sibling _vararg_count ret;
! glk_window_get_sibling (win)
! And now the @glk call
@glk 48 _vararg_count ret;
return ret;
];
[ glk_window_clear _vararg_count ret;
! glk_window_clear (win)
! And now the @glk call
@glk 42 _vararg_count ret;
return ret;
];
[ glk_window_move_cursor _vararg_count ret;
! glk_window_move_cursor (win xpos ypos)
! And now the @glk call
@glk 43 _vararg_count ret;
return ret;
];
[ glk_window_get_stream _vararg_count ret;
! glk_window_get_stream (win)
! And now the @glk call
@glk 44 _vararg_count ret;
return ret;
];
[ glk_window_set_echo_stream _vararg_count ret;
! glk_window_set_echo_stream (win str)
! And now the @glk call
```

```
@glk 45 _vararg_count ret;
return ret;
];
[ glk_window_get_echo_stream _vararg_count ret;
! glk_window_get_echo_stream (win)
! And now the @glk call
@glk 46 _vararg_count ret;
return ret;
];
[ glk_set_window _vararg_count ret;
! glk_set_window (win)
! And now the @glk call
@glk 47 _vararg_count ret;
return ret;
];
[ glk_stream_open_file _vararg_count ret;
! glk_stream_open_file (fileref fmode rock)
! And now the @glk call
@glk 66 _vararg_count ret;
return ret;
];
[ glk_stream_open_memory _vararg_count ret;
! glk_stream_open_memory (buf buflen fmode rock)
! And now the @glk call
@glk 67 _vararg_count ret;
return ret;
];
[ glk_stream_close _vararg_count ret;
! glk_stream_close (str result)
! And now the @glk call
@glk 68 _vararg_count ret;
return ret;
];
[ glk_stream_iterate _vararg_count ret;
! glk_stream_iterate (str rockptr)
! And now the @glk call
@glk 64 _vararg_count ret;
return ret;
];
[ glk_stream_get_rock _vararg_count ret;
! glk_stream_get_rock (str)
! And now the @glk call
@glk 65 _vararg_count ret;
return ret;
];
[ glk_stream_set_position _vararg_count ret;
! glk_stream_set_position (str pos seekmode)
! And now the @glk call
@glk 69 _vararg_count ret;
return ret;
];
```

```
[ glk_stream_get_position _vararg_count ret;
! glk_stream_get_position (str)
! And now the @glk call
@glk 70 _vararg_count ret;
return ret;
];
[ glk_stream_set_current _vararg_count ret;
! glk_stream_set_current (str)
! And now the @glk call
@glk 71 _vararg_count ret;
return ret;
];
[ glk_stream_get_current _vararg_count ret;
! glk_stream_get_current ()
! And now the @glk call
@glk 72 _vararg_count ret;
return ret;
];
[ glk_put_char _vararg_count ret;
! glk_put_char (ch)
! And now the @glk call
@glk 128 _vararg_count ret;
return ret;
];
[ glk_put_char_stream _vararg_count ret;
! glk_put_char_stream (str ch)
! And now the @glk call
@glk 129 _vararg_count ret;
return ret;
];
[ glk_put_string _vararg_count ret;
! glk_put_string (s)
! And now the @glk call
@glk 130 _vararg_count ret;
return ret;
];
[ glk_put_string_stream _vararg_count ret;
! glk_put_string_stream (str s)
! And now the @glk call
@glk 131 _vararg_count ret;
return ret;
];
[ glk_put_buffer _vararg_count ret;
! glk_put_buffer (buf len)
! And now the @glk call
@glk 132 _vararg_count ret;
return ret;
];
[ glk_put_buffer_stream _vararg_count ret;
! glk_put_buffer_stream (str buf len)
! And now the @glk call
```

```
@glk 133 _vararg_count ret;
return ret;
];
[ glk_set_style _vararg_count ret;
! glk_set_style (styl)
! And now the @glk call
@glk 134 _vararg_count ret;
return ret;
];
[ glk_set_style_stream _vararg_count ret;
! glk_set_style_stream (str styl)
! And now the @glk call
@glk 135 _vararg_count ret;
return ret;
];
[ glk_get_char_stream _vararg_count ret;
! glk_get_char_stream (str)
! And now the @glk call
@glk 144 _vararg_count ret;
return ret;
];
[ glk_get_line_stream _vararg_count ret;
! glk_get_line_stream (str buf len)
! And now the @glk call
@glk 145 _vararg_count ret;
return ret;
];
[ glk_get_buffer_stream _vararg_count ret;
! glk_get_buffer_stream (str buf len)
! And now the @glk call
@glk 146 _vararg_count ret;
return ret;
];
[ glk_stylehint_set _vararg_count ret;
! glk_stylehint_set (wintype styl hint val)
! And now the @glk call
@glk 176 _vararg_count ret;
return ret;
];
[ glk_stylehint_clear _vararg_count ret;
! glk_stylehint_clear (wintype styl hint)
! And now the @glk call
@glk 177 _vararg_count ret;
return ret;
];
[ glk_style_distinguish _vararg_count ret;
! glk_style_distinguish (win styl1 styl2)
! And now the @glk call
@glk 178 _vararg_count ret;
return ret;
];
```

```
[ glk_style_measure _vararg_count ret;
! glk_style_measure (win styl hint result)
! And now the @glk call
@glk 179 _vararg_count ret;
return ret;
];
[ glk_fileref_create_temp _vararg_count ret;
! glk_fileref_create_temp (usage rock)
! And now the @glk call
@glk 96 _vararg_count ret;
return ret;
];
[ glk_fileref_create_by_name _vararg_count ret;
! glk_fileref_create_by_name (usage name rock)
! And now the @glk call
@glk 97 _vararg_count ret;
return ret;
];
[ glk_fileref_create_by_prompt _vararg_count ret;
! glk_fileref_create_by_prompt (usage fmode rock)
! And now the @glk call
@glk 98 _vararg_count ret;
return ret;
];
[ glk_fileref_create_from_fileref _vararg_count ret;
! glk_fileref_create_from_fileref (usage fref rock)
! And now the @glk call
@glk 104 _vararg_count ret;
return ret;
];
[ glk_fileref_destroy _vararg_count ret;
! glk_fileref_destroy (fref)
! And now the @glk call
@glk 99 _vararg_count ret;
return ret;
];
[ glk_fileref_iterate _vararg_count ret;
! glk_fileref_iterate (fref rockptr)
! And now the @glk call
@glk 100 _vararg_count ret;
return ret;
];
[ glk_fileref_get_rock _vararg_count ret;
! glk_fileref_get_rock (fref)
! And now the @glk call
@glk 101 _vararg_count ret;
return ret;
];
[ glk_fileref_delete_file _vararg_count ret;
! glk_fileref_delete_file (fref)
! And now the @glk call
```

```
@glk 102 _vararg_count ret;
return ret;
];
[ glk_fileref_does_file_exist _vararg_count ret;
! glk_fileref_does_file_exist (fref)
! And now the @glk call
@glk 103 _vararg_count ret;
return ret;
];
[ glk_select _vararg_count ret;
! glk_select (event)
! And now the @glk call
@glk 192 _vararg_count ret;
return ret;
];
[ glk_select_poll _vararg_count ret;
! glk_select_poll (event)
! And now the @glk call
@glk 193 _vararg_count ret;
return ret;
];
[ glk_request_timer_events _vararg_count ret;
! glk_request_timer_events (millisecs)
! And now the @glk call
@glk 214 _vararg_count ret;
return ret;
];
[ glk_request_line_event _vararg_count ret;
! glk_request_line_event (win buf maxlen initlen)
! And now the @glk call
@glk 208 _vararg_count ret;
return ret;
];
[ glk_request_char_event _vararg_count ret;
! glk_request_char_event (win)
! And now the @glk call
@glk 210 _vararg_count ret;
return ret;
];
[ glk_request_mouse_event _vararg_count ret;
! glk_request_mouse_event (win)
! And now the @glk call
@glk 212 _vararg_count ret;
return ret;
];
[ glk_cancel_line_event _vararg_count ret;
! glk_cancel_line_event (win event)
! And now the @glk call
@glk 209 _vararg_count ret;
return ret;
];
```

```
[ glk_cancel_char_event _vararg_count ret;
! glk_cancel_char_event (win)
! And now the @glk call
@glk 211 _vararg_count ret;
return ret;
];
[ glk_cancel_mouse_event _vararg_count ret;
! glk_cancel_mouse_event (win)
! And now the @glk call
@glk 213 _vararg_count ret;
return ret;
];
[ glk_buffer_to_lower_case_uni _vararg_count ret;
! glk_buffer_to_lower_case_uni (buf len numchars)
! And now the @glk call
@glk 288 _vararg_count ret;
return ret;
];
[ glk_buffer_to_upper_case_uni _vararg_count ret;
! glk_buffer_to_upper_case_uni (buf len numchars)
! And now the @glk call
@glk 289 _vararg_count ret;
return ret;
];
[ glk_buffer_to_title_case_uni _vararg_count ret;
! glk_buffer_to_title_case_uni (buf len numchars lowerrest)
! And now the @glk call
@glk 290 _vararg_count ret;
return ret;
];
[ glk_put_char_uni _vararg_count ret;
! glk_put_char_uni (ch)
! And now the @glk call
@glk 296 _vararg_count ret;
return ret;
];
[ glk_put_string_uni _vararg_count ret;
! glk_put_string_uni (s)
! And now the @glk call
@glk 297 _vararg_count ret;
return ret;
];
[ glk_put_buffer_uni _vararg_count ret;
! glk_put_buffer_uni (buf len)
! And now the @glk call
@glk 298 _vararg_count ret;
return ret;
];
[ glk_put_char_stream_uni _vararg_count ret;
! glk_put_char_stream_uni (str ch)
! And now the @glk call
```

```
@glk 299 _vararg_count ret;
return ret;
];
[ glk_put_string_stream_uni _vararg_count ret;
! glk_put_string_stream_uni (str s)
! And now the @glk call
@glk 300 _vararg_count ret;
return ret;
];
[ glk_put_buffer_stream_uni _vararg_count ret;
! glk_put_buffer_stream_uni (str buf len)
! And now the @glk call
@glk 301 _vararg_count ret;
return ret;
];
[ glk_get_char_stream_uni _vararg_count ret;
! glk_get_char_stream_uni (str)
! And now the @glk call
@glk 304 _vararg_count ret;
return ret;
];
[ glk_get_buffer_stream_uni _vararg_count ret;
! glk_get_buffer_stream_uni (str buf len)
! And now the @glk call
@glk 305 _vararg_count ret;
return ret;
];
[ glk_get_line_stream_uni _vararg_count ret;
! glk_get_line_stream_uni (str buf len)
! And now the @glk call
@glk 306 _vararg_count ret;
return ret;
];
[ glk_stream_open_file_uni _vararg_count ret;
! glk_stream_open_file_uni (fileref fmode rock)
! And now the @glk call
@glk 312 _vararg_count ret;
return ret;
];
[ glk_stream_open_memory_uni _vararg_count ret;
! glk_stream_open_memory_uni (buf buflen fmode rock)
! And now the @glk call
@glk 313 _vararg_count ret;
return ret;
];
[ glk_request_char_event_uni _vararg_count ret;
! glk_request_char_event_uni (win)
! And now the @glk call
@glk 320 _vararg_count ret;
return ret;
];
```

```
[ glk_request_line_event_uni _vararg_count ret;
! glk_request_line_event_uni (win buf maxlen initlen)
! And now the @glk call
@glk 321 _vararg_count ret;
return ret;
];
[ glk_image_draw _vararg_count ret;
! glk_image_draw (win image val1 val2)
! And now the @glk call
@glk 225 _vararg_count ret;
return ret;
];
[ glk_image_draw_scaled _vararg_count ret;
! glk_image_draw_scaled (win image val1 val2 width height)
! And now the @glk call
@glk 226 _vararg_count ret;
return ret;
];
[ glk_image_get_info _vararg_count ret;
! glk_image_get_info (image width height)
! And now the @glk call
@glk 224 _vararg_count ret;
return ret;
];
[ glk_window_flow_break _vararg_count ret;
! glk_window_flow_break (win)
! And now the @glk call
@glk 232 _vararg_count ret;
return ret;
];
[ glk_window_erase_rect _vararg_count ret;
! glk_window_erase_rect (win left top width height)
! And now the @glk call
@glk 233 _vararg_count ret;
return ret;
];
[ glk_window_fill_rect _vararg_count ret;
! glk_window_fill_rect (win color left top width height)
! And now the @glk call
@glk 234 _vararg_count ret;
return ret;
];
[ glk_window_set_background_color _vararg_count ret;
! glk_window_set_background_color (win color)
! And now the @glk call
@glk 235 _vararg_count ret;
return ret;
];
[ glk_schannel_create _vararg_count ret;
! glk_schannel_create (rock)
! And now the @glk call
```

```
@glk 242 _vararg_count ret;
return ret;
];
[ glk_schannel_destroy _vararg_count ret;
! glk_schannel_destroy (chan)
! And now the @glk call
@glk 243 _vararg_count ret;
return ret;
];
[ glk_schannel_iterate _vararg_count ret;
! glk_schannel_iterate (chan rockptr)
! And now the @glk call
@glk 240 _vararg_count ret;
return ret;
];
[ glk_schannel_get_rock _vararg_count ret;
! glk_schannel_get_rock (chan)
! And now the @glk call
@glk 241 _vararg_count ret;
return ret;
];
[ glk_schannel_play _vararg_count ret;
! glk_schannel_play (chan snd)
! And now the @glk call
@glk 248 _vararg_count ret;
return ret;
];
[ glk_schannel_play_ext _vararg_count ret;
! glk_schannel_play_ext (chan snd repeats notify)
! And now the @glk call
@glk 249 _vararg_count ret;
return ret;
];
[ glk_schannel_stop _vararg_count ret;
! glk_schannel_stop (chan)
! And now the @glk call
@glk 250 _vararg_count ret;
return ret;
];
[ glk_schannel_set_volume _vararg_count ret;
! glk_schannel_set_volume (chan vol)
! And now the @glk call
@glk 251 _vararg_count ret;
return ret;
];
[ glk_sound_load_hint _vararg_count ret;
! glk_sound_load_hint (snd flag)
! And now the @glk call
@glk 252 _vararg_count ret;
return ret;
];
```

```
[ glk_set_hyperlink _vararg_count ret;
! glk_set_hyperlink (linkval)
! And now the @glk call
@glk 256 _vararg_count ret;
return ret;
];
[ glk_set_hyperlink_stream _vararg_count ret;
! glk_set_hyperlink_stream (str linkval)
! And now the @glk call
@glk 257 _vararg_count ret;
return ret;
];
[ glk_request_hyperlink_event _vararg_count ret;
! glk_request_hyperlink_event (win)
! And now the @glk call
@glk 258 _vararg_count ret;
return ret;
];
[ glk_cancel_hyperlink_event _vararg_count ret;
! glk_cancel_hyperlink_event (win)
! And now the @glk call
@glk 259 _vararg_count ret;
return ret;
];
#ENDIF;
```

§**4. Rocks.** These are unique ID codes used to mark resources; think of them as inedible cookies.

```
Constant GG_MAINWIN_ROCK        201;
Constant GG_STATUSWIN_ROCK      202;
Constant GG_QUOTEWIN_ROCK       203;
Constant GG_SAVESTR_ROCK        301;
Constant GG_SCRIPTSTR_ROCK      302;
Constant GG_COMMANDWSTR_ROCK    303;
Constant GG_COMMANDRSTR_ROCK    304;
Constant GG_SCRIPTFREF_ROCK     401;
Constant GG_FOREGROUNDCHAN_ROCK 410;
Constant GG_BACKGROUNDCHAN_ROCK 411;
```

§**5. Stubs.** These are I6 library-style entry point routines, not used by I7, but retained in case I7 extensions want to do interesting things with Glulx.

```
#Stub HandleGlkEvent    2;
#Stub IdentifyGlkObject 4;
#Stub InitGlkWindow     1;
```

§**6. Starting Up.** `VM_Initialise()` is almost the first routine called, except that the "starting the virtual machine" activity is allowed to go first.

Arrangements are a little different here from on the Z-machine, because some data is retained in the case of a restart.

(Many thanks are due to Eliuk Blau, who found several tricky timing errors here and elsewhere in the Glulx-specific code. Frankly, I feel like hanging a sign on the following routines which reads "Congratulations on bringing light to the Dark Room.")

```
[ VM_Initialise res sty;
    @gestalt 4 2 res; ! Test if this interpreter has Glk...
    if (res == 0) quit; ! ...without which there would be nothing we could do

    unicode_gestalt_ok = false;
    if (glk_gestalt(gestalt_Unicode, 0))
        unicode_gestalt_ok = true;

    ! Set the VM's I/O system to be Glk.
    @setiosys 2 0;

    ! First, we must go through all the Glk objects that exist, and see
    ! if we created any of them. One might think this strange, since the
    ! program has just started running, but remember that the player might
    ! have just typed "restart".

    GGRecoverObjects();

    ! Sound channel initialisation, and RNG fixing, must be done now rather
    ! than later in case InitGlkWindow() returns a non-zero value.

    if (glk_gestalt(gestalt_Sound, 0)) {
        if (gg_foregroundchan == 0)
            gg_foregroundchan = glk_schannel_create(GG_FOREGROUNDCHAN_ROCK);
        if (gg_backgroundchan == 0)
            gg_backgroundchan = glk_schannel_create(GG_BACKGROUNDCHAN_ROCK);
    }

    #ifdef FIX_RNG;
    @random 10000 i;
    i = -i-2000;
    print "[Random number generator seed is ", i, "]^";
    @setrandom i;
    #endif; ! FIX_RNG

    res = InitGlkWindow(0);
    if (res ~= 0) return;

    ! Now, gg_mainwin and gg_storywin might already be set. If not, set them.

    if (gg_mainwin == 0) {
        ! Open the story window.
        res = InitGlkWindow(GG_MAINWIN_ROCK);
        if (res == 0) {
            ! Left-justify the header style
            glk_stylehint_set(wintype_TextBuffer, style_Header, stylehint_Justification, 0);
            ! Try to make emphasized type in italics and not boldface
            glk_stylehint_set(wintype_TextBuffer, style_Emphasized, stylehint_Weight, 0);
            glk_stylehint_set(wintype_TextBuffer, style_Emphasized, stylehint_Oblique, 1);
            gg_mainwin = glk_window_open(0, 0, 0, wintype_TextBuffer, GG_MAINWIN_ROCK);
        }
        if (gg_mainwin == 0) quit; ! If we can't even open one window, give in
```

```
    } else {
        ! There was already a story window. We should erase it.
        glk_window_clear(gg_mainwin);
    }
    if (gg_statuswin == 0) {
        res = InitGlkWindow(GG_STATUSWIN_ROCK);
        if (res == 0) {
            statuswin_cursize = statuswin_size;
            for (sty=0: sty<style_NUMSTYLES: sty++)
                glk_stylehint_set(wintype_TextGrid, sty, stylehint_ReverseColor, 1);
            gg_statuswin =
                glk_window_open(gg_mainwin, winmethod_Fixed + winmethod_Above,
                    statuswin_cursize, wintype_TextGrid, GG_STATUSWIN_ROCK);
        }
    }
    ! It's possible that the status window couldn't be opened, in which case
    ! gg_statuswin is now zero. We must allow for that later on.
    glk_set_window(gg_mainwin);
    InitGlkWindow(1);
];
[ GGRecoverObjects id;
    ! If GGRecoverObjects() has been called, all these stored IDs are
    ! invalid, so we start by clearing them all out.
    ! (In fact, after a restoreundo, some of them may still be good.
    ! For simplicity, though, we assume the general case.)
    gg_mainwin = 0;
    gg_statuswin = 0;
    gg_quotewin = 0;
    gg_scriptfref = 0;
    gg_scriptstr = 0;
    gg_savestr = 0;
    statuswin_cursize = 0;
    gg_foregroundchan = 0;
    gg_backgroundchan = 0;
    #Ifdef DEBUG;
    gg_commandstr = 0;
    gg_command_reading = false;
    #Endif; ! DEBUG
    ! Also tell the game to clear its object references.
    IdentifyGlkObject(0);
    id = glk_stream_iterate(0, gg_arguments);
    while (id) {
        switch (gg_arguments-->0) {
            GG_SAVESTR_ROCK: gg_savestr = id;
            GG_SCRIPTSTR_ROCK: gg_scriptstr = id;
            #Ifdef DEBUG;
            GG_COMMANDWSTR_ROCK: gg_commandstr = id;
                                 gg_command_reading = false;
            GG_COMMANDRSTR_ROCK: gg_commandstr = id;
                                 gg_command_reading = true;
            #Endif; ! DEBUG
            default: IdentifyGlkObject(1, 1, id, gg_arguments-->0);
```

```
        }
        id = glk_stream_iterate(id, gg_arguments);
    }
    id = glk_window_iterate(0, gg_arguments);
    while (id) {
        switch (gg_arguments-->0) {
            GG_MAINWIN_ROCK: gg_mainwin = id;
            GG_STATUSWIN_ROCK: gg_statuswin = id;
            GG_QUOTEWIN_ROCK: gg_quotewin = id;
            default: IdentifyGlkObject(1, 0, id, gg_arguments-->0);
        }
        id = glk_window_iterate(id, gg_arguments);
    }
    id = glk_fileref_iterate(0, gg_arguments);
    while (id) {
        switch (gg_arguments-->0) {
            GG_SCRIPTFREF_ROCK: gg_scriptfref = id;
            default: IdentifyGlkObject(1, 2, id, gg_arguments-->0);
        }
        id = glk_fileref_iterate(id, gg_arguments);
    }
    if (glk_gestalt(gestalt_Sound, 0)) {
        id = glk_schannel_iterate(0, gg_arguments);
        while (id) {
            switch (gg_arguments-->0) {
                GG_FOREGROUNDCHAN_ROCK: gg_foregroundchan = id;
                GG_BACKGROUNDCHAN_ROCK: gg_backgroundchan = id;
            }
            id = glk_schannel_iterate(id, gg_arguments);
        }
        if (gg_foregroundchan ~= 0) { glk_schannel_stop(gg_foregroundchan); }
        if (gg_backgroundchan ~= 0) { glk_schannel_stop(gg_backgroundchan); }
    }
    ! Tell the game to tie up any loose ends.
    IdentifyGlkObject(2);
];
```

§**7. Enable Acceleration.** This enables use of March 2009 extension to Glulx which optimises the speed of Inform-compiled story files by moving the work of I6 veneer routines into the interpreter itself. It should have no effect on earlier versions of the Glulx VM, which will lack the gestalt for this feature, but nor should it do any harm.

```
[ ENABLE_GLULX_ACCEL_R addr res;
    @gestalt 9 0 res;
    if (res == 0) return;
    addr = #classes_table;
    @accelparam 0 addr;
    @accelparam 1 INDIV_PROP_START;
    @accelparam 2 Class;
    @accelparam 3 Object;
    @accelparam 4 Routine;
    @accelparam 5 String;
    addr = #globals_array + WORDSIZE * #g$self;
    @accelparam 6 addr;
    @accelparam 7 NUM_ATTR_BYTES;
    addr = #cpv__start;
    @accelparam 8 addr;
    @accelfunc 1 Z__Region;
    @accelfunc 2 CP__Tab;
    @accelfunc 3 RA__Pr;
    @accelfunc 4 RL__Pr;
    @accelfunc 5 OC__Cl;
    @accelfunc 6 RV__Pr;
    @accelfunc 7 OP__Pr;
    rfalse;
];
```

§**8. Release Number.** Like all software, IF story files have release numbers to mark revised versions being circulated: unlike most software, and partly for traditional reasons, the version number is recorded not in some print statement or variable but is branded on, so to speak, in a specific memory location of the story file header.

`VM_Describe_Release()` describes the release and is used as part of the "banner", IF's equivalent to a title page.

```
[ VM_Describe_Release i;
    print "Release ";
    @aloads ROM_GAMERELEASE 0 i;
    print i;
    print " / Serial number ";
    for (i=0 : i<6 : i++) print (char) ROM_GAMESERIAL->i;
];
```

§**9. Keyboard Input.** The VM must provide three routines for keyboard input:

(a) `VM_KeyChar()` waits for a key to be pressed and then returns the character chosen as a ZSCII character.
(b) `VM_KeyDelay(N)` waits up to $N/10$ seconds for a key to be pressed, returning the ZSCII character if so, or 0 if not.
(c) `VM_ReadKeyboard(b, t)` reads a whole newline-terminated command into the buffer `b`, then parses it into a word stream in the table `t`.

There are elaborations to due with mouse clicks, but this isn't the place to document all of that.

```
[ VM_KeyChar win nostat done res ix jx ch;
    jx = ch; ! squash compiler warnings
    if (win == 0) win = gg_mainwin;
    if (gg_commandstr ~= 0 && gg_command_reading ~= false) {
        done = glk_get_line_stream(gg_commandstr, gg_arguments, 31);
        if (done == 0) {
            glk_stream_close(gg_commandstr, 0);
            gg_commandstr = 0;
            gg_command_reading = false;
            ! fall through to normal user input.
        } else {
            ! Trim the trailing newline
            if (gg_arguments->(done-1) == 10) done = done-1;
            res = gg_arguments->0;
            if (res == '\') {
                res = 0;
                for (ix=1 : ix<done : ix++) {
                    ch = gg_arguments->ix;
                    if (ch >= '0' && ch <= '9') {
                        @shiftl res 4 res;
                        res = res + (ch-'0');
                    } else if (ch >= 'a' && ch <= 'f') {
                        @shiftl res 4 res;
                        res = res + (ch+10-'a');
                    } else if (ch >= 'A' && ch <= 'F') {
                        @shiftl res 4 res;
                        res = res + (ch+10-'A');
                    }
                }
            }
            jump KCPContinue;
        }
    }
    done = false;
    glk_request_char_event(win);
    while (~~done) {
        glk_select(gg_event);
        switch (gg_event-->0) {
        5: ! evtype_Arrange
            if (nostat) {
                glk_cancel_char_event(win);
                res = $80000000;
                done = true;
                break;
            }
```

```
            DrawStatusLine();
        2: ! evtype_CharInput
            if (gg_event-->1 == win) {
                res = gg_event-->2;
                done = true;
                }
        }
        ix = HandleGlkEvent(gg_event, 1, gg_arguments);
        if (ix == 2) {
            res = gg_arguments-->0;
            done = true;
        } else if (ix == -1)  done = false;
    }
    if (gg_commandstr ~= 0 && gg_command_reading == false) {
        if (res < 32 || res >= 256 || (res == '\' or ' ')) {
            glk_put_char_stream(gg_commandstr, '\');
            done = 0;
            jx = res;
            for (ix=0 : ix<8 : ix++) {
                @ushiftr jx 28 ch;
                @shiftl jx 4 jx;
                ch = ch & $0F;
                if (ch ~= 0 || ix == 7) done = 1;
                if (done) {
                    if (ch >= 0 && ch <= 9) ch = ch + '0';
                    else                    ch = (ch - 10) + 'A';
                    glk_put_char_stream(gg_commandstr, ch);
                }
            }
        } else {
            glk_put_char_stream(gg_commandstr, res);
        }
        glk_put_char_stream(gg_commandstr, 10); ! newline
    }
.KCPContinue;
    return res;
];
[ VM_KeyDelay tenths  key done ix;
    glk_request_char_event(gg_mainwin);
    glk_request_timer_events(tenths*100);
    while (~~done) {
        glk_select(gg_event);
        ix = HandleGlkEvent(gg_event, 1, gg_arguments);
        if (ix == 2) {
            key = gg_arguments-->0;
            done = true;
        }
        else if (ix >= 0 && gg_event-->0 == 1 or 2) {
            key = gg_event-->2;
            done = true;
        }
    }
    glk_cancel_char_event(gg_mainwin);
```

```
    glk_request_timer_events(0);
    return key;
];
[ VM_ReadKeyboard  a_buffer a_table done ix;
    if (gg_commandstr ~= 0 && gg_command_reading ~= false) {
        done = glk_get_line_stream(gg_commandstr, a_buffer+WORDSIZE,
            (INPUT_BUFFER_LEN-WORDSIZE)-1);
        if (done == 0) {
            glk_stream_close(gg_commandstr, 0);
            gg_commandstr = 0;
            gg_command_reading = false;
            ! L__M(##CommandsRead, 5); would come after prompt
            ! fall through to normal user input.
        }
        else {
            ! Trim the trailing newline
            if ((a_buffer+WORDSIZE)->(done-1) == 10) done = done-1;
            a_buffer-->0 = done;
            VM_Style(INPUT_VMSTY);
            glk_put_buffer(a_buffer+WORDSIZE, done);
            VM_Style(NORMAL_VMSTY);
            print "^";
            jump KPContinue;
        }
    }
    done = false;
    glk_request_line_event(gg_mainwin, a_buffer+WORDSIZE, INPUT_BUFFER_LEN-WORDSIZE, 0);
    while (~~done) {
        glk_select(gg_event);
        switch (gg_event-->0) {
        5: ! evtype_Arrange
            DrawStatusLine();
        3: ! evtype_LineInput
            if (gg_event-->1 == gg_mainwin) {
                a_buffer-->0 = gg_event-->2;
                done = true;
            }
        }
        ix = HandleGlkEvent(gg_event, 0, a_buffer);
        if (ix == 2) done = true;
        else if (ix == -1) done = false;
    }
    if (gg_commandstr ~= 0 && gg_command_reading == false) {
        glk_put_buffer_stream(gg_commandstr, a_buffer+WORDSIZE, a_buffer-->0);
        glk_put_char_stream(gg_commandstr, 10); ! newline
    }
.KPContinue;
    VM_Tokenise(a_buffer,a_table);
    ! It's time to close any quote window we've got going.
    if (gg_quotewin) {
        glk_window_close(gg_quotewin, 0);
        gg_quotewin = 0;
    }
```

```
    #ifdef ECHO_COMMANDS;
    print "** ";
    for (ix=WORDSIZE: ix<(a_buffer-->0)+WORDSIZE: ix++) print (char) a_buffer->ix;
    print "^";
    #endif; ! ECHO_COMMANDS
];
```

§**10. Buffer Functions.** A "buffer", in this sense, is an array containing a stream of characters typed from the keyboard; a "parse buffer" is an array which resolves this into individual words, pointing to the relevant entries in the dictionary structure. Because each VM has its own format for each of these arrays (not to mention the dictionary), we have to provide some standard operations needed by the rest of the template as routines for each VM.

`VM_CopyBuffer(to, from)` copies one buffer into another.

`VM_Tokenise(buff, parse_buff)` takes the text in the buffer `buff` and produces the corresponding data in the parse buffer `parse_buff` – this is called tokenisation since the characters are divided into words: in traditional computing jargon, such clumps of characters treated syntactically as units are called tokens.

`LTI_Insert` is documented in the DM4 and the `LTI` prefix stands for "Language To Informese": it's used only by translations into non-English languages of play, and is not called in the template.

```
[ VM_CopyBuffer bto bfrom i;
    for (i=0: i<INPUT_BUFFER_LEN: i++) bto->i = bfrom->i;
];
[ VM_PrintToBuffer buf len a b c;
    if (b) {
        if (metaclass(a) == Object && a.#b == WORDSIZE
            && metaclass(a.b) == String)
            buf-->0 = Glulx_PrintAnyToArray(buf+WORDSIZE, len, a.b);
        else if (metaclass(a) == Routine)
            buf-->0 = Glulx_PrintAnyToArray(buf+WORDSIZE, len, a, b, c);
        else
            buf-->0 = Glulx_PrintAnyToArray(buf+WORDSIZE, len, a, b);
    }
    else if (metaclass(a) == Routine)
        buf-->0 = Glulx_PrintAnyToArray(buf+WORDSIZE, len, a, b, c);
    else
        buf-->0 = Glulx_PrintAnyToArray(buf+WORDSIZE, len, a);
    if (buf-->0 > len) buf-->0 = len;
    return buf-->0;
];
[ VM_Tokenise buf tab
    cx numwords len bx ix wx wpos wlen val res dictlen entrylen;
    len = buf-->0;
    buf = buf+WORDSIZE;
    ! First, split the buffer up into words. We use the standard Infocom
    ! list of word separators (comma, period, double-quote).
    cx = 0;
    numwords = 0;
    while (cx < len) {
        while (cx < len && buf->cx == ' ') cx++;
        if (cx >= len) break;
        bx = cx;
```

```
        if (buf->cx == '.' or ',' or '"') cx++;
        else {
            while (cx < len && buf->cx ~= ' ' or '.' or ',' or '"') cx++;
        }
        tab-->(numwords*3+2) = (cx-bx);
        tab-->(numwords*3+3) = WORDSIZE+bx;
        numwords++;
        if (numwords >= MAX_BUFFER_WORDS) break;
    }
    tab-->0 = numwords;

    ! Now we look each word up in the dictionary.

    dictlen = #dictionary_table-->0;
    entrylen = DICT_WORD_SIZE + 7;

    for (wx=0 : wx<numwords : wx++) {
        wlen = tab-->(wx*3+2);
        wpos = tab-->(wx*3+3);

        ! Copy the word into the gg_tokenbuf array, clipping to DICT_WORD_SIZE
        ! characters and lower case.
        if (wlen > DICT_WORD_SIZE) wlen = DICT_WORD_SIZE;
        cx = wpos - WORDSIZE;
        for (ix=0 : ix<wlen : ix++) gg_tokenbuf->ix = VM_UpperToLowerCase(buf->(cx+ix));
        for (: ix<DICT_WORD_SIZE : ix++) gg_tokenbuf->ix = 0;

        val = #dictionary_table + WORDSIZE;
        @binarysearch gg_tokenbuf DICT_WORD_SIZE val entrylen dictlen 1 1 res;
        tab-->(wx*3+1) = res;
    }
];

[ LTI_Insert i ch  b y;

    ! Protect us from strict mode, as this isn't an array in quite the
    ! sense it expects
    b = buffer;

    ! Insert character ch into buffer at point i.
    ! Being careful not to let the buffer possibly overflow:
    y = b-->0;
    if (y > INPUT_BUFFER_LEN) y = INPUT_BUFFER_LEN;

    ! Move the subsequent text along one character:
    for (y=y+WORDSIZE : y>i : y--) b->y = b->(y-1);
    b->i = ch;

    ! And the text is now one character longer:
    if (b-->0 < INPUT_BUFFER_LEN) (b-->0)++;
];
```

§**11. Dictionary Functions.** Again, the dictionary structure is differently arranged on the different VMs. This is a data structure containing, in compressed form, the text of all the words to be recognised by tokenisation (above). In I6 for Glulx, a dictionary word is represented at run-time by its record's address in the dictionary.

`VM_InvalidDictionaryAddress(A)` tests whether `A` is a valid record address in the dictionary data structure. In Glulx, dictionary records might in theory be anywhere in the 2 GB or so of possible memory, but we can rule out negative addresses. (This allows −1, say, to be used as a value meaning "not a valid dictionary word".)

`VM_DictionaryAddressToNumber(A)` and `VM_NumberToDictionaryAddress(N)` convert between word addresses and their run-time representations: since, on Glulx, they are the same, these are each the identity function.

```
[ VM_InvalidDictionaryAddress addr;
    if (addr < 0) rtrue;
    rfalse;
];
[ VM_DictionaryAddressToNumber w; return w; ];
[ VM_NumberToDictionaryAddress n; return n; ];
Array gg_tokenbuf -> DICT_WORD_SIZE;
[ GGWordCompare str1 str2 ix jx;
    for (ix=0 : ix<DICT_WORD_SIZE : ix++) {
        jx = (str1->ix) - (str2->ix);
        if (jx ~= 0) return jx;
    }
    return 0;
];
```

§**12. SHOWVERB support.** Further VM-specific tables cover actions and attributes, and these are used by the SHOWVERB testing command.

```
#Ifdef DEBUG;
[ DebugAction a str;
    if (a >= 4096) { print "<fake action ", a-4096, ">"; return; }
    if (a < 0 || a >= #identifiers_table-->7) print "<invalid action ", a, ">";
    else {
        str = #identifiers_table-->6;
        str = str-->a;
        if (str) print (string) str; else print "<unnamed action ", a, ">";
    }
];
[ DebugAttribute a str;
    if (a < 0 || a >= NUM_ATTR_BYTES*8) print "<invalid attribute ", a, ">";
    else {
        str = #identifiers_table-->4;
        str = str-->a;
        if (str) print (string) str; else print "<unnamed attribute ", a, ">";
    }
];
#Endif;
```

§**13. Command Tables.** The VM is also generated containing a data structure for the grammar produced by I6's `Verb` and `Extend` directives: this is essentially a list of command verbs such as DROP or PUSH, together with a list of synonyms, and then the grammar for the subsequent commands to be recognised by the parser.

```
[ VM_CommandTableAddress i;
    return (#grammar_table)-->(i+1);
];
[ VM_PrintCommandWords i wd j dictlen entrylen;
    dictlen = #dictionary_table-->0;
    entrylen = DICT_WORD_SIZE + 7;
    for (j=0 : j<dictlen : j++) {
        wd = #dictionary_table + WORDSIZE + entrylen*j;
        if (DictionaryWordToVerbNum(wd) == i)
            print "'", (address) wd, "' ";
    }
];
```

§**14. Random Number Generator.** No routine is needed for extracting a random number, since I6's built-in `random` function does that, but it's useful to abstract the process of seeding the RNG so that it produces a repeatable sequence of "random" numbers from here on: the necessary opcodes are different for the two VMs.

```
[ VM_Seed_RNG n;
    @setrandom n;
];
```

§**15. Memory Allocation.** This is dynamic memory allocation: something which is never practicable in the Z-machine, because the whole address range is already claimed, but which is viable on recent revisions of Glulx.

```
[ VM_AllocateMemory amount i;
    @gestalt 7 0 i;
    if (i == 0) return i;
    @malloc amount i;
    return i;
];
[ VM_FreeMemory address i;
    @gestalt 7 0 i;
    if (i == 0) return;
    @mfree address;
];
```

§**16. Audiovisual Resources.** The Z-machine only barely supports figures and sound effects, so Glulx is the preferred VM to choose if they are wanted. Properly speaking, it's not Glulx which supports these, but its I/O layer Glk, and implementations of Glk are free to support them or not as they please: "cheapglk", a dumb terminal version, does not, for instance. We therefore have to investigate the "gestalt" to find out.

```
[ VM_Picture resource_ID;
    if (glk_gestalt(gestalt_Graphics, 0)) {
        glk_image_draw(gg_mainwin, resource_ID, imagealign_InlineCenter, 0);
    } else {
        print "[Picture number ", resource_ID, " here.]^";
    }
];
[ VM_SoundEffect resource_ID;
    if (glk_gestalt(gestalt_Sound, 0)) {
        glk_schannel_play(gg_foregroundchan, resource_ID);
    } else {
        print "[Sound effect number ", resource_ID, " here.]^";
    }
];
```

§**17. Typography.** Glk makes an attempt to present typographic styles as being a matter of semantic markup rather than controlling the actual appearance of text: the idea is that the story file should want to print something in a heading kind of way, and then the interpreter – guided by the player's reading preferences – might set that in bold, or larger type, or red ink, or any combination of the three, or with other effects entirely. This is not the place to discuss whether that was a wise decision for Glk to take (it really, really, *really* wasn't): we can only play along.

```
[ VM_Style sty;
    switch (sty) {
        NORMAL_VMSTY:     glk_set_style(style_Normal);
        HEADER_VMSTY:     glk_set_style(style_Header);
        SUBHEADER_VMSTY:  glk_set_style(style_Subheader);
        NOTE_VMSTY:       glk_set_style(style_Note);
        ALERT_VMSTY:      glk_set_style(style_Alert);
        BLOCKQUOTE_VMSTY: glk_set_style(style_BlockQuote);
        INPUT_VMSTY:      glk_set_style(style_Input);
    }
];
```

§**18. Character Casing.** The following are the equivalent of `tolower` and `toupper`, the traditional C library functions for forcing letters into lower and upper case form, for the ZSCII character set. Note that Glulx can also use Unicode characters for some purposes (Unicode was a relatively late addition to the Glulx standard), and we make good use of this when storing indexed text.

```
[ VM_UpperToLowerCase c; return glk_char_to_lower(c); ];
[ VM_LowerToUpperCase c; return glk_char_to_upper(c); ];
```

§**19. Glulx-Only Printing Routines.** Partly because of the smallness of the range of representable values in the Z-machine, there is little run-time type-checking that can be done: for instance a dictionary address cannot be distinguished from a function address because they are encoded differently, so that a function address (which is packed) could well coincide with that of a dictionary word (which is not). On Glulx these restrictions are somewhat lifted, so that it's possible to write a routine which can look at a value, work out what it must mean, and print it suitably. This is only possible up to a point – for instance, it can't distinguish an integer from a function address – and in I7 the use of this sort of trick is much less important because type-checking in the NI compiler handles the problem much better. Still, we retain some Glulx-only features because they are convenient for writing external files to disc, for instance, something which the Z-machine can't do in any case.

`Glulx_PrintAnything` handles strings, functions (with optional arguments), objects, object properties (with optional arguments), and dictionary words.

`Glulx_PrintAnyToArray` does the same, but the output is sent to a byte array in memory. The first two arguments must be the array address and length; subsequent arguments are as for `Glulx_PrintAnything`. The return value is the number of characters output. If the output is longer than the array length given, the extra characters are discarded, so the array does not overflow. (However, the return value is the total length of the output, including discarded characters.) The character set stored here is ZSCII, not Unicode.

`Glulx_ChangeAnyToCString` calls `Glulx_PrintAnyToArray` on a particular array, then amends the result to make it a C-style string – that is, a sequence of byte-sized characters which are null terminated. The character set stored here is once again ZSCII, not Unicode.

```
! Glulx_PrintAnything()                    <nothing printed>
! Glulx_PrintAnything(0)                   <nothing printed>
! Glulx_PrintAnything("string");           print (string) "string";
! Glulx_PrintAnything('word')              print (address) 'word';
! Glulx_PrintAnything(obj)                 print (name) obj;
! Glulx_PrintAnything(obj, prop)           obj.prop();
! Glulx_PrintAnything(obj, prop, args...)  obj.prop(args...);
! Glulx_PrintAnything(func)                func();
! Glulx_PrintAnything(func, args...)       func(args...);
[ Glulx_PrintAnything _vararg_count obj mclass;
    if (_vararg_count == 0) return;
    @copy sp obj;
    _vararg_count--;
    if (obj == 0) return;
    if (obj->0 == $60) {
        ! Dictionary word. Metaclass() can't catch this case, so we do it manually
        print (address) obj;
        return;
    }
    mclass = metaclass(obj);
    switch (mclass) {
    nothing:
        return;
    String:
        print (string) obj;
        return;
    Routine:
        ! Call the function with all the arguments which are already
        ! on the stack.
        @call obj _vararg_count 0;
        return;
```

```
    Object:
        if (_vararg_count == 0) {
            print (name) obj;
        }
        else {
            ! Push the object back onto the stack, and call the
            ! veneer routine that handles obj.prop() calls.
            @copy obj sp;
            _vararg_count++;
            @call CA__Pr _vararg_count 0;
        }
        return;
    }
];
[ Glulx_PrintAnyToArray _vararg_count arr arrlen str oldstr len;
    @copy sp arr;
    @copy sp arrlen;
    _vararg_count = _vararg_count - 2;
    oldstr = glk_stream_get_current();
    str = glk_stream_open_memory(arr, arrlen, 1, 0);
    if (str == 0) return 0;
    glk_stream_set_current(str);
    @call Glulx_PrintAnything _vararg_count 0;
    glk_stream_set_current(oldstr);
    @copy $ffffffff sp;
    @copy str sp;
    @glk $0044 2 0; ! stream_close
    @copy sp len;
    @copy sp 0;
    return len;
];
Constant GG_ANYTOSTRING_LEN 66;
Array AnyToStrArr -> GG_ANYTOSTRING_LEN+1;
[ Glulx_ChangeAnyToCString _vararg_count ix len;
    ix = GG_ANYTOSTRING_LEN-2;
    @copy ix sp;
    ix = AnyToStrArr+1;
    @copy ix sp;
    ix = _vararg_count+2;
    @call Glulx_PrintAnyToArray ix len;
    AnyToStrArr->0 = $E0;
    if (len >= GG_ANYTOSTRING_LEN)
        len = GG_ANYTOSTRING_LEN-1;
    AnyToStrArr->(len+1) = 0;
    return AnyToStrArr;
];
```

§**20. The Screen.** Our generic screen model is that the screen is made up of windows: we tend to refer only to two of these, the main window and the status line, but others may also exist from time to time. Windows have unique ID numbers: the special window ID −1 means "all windows" or "the entire screen", which usually amounts to the same thing.

Screen height and width are measured in characters, with respect to the fixed-pitch font used for the status line. The main window normally contains variable-pitch text which may even have been kerned, and character dimensions make little sense there.

```
[ VM_ClearScreen window;
    if (window == WIN_ALL or WIN_MAIN) {
        glk_window_clear(gg_mainwin);
        if (gg_quotewin) {
            glk_window_close(gg_quotewin, 0);
            gg_quotewin = 0;
        }
    }
    if (gg_statuswin && window == WIN_ALL or WIN_STATUS) glk_window_clear(gg_statuswin);
];
[ VM_ScreenWidth  id;
    id=gg_mainwin;
    if (gg_statuswin && statuswin_current) id = gg_statuswin;
    glk_window_get_size(id, gg_arguments, 0);
    return gg_arguments-->0;
];
[ VM_ScreenHeight;
    glk_window_get_size(gg_mainwin, 0, gg_arguments);
    return gg_arguments-->0;
];
```

§**21. Window Colours.** Our generic screen model is that the screen is made up of windows, each of which can have its own foreground and background colours.

The colour of individual letters or words of type is not controllable in Glulx, to the frustration of many, and so the template layer of I7 has no framework for handling this (even though it is controllable on the Z-machine, which is greatly superior in this respect).

```
[ VM_SetWindowColours f b window doclear  i fwd bwd swin;
    if (clr_on && f && b) {
        if (window) swin = 5-window; ! 4 for TextGrid, 3 for TextBuffer
        fwd = MakeColourWord(f);
        bwd = MakeColourWord(b);
        for (i=0 : i<style_NUMSTYLES: i++) {
            if (f == CLR_DEFAULT || b == CLR_DEFAULT) {  ! remove style hints
                glk_stylehint_clear(swin, i, stylehint_TextColor);
                glk_stylehint_clear(swin, i, stylehint_BackColor);
            } else {
                glk_stylehint_set(swin, i, stylehint_TextColor, fwd);
                glk_stylehint_set(swin, i, stylehint_BackColor, bwd);
            }
        }
        ! Now re-open the windows to apply the hints
        if (gg_statuswin) glk_window_close(gg_statuswin, 0);
```

```
        gg_statuswin = 0;
        if (doclear || ( window ~= 1 && (clr_fg ~= f || clr_bg ~= b) ) ) {
            glk_window_close(gg_mainwin, 0);
            gg_mainwin = glk_window_open(0, 0, 0, wintype_TextBuffer, GG_MAINWIN_ROCK);
            if (gg_scriptstr ~= 0)
                glk_window_set_echo_stream(gg_mainwin, gg_scriptstr);
        }
        gg_statuswin =
            glk_window_open(gg_mainwin, winmethod_Fixed + winmethod_Above,
                statuswin_cursize, wintype_TextGrid, GG_STATUSWIN_ROCK);
        if (statuswin_current && gg_statuswin) VM_MoveCursorInStatusLine(); else VM_MainWindow();
        if (window ~= 2) {
            clr_fgstatus = f;
            clr_bgstatus = b;
        }
        if (window ~= 1) {
            clr_fg = f;
            clr_bg = b;
        }
    }
];
[ VM_RestoreWindowColours; ! used after UNDO: compare I6 patch L61007
    if (clr_on) { ! check colour has been used
        VM_SetWindowColours(clr_fg, clr_bg, 2); ! make sure both sets of variables are restored
        VM_SetWindowColours(clr_fgstatus, clr_bgstatus, 1, true);
        VM_ClearScreen();
    }
];
[ MakeColourWord c;
    if (c > 9) return c;
    c = c-2;
    return $ff0000*(c&1) + $ff00*(c&2 ~= 0) + $ff*(c&4 ~= 0);
];
```

§**22. Main Window.** The part of the screen on which commands and responses are printed, which ordinarily occupies almost all of the screen area.

`VM_MainWindow()` switches printing back from another window, usually the status line, to the main window.

```
[ VM_MainWindow;
    glk_set_window(gg_mainwin); ! set_window
    statuswin_current=0;
];
```

§**23. Status Line.** Despite the name, the status line need not be a single line at the top of the screen: that's only the conventional default arrangement. It can expand to become the equivalent of an old-fashioned VT220 terminal, with menus and grids and mazes displayed lovingly in character graphics, or it can close up to invisibility.

`VM_StatusLineHeight(n)` sets the status line to have a height of `n` lines of type. (The width of the status line is always the width of the whole screen, and the position is always at the top, so the height is the only controllable aspect.) The $n = 0$ case makes the status line disappear.

`VM_MoveCursorInStatusLine(x, y)` switches printing to the status line, positioning the "cursor" – the position at which printing will begin – at the given character grid position (x, y). Line 1 represents the top line; line 2 is underneath, and so on; columns are similarly numbered from 1 at the left.

```
[ VM_StatusLineHeight hgt;
    if (gg_statuswin == 0) return;
    if (hgt == statuswin_cursize) return;
    glk_window_set_arrangement(glk_window_get_parent(gg_statuswin), $12, hgt, 0);
    statuswin_cursize = hgt;
];
[ VM_MoveCursorInStatusLine line column;
    if (gg_statuswin == 0) return;
    glk_set_window(gg_statuswin);
    if (line == 0) { line = 1; column = 1; }
    glk_window_move_cursor(gg_statuswin, column-1, line-1);
    statuswin_current=1;
];
```

§**24. Quotation Boxes.** On the Z-machine, quotation boxes are produced by stretching the status line, but on Glulx they usually occupy windows of their own. If it isn't possible to create such a window, so that `gg_quotewin` is zero below, the quotation text just appears in the main window.

```
[ Box__Routine maxwid arr ix lines lastnl parwin;
    maxwid = 0; ! squash compiler warning
    lines = arr-->0;
    if (gg_quotewin == 0) {
        gg_arguments-->0 = lines;
        ix = InitGlkWindow(GG_QUOTEWIN_ROCK);
        if (ix == 0)
            gg_quotewin =
                glk_window_open(gg_mainwin, winmethod_Fixed + winmethod_Above,
                    lines, wintype_TextBuffer, GG_QUOTEWIN_ROCK);
    } else {
        parwin = glk_window_get_parent(gg_quotewin);
        glk_window_set_arrangement(parwin, $12, lines, 0);
    }
    lastnl = true;
    if (gg_quotewin) {
        glk_window_clear(gg_quotewin);
        glk_set_window(gg_quotewin);
        lastnl = false;
    }
    VM_Style(BLOCKQUOTE_VMSTY);
    for (ix=0 : ix<lines : ix++) {
```

```
        print (string) arr-->(ix+1);
        if (ix < lines-1 || lastnl) new_line;
    }
    VM_Style(NORMAL_VMSTY);
    if (gg_quotewin) glk_set_window(gg_mainwin);
];
```

§**25. GlkList Command.** GLKLIST is a testing command best used by those who understand Glulx and its ways: it isn't documented in the I7 manual, because it is pretty inscrutable for "real" users, but it's probably worth keeping just the same.

```
#Ifdef DEBUG;
[ GlkListSub id val;
    id = glk_window_iterate(0, gg_arguments);
    while (id) {
        print "Window ", id, " (", gg_arguments-->0, "): ";
        val = glk_window_get_type(id);
        switch (val) {
        1: print "pair";
        2: print "blank";
        3: print "textbuffer";
        4: print "textgrid";
        5: print "graphics";
        default: print "unknown";
        }
        val = glk_window_get_parent(id);
        if (val) print ", parent is window ", val;
        else     print ", no parent (root)";
        val = glk_window_get_stream(id);
        print ", stream ", val;
        val = glk_window_get_echo_stream(id);
        if (val) print ", echo stream ", val;
        print "^";
        id = glk_window_iterate(id, gg_arguments);
    }
    id = glk_stream_iterate(0, gg_arguments);
    while (id) {
        print "Stream ", id, " (", gg_arguments-->0, ")^";
        id = glk_stream_iterate(id, gg_arguments);
    }
    id = glk_fileref_iterate(0, gg_arguments);
    while (id) {
        print "Fileref ", id, " (", gg_arguments-->0, ")^";
        id = glk_fileref_iterate(id, gg_arguments);
    }
    if (glk_gestalt(gestalt_Sound, 0)) {
        id = glk_schannel_iterate(0, gg_arguments);
        while (id) {
            print "Soundchannel ", id, " (", gg_arguments-->0, ")^";
            id = glk_schannel_iterate(id, gg_arguments);
        }
    }
```

```
];
{-testing-command:glklist}
    *                                           -> Glklist;
#Endif;
```

§**26. Undo.** These are really emulations of the Z-machine's conventions on UNDO: Glulx's undo opcodes used different result codes while providing essentially the same functionality, for reasons which are opaque, but no trouble is caused thereby.

```
[ VM_Undo result_code;
    @restoreundo result_code;
    return (~~result_code);
];
[ VM_Save_Undo result_code;
    @saveundo result_code;
    if (result_code == -1) { GGRecoverObjects(); return 2; }
    return (~~result_code);
];
```

§27. Quit The Game Rule.

```
[ QUIT_THE_GAME_R;
    if (actor ~= player) rfalse;
    GL__M(##Quit, 2); if (YesOrNo()~=0) quit;
];
```

§28. Restart The Game Rule.

```
[ RESTART_THE_GAME_R;
    if (actor ~= player) rfalse;
    GL__M(##Restart, 1);
    if (YesOrNo() ~= 0) {
        @restart;
        GL__M(##Restart, 2);
    }
];
```

§29. Restore The Game Rule.

```
[ RESTORE_THE_GAME_R res fref;
    if (actor ~= player) rfalse;
    fref = glk_fileref_create_by_prompt($01, $02, 0);
    if (fref == 0) jump RFailed;
    gg_savestr = glk_stream_open_file(fref, $02, GG_SAVESTR_ROCK);
    glk_fileref_destroy(fref);
    if (gg_savestr == 0) jump RFailed;
    @restore gg_savestr res;
    glk_stream_close(gg_savestr, 0);
    gg_savestr = 0;
    .RFailed;
    GL__M(##Restore, 1);
];
```

§30. Save The Game Rule.

```
[ SAVE_THE_GAME_R res fref;
    if (actor ~= player) rfalse;
    fref = glk_fileref_create_by_prompt($01, $01, 0);
    if (fref == 0) jump SFailed;
    gg_savestr = glk_stream_open_file(fref, $01, GG_SAVESTR_ROCK);
    glk_fileref_destroy(fref);
    if (gg_savestr == 0) jump SFailed;
    @save gg_savestr res;
    if (res == -1) {
        ! The player actually just typed "restore". We're going to print
        !  GL__M(##Restore,2); the Z-Code Inform library does this correctly
        ! now. But first, we have to recover all the Glk objects; the values
        ! in our global variables are all wrong.
        GGRecoverObjects();
        glk_stream_close(gg_savestr, 0); ! stream_close
        gg_savestr = 0;
        return GL__M(##Restore, 2);
    }
    glk_stream_close(gg_savestr, 0); ! stream_close
    gg_savestr = 0;
    if (res == 0) return GL__M(##Save, 2);
    .SFailed;
    GL__M(##Save, 1);
];
```

§31. Verify The Story File Rule. This is a fossil now, really, but in the days of Infocom, the 110K story file occupying an entire disc was a huge data set: floppy discs were by no means a reliable medium, and cheap hardware often used hit-and-miss components, as on the notorious Commodore 64 disc controller. If somebody experienced an apparent bug in play, it could easily be that he had a corrupt disc or was unable to read data of that density. So the VERIFY command, which took up to ten minutes on some early computers, would chug through the entire story file and compute a checksum, compare it against a known result in the header, and determine that the story file could or could not properly be read. The Z-machine provided this service as an opcode, and so Glulx followed suit.

```
[ VERIFY_THE_STORY_FILE_R res;
    if (actor ~= player) rfalse;
    @verify res;
    if (res == 0) return GL__M(##Verify, 1);
    GL__M(##Verify, 2);
];
```

§32. Switch Transcript On Rule.

```
[ SWITCH_TRANSCRIPT_ON_R;
    if (actor ~= player) rfalse;
    if (gg_scriptstr ~= 0) return GL__M(##ScriptOn, 1);
    if (gg_scriptfref == 0) {
        gg_scriptfref = glk_fileref_create_by_prompt($102, $05, GG_SCRIPTFREF_ROCK);
        if (gg_scriptfref == 0) jump S1Failed;
    }
    ! stream_open_file
    gg_scriptstr = glk_stream_open_file(gg_scriptfref, $05, GG_SCRIPTSTR_ROCK);
    if (gg_scriptstr == 0) jump S1Failed;
    glk_window_set_echo_stream(gg_mainwin, gg_scriptstr);
    GL__M(##ScriptOn, 2);
    VersionSub();
    return;
    .S1Failed;
    GL__M(##ScriptOn, 3);
];
```

§33. Switch Transcript Off Rule.

```
[ SWITCH_TRANSCRIPT_OFF_R;
    if (actor ~= player) rfalse;
    if (gg_scriptstr == 0) return GL__M(##ScriptOff,1);
    GL__M(##ScriptOff, 2);
    glk_stream_close(gg_scriptstr, 0); ! stream_close
    gg_scriptstr = 0;
];
```

§34. Announce Story File Version Rule.

```
[ ANNOUNCE_STORY_FILE_VERSION_R ix;
    if (actor ~= player) rfalse;
    Banner();
    print "Identification number: ";
    for (ix=6: ix <= UUID_ARRAY->0: ix++) print (char) UUID_ARRAY->ix;
    print "^";
    @gestalt 1 0 ix;
    print "Interpreter version ", ix / $10000, ".", (ix & $FF00) / $100,
    ".", ix & $FF, " / ";
    @gestalt 0 0 ix;
    print "VM ", ix / $10000, ".", (ix & $FF00) / $100, ".", ix & $FF, " / ";
    print "Library serial number ", (string) LibSerial, "^";
    #Ifdef LanguageVersion;
    print (string) LanguageVersion, "^";
    #Endif; ! LanguageVersion
    ShowExtensionVersions();
    say__p = 1;
];
```

§**35. Descend To Specific Action Rule.** There are 100 or so actions, typically, and this rule is for efficiency's sake: rather than perform 100 or so comparisons to see which routine to call, we indirect through a jump table. The routines called are the `-Sub` routines: thus, for instance, if `action` is `##Wait` then `WaitSub` is called. It is essential that this routine not be called for fake actions: in I7 use this is guaranteed, since fake actions are not allowed into the action machinery at all.

Strangely, Glulx's action routines table is numbered in an off-by-one way compared to the Z-machine's: hence the `+1`.

```
[ DESCEND_TO_SPECIFIC_ACTION_R;
    indirect(#actions_table-->(action+1));
    rtrue;
];
```

FileIO Template B/iot

Purpose

Reading and writing external files, in the Glulx virtual machine only.

---

§**1. Structure.** The I7 kind of value "auxiliary-file" is an `-->` array, holding a memory structure containing information about external files. The following constants specify memory offsets and values. Note the safety value stored as the first word of the structure: this helps protect the routines below from accidents. (16339, besides being prime, is a number interesting to the author of Inform since it was the examination board identifying number of his school, and so had to be filled in on all of the many papers he sat during his formative years.)

```
Constant AUXF_MAGIC = 0; ! First word holds a safety constant
Constant AUXF_MAGIC_VALUE = 16339; ! Should be first word of any valid file structure
Constant AUXF_STATUS = 1; ! One of the following:
    Constant AUXF_STATUS_IS_UNUSED = 0; ! Never used
    Constant AUXF_STATUS_IS_CLOSED = 1; ! Currently closed
    Constant AUXF_STATUS_IS_OPEN_FOR_READ = 2;
    Constant AUXF_STATUS_IS_OPEN_FOR_WRITE = 3;
    Constant AUXF_STATUS_IS_OPEN_FOR_APPEND = 4;
Constant AUXF_BINARY = 2; ! False for text files (I7 default), true for binary
Constant AUXF_FREF = 3; ! Glulx file reference number (if file has been used)
Constant AUXF_STREAM = 4; ! Stream for an open file (meaningless otherwise)
Constant AUXF_FILENAME = 5; ! Packed address of constant string
Constant AUXF_IFID_OF_OWNER = 6; ! UUID_ARRAY if owned by this project, or
    ! string array of IFID of owner wrapped in //...//, or NULL to leave open
```

§**2. Instances.** These structures are not dynamically created: they are precompiled by the NI compiler, already filled in with the necessary values. The following command generates them.

```
{-call:external_file_arrays}
```

§**3. Errors.** This is used for I/O errors of all kinds: it isn't within the Glulx-only code because one of the errors is to try to use these routines on the Z-machine.

```
[ FileIO_Error auxf err_text;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE)) {
        print "^*** Error on unknown file: ", (string) err_text, " ***^";
    } else
        print "^*** Error on file '", (string) auxf-->AUXF_FILENAME, "': ",
            (string) err_text, " ***^";
    RunTimeProblem(RTP_FILEIOERROR);
    return 0;
];
```

§**4. Glulx Material.**

```
#IFDEF TARGET_GLULX;
```

§**5. Obtain File Reference.** Obtain the file reference for a file (which need not exist). `auxf` is the memory location of the auxiliary file structure; a file reference is a Glulx concept.

```
[ FileIO_Fref auxf  fref usage;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE)) rfalse;
    if (auxf-->AUXF_STATUS == 0) {
        if (auxf-->AUXF_BINARY) usage = fileusage_BinaryMode;
        else usage = fileusage_TextMode;
        auxf-->AUXF_FREF =
            glk_fileref_create_by_name(fileusage_Data + usage,
                Glulx_ChangeAnyToCString(auxf-->AUXF_FILENAME), 0);
        auxf-->AUXF_STATUS = AUXF_STATUS_IS_CLOSED;
    }
    return auxf-->AUXF_FREF;
];
```

§**6. Existence.** Determine whether a file exists on disc. Note that we have no concept of directories, or the file system structure on the host machine: indeed, it is entirely up to the Glulx VM what it does when asked to look for a file. By convention, though, files for a project are stored in the same folder as the story file when out in the wild; when a project is developed within the Inform user interface, they are either (for preference) stored in a `Files` subfolder of the `Materials` folder for a project, or else stored alongside the Inform project file.

```
[ FileIO_Exists auxf  fref;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE)) rfalse;
    fref = FileIO_Fref(auxf);
    return glk_fileref_does_file_exist(fref);
];
```

§**7. Readiness.** One of our problems is that a file might be being used by another application: perhaps even by another story file running in a second incarnation of Glulx, like a parallel world of which we can know nothing. We actually want to allow for this sort of thing, because one use for external files in I7 is as a sort of communications conduit for assisting applications.

Most operating systems solve this problem by means of locking a file, or by creating a second lock-file, the existence of which indicates ownership of the original. We haven't got much access to the file-system, though: what we do is to set the first character of the file to an asterisk to mark it as complete and ready for reading, or to a hyphen to mark it as a work in progress.

`FileIO_Ready` determines whether or not a file is ready to be read from: it has to exist on disc, and to be openable, and also to be ready in having this marker asterisk.

`FileIO_MarkReady` changes the readiness state of a file, writing the asterisk or hyphen into the initial character as needed.

```
[ FileIO_Ready auxf  fref str ch;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE)) rfalse;
    fref = FileIO_Fref(auxf);
    if (glk_fileref_does_file_exist(fref) == false) rfalse;
    str = glk_stream_open_file(fref, filemode_Read, 0);
```

```
    ch = glk_get_char_stream(str);
    glk_stream_close(str, 0);
    if (ch ~= '*') rfalse;
    rtrue;
];
[ FileIO_MarkReady auxf readiness  fref str ch;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE))
        return FileIO_Error(auxf, "tried to open a non-file");
    fref = FileIO_Fref(auxf);
    if (auxf-->AUXF_STATUS ~= AUXF_STATUS_IS_CLOSED)
        return FileIO_Error(auxf, "only closed files can be marked");
    str = glk_stream_open_file(fref, filemode_ReadWrite, 0);
    glk_stream_set_position(str, 0, 0); ! seek start
    if (readiness) ch = '*'; else ch = '-';
    glk_put_char_stream(str, ch); ! mark as complete
    glk_stream_close(str, 0);
];
```

§8. Open File.

```
[ FileIO_Open auxf write_flag append_flag
    fref str mode ix ch not_this_ifid owner force_header;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE))
        return FileIO_Error(auxf, "tried to open a non-file");
    fref = FileIO_Fref(auxf);
    if (auxf-->AUXF_STATUS ~= AUXF_STATUS_IS_CLOSED)
        return FileIO_Error(auxf, "tried to open a file already open");
    if (write_flag) {
        if (append_flag) {
            mode = filemode_WriteAppend;
            if (glk_fileref_does_file_exist(fref) == false)
                force_header = true;
        }
        else mode = filemode_Write;
    } else {
        mode = filemode_Read;
        if (glk_fileref_does_file_exist(fref) == false)
            return FileIO_Error(auxf, "tried to open a file which does not exist");
    }
    str = glk_stream_open_file(fref, mode, 0);
    if (str == 0)
        return FileIO_Error(auxf, "tried to open a file but failed");
    auxf-->AUXF_STREAM = str;
    if (write_flag) {
        if (append_flag)
            auxf-->AUXF_STATUS = AUXF_STATUS_IS_OPEN_FOR_APPEND;
        else
            auxf-->AUXF_STATUS = AUXF_STATUS_IS_OPEN_FOR_WRITE;
        glk_stream_set_current(str);
        if ((append_flag == FALSE) || (force_header)) {
            print "- ";
            for (ix=6: ix <= UUID_ARRAY->0: ix++) print (char) UUID_ARRAY->ix;
```

```
            print " ", (string) auxf-->AUXF_FILENAME, "^";
        }
    } else {
        auxf-->AUXF_STATUS = AUXF_STATUS_IS_OPEN_FOR_READ;
        ch = FileIO_GetC(auxf);
        if (ch ~= '-' or '*') { jump BadFile; }
        if (ch == '-')
            return FileIO_Error(auxf, "tried to open a file which was incomplete");
        ch = FileIO_GetC(auxf);
        if (ch ~= ' ') { jump BadFile; }
        ch = FileIO_GetC(auxf);
        if (ch ~= '/') { jump BadFile; }
        ch = FileIO_GetC(auxf);
        if (ch ~= '/') { jump BadFile; }
        owner = auxf-->AUXF_IFID_OF_OWNER;
        ix = 3;
        if (owner == UUID_ARRAY) ix = 8;
        if (owner ~= NULL) {
            for (: ix <= owner->0: ix++) {
                ch = FileIO_GetC(auxf);
                if (ch == -1) { jump BadFile; }
                if (ch ~= owner->ix) not_this_ifid = true;
                if (ch == ' ') break;
            }
            if (not_this_ifid == false) {
                ch = FileIO_GetC(auxf);
                if (ch ~= ' ') { jump BadFile; }
            }
        }
        while (ch ~= -1) {
            ch = FileIO_GetC(auxf);
            if (ch == 10 or 13) break;
        }
        if (not_this_ifid) {
            auxf-->AUXF_STATUS = AUXF_STATUS_IS_CLOSED;
            glk_stream_close(str, 0);
            return FileIO_Error(auxf,
                "tried to open a file owned by another project");
        }
    }
    return auxf-->AUXF_STREAM;
    .BadFile;
    auxf-->AUXF_STATUS = AUXF_STATUS_IS_CLOSED;
    glk_stream_close(str, 0);
    return FileIO_Error(auxf, "tried to open a file which seems to be malformed");
];
```

§9. Close File.

```
[ FileIO_Close auxf;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE))
        return FileIO_Error(auxf, "tried to close a non-file");
    if (auxf-->AUXF_STATUS ~=
        AUXF_STATUS_IS_OPEN_FOR_READ or
        AUXF_STATUS_IS_OPEN_FOR_WRITE or
        AUXF_STATUS_IS_OPEN_FOR_APPEND)
        return FileIO_Error(auxf, "tried to close a file which is not open");
    if ((auxf-->AUXF_BINARY == false) &&
        (auxf-->AUXF_STATUS ==
        AUXF_STATUS_IS_OPEN_FOR_WRITE or
        AUXF_STATUS_IS_OPEN_FOR_APPEND)) {
        glk_set_window(gg_mainwin);
    }
    if (auxf-->AUXF_STATUS ==
        AUXF_STATUS_IS_OPEN_FOR_WRITE or
        AUXF_STATUS_IS_OPEN_FOR_APPEND) {
        glk_stream_set_position(auxf-->AUXF_STREAM, 0, 0); ! seek start
        glk_put_char_stream(auxf-->AUXF_STREAM, '*'); ! mark as complete
    }
    glk_stream_close(auxf-->AUXF_STREAM, 0);
    auxf-->AUXF_STATUS = AUXF_STATUS_IS_CLOSED;
];
```

§10. Get Character.

```
[ FileIO_GetC auxf;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE)) return -1;
    if (auxf-->AUXF_STATUS ~= AUXF_STATUS_IS_OPEN_FOR_READ) return -1;
    return glk_get_char_stream(auxf-->AUXF_STREAM);
];
```

§11. Put Character.

```
[ FileIO_PutC auxf char;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE))
        return FileIO_Error(auxf, "tried to write to a non-file");
    if (auxf-->AUXF_STATUS ~=
        AUXF_STATUS_IS_OPEN_FOR_WRITE or
        AUXF_STATUS_IS_OPEN_FOR_APPEND)
        return FileIO_Error(auxf,
            "tried to write to a file which is not open for writing");
    return glk_put_char_stream(auxf-->AUXF_STREAM, char);
];
```

§**12. Print Line.** We read characters from the supplied file until the next newline character. (We allow for that to be encoded as either a single `0a` or a single `0d`.) Each character is printed, and at the end we print a newline.

```
[ FileIO_PrintLine auxf ch;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE))
        return FileIO_Error(auxf, "tried to write to a non-file");
    for (::) {
        ch = FileIO_GetC(auxf);
        if (ch == -1) rfalse;
        if (ch == 10 or 13) { print "^"; rtrue; }
        print (char) ch;
    }
];
```

§**13. Print Contents.** Repeating this until the file runs out is equivalent to the Unix command `cat`, that is, it copies the stream of characters from the file to the output stream. (This might well be another file, just as with `cat`, in which case we have a copy utility.)

```
[ FileIO_PrintContents auxf tab;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE))
        return FileIO_Error(auxf, "tried to access a non-file");
    if (auxf-->AUXF_BINARY)
        return FileIO_Error(auxf, "printing text will not work with binary files");
    if (FileIO_Open(auxf, false) == 0) rfalse;
    while (FileIO_PrintLine(auxf)) ;
    FileIO_Close(auxf);
    rtrue;
];
```

§**14. Print Text.** The following writes a given piece of text as the new content of the file, either as the whole file (if `append_flag` is false) or adding only to the end (if true).

```
[ FileIO_PutContents auxf text append_flag  str ch;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE))
        return FileIO_Error(auxf, "tried to access a non-file");
    if (auxf-->AUXF_BINARY)
        return FileIO_Error(auxf, "writing text will not work with binary files");
    str = FileIO_Open(auxf, true, append_flag);
    if (str == 0) rfalse;
    @push say__p; @push say__pc;
    ClearParagraphing();
    PrintText(text);
    FileIO_Close(auxf);
    @pull say__pc; @pull say__p;
    rfalse;
];
```

§**15. Serialising Tables.** The most important data structures to "serialise" – that is, to convert from their binary representations in memory into text representations in an external file – are Tables. Here we only carry out the file-handling; the actual translations are in "Tables.i6t".

```
[ FileIO_PutTable auxf tab rv;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE))
        return FileIO_Error(auxf, "tried to write table to a non-file");
    if (auxf-->AUXF_BINARY)
        return FileIO_Error(auxf, "writing a table will not work with binary files");
    if (FileIO_Open(auxf, true) == 0) rfalse;
    rv = TablePrint(tab);
    FileIO_Close(auxf);
    if (rv) return RunTimeProblem(RTP_TABLE_CANTSAVE, tab);
    rtrue;
];
[ FileIO_GetTable auxf tab;
    if ((auxf == 0) || (auxf-->AUXF_MAGIC ~= AUXF_MAGIC_VALUE))
        return FileIO_Error(auxf, "tried to read table from a non-file");
    if (auxf-->AUXF_BINARY)
        return FileIO_Error(auxf, "reading a table will not work with binary files");
    if (FileIO_Open(auxf, false) == 0) rfalse;
    TableRead(tab, auxf);
    FileIO_Close(auxf);
    rtrue;
];
```

§**16. Z-Machine Stubs.** These routines do the minimum possible, but equally, they only generate a run-time problem when there is no alternative.

```
#IFNOT; ! TARGET_GLULX
[ FileIO_Exists auxf; rfalse; ];
[ FileIO_Ready auxf; rfalse; ];
[ FileIO_GetC auxf; return -1; ];
[ FileIO_PutTable auxf tab;
    return FileIO_Error(auxf, "external files can only be used under Glulx");
];
[ FileIO_MarkReady auxf status; FileIO_PutTable(auxf); ];
[ FileIO_GetTable auxf tab; FileIO_PutTable(auxf); ];
[ FileIO_PrintContents auxf; FileIO_PutTable(auxf); ];
[ FileIO_PutContents auxf; FileIO_PutTable(auxf); ];
#ENDIF; ! TARGET_GLULX
```