INWEB

The Program

Complete Program

Build 4/090319 Graham Nelson

User Manual for Inweb

P/man

Purpose

A manual for inweb, a simple literate-programming tool used by the Inform project.

§**1. Introduction.** `inweb` is a command line tool for literate programming, a doctrine invented by Donald E. Knuth in the early 1980s. `inweb` stands in a genre of LP tools ultimately all deriving from Knuth's `WEB`, and in particular borrows from syntaxes used by `CWEB`, a collaboration between Knuth and Sylvio Levy.

In literate programming, a program is written as a "web", a hybrid of code and commentary. For us, a web is a folder containing the strands of code and documentation, and also a few other ingredients needed to weave and tangle the final results.

On a typical run, `inweb` is asked to perform a given operation on an existing web. It has four fundamental modes:

(w) weaving – compiling the many sections into a single file of TEX source, running the result through `PDFTeX` (or some other plain TEX variant: `XeTeX`, `PDFeTeX`, etc.) and then, optionally, displaying it;
(t) tangling – compiling the many sections into a single file of ANSI C source code ready for compilation;
(a) analysing – a mixed bag of non-generative tasks, but for instance looking through the sections for infelicities, displaying how data structures are used, etc.;
(c) creating – making a new web.

§**2. Getting started.** `inweb` is itself supplied as a web. The easiest way to install and use it is to create a folder for webs, and to to place `inweb` inside this; then work in a terminal window with the webs folder as the current working directory. Suppose we have two webs, like so:

```
webs
    cBlorb
    inweb
```

The actual program inside a web, ready for a compiler or an interpreter is its "tangled" form. In the case of `inweb` this is ultimately a Perl script, so it requires no compilation, but it does need a Perl installation (built into Linux, Mac OS X and other Unix variants already, and available with Cygwin on Windows). `inweb` is supplied with its tangled form already in place, as the Perl script `inweb/Tangled/inweb.pl`. We can test this like so:*

```
webs$ ls
cBlorb   inweb
webs$ perl inweb/Tangled/inweb.pl -tangle cBlorb
inweb 4/090319 (Inform Tools Suite)
"cBlorb" 15 structure(s): 4 chapter(s) : 12 section(s) : 250 paragraph(s) : 4451 line(s)
Tangled: cBlorb/Tangled/cBlorb.c
webs$
```

This works through the cBlorb web and generates its own tangled form – `cBlorb/Tangled/cBlorb.c`, which is now a conventional C program and can be compiled with `gcc` or similar, and then run. Alternatively:

```
webs$ perl inweb/Tangled/inweb.pl cBlorb -weave
```

* Users of the `bash` shell may want to `alias inweb='perl inweb/Tangled/inweb.pl'` to save a good deal of typing.

```
inweb 4/090319 (Inform Tools Suite)
"cBlorb" 15 structure(s): 4 chapter(s) : 12 section(s) : 250 paragraph(s) : 4451 line(s)
[0: 94pp 358K]
webs$
```

This generates TEX source for the human-readable form of `cBlorb` and runs it through a variant of TEX called `pdftex`, and then tries to open this. Of course this can only happen if `pdftex` is installed; in practice, you may need to alter the simple configuration file stored at:

```
inweb/Materials/inweb-configuration.txt
```

in order to tell `inweb` how to invoke `pdftex`.

The woven web is a PDF file, then, stored at

```
inweb/Woven/Complete.pdf
```

`inweb` tries to open this on-screen automatically, on the grounds that you probably want to see it, but there's no standard cross-platform way to do this; again, you may need to tweak the configuration file.

We don't have to weave the entire web in one piece – often we're interested in just one section or chapter. For instance,

```
webs$ perl inweb/Tangled/inweb.pl cBlorb -weave 2
inweb 4/090319 (Inform Tools Suite)
"cBlorb" 15 structure(s): 4 chapter(s) : 12 section(s) : 250 paragraph(s) : 4451 line(s)
[2: 11pp 117K]
webs$
```

This time we specified "2", meaning "Chapter 2", as what to weave. We now have:

```
webs$ ls cBlorb/Woven
Chapter-2.pdf   Complete.pdf
```

The most dramatic action we can take is to set off a "swarm" of weaves – one for every section, one for every chapter, one for the entire source, and all accompanied by an indexing website, ready to be uploaded to a web server.

```
webs$ perl inweb/Tangled/inweb.pl cBlorb -weave sections
inweb 4/090319 (Inform Tools Suite)
"cBlorb" 15 structure(s): 4 chapter(s) : 12 section(s) : 250 paragraph(s) : 4451 line(s)
[P/man: 8pp 94K]
[1/main: 6pp 95K]
[1/mem: 9pp 112K]
[1/text: 6pp 81K]
[1/blurb: 6pp 83K]
[2/blorb: 10pp 109K]
[3/rel: 8pp 98K]
[3/sol: 10pp 105K]
[3/links: 3pp 73K]
[3/place: 3pp 74K]
[3/templ: 3pp 69K]
[3/web: 21pp 154K]
[P: 9pp 102K]
[1: 28pp 177K]
[2: 11pp 117K]
[3: 49pp 233K]
[0: 94pp 358K]
Weaving index file: Woven/index.html
Copying additional index file: Woven/download.gif
Copying additional index file: Woven/lemons.jpg
```

§**3.** Suppose we now take a look inside the `cBlorb` web. We find:

```
webs$ ls cBlorb
Chapter 1  Chapter 2   Chapter 3   Contents.w  Materials   Preliminaries   Tangled    Woven
```

This is typical for a medium-sized web – one large enough to be worth dividing into chapters.

(a) We have already seen the `Woven` and `Tangled` folders. Every web contains these, and they hold the results of weaving and tangling, respectively. They are always such that the entire contents can be thrown away without loss, since they can always be generated again.

(b) The `Materials` folder is optional, and contains auxiliary files needed when the program expressed by the web is run – in the case of `cBlorb`, it contains a configuration file.

(c) There is also an optional `Figures` folder for any images included in the text, but `cBlorb` has none.

(d) The program itself is cut up into "sections", each being a file with the extension ".w" (for "web").*
  (1) One section, `Contents.w`, is special – it must be present, it must be called that, and it must be in the main web folder.
  (2) All other sections (and there must be at least one other) are filed either in chapter folders – here, "Preliminaries", "Chapter 1", "Chapter 2" and "Chapter 3" – or else, for a smaller (unchaptered) web, are in a single folder called "Sections".

`inweb` is intended for medium-sized programs which will likely have dozens of these sections (the main `inform7` web has about 225). A section is a structural block, typically containing 500 to 1000 lines of material, which has its own name. An ideal section file makes a standalone essay, describing and implementing a single well-defined component of the whole program.

§**4. A minimal Hello web.** Many software tools scale up badly, in that they work progressively less well on larger tasks: `inweb` scales down badly. So the "Hello" project we'll make here looks very cumbersome for so tiny a piece of code, but it does work.

We first make the `Hello` web. We could make this by hand, but it's easier to ask `inweb` itself to do so:

```
webs$ perl inweb/Tangled/inweb.pl -create Hello
inweb 4/090319 (Inform Tools Suite)
Hello
Hello/Figures
Hello/Materials
Hello/Sections
Hello/Tangled
Hello/Woven
inweb/Materials/Contents.w -> Hello/Contents.w
inweb/Materials/Main.w -> Hello/Sections/Main.w
```

The output here shows `inweb` creating a little nest of folders, and then copying two files into them. The `Hello` web now exists, and works – it can be tangled or woven, and is a "Hello world" program written in C.

---

* In other literate programming tools, notably Knuth's `WEB` and `CWEB`, the term "section" is used for what is probably one or two paragraphs of English prose followed by a single code excerpt. In `inweb` terms, that's a "paragraph".

§**5.** Uniquely, the "Contents.w" section provides neither typeset output nor compiled code: it is instead a roster telling `inweb` about the rest of the web, and how the other sections are organised. It has a completely different syntax from all other sections.

The contents section for `Hello` might be created like so:

```
Title: New
Author: Anonymous
Purpose: A newly created program.
Language: C
Licence: This program is unpublished.
Build Number: 1

Sections
 Main
```

This opens with a block of name-value pairs specifying some bibliographic details; there is then a skipped line, and the roster of sections begins.

So, we have to fill in the details. Note that the program's Title is not the same as the folder-name for the web, which is useful if the web contains multiple programs (see below) or if it has a long or file-system-unfriendly name. The Purpose should be brief enough to fit onto one line. Licence can also have the US spelling, License; `inweb` treats these as equivalent. The Build Number can have any format we like, and is optional. There are other optional values here, too: see below.

The Language is the programming language in which the code is written. At present `inweb` supports:

```
C   C++   C for Inform   Perl   Inform 6   Inform 7   Plain Text
```

Perhaps "supports" ought to be in quotation marks, because `inweb` doesn't need to know much about the underlying programming language. It would be easy to add others; this selection just happens to be the ones we need for the Inform project. ("C for Inform" is an idiosyncratic extension of the C programming language used by the core Inform software; nobody else will ever need it.)

§**6.** After the header block of details, then, we have the roster of sections. This is like a contents page – the order is the order in which the sections are presented on any website, or in any of the larger PDFs woven. For a short, unchaptered web, we might have for instance:

```
Sections
 Program Control
 Command Line and Configuration
 Scan Documentation
 HTML and Javascript
 Renderer
```

And then `inweb` will expect to find, for instance, the section file "Scan Documentation.w" in the "Sections" folder.

A chaptered web, however, won't have a "Sections" folder. It will have a much longer roster, such as:

```
Preliminaries
 Preface
 Thematic Index
 Licence and Copyright Declaration
 Literate Programming
 BNF Grammar

Chapter 1: Definitions
"In which some globally-used constants are defined and the standard C libraries
are interfaced with, with all the differences between platforms (Mac OS X,
Windows, Linux, Solaris, Sugar/XO and so forth) taken care of once and for all."
```

```
 Basic Definitions
 Platform-Specific Definitions

Chapter 2: Memory, Files, Problems and Logs
"In which are low-level services for memory allocation and deallocation,
file input/output, HTML and JavaScript generation, the issuing of Problem
messages, and the debugging log file."
 Memory
 Streams
 Filenames
```

... and so on...

```
Appendix A: The Standard Rules (Independent Inform 7)
"This is the body of Inform 7 source text automatically included with every
project run through the NI compiler, and which defines most of what end users
see as the Inform language."
 SR0 - Preamble
 SR1 - Physical World Model
 SR2 - Variables and Rulebooks
```

Here the sections appear in folders called Preliminaries, Chapter 1, Chapter 2, ..., Appendix A. (These are the only possibilities: `inweb` doesn't allow other forms of name for blocks of sections.)

In case of any doubt we can use the following command-line switch to see how `inweb` is actually reading its sections in:

```
perl inweb/Tangled/inweb.pl -catalogue -verbose-about-input
```

§**7.** If we look at the single, minimal "Main.w" section in the "Hello" web created above, we find something very rudimentary:

```
S/main: Main.

@Purpose: This is the entire program.

@-------------------------------------------------------------------------------

@ Hello, reading world!

@c
int main(int argc, char *argv[]) {
    printf("Hello, computing world!\n");
    return 0;
}
```

This layout will be explained further below, but briefly: each section other than the Contents opens with a titling line, then a pithy statement of its purpose. It then (optionally) contains some definitions and other general discussion up as far as "the bar", followed by code and more discussion below it. Broadly speaking, the content above the bar is what you need to know to use the material in the section; the content below the bar is how it actually works. (The distinction is a little like that between the `.h` "header files" used by C programmers, and the `.c` files of the code these describe.)

As can be seen, the `@` escape character is very significant to `inweb`. In particular, in the first column it marks a structural junction in the file – the Purpose and the bar are examples of this, and so is the single "paragraph" of code below the bar, which begins with the solitary `@` before the word "Hello".

A paragraph has three parts, each optional, but always in the following sequence: some textual commentary (here "Hello..."), some definitions (here there are none), then some code after an `@c` marker (here, the C function `main(argc, argc)`).

A section need not contain any code at all, in fact: this manual is itself an `inweb` section, in which every paragraph contains only commentary.

§**8. Chapter and section names.** There is in principle no limit to the number of sections in an unchaptered web. But once there are more than nine or ten, it is usually a good idea to group them into higher-level blocks. `inweb` calls these blocks "chapters", though they aren't always named that way. The possibilities are as follows:

(a) "Preliminaries". Like the preliminary pages of a book – preface, licence details perhaps, some expository material about the method. The actual code ought to begin in Chapter 1 (though `indoc` doesn't require that).

(b) "Chapter 1", "Chapter 2", and so on.

(c) "Appendix A", "Appendix B", ... and so on up to "Appendix O" but no further.

Each chapter has a one-character abbreviation: P, 1, 2, ..., A, B, ...; thus

```
perl inweb/Tangled/inweb.pl cBlorb -weave B
```

weaves a PDF of Appendix B alone. In general, it doesn't make sense to tangle a single chapter alone, because they are all parts of one large program, but there's a way to specify that certain chapters contain independent material – this allows for the web to hold one big C program (say) and Appendix A a text file of settings vital for its running; and in that case it would indeed make sense to tangle just Appendix A, generating the text file.

§**9.** A section name must contain only filename-safe characters, and it's probably wise to make them filename-safe on all platforms: so don't include either kind of slash, or a colon, and in general go easy on punctuation marks.

To be as descriptive as possible, section names need to be somewhat like chapter titles in books, and this means they tend to be inconveniently long for tabulation, or for references in small type.

Each section therefore also has an abbreviated name, which is quoted in its titling line. This should always have the form of a chapter number, followed by a slash, followed by a short (usually two to five character) alphanumeric name. For instance, the `inform7` section "Problems, Level 3" has abbreviated name `2/prob3`, since it falls in Chapter 2.

In a web with no chapters, the chapter part should be S, for section. (Hence `S/hello` above.) Sections in the Preliminaries chapter, if there is one, are prefaced P; sections in Appendix A, B, ..., are prefaced with the relevant letter.

Within each section, paragraphs above the bar are set as ¶1, ¶2, ... while those below are §1, §2, ..., and this gives a concise notation identifying any paragraph in the whole program: for instance,

2/prob3.§2

means paragraph 2 below the bar in the "Problems, Level 3" section of chapter 2.

§**10. Weaving.** The weaver produces a typeset version of all or part of the web, or possibly an index to it. In general, we activate the weaver by running `inweb` like so:

```
perl inweb/Tangled/inweb.pl cBlorb -weave W
```

where `W` is the "target". This can be any of the following:

(1) `all`: the whole thing;
(2) a chapter number 1, 2, 3, ..., or an appendix letter A, B, ...;
(3) the letter P, meaning the collected Preliminaries;
(4) the abbreviated name for a section, such as `2/prob3`.

The default target is `all`.

§**11.** In addition, three special targets run the weaver in "swarm mode", which could perhaps be more happily named. This automates a mass of individual weaving tasks to generate multiple PDFs suitable for offering as downloads from a website. (It's more efficient than simply batch-processing uses of `inweb` from the shell, since the source only needs to be scanned once. It's also a lot less trouble.) These targets are, in ascending order of size:

(5) `index`: create a web page from a template which gives a tidy download directory of the chapters and sections (see below);
(6) `chapters`: weave a single PDF for each chapter block (Preliminaries, if present, all numbered chapters present, all lettered appendices present), then also `index`;
(7) `sections`: weave a single PDF for each section (other than the contents), then also weave `chapters` and `index`.

Lastly, though this will only make sense for the `sections` target, it's possible to restrict the weaver to the sections in a given chapter or Appendix only, using the `-only` command-line switch: so

```
perl inweb/Tangled/inweb.pl inform7 -weave -only A sections
```

weaves the sections of Appendix A only, and makes up the index page to show only those. (This is useful for publishing only part of a web.)

§**12.** The weaving process consists of several steps:

(i) A suitable file of TEX source, with a name such as `Chapter-2.tex`, `2-prob3.tex`, etc., is written out into the `Woven` subfolder of the web. Spaces are removed from its filename since TEX typically has dire problems with filenames including spaces.

(ii) This is then run through a TEX-to-PDF tool such as `pdftex` or `XeTeX`. (The choice of and path to this can be altered in the `inweb` configuration file.) The console output is transcribed to a file rather than being echoed on screen: `inweb` instead prints a concise summary such as

```
[2/prob3: 12pp 99K]
```

meaning that a 12-page PDF file, 99K in size, has been generated, and that there were no TEX errors – because if there had been, the summary would have said so.

(iii) The TEX source and log file are deleted. So is the console output file showing what the TEX agent verbosely chattered to `stdout`, *unless* there were any errors, in which case it is preserved for the user to investigate.

(iv) If the `inweb` configuration file supplies a command for opening PDFs in the local operating system's PDF viewer, then `inweb` now uses it by default, but this can be explicitly switched on with the `-open` switch or off with the `-closed` one. The configuration file supplied in the standard `inweb` distribution simply uses a command called `open`, which in Mac OS X duplicates the effect of double-clicking the file in the Finder, so that it opens in the user's preferred PDF viewer (Preview, say). Under OS 10.5, Preview automatically detects if an open PDF file has changed and redisplays it if so, so that the author can keep a section permanently open and have it refresh on each run of `inweb`.

§**13. Cover sheets.** The really large PDFs, for chapters and for the whole thing, have a cover sheet attached. The standard design for this is pretty dull, but it can be overridden with an optional value in the "Contents.w" section:

```
Cover Sheet: cover-sheet.tex
```

This names a file, which must be present in the "Materials" folder for the web, of TEX source for the cover sheet. (It already has all of the `inweb` macros loaded, so needn't `\input` anything, and it should not `\end`.)

§**14.** Within the cover sheet copy, doubled square brackets can be used to insert any of the values in the "Contents.w" section – for instance,

```
\noindent{{\sinchhigh\noindent [[Build Number]]}}
```

In addition:

(a) `[[Cover Sheet]]` expands to the default cover sheet – this is convenient if all you want to do is to add a note at the bottom of the standard look.
(b) `[[Booklet Title]]` expands to text such as "Chapter 3", appropriate to the weave being made.
(c) `[[Capitalized Title]]` is a form of the title in block capital letters.

§**15. Indexing.** The weaver's `index` target, produced either as a stand-alone or to accompany a swarm of chapters or sections, is generated using a template file. In fact, this can almost any kind of report, or even a multiplicity of reports: the "Contents.w" section can specify one or more templates, like so –

```
Index Template: index.html, sizes.txt
```

If no such setting is made, `inweb` will by default use its own standard template, which is a single HTML index page.

The "Contents.w" can also specify a list of binary files:

```
Index Extras: corporate-logo.gif
```

These are copied verbatim from the web's Materials folder into its Woven one. (When `inweb` is making its default web page, it copies two such images – a botanical painting of some lemons which is for some reason the `inweb` banner, and a download icon.) Of course the effect is the same as if we always kept these files in Woven, but we don't want to do that because we want to preserve the rule that Woven contains no master copies – the contents can always be thrown away without loss.

§**16.** Each index is made by taking the named template file and running it through the "template interpreter" to generate a file of the same name in the `Woven` folder. The template interpreter is basically a filter: that is, it works through one line at a time, and most of the time it simply copies the input to the output. The filtering consists of making the following replacements. Any text in the form `[[...]]` is substituted with the value ..., which can be any of:

(a) A bibliographic variable, set at the top of the `Contents.w` section.
(b) One of the following details about the entire-web PDF (see below):

```
[[Complete Leafname]]  [[Complete Extent]]  [[Complete PDF Size]]
```

(b) One of the following details about the "current chapter" (again, see below):

```
[[Chapter Title]]  [[Chapter Purpose]]  [[Chapter Leafname]]
[[Chapter Extent]]  [[Chapter PDF Size]]  [[Chapter Errors]]
```

The leafname is that of the typeset PDF; the extent is a page count; the errors result is a usually blank report.

(c) One of the following details about the "current section" (again, see below):

```
[[Section Title]]  [[Section Purpose]]  [[Section Leafname]]
[[Section Extent]]  [[Section PDF Size]]  [[Section Errors]]
[[Section Lines]]  [[Section Paragraphs]]  [[Section Mean]]
[[Section Source]]
```

Lines and Paragraphs are counts of the number of each; the Source substitution is the leafname of the original `.w` file. The Mean is the average number of lines per paragraph: where this is large, the section is rather raw and literate programming is not being used to the full.

§**17.** But the template interpreter isn't merely "editing the stream", because it can also handle repetitions. The following commands must occupy entire lines:

`[[Repeat Chapter]]` and `[[Repeat Section]]` begin blocks of lines which are repeated for each chapter or section: the material to be repeated continues to the matching `[[End Repeat]]` line. The "current chapter or section" mentioned above is the one selected in the current innermost loop of that description.

`[[Select ...]]` and `[[End Select]` form a block which behaves like a repetition, but happens just once, for the named chapter or section.

For example, the following pattern:

```
To take chapter 3 as an example, for instance, we find:
[[Select 3]]
[[Repeat Section]]
    Section [[Section Title]]: [[Section Code]]: [[Section Lines]] lines.
[[End Repeat]]
[[End Select]]
```

weaves a report somewhat like this:

```
To take chapter 3 as an example, for instance, we find:
    Section Lexer: 3/lex: 1011 lines.
    Section Read Source Text: 3/read: 394 lines.
    Section Lexical Writing Back: 3/lwb: 376 lines.
    Section Lexical Services: 3/lexs: 606 lines.
    Section Vocabulary: 3/vocab: 338 lines.
    Section Built-In Words: 3/words: 1207 lines.
```

§**18. Tangling.** This is more simply described. For almost all webs, there is only one possible way to tangle:

```
perl inweb/Tangled/inweb.pl cBlorb -tangle
```

However, if we want the tangled result to have a different name or destination from the normal one, we can write:

```
perl inweb/Tangled/inweb.pl cBlorb -tangle-to ../stuff/etc/cblorb.c
```

Exactly what happens during a tangle will be described later, when we get to details on the syntax of section files. Basically, it makes the entire program as a single source file with the commentary removed, and (for C-like languages, anyway) restrings it all into a convenient order: for instance all C functions are predeclared, structure definitions are made in an order such that if A contains B as an element then B is declared before A regardless of where they occur in the source text, all above-the-bar definition material is available from every section, and so forth.

§**19.** As trailed above, it *is* legal in some circumstances to tangle only part of a web. The command syntax is just like that for weaving:

```
perl inweb/Tangled/inweb.pl inform7 -tangle A
perl inweb/Tangled/inweb.pl inform7 -tangle B/light
```

However, this is only allowed if the chapter or section involved was listed in the "Contents.w" roster as being Independent. This explains why we had:

```
Appendix A: The Standard Rules (Independent Inform 7)
"This is the body of Inform 7 source text automatically included with every
project run through the NI compiler, and which defines most of what end users
see as the Inform language."
 SR0 - Preamble
 SR1 - Physical World Model
```

The bracketed "(Independent)" marks out Appendix A as a different tangle target to the rest of the web. In this case, we've also marked it out as having a different language – the rest is a C program, but this is an Inform 7 one.

Similarly, we can mark a section as independent:

```
 Light Template (Independent Inform 6)
```

Independent chapters and sections are missed out when tangling the main part of the web, of course.

§**20. Analysing.** We provide miscellaneous tools which were created to locate bad practices in the main `inform7` web; other people may not find them useful at all.

`-analyse-structure name`

prints a summary of which sections other than the owner access which elements of the data structure `name`. If the structure is private to its owner, then there will be no such shared elements and no output will be produced. (`inweb` works only textually, so use of macros can conceal such sharing; and of course `inweb` only recognises data structures in C-like languages.)

`-catalogue`

prints an annotated table of contents of the web source, listing each section and its C structures, together with a note of any other sections sharing these data structures.

`-functions`

prints a much longer catalogue, including names of all functions defined in the sections.

`-make-graphs`

outputs a set of `.dot` files called `chapter2.dot`, `chapter3.dot` and so on into the `Tangled` folder. These are dependency graphs, showing which sections in each chapter make function calls to which other sections. Dot files are the source code used by the graph-drawing program `dot`. If the further setting...

`-make-graphs -convert-graphs`

...is made, then the dependency graphs are run automatically through the `dot` utility to produce PNG images in the `Figures` folder. This can only work if `dot` is installed: the `inweb` configuration file specifies the filename for it. The result can look somewhat like an octopus rewiring a telephone exchange, though.

`inweb -voids`

shows which data structures include elements of type `void *`, something which we might deprecate since any long term storage as a `void *` must mean that a pointer will need eventually to be cast back to an explicit pointer type, which is unsafe.

§**21. The web source code format.** To recap: the web is hierarchically organised on four levels – chapter, section, named paragraph, anonymous paragraph. (We haven't seen named paragraphs yet, but they are about to appear.) Each `.w` file corresponds to a single section, except for the "Contents.w" section, which is special and to which the following does not apply.

§**22. Below the bar.** Broadly speaking, material below the bar is the program itself: the routines of code, intermingled with commentary. This is a sequence of paragraphs.

Each paragraph is introduced by an "at" symbol `@` in column 1. (Outside of column 1, an `@` is a literal at symbol, except for the `@< ... @>` notation – for which see below.) Some paragraphs are named, while others are anonymous. Here are the three varieties of paragraph break:

```
@ This begins an anonymous paragraph, and runs straight into text...

@p Named Paragraph. The text after the symbol, up to the full stop, is the title.
```

`@pp` produces a named paragraph but also forces a page-break before it, so that it will start at the top of a fresh page.

§**23.** The content of a paragraph is divided into comment, definitions and code, always occurring in that order. Definitions and code are each optional, but there is always comment – even if it is sometimes just a space where comment could have been written but wasn't.

Text following a `@` marker, or following the end of the title of a `@p` marker, is taken as the comment. Comment material is ignored by the tangler, and contributes nothing to the final compiled program.

Definitions are indicated by an `@d` escape which begins a line. A definition can be read almost exactly as if it were a `#define` preprocessor macro for C, but there are two differences: firstly, the tangler automatically gathers up all definitions and moves them (in order) to the start of the C code, so there is no need for a definition to be made earlier in the web source than it is used; and secondly, a definition automatically continues to the next `@` escape without any need for continuation backslashes. This means that long, multi-line macros can be written much as ordinary code.

Code begins with a `@c` escape which begins a line. There can be at most one of these in any given paragraph. From that escape to the end of the paragraph, the content is literal C code.

The following example makes a long macro definition as a template, just to demonstrate the point about multi-line definitions:

```
@p Example Paragraph. Here I could write a whole essay, or nothing much.
@d MAX_BANANA_SHIPMENT 100
@d EAT_FRUIT(variety)
    int consume_by_##variety(variety *frp) {
        return frp->eat_by_date;
    }
@c
banana my_banana; /* initialised somewhere else, let's suppose */
EAT_FRUIT(banana) /* expands with the definition above */
void consider_fruit(void) {
    printf("The banana has an eat-by date of %d.", consume_by_banana(&my_banana));
}
```

Note that the C code can contain comments, just as any C program can. These are typeset using TEX in the woven output, just as the comment matter at the start of the paragraph is, except that they appear in italic type:

```
banana my_banana;                                   initialised somewhere else, let's suppose
EAT_FRUIT(banana)                                   expands with the definition above
```

§**24. Folding code.** The single most important feature of `inweb`, and other literate programming systems, is the ability to fold up pieces of code into what look like lines of pseudocode. For example,

```
@ So, this is a program to see if even numbers from 4 to 100 can all
be written as a sum of two primes. Christian Goldbach asked Euler in 1742
if every even number can be written this way, and we still don't know.

@c
int main(int argc, char *argv[]) {
    int i;
    for (i=4; i<100; i=i+2) {
        printf("%d =", i);
        @<Solve Goldbach's conjecture for i@>;
        printf("\n");
    }
}
```

Here, the interesting part of the code has been abstracted into a named paragraph. The definition could then follow:

```
@ We'll print each different pair of primes adding up to $i$. We
only check in the range $2\leq j\leq i/2$ to avoid counting pairs
twice over (thus $8 = 3+5 = 5+3$, but that's hardly two different ways).

@<Solve Goldbach's conjecture for i@> =
    int j;
    for (j=2; j<=i/2; j++)
        if ((isprime(j)) && (isprime(i-j)))
            printf(" %d+%d", j, i-j);
```

What did we gain by this? Really the point was to simplify the presentation by presenting the code in layers. There was an outer layer doing routine book-keeping, and an inner part which had to understand some basic number theory, and each layer is easier to understand without having to look at the other one.

Of course, that kind of code abstraction is also what a function call does. But there are several differences: (a) we don't have to pass arguments to it, because it isn't another function – it's within this function and has access to all its variables; (b) we aren't restricted to a C identifier, so we can write a more natural description of what it does.

§**25.** Two important differences between `inweb` and its ancestor `CWEB` in how they handle folded code:

(1) In `inweb`, the code is tangled within braces. (Well, for C-like languages which brace blocks of code, anyway.) This means that the scope of the variable `j` defined above is just within the "Solve..." paragraph. It also means that the following loops over the whole code in the paragraph, as we might expect:

```
for (k=1; k<=5; k++) @<Do something really complex depending on k@>;
```

(2) `inweb` does *not* follow the disastrous rule of `CWEB` that every name is equal to every other name of which it is an initial substring, so that, say, "Finish" would be considered the same name as "Finish with error". This was a rule Knuth adopted to save typing – he habitually wrote elephantine names, the classic being §1000 in the TEX source code, which reads:

⟨ If the current page is empty and node *p* is to be deleted, **goto** *done1*; otherwise use node *p* to update the state of the current page; if this node is an insertion, **goto** *contribute*; otherwise if this node is not legal breakpoint, **goto** *contribute* or *update_heights*; otherwise set *pi* to the penalty associated with this breakpoint ⟩

§**26. The bar.** The section is bisected by a horizontal bar:

```
@-------------------------------------------------------------------------------
```

which acts as a dividing line. (A vertical bar would be more of a challenge for coders and editors, but there we are.) In fact this can have any width from four hyphens upwards, so the smallest legal bar would be: `@----`.

§**27. Above the bar.** Broadly speaking, material above the bar identifies the section and how it relates to other sections in the web. No functions should be declared here: it should be structures, definitions, global variables and arrays, commentary.

The titling line at the top of the section has already been described: it has the form

```
abbrev: Unabbreviated Name.
```

and it is always followed by a blank line, and then by:

```
@Purpose: ...
```

This should be a brief summary of what the code in this section is for. For instance, "To manage initial and current values of named values, which may be either constants or variables, but which have global scope."

§**28.** When the web is for a C-like language, we may want to include an interface declaration next –

```
@Interface:
```

If present, this is followed by a sequence of lines explicitly declaring the sections which use the section, or are used by it, and also what data structures it owns. Examples of the four kinds of interface declaration follow:

```
-- Used by Chapter 6/Compile Atoms.w
-- Uses Chapter 2/Index File Service.w
-- Owns struct quantity (private)
-- Defines {-callv:index_quantities}
```

"Used by" and "Uses" refers to calling functions – we are used by X if code in section X calls one of our functions; we use section Y if any of our code calls a function defined in Y.

The ownership declaration indicates that this section will `typedef` a C structure called `quantity`, and that it is private: `inweb` will not permit code in any other section to access its data. Structures need not be private; for instance:

```
-- Owns struct phrase (public)
   !- shared with Chapter 7/Data Type Checking.w
   !- shared with Chapter 7/Type Checking.w
```

The `Defines` line is meaningful only for the `inform7` web and shows which sections of the Inform 7 compiler support which I6 template escapes.

By default, `inweb` does not require each section to have a correct Interface declaration – for most small webs, it's too much trouble for no particular gain. But if the "Contents.w" section includes the setting –

```
Strict Usage Rules: On
```

then `inweb` will indeed complain if the interface is wrongly declared. As a half-way sort of strictness, we can also set

```
Declare Section Usage: Off
```

which relieves us of the more tedious job of making "Used by" and "Uses" declarations, and polices only the other two forms.

Thus, *unless the Contents section goes out if its way to ask for this, there's no need ever to declare the Interface.*

§**29.** Another optional paragraph follows:

```
@Grammar:
```

This is purely for documentation and is not mechanically verified as accurate. It gathers up pieces of the BNF grammar for Inform 7's language, so that these can be documented all over the code and gathered together into a single description automatically by `inweb`. (See below.) Typical grammar notation would be:

```
<rule-usage>
 := <rulebook-usage> [during <scene-name>]

<rulebook-usage>
 := <rulebook-invocation> [rule]
 := [rule [for]] <rulebook-invocation>
 ... [ [ <bridging> ] <rule-parameter> ]
 ... [when/while <condition> ]
```

§**30.** The last body of material above the bar, which is yet again optional, is written:

```
@Definitions:
```

Beneath this heading, if it is present, is a sequence of 1 or more paragraphs exactly like those below the bar, except that no functions should be defined.

This is a good place to make global variable, array, constant, or type definitions. The tangler automatically moves these to the start of the code and ensures that everything works regardless of ordering. Note that no C function predeclarations are ever needed: again, the tangler makes those itself automatically. Functions can therefore be declared and used in any order.

§**31. Calling functions across sections.** If we're writing a C-like language, and if we've chosen to set Strict Usage Rules to "On" (by default it's "Off"), then we are required to mark any point at which a function in one section makes itself available to be called by functions in another.

This is done with C-style comments, thus. A two-star function is called by other sections in the same chapter:

```
/**/ void called_from_rest_of_chapter(void) { ...
```

A three-star function is one called from other chapters:

```
/***/ void called_from_other_chapters(void) { ...
```

A four-star function is called by the `.i6` top-level interpreter – this will only be the case in the `inform7` web:

```
/****/ void called_by_our_User_above(void) { ...
```

And for completeness, the one and only five-star function is `main`:

```
/*****/ int main(int argc, char *argv[]) { ...
```

Strict Usage Rules really are strict, and `inweb` checks such definitions; it will halt a tangle if functions are incorrectly labelled.

§**32. Special extensions made to TEX syntax.** Comment material in paragraphs is set more or less as standard plain TEX, but with a suite of convenient macros available, and with a number of convenient enhancements provided by `inweb` as it weaves.

A line in the following form:

```
/* PAGEBREAK */
```

forces a page break, either in comment or in code. In the absence of such explicit instructions, TEX will use its standard (sometimes unfortunate) page breaking algorithm, ameliorated by the spacing rules followed by `inweb`.

Lines beginning with what look like bracketed list numbers or letters are set as such, running on into little indented paragraphs. Thus

```
(a) Intellectual property has the shelf life of a banana. (Bill Gates)
(b) He is the very pineapple of politeness! (Richard Brinsley Sheridan)
(c) Harvard takes perfectly good plums as students, and turns them into
prunes. (Frank Lloyd Wright)
```

will be typeset thus:

(a) Intellectual property has the shelf life of a banana. (Bill Gates)
(b) He is the very pineapple of politeness! (Richard Brinsley Sheridan)
(c) Harvard takes perfectly good plums as students, and turns them into prunes. (Frank Lloyd Wright)

A line which begins `(...)` will be treated as a continuation of indented matter (following on from some break-off such as a source quotation). A line which begins `(-X)` will be treated as if it were `(X)`, but indented one tab stop further in, like so:

(d) Pick a song and sing a yellow nectarine. (Scott Weiland)

If a series of lines is indented with tab characters and consists of courier-type code extracts, it will be set as a running-on series of code lines.

A line written thus:

```
>> The monkey carries the blue scarf.
```

is typeset as an extract of I7 source text thus:

```
The monkey carries the blue scarf.
```

§**33.** Pictures must be in PNG or PDF format and can be included with lines like:

```
[[Figure: Fig_0_1.pdf]]
[[Figure: 10cm: Fig_0_2.png]]
```

In the second example, we constrain the width of the image to be exactly that given: it is scaled accordingly.

The weaver expects that any pictures needed will be stored in a subfolder of the web called `Figures`: for instance, the weaver would seek `Fig_2_3.pdf` at pathname `Figures/Fig_2_3.pdf`.

§**34.** The BNF grammar, as gathered from the special Grammar paragraphs across all the sections, can be output thus:

```
[[BNF Grammar]]
```

`inweb` collates all the productions and arranges them hierarchically, giving their locations in the code as subscripts. The above command outputs every production not yet output. This is so that certain productions can be output first, thus:

```
[[BNF Grammar: heading-sentence, table-sentence]]
```

which shows just `<heading-sentence>` and `<table-sentence>`.

§**35.** Finally, note that vertical strokes `|` can be used in comment to go into courier type, to indicate pieces of code. For instance, `the |pineapple| variable` would typeset to "the `pineapple` variable".

§**36. The "C for Inform" language.** This is a small extension of C, provided at the tangling stage within `inweb`, for the `inform7` web only: no other project can usefully employ it.

Here the code to be tangled is ANSI standard C, but with the following additional syntaxes, which are expanded into valid C by `inweb`. Note that although I7 makes heavy use of memory allocation and linked list navigation macros, `CREATE` and `LOOP_OVER`, they and similar constructions are defined as ordinary C preprocessor macros and are documented where they are defined. They are not formally part of the `inweb` language.

The extension made is to assist with parsing sequences of words, something that I7 does frequently and with repetitive code which is nevertheless easy to get wrong.

Note that I7 numbers the words in the source text upwards contiguously from 0. It represents an excerpt of text as a pair of integers, often conventionally using the variables `w1` and `w2` to store these, so these will be used in the examples below; but any C variables would do just as well.

Many data structures used by I7 are implemented with C structures to represent objects (for instance, a `quantity` represents a named constant or variable) and the elements `word_ref1` and `word_ref2` are conventionally used to store the name of the object in question. This means that we often need to copy such a pair of word numbers: for this purpose, we can write

```
[[w1, w2 <-- something]];
```

which copies the name of the `something` object into `w1` and `w2`, expanding into the code:

```
w1 = something->word_ref1; w2 = something->word_ref2;
```

§**37.** Similarly, we can give the name of an instance of a structure in place of the initial pair of word variables, in which case the `word_ref1` and `word_ref2` elements are read. Thus:

```
[[q == blackcurrant jelly]]
```

is equivalent to:

```
[[w1, w2 <-- q]]; [[w1, w2 == blackcurrant jelly]]
```

§**38.** I7 identifies words against a vocabulary, and uses conventionally named variables for standard words in this vocabulary: for instance the variable `first_V` points to the entry for the word "first". To check whether a given range of words matches a given possible text, we need to perform a number of tedious comparisons of vocabulary pointers. This tends to result in lengthy conditions. Instead:

```
[[w1, w2 == star fruit]]
```

will expand to suitable code which checks that `w1` and `w2` represent a valid word range, of length exactly 2 words, and further that the first word is "star" and the second is "fruit". (This will only work if `star_V` and `fruit_V` can be found in the definitions of built-in words in chapter 3 of I7.) Any number of words can be given, from 1 upwards. Note that if `w1` is negative, conventionally meaning "no text", then the condition evaluates to false without any risk of accessing invalid areas of memory such as the one for "word −1". Comparisons are case insensitive.

We can also specify alternative single words with a slash character, thus:

```
[[w1, w2 == an apple/orange for breakfast]]
```

matches exactly two possible texts, "an apple for breakfast" and "an orange for breakfast".

A single word can also be compared thus:

```
[[word w1 == peach]]
[[word w1 == peach/plum/nectarine]]
```

§**39.** Three words are special: `###`, `...` and `***`. The first of these means "any single word can go here". Thus:

```
[[w1, w2 == silver ### peeler]]
```

matches "silver lemon peeler" and "silver grape peeler", and so on.

§**40.** We can use the special ellipsis word `...` to indicate "one or more arbitrary words". Thus

```
[[w1, w2 == pomegranate can be cooked with ...]]
```

would match "pomegranate can be cooked with coconut shrimp". (Check out `www.pomegranateworld.com` if you don't believe this.) Thus, for instance,

```
[[w1, w2 == ...]]
```

evaluates to true if and only if `w1` and `w2` represent a valid and non-empty range of words; while

```
[[w1, w2 == ... keeps the doctor away]]
```

matches, say, "an apple every five minutes keeps the doctor away" (this one you probably shouldn't believe) but not "keeps the doctor away".

If we have more than one ellipsis, then we need some temporary storage in order to be able to make the condition work: we need to tell `inweb` what local variables it can use. For instance:

```
[[w1, w2 == ... when ... : w]]
```

looks for the word "when" strictly between `w1` and `w2`, setting the integer variable `w` to the word number of the position of "when". In general, if there are E ellipses, then we need to specify $E - 1$ variables. For instance:

```
[[w1, w2 == ... when ... and ... fruit cocktail is served : w, x]]
```

§**41.** It tends to be convenient to record the word number ranges of these ellipsis excerpts, for further parsing, so we can optionally follow the condition with a `-->` clause. As with `<--` above, this implies that a pair of integer variables is being written (or in fact one per ellipsis). For instance:

```
[[w1, w2 == and now ... --> now1, now2]]
```

is a condition which, if it evaluates to true, has the side effect of setting `now1` and `now2` to the word range, guaranteed to be valid and non-empty, for the text of the ellipsis `....` (If evaluated to false, the variables are not changed.) Likewise:

```
[[w1, w2 == ... when ... : w --> w1, w2 ... when1, when2]]
```

will, if evaluated to true, reduce `w1` and `w2` from the full excerpt to just the initial clause, and store the text after "when" in the pair `when1` and `when2`. (As this example demonstrates, it is legal for the input variables to coincide with the output ones, and `inweb` expands the condition carefully to set the output variables in the right order so that no confusion between old and new data occurs.)

More modestly, the following condition executed only for its side-effect:

```
[[w1, w2 == the ... --> w1, w2]];
```

strips any initial "the" from the text, if the text is well-founded, and provided that further words do follow the "the".

§**42.** A variation on the ellipsis is used to mark "0 or more words" and occurs only at the end of an excerpt. Thus:

```
[[w1, w2 == *** fruit cocktail]]
```

tests to see if the word range can contain two words or more, and that the final two words if so are "fruit" and then "cocktail". `***` can be used only at the end of an excerpt.

§**43. Why inweb was written.** "An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our safety than unsafe cars, toxic pesticides, or accidents at nuclear power stations." (Tony Hoare, quoted in Donald MacKenzie, *Mechanizing Proof* (MIT Press, 2001)). The traditional response is to formalise such tools: to give a mathematical description of input and output, and to prove that the compiler does indeed transform one into the other. But the ideal of the fully verified, fully verifying compiler remains a distant one, and it seems altogether likely that when we do get it, it will be as difficult to use as a mathematical theorem-prover. In any case, there are obvious difficulties when the input syntax is natural language, and not easily given formal expression.

My biggest concern in coding Inform has been to find a way to write it which would give some confidence in its correctness, and to make it maintainable by other people besides myself. Having little faith in "neat" AI approaches to program correctness – the ideal advanced by Tony Hoare's Grand Challenge project for a truly verifying compiler – I turned to the "scruffy" side. Knuth's literate programming dogma is a different kind of program verification. The aim is to write a program which is as much an argument for its own correctness as it is code. This is done not so much with formalities – preconditions and contracts à la Eiffel, or heavy use of assertions – as with something closer to a proof as it might appear in a scientific journal. The text mixed in with code is aimed at a human reader.

A key step in literate programming is publishing code, which is not the same thing as making code available for download. It means tidying up and properly explaining code, and is a process much like writing up roughly-correct ideas for publication in journals – the act of tidying up a final article reveals many small holes: odd cases not thought of, steps missed out. The process is like a systematic code review, but it's more than that; it involves a certain pledge by the author that the code is, to the best of his understanding, right.

§**44.** When he wound up his short-lived *Literate Programming* column in CACM (vol. 33, no. 3, 1990) – where it had been the successor to the legendary *Programming Pearls* column – Christopher Wyk made the perceptive criticism that "no one has yet volunteered to write a program using another's system for literate programming". Nor have I, but I did try, and perhaps it may be helpful to explain why `inweb` was written.

When Inform 7 began to be coded, in 2003, it used `CWEB` – a combination of programs called `cweave` and `ctangle` by Sylvio Levy, after Knuth, and to some extent the lineal descendant today of Knuth's original Pascal-based `WEB`. A short manual for `CWEB`, available online, is also on sale in printed form: the software itself can be downloaded from the Comprehensive TEX Archive `CTAN`. I am going to say some unkind things about `CWEB` in a moment, so let me first acknowledge the great benefit which I have derived from it, and record my appreciation of Levy and Knuth's work.

`CWEB` seemed to me the most established and durable of the options available, and the alternatives, such as `noweb` and `funnelweb`, did not seem especially advantageous for a C program. I was also unsure that either would be very well maintained or supported (which is ironic, given that the author of `noweb` later became an Inform user, filing numerous helpful bug report forms). Though it was arguably abandonware, `CWEB` seemed likely to be reliable, given its heritage and continuing use by the TEX community.

`CWEB` is, it must be said, clearly out of sorts with modern practice. It is an aggressively Knuthian tool, rooted in the computing paradigms of the 1970s, when nothing was WYSIWYG and the escape character was king. Hardly anybody reads `CWEB` source fluently, even though it is a simple system with a tiny manual. Development environments have never heard of it; it won't syntax-colour properly. The holistic approach subverts the business of linking independently compiled pieces of a large program together. So my first experiments were carried out in a spirit of scepticism. The prospect of making C more legible by interspersing it with TEX macros seemed a remote one, and since both C and TEX already have quite enough escape characters as it is, the arrival of yet another – the `@` sign – did not gladden the heart. And yet. `@` signs began to appear like mushrooms in the Inform source, and I found that I was indeed habitually documenting what I was doing rather more than usual, and gathering conceptually similar material together, and trying to structure the code around "stories" of what was happening. Literate programming was working, in fact.

A lengthy process of fighting against hard limits and unwarranted assumptions in `CWEB` followed, and by 2005 both `cweave` and `ctangle` had had to be hacked to be viable on the growing Inform source. For one thing, both contained unnecessary constraints on the size of the source code, making them capable of compiling

TEX and Metafont – their primary task – but little more. Both tools use tricksy forms of bitmap storage making these limits difficult to overcome. `ctangle` proved to be fairly robust, if slow: its only defect as such was an off-by-one bug affecting large webs, which I never did resolve, causing all line numbering to be out by one line in `gdb` stack backtraces, internal error messages and the like. But the problems of `cweave` went beyond minor inconvenience. It parses C with a top-down grammar of productions in order to work out the ideal layout of the code, totally ignoring the actual layout as one has typed it. This "ideal" layout is usually worse than a simple syntax-colouring text editor could manage, and often much worse.

More seriously, the productions for C used by `CWEB` do not properly cope with macros, and as often happens with such grammars, when they go wrong they go horribly wrong. A glitch somewhere in a function causes the entire body of code to be misinterpreted as, I don't know, a variable. When that glitch is not in fact a coding error but legal and indispensable code, the results are horrible. For some months I rewrote and rewrote `cwebmac.tex`, the typically incomprehensible suite of TEX macros supplied with `CWEB`, but I was forced to go through stages of both pre- and post-processing in order to get anything like what I wanted out of `cweave`. It is, unfortunately, almost impossible to amend or customise `cweave`: it hangs entirely on the parsing grammar for C, and this is presented as cryptic productions which have clearly been mechanically generated from some higher-level description which is *not* available. It is a considerable irony that the `CWEB` source was meant to demonstrate its own virtues as a presentation of openly legible code, yet as a program it now clearly fails clause 2 of the Open Source Initiative's definition of "open source":

The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

And this is a real hazard with literate programming (and is the reason `inweb` provides features to present auxiliary files in multiple languages within the same web). Disagree with Knuth and Levy about the ideal display position of braces? Prefer the One True Brace Style, for instance? Want to omit those little skips between variable declarations and lines of code? Think cases in `switch` statements ought to be indented a little? Want static arrays to be tabulated? Think you ought to be allowed to comment out pieces of code? You are in for hours of misery.

And so for about a year, from autumn 2005 to autumn 2006, I tangled but I did not weave. This was hardly a showcase for the literate programming ideal.

In December 2006, I finally conceded that `ctangle` was no longer adequate either. Inform was pushing against further limits, harder still to raise, but the real problem was that `ctangle` made it difficult to encapsulate code in any kind of modular way. For instance, although `ctangle` would indeed collate `@d` definitions (the equivalent of standard C's `#define`s) from the whole web of source code and output them together in a preliminary block at the start of the C to be compiled, it did not perform the same service for structure declarations. Nor would it predeclare functions automatically (something I found annoying enough that for some time, I used a hacky Perl script to do for me). The result was that, for some years, Chapter 1 of the Inform source code consisted of a gigantic string of data structure definitions, with commentary attached: it amounted to more than 100pp of close-type A4 when woven into a PDF, and became so gargantuan that it seemed to me a counter-example to Fred Brooks's line "give me the table structures, and I can see what the program does".

All in all, then, `CWEB`'s suitability and performance gradually deteriorated as Inform grew. At first, `CWEB` made good on its essential promise to produce an extensively documented program and an accompanying book of its source code. I believe it really would be a good tool for, say, presenting reference code for new algorithms. But a tipping point was reached at about 50,000 lines of source where `CWEB`'s blindness to the idea of modular, encapsulated code was actively hindering me from organising the source better. (While `CWEB` has notionally been extended to `C++`, it truly belongs to the prelapsarian world of early C hacking.) I needed a way to modularise the source further, and `inweb` was born.

1 Top Level

`1/pc`: *Program Control.w* The top level, which decides what is to be done and then carries this plan out.

`1/cli`: *Command Line and Errors.w* To parse the command line arguments with which inweb was called, and to handle any errors it needs to issue.

Program Control 1/pc

Purpose

The top level, which decides what is to be done and then carries this plan out.

Definitions

¶1. `inweb` is itself written as an `inweb` web. This hazardous circularity is traditional in literate-programming circles, much as people like to make compilers compile themselves: after all, if even the authors aren't willing to trust the code, why should anyone else? So this is a kind of demonstration of self-belief. `inweb` is probably fairly correct, because it is capable of tangling a program (`inweb`) which can then tangle a really complex and densely annotated program (Inform) which then passes over 1000 difficult test cases.

In my case, though, it was done simply because `inweb` had become that most evil of all system tools: a 4000-line Perl script resulting from bursts of careless invention every few months. When Woody Allen said that all literature is a footnote to Faust, he probably wasn't thinking of the Camel book, Perl's notorious manual, but he should have been. What can you say about a programming language where functions have no parameters but have to read them from an array named `$_`, where `local` defines a (partly) global variable, and where `my $a, $b` creates one local and one global, thanks to the comma being interpreted as marking a thrown-away evaluation of `my $a` (which, *obviously*, returns a value) and then an evaluation in void context of the nonexistent `$b`, which is helpfully and silently created as a global variable in the process? And yet, Perl is so quick... so easy... all that memory is yours at the slightest whim... and why work out how to derive data in format B from the data you already have in format A, when the devil on your shoulder whispers that you could just as easily store it both ways and save the trouble?

More seriously, Perl was used because it was the most standard scripting language with dynamic memory allocation available in 2003, when Inform 7 began: the largely preferable Python and Ruby had not then established themselves as fixtures.

¶2. In Perl floating-point arithmetic is used by default, since this is just what you probably wouldn't expect, and division in particular is slow in some Unix environments because it emulates floating point instructions rather than using hardware support: this was a nuisance when Inform was first ported to Unix boxes. So we gain some speed by writing what looks like a module inclusion, but is in fact a compiler `pragma`.

```
use integer;
```

¶3. Now we define the build of Inweb:

```
define $INWEB_BUILD "inweb [[Build Number]]"
```

¶4. As we will discover when we read it in, a web is either "chaptered" (its sections are distributed among Preliminaries, Chapter 1, ..., and some Appendices) or else "unchaptered", with all sections lumped together in "Sections".

```
$web_is_chaptered = 1;
```

¶5. `inweb` has a single fundamental mode of operation: on any given run, it is either tangling, weaving or analysing. These processes use the same input and parsing code, but then do very different things to produce their output, so the fork in the road is not met until halfway through `inweb`'s execution.

```
define $NO_MODE 0
define $ANALYSE_MODE 1
define $TANGLE_MODE 2
define $WEAVE_MODE 3
define $CREATE_MODE 4                    a special mode for creating a new web, not acting on an existing one

$web_mode = $NO_MODE;                    a value used to mean "not set yet"
```

¶6. It can also (in some functions, anyway) work on only selected portions of the web: such a selected piece is called a "target", and its name is called a "sigil". This is stored in the following textual variable, and can be a section name like `2/pine`, a chapter number like `12`, an appendix letter `A` or the preliminaries block `P`, or the special value `0` to mean the entire web.

```
$sigil_of_target = "0";                  By default, the entire web is the target
```

¶7. In "swarm mode", however, the user chooses a multiplicity of targets rather than just one, in which case the above value is meaningless.

```
define $SWARM_OFF 0
define $SWARM_INDEX 1                    Make index(es) as if swarming, but don't actually swarm
define $SWARM_CHAPTERS 2                 Swarm the chapters
define $SWARM_SECTIONS 3                 That, and also all of the individual sections

$swarm_mode = $SWARM_OFF;
```

¶8. When tangling, we write a file whose name is:

```
$tangle_to = "";                         This is set either by the command line or from the Contents
```

---

§**1.** This is the whole program in a nutshell, and it's a pretty old-school program: some input, some thinking, a choice of three forms of output. Perl scripts of course begin outside any function; this section is the only code at the top level in that way.

```
print "$INWEB_BUILD (Inform Tools Suite)\n";
make_command_line_settings();
if (($web_mode == $NO_MODE) || ($web_setting eq "")) ⟨Show command-line usage and exit 2⟩;
read_configuration_file();
if ($create_setting ne "") ⟨Create a new web 11⟩
else {
    ⟨Read in the web 3⟩;
    ⟨Display the statistics line 4⟩;
    if ($web_mode == $ANALYSE_MODE) ⟨Analyse the web 5⟩;
    if ($web_mode == $TANGLE_MODE) ⟨Tangle the web 6⟩;
    if ($web_mode == $WEAVE_MODE) ⟨Weave the web 10⟩;
}
if ($no_inweb_errors == 0) { exit(0); } else { exit(1); }
```

§**2.** There's no sense writing out the whole manual; this usage note is intended for people who run across inweb and have no idea what it is.

⟨Show command-line usage and exit 2⟩ ≡

```
print "[[Purpose]]\n";
print "Usage: inweb webname -action [-options] [target]\n";
print "  where 'webname' is a folder containing a web (an inweb project),\n";
print "  The most useful -action commands are:\n";
print "    -create: make a new web, creating its folder and contents\n";
print "    -tangle: make the program described in the web\n";
print "    -weave: make a human-readable booklet of the web\n";
print "  For options and less commonly used actions, see the inweb manual.\n";
exit(0);
```

This code is used in §1.

§**3.** `inweb` has basic support for a wider range of languages (the ones we need for the Inform project, really), but does detailed work only on C and on the Inform extension of C.

⟨Read in the web 3⟩ ≡

```
read_literate_source();
language_set($bibliographic_data{"Language"});
if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
    create_base_types_hash();
}
parse_literate_source();
language_set($bibliographic_data{"Language"});
if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
    parse_C_like_sections();
}
```

This code is used in §1.

§**4.** To raise the morale of the user, really:

⟨Display the statistics line 4⟩ ≡

```
print "\"", $bibliographic_data{"Title"}, "\" ";
if ($shared_structures+$private_structures > 0) {
    print $shared_structures+$private_structures, " structure(s)";
    if ($shared_structures > 0) { print ", $shared_structures shared"; }
    print ": ";
}
if ($no_chapters > 0) { print $no_chapters, " chapter(s) : "; }
print $no_sections, " section(s) : ", $no_paragraphs, " paragraph(s) : ",
$no_lines, " line(s)\n";
```

This code is used in §1.

§**5.** "Analysis" invokes any combination of four diagnostic tools:

⟨Analyse the web 5⟩ ≡
```
if ($swarm_mode != $SWARM_OFF) {
    inweb_fatal_error("only specific parts of the web can be analysed");
}
if ($catalogue_switch == 1) { catalogue_the_sections($sigil_of_target, 0); }
if ($functions_switch == 1) { catalogue_the_sections($sigil_of_target, 1); }
if ($voids_switch == 1) { catalogue_void_pointers($sigil_of_target); }
if ($make_graphs_switch == 1) { compile_graphs($sigil_of_target); }
if ($scan_switch == 1) { scan_line_categories($sigil_of_target); }
```
This code is used in §1.

§**6.** We can tangle to any one of what might be several targets, numbered upwards from 0. Target 0 always exists, and is the main program forming the web. For many webs, this will in fact be the only target, but `inweb` also allows marked sections of a web to be independent targets – the idea here is to allow an Appendix in the web to contain a configuration file, or auxiliary program, needed for the main program to work; this might be written in a quite different language from the rest of the web, and tangles to a different output, but needs to be part of the web since it's essential to an understanding of the whole system.

In this section we determine `$tn`, the target number wanted, and `$tangle_to`, the filename of the tangled code to write. This may have been set at the command line (in which case `$tangle_setting` contains the filename), but otherwise we impose a sensible choice based on the target.

⟨Tangle the web 6⟩ ≡
```
my $tn = 0;
if ($sigil_of_target eq "0") {
    ⟨Work out main tangle destination 7⟩;
} elsif (exists($sigil_section{$sigil_of_target})) {
    ⟨Work out an independent tangle destination, from one section of the web 8⟩;
} else {
    ⟨Work out an independent tangle destination, from one chapter of the web 9⟩;
}
if ($tangle_to eq "") { inweb_fatal_error("no tangle destination known"); }
$tangle_to = $web_setting."Tangled/".$tangle_to;
if ($tangle_setting ne "") { $tangle_to = $tangle_setting; }
tangle_source($tn, $tangle_to);
print "Tangled: $tangle_to\n";
```
This code is used in §1.

§**7.** Here the target number is 0, and the tangle is of the main part of the web, which for many small webs will be the entire thing.

⟨Work out main tangle destination 7⟩ ≡
```
$tn = 0;
if (exists $bibliographic_data{"Short Title"}) {
    $tangle_to = $bibliographic_data{"Short Title"};
} else {
    $tangle_to = $bibliographic_data{"Title"};
}
language_set($bibliographic_data{"Main Language"});
$tangle_to .= language_file_extension();
```
This code is used in §6.

§**8.** If someone tangles, say, `2/eg` then the default filename is "Example Section".

⟨Work out an independent tangle destination, from one section of the web 8⟩ ≡
```
$tn = $section_tangle_target[$sigil_section{$sigil_of_target}];
if ($tn == 0) { inweb_fatal_error("section cannot be independently tangled"); }
$tangle_to = $section_leafname[$sigil_section{$sigil_of_target}];
```

This code is used in §6.

§**9.** If someone tangles, say, `B` then the default filename is "Appendix B".

⟨Work out an independent tangle destination, from one chapter of the web 9⟩ ≡
```
my $cn;
for ($cn=0; $cn<$no_chapters; $cn++) {
    if ($chapter_tangle_target[$cn] > 0) {
        if ($sigil_of_target eq $chapter_sigil[$cn]) {
            my $brief = $chapter_title[$cn];
            $brief =~ s/^.*?\:\s*//;
            $tangle_to = $brief;
            $tn = $chapter_tangle_target[$cn];
            last;
        }
    }
}
if ($tn == 0) {
    inweb_fatal_error("only the entire web, or specific sections, can be tangled");
}
```

This code is used in §6.

§**10.** Weaving is not actually easier, it's just more thoroughly delegated:

⟨Weave the web 10⟩ ≡
```
if ($swarm_mode == $SWARM_OFF) {
    my $shall_we_open = $open_pdf_switch;
    if ($shall_we_open == -1) {                    i.e., if it wasn't set at the command line
        if ($open_command_configuration ne "") { $shall_we_open = 1; }
        else { $shall_we_open = 0; }
    }
    weave_sigil($sigil_of_target, $shall_we_open);
} else { weave_swarm(); }
```

This code is used in §1.

§**11.** Lastly, here's a small utility for creating a new web – a slightly fiddly business, so just about worth automating.

⟨Create a new web 11⟩ ≡

```
system("mkdir -pv '".$create_setting."'");
system("mkdir -pv '".$create_setting."/Figures'");
system("mkdir -pv '".$create_setting."/Materials'");
system("mkdir -pv '".$create_setting."/Sections'");
system("mkdir -pv '".$create_setting."/Tangled'");
system("mkdir -pv '".$create_setting."/Woven'");
system("cp -nv '".$path_to_inweb_setting."inweb/Materials/Contents.w' '".
    $create_setting."'");
system("cp -nv '".$path_to_inweb_setting."inweb/Materials/Main.w' '".
    $create_setting."/Sections'");
```

This code is used in §1.

Command Line and Errors 1/cli

Purpose

To parse the command line arguments with which inweb was called, and to handle any errors it needs to issue.

Definitions

¶1. The command line options set the following variables. True/false options have `*_switch` variables, while textual ones are `*_setting`. (They also cause the two main program modes to be set: these are defined in Program Control.)

```
$analyse_structure_setting = "";
$catalogue_switch = 0;
$functions_switch = 0;
$convert_graphs_switch = 0;
$make_graphs_switch = 0;
$open_pdf_switch = -1;
$scan_switch = 0;
$tangle_setting = "";
$verbose_about_input_switch = 0;
$voids_switch = 0;
$web_setting = "";
$create_setting = "";
$only_setting = "";
$complete_PDF_leafname = "Complete.pdf";
```

`-analyse-structure`*: name of typedef struct to show usage of*
`-catalogue`*: print catalogue of sections*
`-functions`*: print catalogue of functions within sections*
`-convert-graphs`*: run* `-make-graphs` *output through 'dot'*
`-make-graphs`*: compile code to generate graphs*
`-open-pdf`*: open any woven PDF in the OS once it is made*
`-scan`*: simply show the syntactic scan of the source*
`-tangle X`*: the pathname X, if supplied*
`-verbose-about-input`*: print names of files read to stdout*
`-voids`*: print void pointer usage*
set by `-web`*: project folder relative to cwd*
set by `-create`*: name of a new project to create*
set by `-only`*: restrict a swarm to this chapter*
overridden when `-only` *is used*

¶2. In order to run, `inweb` needs to know where it is installed – this enables it to find its configuration file, the macros file, and so on. Unless told otherwise on the command line, we'll assume `inweb` is present in the current working directory.

```
$path_to_inweb_setting = "";
```

¶3. The two external tools we may need to be present, beyond standard shell tools:

```
$pdftex_configuration = 'pdftex';
$dot_utility_configuration = 'dot';
```

¶4. We count the errors in order to be able to exit with a suitable exit code.

```
$no_inweb_errors = 0;
```

---

§1. Reading the command line.

```
sub make_command_line_settings {
    my $i;
    my $targets = 0;
    ARGUMENT: for ($i=0; $i<=$#ARGV; $i++) {
        my $opt = $ARGV[$i];
        my $non_switch_follows = 0;
        if (($i < $#ARGV) && (not($ARGV[$i+1] =~ m/^-/))) { $non_switch_follows = 1; }
        if ($opt =~ m/^-/) ⟨Parse this as a switch 2⟩
        else {
            if ($web_setting eq "") { $web_setting = $opt.'/'; }
            else ⟨Parse this as a target sigil 7⟩;
        }
    }
}
```

§2.

⟨Parse this as a switch 2⟩ ≡

```
    $opt =~ s/^\-\-/\-/;                                    allow a doubled-dash as equivalent to one
    if ($opt eq "-test-extensions") {
        print "(Test inweb's implementation of Inform's C extensions)\n";
        full_test_double_squares();
        exit(0);
    }
    if ($opt eq "-verbose-about-input") {
        $verbose_about_input_switch = 1; next ARGUMENT;
    }
    if ($opt eq "-at") {
        if ($non_switch_follows == 1) {
            $path_to_inweb_setting = $ARGV[$i+1]; $i++; next ARGUMENT;
        }
        inweb_fatal_error("-at must be followed by the pathname where inweb lives");
    }
    ⟨Parse analysis options 3⟩;
    ⟨Parse weaver options 4⟩;
    ⟨Parse tangler options 5⟩;
    ⟨Parse creation option 6⟩;
    inweb_fatal_error("unknown command line switch: $opt");
```

This code is used in §1.

§3.

⟨Parse analysis options 3⟩ ≡

```
if ($opt eq "-analyse-structure") {
    if ($non_switch_follows) {
        $analyse_structure_setting = $ARGV[$i+1]; $i++;
        enter_main_mode($ANALYSE_MODE);
        next ARGUMENT;
    }
    inweb_fatal_error("-analyse-structure must be followed by a structure name");
}
if ($opt eq "-catalogue") {
    $catalogue_switch = 1; enter_main_mode($ANALYSE_MODE); next ARGUMENT;
}
if ($opt eq "-functions") {
    $functions_switch = 1; enter_main_mode($ANALYSE_MODE); next ARGUMENT;
}
if ($opt eq "-voids") {
    $voids_switch = 1; enter_main_mode($ANALYSE_MODE); next ARGUMENT;
}
if ($opt eq "-make-graphs") {
    $make_graphs_switch = 1; enter_main_mode($ANALYSE_MODE); next ARGUMENT;
}
if ($opt eq "-convert-graphs") {
    $make_graphs_switch = 1; $convert_graphs_switch = 1;
    enter_main_mode($ANALYSE_MODE); next ARGUMENT;
}
if ($opt eq "-scan") {
    $scan_switch = 1; enter_main_mode($ANALYSE_MODE); next ARGUMENT;
}
```

This code is used in §2.

§4.

⟨Parse weaver options 4⟩ ≡

```
if ($opt eq "-weave") {
    enter_main_mode($WEAVE_MODE); next ARGUMENT;
}
if ($opt eq "-open") {
    $open_pdf_switch = 1; enter_main_mode($WEAVE_MODE); next ARGUMENT;
}
if ($opt eq "-closed") {
    $open_pdf_switch = 0; enter_main_mode($WEAVE_MODE); next ARGUMENT;
}
if ($opt eq "-only") {
    if ($non_switch_follows) {
        $only_setting = $ARGV[$i+1]; $i++;
        enter_main_mode($WEAVE_MODE);
        next ARGUMENT;
    }
    inweb_fatal_error("-only must be followed by a chapter number or appendix letter");
}
```

This code is used in §2.

§**5.**

⟨Parse tangler options 5⟩ ≡
```
if ($opt eq "-tangle") {
    enter_main_mode($TANGLE_MODE); next ARGUMENT;
}
if ($opt eq "-tangle-to") {
    if ($non_switch_follows) {
        $tangle_setting = $ARGV[$i+1]; $i++;
        enter_main_mode($TANGLE_MODE); next ARGUMENT;
    }
    inweb_fatal_error("-tangle-to must be followed by a filename to write");
}
```

This code is used in §2.

§**6.** The single creation option is an exception, since it doesn't act on an existing web:

⟨Parse creation option 6⟩ ≡
```
if ($opt eq "-create") {
    if ($non_switch_follows == 1) {
        $create_setting = $ARGV[$i+1]; $web_setting = $create_setting; $i++;
        enter_main_mode($CREATE_MODE);
        next ARGUMENT;
    }
    inweb_fatal_error("-create must be followed by the pathname of a web");
}
```

This code is used in §2.

§**7.** A command-line argument not starting with a hyphen, and not already soaked up by a preceding argument such as `-tangle-to`, is a target sigil such as `2/eg` or `B`. Note that appendices are lettered A to O, but that `P` means the preliminary pages.

⟨Parse this as a target sigil 7⟩ ≡
```
$targets++;
if ($targets > 1) { inweb_fatal_error("at most one target may be given"); }
$swarm_mode = $NO_SWARM;
if ($opt eq "index") {
    $swarm_mode = $SWARM_INDEX;
} elsif ($opt eq "chapters") {
    $swarm_mode = $SWARM_CHAPTERS;
} elsif ($opt eq "sections") {
    $swarm_mode = $SWARM_SECTIONS;
} elsif ($opt eq "all") {
    $sigil_of_target = "0";
} elsif ($opt =~ m/\//) {
    $sigil_of_target = $opt;
} elsif ($opt =~ m/^\d+$/) {
    $sigil_of_target = $opt;
} elsif ($opt =~ m/^[A-O]$/) {
    $sigil_of_target = $opt;
} elsif ($opt =~ m/^P$/) {
    $sigil_of_target = $opt;
} else {
```

```
        inweb_error("target not recognised: $opt");
        print "The legal targets are:\n";
        print "   all: complete web\n";
        print "   P: all preliminaries\n";
        print "   1: Chapter 1 (and so on)\n";
        print "   A: Appendix A (and so on, up to Appendix O)\n";
        print "   3/eg: section with abbreviated name \"3/eg\" (and so on)\n";
        print "   index: HTML page indexing project\n";
        print "   chapters: all individual chapters\n";
        print "   sections: all individual sections\n";
        exit(1);
    }
```

This code is used in §1.

§**8.** We can only be in a single mode at a time:

```
sub enter_main_mode {
    my $new_mode = $_[0];
    if ($web_mode == $NO_MODE) { $web_mode = $new_mode; }
    if ($web_mode != $new_mode) {
        inweb_fatal_error("can only do one at a time - weaving, tangling or analysing");
    }
}
```

§**9. The configuration file.** `indoc` has only a tiny configuration file, mainly to point it to other tools it may need to use. Note that it needs none of these for tangling, so it doesn't actually matter if the settings are wrong in such a run.

```
sub read_configuration_file {
    my $cl;
    open(CONFIG, $path_to_inweb_setting.'inweb/Materials/inweb-configuration.txt')
        or die "inweb: can't open configuration file";
    while ($cl = <CONFIG>) {
        $cl =~ m/^\s*(.*?)\s*$/; $cl = $1;
        if ($cl =~ m/^\#/) { next; }                                    skip comment lines
        if ($cl eq "") { next; }                                        skip blank lines
        if ($cl =~ m/^(\S+)\s*=\s*(.*?)$/) {
            my $setting = $1;
            my $value = $2;
            ⟨Make one of the configuration settings 10⟩;
        }
    }
    close CONFIG;
}
```

§**10.** There's very little to see here:

⟨Make one of the configuration settings 10⟩ ≡

```
    if ($setting eq "pdftex") { $pdftex_configuration = $value; next; }
    if ($setting eq "dot") { $dot_utility_configuration = $value; next; }
    if ($setting eq "open-command") { $open_command_configuration = $value; next; }
    inweb_error("inweb: bad configuration setting ($setting)");
```

This code is used in §9.

§**11. Error messages.** Ah, they kill you; or they don't.

```
sub inweb_fatal_error {
    my $message = $_[0];
    print STDERR "inweb: $message\n";
    exit(1);
}
sub inweb_error {
    my $message = $_[0];
    $no_inweb_errors++;
    print STDERR "inweb: $message\n";
}
sub inweb_error_at {
    my $message = $_[0];
    my $file = $_[1];
    my $line = $_[2];
    $no_inweb_errors++;
    print STDERR "inweb: $message\n";
    print STDERR "  (", $file, " line ", $line, ")\n";
}
sub inweb_error_at_program_line {
    my $message = $_[0];
    my $i = $_[1];
    my $sec = $line_sec[$i];
    inweb_error_at($message, $section_pathname_relative_to_web[$sec], $line_source_file_line[$i]);
}
```

2 Parsing a Web

`2/read`: *Reading Sections.w* To read the Contents section of the web, and through that each of the other sections in turn, and to collate all of this material into one big linear array of source-code lines.

`2/lcats`: *Line Categories.w* We are going to need to identify lines of source code as falling into 18 different categories – the start of a definition, a piece of a comment, and so on. In this section we define constants to enumerate these categories, and provide a debugging routine to show the classification we are using on the web we've just read.

`2/parse`: *The Parser.w* To work through the program read in, assigning each line its category, and noting down other useful information as we go.

`2/ident`: *Identifiers.w* To find the identifier names of functions and structures, and monitor in which sections of the program they are used; and so to police the accuracy of declarations at the head of each section.

Reading Sections 2/read

Purpose

To read the Contents section of the web, and through that each of the other sections in turn, and to collate all of this material into one big linear array of source-code lines.

Definitions

¶1. This section describes a single, one-time event which happens early in every run of `inweb`: the reading in of the entire text of the web into memory, and the building of a suitable data structure to hold all this information, neatly docketed and reflecting its three layers – chapter, section, line.

It begins by reading the contents section, which really isn't a section at all (and perhaps we shouldn't pretend that it is by the use of the `.w` file extension, but we probably want it to have the same file extension, and its syntax is chosen so that syntax-colouring for regular sections doesn't make it look odd). When the word "section" is used in the `inweb` code, it almost always means "section other than the contents".

When the reading in is complete, the following variables have their final values:

```
$no_lines = 0;                                  Total lines in literate source, excluding contents
$no_sections = 0;                               Again, excluding contents: it will eventually be at least 1
$no_chapters = 0;                               Similarly, this will be at least 1
$no_tangle_targets = 0;                         And again
```

¶2. Once this phase is completed, the following arrays exist. We are pretty profligate with memory, but we can afford to be. (In the Parser section, we shall build even further arrays, but those aren't discussed here.)

For each individual chapter, numbered from 0 to `$no_chapters` minus 1 in order of declaration on the contents page, we store the following:

(C1) `chapter_sigil[]` is the sigil for the chapter owning this section: that is, the textual abbreviations used to identify chapters on the command line and elsewhere: `P` for Preliminaries, `7` for Chapter 7, `C` for Appendix C. In an apparently unchaptered project, the single chapter is called Sections and has sigil `S`: every section belongs to it.

(C2) `$chapter_title[]` is the title of the chapter, exactly as it is given in the Contents: for instance, "Chapter 3: Fresh Water Fish".

(C3) `$chapter_rubric[]` is the textual description of the chapter's purpose, as given in the Contents.

(C4) `$chapter_woven_pdf_leafname[]` is a leafname like `Appendix-A.pdf`, suitable for the woven PDF version of this individual chapter.

(C5) `$chapter_tangle_target[]` is the tangle ID number for tangling, or 0 if this is not marked for independent tangling.

¶3. Second, for each section other than the Contents (numbered 0 up to `$no_sections` minus 1, and in their canonical reading order, i.e., the order in which a human reader sees them in the typeset book):

(S1) `$section_chap[]` gives the chapter number to which the section belongs, which is between 0 and `$no_chapters` minus 1. (In a project without declared chapters, this will be 0, meaning the "Sections" pseudo-chapter.)

(S2) `$section_extent[]` is the number of lines in the section (which might be one more than the number of lines in its file, if it includes a chapter heading line).

(S3) `$section_pathname_relative_to_web[]` is the pathname of the section file within its own web folder.

(S4) `$section_leafname[]` is the leafname at the end of that. Obviously, it could be derived from (S3) without too much effort, but we often want it and it costs little to store, and since neither (S3) nor (S4) ever change we may as well keep both. It is also convenient to have a cross-referencing hash which lets us invert array number (S4): `$section_number_from_leafname{}`.

(ST) `$section_tangle_target[]` is the tangle ID number for the section – generally 0 to mean that it's part of the main tangle.

¶4. As this last implies, individual tangle targets are numbered from 0 up to `$no_tangle_targets` minus 1. Other than target 0, each target contains only a single section or a single chapter.

(A1) `$tangle_target_language[]` is the language of the tangle.

¶5. Lastly, for each individual line (numbered 0 up to `$no_lines` minus 1, and in their canonical reading order):

(L1) `$line_text[]` is the text as read in.

(L2) `$line_text_raw[]` is a duplicate for now. Later, the `$line_text[]` will be altered by parsing, whereas this won't, so it gives us access to the original form of the line.

(L3) `$line_sec[]` gives the section number (an index to the section arrays above) of the originating section. For the interleaved chapter heading lines placed between automatically chapters, the section number is the one just about to start, that is, the first section of the new chapter.

(L4) `$line_source_file_line[]` gives the line number, from 1, within the section file; or, for interleaved chapter headings, is 0.

---

§**1. Reading the contents page.** We read in the contents first, since that triggers everything else, by forcing reads of each section file in turn.

At the end, we have lines numbered 0 to `$no_lines-1` read in: this is the complete literate source of the web, so that we have the equivalent in memory of one long web file. Most of the lines come straight from the source files, but a few chapter heading lines are inserted if this is a multi-chapter web.

```
sub read_literate_source {
    read_contents_page("Contents.w");
}
```

§**2.** So here goes. The contents section has a syntax quite different from all other sections, and sets out bibliographic information about the web, the sections and their organisation, and so on.

```
sub read_contents_page {
    my $pathname_of_contents = $web_setting.$_[0];
    my $cline;                                          Line read in from the contents file
    my $clc = 0;                          Line count within contents section (for use in error messages)
    my $scanning_bibliographic_block = 1;                                  Top bit, or bottom bit?
    my $scanning_chapter_purpose = 0;                          Reading the bit just after the new chapter?
    my $path_to_chapter_being_read = "";                        Where sections in the current chapter live
    my $titling_line_to_insert = "";                                    To be inserted automagically
    $bibliographic_data{"Declare Section Usage"} = "On";                                    The default
    $bibliographic_data{"Strict Usage Rules"} = "Off";                                      The default
    open CP, $pathname_of_contents
        or die "inweb: can't open contents section at: $pathname_of_contents\n";
    while ($cline = <CP>) {
        $cline =~ s/\s+$//; $clc++;
        if ($cline eq "") { $scanning_bibliographic_block = 0; next; }
        if ($scanning_bibliographic_block == 1) ⟨Read the bibliographic data block at the top 3⟩;
        ⟨Read the roster of sections at the bottom 7⟩;
    }
    close CP;
    if ($verbose_about_input_switch == 1) {
        print "Read contents section: '", $leafname, "' (", $cline, " lines)\n";
    }
    ⟨Check that the required bibliographic data was supplied 4⟩;
    ⟨Create the main tangle target 6⟩;
}
```

§**3.** The bibliographic data gives lines in any order specifying values of variables with fixed names; a blank line ends the block.

⟨Read the bibliographic data block at the top 3⟩ ≡

```
    if ($cline =~ m/^(.*?)\:\s*(.*?)\s*$/) {
        my $key = $1;
        my $value = $2;
        if ($key eq "License") { $key = "Licence"; }                              alias US to UK spelling
        if (($key eq "Title") || ($key eq "Short Title") || ($key eq "Author") ||
            ($key eq "Purpose") || ($key eq "Licence") ||
            ($key eq "Build Number") || ($key eq "Language") ||
            ($key eq "Index Extras") || ($key eq "Index Template") ||
            ($key eq "Cover Sheet") || ($key eq "Namespaces") ||
            ($key eq "Strict Usage Rules") || ($key eq "Declare Section Usage")) {
            $bibliographic_data{$key} = $value;
            if ((($key eq "Strict Usage Rules") || ($key eq "Declare Section Usage") ||
                ($key eq "Namespaces")) &&
                ($value ne "On") && ($value ne "Off")) {
                inweb_error_at("This setting must be 'On' or 'Off'", "Contents.w", $clc);
            }
        } else { inweb_error_at("no such bibliographic datum as '$key'", "Contents.w", $clc); }
    } else { inweb_error_at("expected 'Setting: Value' but found '$cline'", "Contents.w", $clc); }
    next;
```

This code is used in §2.

§**4.** Some bibliographic data settings are compulsory:

⟨Check that the required bibliographic data was supplied 4⟩ ≡
```
ensure_setting_of("Title"); ensure_setting_of("Author");
ensure_setting_of("Purpose"); ensure_setting_of("Language");
$bibliographic_data{"Inweb Build"} = $INWEB_BUILD;
$bibliographic_data{"Main Language"} = $bibliographic_data{"Language"};
```

This code is used in §2.

§**5.** Which requires the following:

```
sub ensure_setting_of {
    my $setting = $_[0];
    if (not (exists ($bibliographic_data{$setting}))) {
        inweb_fatal_error("The Contents.w section does not specify '$setting: ...'");
    }
}
```

§**6.** Tangle target 0 is the main program contained in the web we're reading; the programming language used by this will the one given by the `Language:` field in the contents section.

⟨Create the main tangle target 6⟩ ≡
```
$tangle_target_language[0] = $bibliographic_data{"Language"};
$no_tangle_targets++;
```

This code is used in §2.

§**7.** In the bulk of the contents, we find indented lines for sections and unindented ones for chapters.

⟨Read the roster of sections at the bottom 7⟩ ≡
```
$cline =~ m/^(\s*)(.*?)$/;                                    A pattern which cannot fail to match
my $whitespace = $1; local $title = $2;
if ($whitespace eq "") {
    if ($cline =~ m/^\"(.*)$/) { $scanning_chapter_purpose = 1; $cline = $1; }
    if ($scanning_chapter_purpose == 1) ⟨Record the purpose of the current chapter 8⟩
    else ⟨Read about a new chapter 9⟩;
} else ⟨Read about, and read in, a new section 12⟩;
next;
```

This code is used in §2.

§**8.** After a declared chapter heading, subsequent lines form its purpose, until we reach a closed quote: we then stop, but remove the quotation marks. Because we like a spoonful of syntactic sugar on our porridge, that's why.

⟨Record the purpose of the current chapter 8⟩ ≡
```
if ($cline =~ m/^(.*)\"\s*$/) { $cline = $1; $scanning_chapter_purpose = 0; }
if ($chapter_rubric[$no_chapters-1] ne "") {
    $chapter_rubric[$no_chapters-1] .= " ";
}
$chapter_rubric[$no_chapters-1] .= $cline;
next;
```

This code is used in §7.

§**9.** The title tells us everything we need to know about a chapter:

⟨Read about a new chapter 9⟩ ≡

```
my $new_chapter_sigil = "";                                   e.g., P, 1, 2, 3, A, B, ...
my $pdf_leafname = "";
my $ind_target = 0;
if ($title =~ m/^(.*?)\s*\(\s*Independent\s*(.*?)\s*\)\s*$/)
    ⟨Mark this chapter as an independent tangle target 10⟩;
if ($title eq "Sections") {
    $new_chapter_sigil = "S"; $path_to_chapter_being_read = "Sections/";
    $titling_line_to_insert = "";
    $pdf_leafname = "Sections.pdf";
    $web_is_chaptered = 0;
} elsif ($title eq "Preliminaries") {
    $new_chapter_sigil = "P"; $path_to_chapter_being_read = "Preliminaries/";
    $titling_line_to_insert = "";
    $pdf_leafname = "Preliminaries.pdf";
    $web_is_chaptered = 1;
} elsif ($title =~ m/^Chapter\s+(\d+)\:\s*(.*?)$/) {
    $new_chapter_sigil = $1; $path_to_chapter_being_read = "Chapter $1/";
    $titling_line_to_insert = $title.".";
    $pdf_leafname = "Chapter-$1.pdf";
    $web_is_chaptered = 1;
} elsif ($title =~ m/^Appendix\s+([A-O])\:\s*(.*?)$/) {
    $new_chapter_sigil = $1; $path_to_chapter_being_read = "Appendix $1/";
    $titling_line_to_insert = $title.".";
    $pdf_leafname = "Appendix-$1.pdf";
    $web_is_chaptered = 1;
} else {
    inweb_error_at("segment '$title' not understood", "Contents.w", $clc);
    print STDERR "(Must be 'Chapter <number>: Title', 'Appendix <letter A to O>: Title',\n";
    print STDERR "'Preliminaries' or 'Sections')\n";
}
⟨Create the new chapter with these details 11⟩;
```

This code is used in §7.

§**10.** A chapter whose title marks it as Independent becomes a new tangle target, with the same language as the main web unless stated otherwise.

⟨Mark this chapter as an independent tangle target 10⟩ ≡

```
$title = $1; $lang = $2;
$current_tangle_target = ++$no_tangle_targets;
$ind_target = $current_tangle_target;
$tangle_target_language[$no_tangle_targets] = $bibliographic_data{"Language"};
if ($lang ne "") { $tangle_target_language[$no_tangle_targets] = $lang; }
```

This code is used in §9.

§**11.**

⟨Create the new chapter with these details 11⟩ ≡

```
$chapter_sigil[$no_chapters] = $new_chapter_sigil;
$chapter_title[$no_chapters] = $title;
$chapter_rubric[$no_chapters] = "";
$chapter_tangle_target[$no_chapters] = $ind_target;
$chapter_woven_pdf_leafname[$no_chapters] = $pdf_leafname;
if ($ind_target == 0) { $current_tangle_target = 0; }
$no_chapters++;
```

This code is used in §9.

§**12.** That's enough on creating chapters. This is the more interesting business of registering a new section within a chapter – more interesting because we also read in and process its file.

⟨Read about, and read in, a new section 12⟩ ≡

```
my $source_file_extension = ".w";
if ($title =~ m/^(.*?)\s*\(\s*Independent\s*(.*?)\s*\)\s*$/) {
    ⟨Mark this section as an independent tangle target 13⟩;
} else {
    $section_tangle_target[$no_sections] = $current_tangle_target;
}
$section_chap[$no_sections] = $no_chapters - 1;
$section_extent[$no_sections] = 0;
my $path_to_section = $path_to_chapter_being_read.$title.$source_file_extension;
$path_to_section =~ s/ Template\.i6t$/\.i6t/;
$section_pathname_relative_to_web[$no_sections] = $path_to_section;
$section_leafname[$no_sections] = $path_to_section;
if ($section_leafname[$no_sections] =~ m/\/([^\/]*?)$/) {
    $section_leafname[$no_sections] = $1;
}
$section_number_from_leafname{$section_leafname[$no_sections]} = $no_sections;
read_file($path_to_section, $titling_line_to_insert, $no_sections);
$no_sections++;
```

This code is used in §7.

§**13.** Just as for chapters, but a section which is an independent target with language "Inform 6" is given the filename extension `.i6t` instead of `.w`. This is to conform with the naming convention used within Inform, where I6 template files – `inweb` files with language Inform 6 – are given the file extensions `.i6t`.

⟨Mark this section as an independent tangle target 13⟩ ≡

```
$title = $1; $lang = $2;
$section_tangle_target[$no_sections] = ++$no_tangle_targets;
$tangle_target_language[$no_tangle_targets] = $bibliographic_data{"Language"};
if ($lang ne "") {
    $tangle_target_language[$no_tangle_targets] = $lang;
    if ($lang eq "Inform 6") { $source_file_extension = ".i6t"; }
}
```

This code is used in §12.

§14. Reading source files. Note that we assume here that trailing whitespace on a line (up to but not including the line break) is not significant in the language being tangled for.

```
sub read_file {
    my $path_relative_to_web = $_[0];
    my $titling_line_for_this_chapter = $_[1];
    my $section_number = $_[2];
    my $file_line_count = 0;
    my $pathname = $web_setting.$path_relative_to_web;
    if (($titling_line_for_this_chapter ne "") &&
        ($titling_of_current_chapter ne $titling_line_for_this_chapter)) {
        $titling_of_current_chapter = $titling_line_for_this_chapter;
        $nl = '@*** '.$titling_of_current_chapter;
        ⟨Accept this as a line belonging to this section and chapter 15⟩;
    }
    open SECTIONF, $pathname or die "inweb: Unable to open $pathname\n";
    while ($nl = <SECTIONF>) {
        $file_line_count++;
        $nl =~ s/\s+$//;                                    remove trailing whitespace and the line break
        ⟨Accept this as a line belonging to this section and chapter 15⟩;
    }
    close SECTIONF;
    if ($verbose_about_input_switch == 1) {
        print "Read section: '", $pathname, "' (", $file_line_count, " lines)\n";
    }
}
```

§15.

⟨Accept this as a line belonging to this section and chapter 15⟩ ≡

The text, with a spare copy protected from the parser's meddling:

```
$line_text[$no_lines] = $nl;
$line_text_raw[$no_lines] = $nl;
```

And where it occurs in the web:

```
$line_sec[$no_lines] = $section_number;
$line_source_file_line[$no_lines] = $file_line_count;
```

And keep count:

```
$no_lines++;
$section_extent[$section_number]++;                    Not the same as the file line count!
```

This code is used in §14.

§16. A utility routine we need because the titling lines are special. Lines with a count of 1 begin their sections, obviously, but there are occasional lines with a count of 0 as well: these are the inserted chapter headings, and only occur in a chaptered web.

```
sub line_is_in_heading_position {
    my $i = $_[0];
    if (($line_source_file_line[$i] == 0) ||
        ($line_source_file_line[$i] == 1)) { return 1; }
    return 0;
}
```

Line Categories 2/lcats

Purpose

We are going to need to identify lines of source code as falling into 18 different categories – the start of a definition, a piece of a comment, and so on. In this section we define constants to enumerate these categories, and provide a debugging routine to show the classification we are using on the web we've just read.

Definitions

¶1. The line categories are enumerated as follows:

```
define $NO_LCAT 0                                    none set as yet
define $COMMENT_BODY_LCAT 1
define $MACRO_DEFINITION_LCAT 2
define $BAR_LCAT 3
define $INDEX_ENTRY_LCAT 4
define $PURPOSE_LCAT 5
define $INTERFACE_LCAT 6
define $GRAMMAR_LCAT 7
define $DEFINITIONS_LCAT 8
define $PARAGRAPH_START_LCAT 9
define $BEGIN_VERBATIM_LCAT 10
define $TEXT_EXTRACT_LCAT 11
define $BEGIN_DEFINITION_LCAT 12
define $GRAMMAR_BODY_LCAT 13
define $INTERFACE_BODY_LCAT 14
define $CODE_BODY_LCAT 15
define $CONT_DEFINITION_LCAT 19
define $SOURCE_DISPLAY_LCAT 16
define $TOGGLE_WEAVING_LCAT 17
define $COMMAND_LCAT 18
```

---

§**1.** The scanner is intended for debugging `inweb`, and simply shows the main result of reading in and parsing the web:

```
sub scan_line_categories {
    my $sigil = $_[0];
    my $confine_to = -1;
    my $sn;
    my $i;
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_sigil[$sn] eq $sigil) {
            $confine_to = $sn;
        }
    }
    for ($i=0; $i<$no_lines; $i++) {
        if (($confine_to >= 0) && ($confine_to != $line_sec[$i])) { next; }
        print sprintf("%04d  %16s  %s\n",
            $i, category_name($line_category[$i]), $line_text[$i]);
    }
}
```

§**2.** And the little routine which prints category names to `stdout`:

```
sub category_name {
    my $cat = $_[0];
    if ($cat == $COMMENT_BODY_LCAT) { return "COMMENT_BODY"; }
    elsif ($cat == $MACRO_DEFINITION_LCAT) { return "MACRO_DEFINITION"; }
    elsif ($cat == $BAR_LCAT) { return "BAR"; }
    elsif ($cat == $INDEX_ENTRY_LCAT) { return "INDEX_ENTRY"; }
    elsif ($cat == $PURPOSE_LCAT) { return "PURPOSE"; }
    elsif ($cat == $INTERFACE_LCAT) { return "INTERFACE"; }
    elsif ($cat == $GRAMMAR_LCAT) { return "GRAMMAR"; }
    elsif ($cat == $DEFINITIONS_LCAT) { return "DEFINITIONS"; }
    elsif ($cat == $PARAGRAPH_START_LCAT) { return "PARAGRAPH_START"; }
    elsif ($cat == $BEGIN_VERBATIM_LCAT) { return "BEGIN_CODE"; }
    elsif ($cat == $TEXT_EXTRACT_LCAT) { return "TEXT_EXTRACT"; }
    elsif ($cat == $BEGIN_DEFINITION_LCAT) { return "BEGIN_DEFINITION"; }
    elsif ($cat == $GRAMMAR_BODY_LCAT) { return "GRAMMAR_BODY"; }
    elsif ($cat == $INTERFACE_BODY_LCAT) { return "INTERFACE_BODY"; }
    elsif ($cat == $CODE_BODY_LCAT) { return "CODE_BODY"; }
    elsif ($cat == $SOURCE_DISPLAY_LCAT) { return "SOURCE_DISPLAY"; }
    elsif ($cat == $TOGGLE_WEAVING_LCAT) { return "TOGGLE_WEAVING"; }
    elsif ($cat == $COMMAND_LCAT) { return "COMMAND"; }
    elsif ($cat == $CONT_DEFINITION_LCAT) { return "CONT_DEFINITION"; }
    else { return "? cat $cat"; }
}
```

The Parser 2/parse

Purpose

To work through the program read in, assigning each line its category, and noting down other useful information as we go.

Definitions

¶1. Like the earlier reading-in stage, the parsing stage always happens in every run of `inweb`, and it happens once only. We use the arrays already built to represent the lines, sections and chapters, and we go on to create new ones, as follows. We know everything about chapters already. However, for the sections we add:

(S5) `$section_sigil[]`, which is the sigil such as `4/fish` identifying the section briefly, and is found on the titling line before the colon. (The hash `$sigil_section{}` provides a convenient inverse to this.) Similarly, `$section_namespace[]` is the optional namespace quoted.

(S6) `$section_toc[]`, which is the text of the brief table of contents of the section, in TEX marked-up form. (The weaver adds this as a headnote when setting the section.)

(S7) `$section_purpose[]`, the text of the purpose given by `@Purpose:`.

(S8) `$section_no_pars[]`, the number of paragraphs in the section.

¶2. And for the lines we add:

(L5) `$line_category[]`, which assigns every line one of the `*_LCAT` values (see the Line Categories section for details): when parsing is complete, no line is permitted still to have category `$NO_LCAT`.

(L6) `$line_operand[]`: the type and value of this depends on the line category, but basically it supplies some useful information about what the line constructs.

(L7) `$line_operand_2[]` is a second, even more optional operand.

(L8) `$line_paragraph_number[]` is the paragraph number within the section to which the line belongs (or 0, if it is somewhere at the top before paragraphs have begun). This doesn't entirely specify the paragraph, though, as there might be a paragraph 2 in Definitions above the bar and then another in the code stuff below: so we must also record the

(L9) `$line_paragraph_ornament[]` which is the TEX for the paragraph "ornament", a ¶ or § sign, according to whether we're above or below the bar.

(L10) `$line_starts_paragraph_on_new_page[]` which is simply a flag used with paragraph headings to cause a page throw, or not.

(L11) `$line_is_comment[]` is set if the weaver should treat the line as regular-type commentary, and clear if it should treat it as verbatim program code or other such quoted matter. (We could deduce this from the line category, but it's more convenient not to have to.)

(L12) `$line_occurs_in_CWEB_macro_definition[]` is a flag needed by the tangler.

¶3. Command-category lines set their (first) operand to one of the following:

```
define $PAGEBREAK_CMD 1
define $BNF_GRAMMAR_CMD 2
define $THEMATIC_INDEX_CMD 3
define $FIGURE_CMD 4
define $INDEX_UNDER_CMD 5
```

¶4. With the parsing done, we discover the remaining layer of structure within the web: the paragraphs. The following count variable exists only for the sake of the statistics line printed out at the end of `inweb`'s run: it counts the number of marked paragraphs in each section, plus a notional 1 for the header part of the section (titling, purpose, interface, grammar, etc.).

```
$no_paragraphs = 0;
```

¶5. Each paragraph has a "weight", which is a measure of its typographic prominence. The paragraph weights are:

(0) an ordinary paragraph with no subheading
(1) paragraph with bold label (operand 2: the title)
(2) paragraph with a subheading (operand 2: the title)
(3) paragraph with a chapter heading (operand 2: the title)

¶6. Some paragraphs define `CWEB` macros in angled brackets, and those need special handling. We parse the following data on them, using hashes keyed by the name of the macro without angle brackets attached:

(W1) `$cweb_macros_paragraph_number{}` is the paragraph number of the definition, as printed in small type after the name in any usage.
(W2) `$cweb_macros_start{}` is the first line number of the definition body.
(W3) `$cweb_macros_end{}` is the last.
(W4) `$cweb_macros_surplus_bit{}` is the possible little first fragment of macro definition on the same line as the declaration opens, following the equals sign. (It's poor style to write macros this way, I think, but for better compatibility with `CWEB`...)

¶7. We also look out for Backus-Naur form grammar in the program, collating it all together. (Obviously, this was written specifically for the Inform documentation, but a lot of programs have grammars of one kind or another, so it might as well be available for any project.) At this parsing stage, collation is all we do. The grammar lines are numbered 0 to `$bnf_grammar_lines` minus 1, and we have arrays:

(G1) `$bnf_grammar[]` holding the text of the line,
(G2) `$bnf_grammar_source[]` holding the line number it came from in the web.

§**1. Sequence of parsing.** As can be seen, we work in two passes. It could all be done in one if we needed to, but this is plenty fast enough, is tidier and makes the code simpler.

```
sub parse_literate_source {
    ⟨Initialise the new arrays created in parsing phase 2⟩;
    determine_line_categories();                                    Pass 1
    establish_canonical_section_names();                            Pass 2
    ⟨Count the number of paragraphs 3⟩;
}
```

§**2.** See Definitions above.

⟨Initialise the new arrays created in parsing phase 2⟩ ≡

```
    my $i;
    for ($i=0; $i<$no_lines; $i++) {
        $line_is_comment[$i] = 0;
        $line_category[$i] = $NO_LCAT;
        $line_paragraph_number[$i] = 0;
        $line_paragraph_ornament[$i] = "";
        $line_starts_paragraph_on_new_page[$i] = 0;
    }
    for ($i=0; $i<$no_sections; $i++) {
        $section_sigil[$i] = "";
        $section_toc[$i] = "";
        $section_purpose[$i] = "";
        $section_no_pars[$i] = 0;
    }
```

This code is used in §1.

§**3.**

⟨Count the number of paragraphs 3⟩ ≡

```
    my $i;
    $no_paragraphs = 0;
    for ($i=0; $i<$no_lines; $i++) {
        if ($line_category[$i] == $PARAGRAPH_START_LCAT) {
            $no_paragraphs++;
        }
    }
```

This code is used in §1.

§4. **First parse.** On the first run through, we assign a category to every line of the source, and set operands as appropriate. We also note down the `$section_purpose[]` and `$section_sigil[]` as we pass.

```
sub determine_line_categories {
    $comment_mode = 1;
    $grammar_mode = 0;
    CATEGORISATION: for ($i=0; $i<$no_lines; $i++) {
        $line_is_comment[$i] = $comment_mode;
        $line_category[$i] = $COMMENT_BODY_LCAT;                    Until set otherwise down below
        my $l = $line_text[$i];
        if (line_is_in_heading_position($i)) {
            language_set($tangle_target_language[$section_tangle_target[$line_sec[$i]]]);
            ⟨Rewrite the heading in CWEB-style paragraph notation but continue 5⟩;
        }
        if ($l =~ m/^\[\[\s*(.*?)\s*\]\]\s*$/)
            ⟨Parse the line as an Inweb command 6⟩;
        if (language_pagebreak_comment($l)) {
            $line_category[$i] = $COMMAND_LCAT;
            $line_operand[$i] = $PAGEBREAK_CMD;
            next CATEGORISATION;
        }
        if ($l =~ m/^\s*\@\<\s*(.*?)\s*\@\>\s*\=\s*(.*?)$/)
            ⟨Note that a CWEB macro is defined here 7⟩;
        if ($l =~ m/^\@(\S*)(.*?)$/) {
            my $command = $1;
            my $remainder = $2;
            my $succeeded = 0;
            ⟨Parse and deal with a structural marker 8⟩;
            if ($succeeded == 0) {
                inweb_error_at_program_line("don't understand @".$command." at:", $i);
            }
        }
        if ($grammar_mode == 1) ⟨Store this line as part of the BNF grammar 15⟩;
        if ($comment_mode == 1) ⟨This is a line destined for the commentary 13⟩;
        if ($comment_mode == 0) ⟨This is a line destined for the verbatim code 14⟩;
    }
}
```

§**5.** In a convenient if faintly sordid manoeuvre, we rewrite a heading line – either a Chapter heading or the titling line of a section – as if it were a CWEB-style paragraph break of a superior kind (`@*`, which is super, or `@**` which is even more so). This code is a residue of the time when `inweb` was essentially a reimplementation of CWEB.

⟨Rewrite the heading in CWEB-style paragraph notation but continue 5⟩ ≡

```
if ($l =~ m/^(([A-Za-z0-9_]+::\s*)*)(\S+\/[a-z][a-z0-9]+)\:\s+(.*)\s*$/) {
    $section_namespace[$line_sec[$i]] = $1;
    $section_sigil[$line_sec[$i]] = $3;
    $sigil_section{$1} = $line_sec[$i];
    $l = '@* '.$4;
    $section_namespace[$line_sec[$i]] =~ s/\s*//g;
} else {
    if (($l =~ m/^Chapter /) || ($l =~ m/^Appendix /)) {
        $l = '@** '.$l;
    }
}
```

This code is used in §4.

§**6.** Note that we report an error if the command isn't one we recognise: we don't simply ignore the squares and let it fall through into the tangler.

⟨Parse the line as an Inweb command 6⟩ ≡

```
my $comm = $1;
$line_category[$i] = $COMMAND_LCAT;
if ($comm =~ m/^(.*?)\s*\:\s*(.*)\s*$/) {
    $comm = $1;
    $line_operand_2[$i] = $2;
}
if ($comm eq "Page Break") { $line_operand[$i] = $PAGEBREAK_CMD; }
elsif ($comm eq "BNF Grammar") { $line_operand[$i] = $BNF_GRAMMAR_CMD; }
elsif ($comm eq "Thematic Index") { $line_operand[$i] = $THEMATIC_INDEX_CMD; }
elsif ($comm =~ m/^Index Under (.*)$/) {
    $thematic_indices{$1} .= $line_operand_2[$i]."...".$i."|";
    $line_operand[$i] = $INDEX_UNDER_CMD;
}
elsif ($comm eq "Figure") { $line_operand[$i] = $FIGURE_CMD; }
else { inweb_error("unknown command $l\n"); }
$line_is_comment[$i] = 1;
```

This code is used in §4.

§**7.** These are woven and tangled much like other comment lines, but we need to notice the macro definition, of course:

⟨Note that a CWEB macro is defined here 7⟩ ≡

```
$line_category[$i] = $MACRO_DEFINITION_LCAT;
$line_operand[$i] = $1;                          The name of the macro
$line_operand_2[$i] = $2;                        Any beginning of its content on the same line
$comment_mode = 0;
$line_is_comment[$i] = 0;
$code_lcat_for_body = $CODE_BODY_LCAT;           Code follows on subsequent lines
next CATEGORISATION;
```

This code is used in §4.

§**8.** A structural marker is introduced by an `@` in column 1, and is a structural division in the current section. There are a number of possibilities. One, where the line is a macro definition, we have already dealt with.

There are some old `CWEB` syntaxes surviving here: `@1` is the same as `@`, `@2` the same as `@p`, `@3` the same as `@*` which is reserved (by us, though not by `CWEB`) for section headings, `@4` the same as `@**` which is reserved (ditto) for chapter headings.

In practice `@*` and `@**` headings do begin on new pages, but that is accomplished by the TEX macros which typeset them, so we don't need to force a page break ourselves: which is why we don't set the `$line_starts_paragraph_on_new_page[]` flag for those.

⟨Parse and deal with a structural marker 8⟩ ≡

```
if ($line_category[$i] == $MACRO_DEFINITION_LCAT) { $succeeded = 1; }
⟨Deal with structural markers above the bar 9⟩;
⟨Deal with the code and extract markers 10⟩;
⟨Deal with the define marker 11⟩;
my $weight = -1;
if ($command eq "") { $weight = 0; }
if ($command eq "p") { $weight = 1; }
if ($command eq "pp") { $weight = 1; $line_starts_paragraph_on_new_page[$i] = 1; }
if ($command eq "*") { $weight = 2; }
if ($command eq "**") { $weight = 2; }
if ($command eq "***") { $weight = 3; }
if ($command =~ m/\*(\d)/) { $weight = eval($1)-1; }
if ($weight >= 0) ⟨Begin a new paragraph of this weight 12⟩;
```

This code is used in §4.

§**9.** The five markers down as far as the bar:

⟨Deal with structural markers above the bar 9⟩ ≡

```
if ($command eq "Purpose:") {
    $line_category[$i] = $PURPOSE_LCAT;
    $line_operand[$i] = $remainder;
    $line_is_comment[$i] = 1;
    $section_purpose[$line_sec[$i]] = $remainder;
    $i2 = $i+1;
    while ($line_text[$i2] =~ m/[a-z]/) {
        $section_purpose[$line_sec[$i]] .= " ".$line_text[$i2];
        $i2++;
    }
    next CATEGORISATION;
}
if ($command eq "Interface:") {
    $line_category[$i] = $INTERFACE_LCAT;
    $line_is_comment[$i] = 1;
    $grammar_mode = 0;
    next CATEGORISATION;
}
if ($command eq "Grammar:") {
    $line_category[$i] = $GRAMMAR_LCAT;
    $line_is_comment[$i] = 1;
    $grammar_mode = 1;
    next CATEGORISATION;
}
```

```
    if ($command eq "Definitions:") {
        $line_category[$i] = $DEFINITIONS_LCAT;
        $line_is_comment[$i] = 1;
        $grammar_mode = 0;
        next CATEGORISATION;
    }
    if ($command =~ m/\-\-\-\-+/) {
        $line_category[$i] = $BAR_LCAT;
        $line_is_comment[$i] = 1;
        $command = "";
        $grammar_mode = 0;
        next CATEGORISATION;
    }
```

This code is used in §8.

§**10.** These have identical behaviour except for whether or not to tangle what follows:

⟨Deal with the code and extract markers 10⟩ ≡

```
    if (($command eq "c") || ($command eq "x")) {
        $line_category[$i] = $BEGIN_VERBATIM_LCAT;
        if ($command eq "x") { $code_lcat_for_body = $TEXT_EXTRACT_LCAT; }
        else { $code_lcat_for_body = $CODE_BODY_LCAT; }
        $comment_mode = 0;
        $succeeded = 1;
        $line_text[$i] = "";
    }
```

This code is used in §8.

§**11.** Definitions are intended to translate to C preprocessor macros, Inform 6 `Constant`s, and so on.

⟨Deal with the define marker 11⟩ ≡

```
    if ($command eq "d") {
        $line_category[$i] = $BEGIN_DEFINITION_LCAT;
        $code_lcat_for_body = $CONT_DEFINITION_LCAT;
        if ($remainder =~ m/^\s*(\S+)\s+(.+?)\s*$/) {
            $line_operand[$i] = $1;                          Name of term defined
            $line_operand_2[$i] = $2;                        Value
        } else {
            $line_operand[$i] = $remainder;                  Name of term defined
            $line_operand_2[$i] = "";                        No value given
        }
        $comment_mode = 0;
        $line_is_comment[$i] = 0;
        $succeeded = 1;
    }
```

This code is used in §8.

§**12.** The noteworthy thing here is the way we fool around with the text on the line of the paragraph opening. This is a hangover from the fact that we are using a line-based system (with significant `@`s in column 1 only) yet imitating `CWEB`, which is based on escape characters and sees only a stream where there's nothing special about line breaks. Anyway,

```
@p The chronology of French weaving. Auguste de Papillon (1734-56) soon
```

is split so that "The chronology of French weaving" is stored as operand 2 (the title) while the line is rewritten to read simply

```
Auguste de Papillon (1734-56) soon
```

so that it can go on to by woven or tangled exactly as the succeeding lines will be.

⟨Begin a new paragraph of this weight 12⟩ ≡

```
$grammar_mode = 0;
$comment_mode = 1;
$line_is_comment[$i] = 1;
$line_category[$i] = $PARAGRAPH_START_LCAT;
$line_operand[$i] = $weight;                                   Weight
$line_operand_2[$i] = "";                                      Title
if (($weight > 0) && ($remainder =~ m/\s+(.*?)\.\s*(.*)$/)) {
    $line_operand_2[$i] = $1;                                  Title up to the full stop
    $line_text[$i] = $2;                                       And then some regular material
} else {
    if ($remainder =~ m/\s+(.+?)$/) { $line_text[$i] = $1; }
    else { $line_text[$i] = ""; }
}
$succeeded = 1;
```

This code is used in §8.

§**13. Any last special rules for lines.** In commentary mode, we look for any fancy display requests, and also look to see if there are interface lines under an `@Interface:` heading (because that happens in commentary mode, though this isn't obvious).

⟨This is a line destined for the commentary 13⟩ ≡

```
if ($line_text[$i] =~ m/^\>\>\s+(.*?)\s*$/) {
    $line_category[$i] = $SOURCE_DISPLAY_LCAT;
    $line_operand[$i] = $1;
}
if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
    if (scan_line_for_interface($i)) {
        $line_category[$i] = $INTERFACE_BODY_LCAT;
    }
}
```

This code is used in §4.

§**14.** Note that in an `@d` definition, a blank line is treated as the end of the definition. (This is unnecessary for C, and is a point of difference with `CWEB`, but is needed for languages which don't allow multi-line definitions.)

⟨This is a line destined for the verbatim code 14⟩ ≡

```
if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
    scan_line_for_interface($i);
}
if (($line_category[$i] != $BEGIN_DEFINITION_LCAT) &&
    ($line_category[$i] != $COMMAND_LCAT)) {
    $line_category[$i] = $code_lcat_for_body;
}
if (($line_category[$i] == $CONT_DEFINITION_LCAT) &&
    ($line_text[$i] =~ m/^\s*$/)) {
    $line_category[$i] = $COMMENT_BODY_LCAT;
    $code_lcat_for_body = $COMMENT_BODY_LCAT;
}
if (language_and_so_on($line_text[$i])) {
    $line_category[$i] = $TOGGLE_WEAVING_LCAT;
}
if ($web_language == $C_FOR_INFORM_LANGUAGE)
    ⟨Detect some NI-specific oddities and throw them into the BNF grammar 16⟩;
```

This code is used in §4.

§**15. The BNF grammar.** Material underneath the optional `@Grammar:` heading is stored away thus for the time being:

⟨Store this line as part of the BNF grammar 15⟩ ≡

```
$line_category[$i] = $GRAMMAR_BODY_LCAT;
$bnf_grammar[$bnf_grammar_lines] = $l;
$bnf_grammar_source[$bnf_grammar_lines] = $i;
$bnf_grammar_lines++;
next CATEGORISATION;
```

This code is used in §4.

§**16.** It's convenient to auto-detect some grammar from specific code patterns in the source code to `NI`, though of course this mustn't be done for any other project:

⟨Detect some NI-specific oddities and throw them into the BNF grammar 16⟩ ≡

```
if ($l =~ m/the_debugging_aspects\[\] =/) {
    $bnf_grammar_source[$bnf_grammar_lines] = $i;
    $bnf_grammar[$bnf_grammar_lines++] = "<debugging-aspect>";
    $da_baseline = $i;
}
if (($l =~ m/^\s+\{\s+\"(.+?)\"\,\s+\"(.*?)\"\,\s+\"(.*?)\"/)
    && ($i < $da_baseline+200)) {
    $bnf_grammar[$bnf_grammar_lines] = "    := $1";
    if ($2 ne "") { $bnf_grammar[$bnf_grammar_lines] .= " $2"; }
    if ($3 ne "") { $bnf_grammar[$bnf_grammar_lines] .= " $3"; }
    $bnf_grammar_lines++;
}
```

This code is used in §14.

§17. Second parse. We work out the values of `$line_paragraph_number[]`, `$line_paragraph_ornament[]`, `$section_toc[]`, `$section_no_pars[]`; and we also create the hashes of details for the `CWEB` macro definitions.

```
sub establish_canonical_section_names {
    my $par_number_counter = 0;
    my $pnum = 0;
    my $ornament = "\\S";
    my $i;
    my $toc = "";
    my $toc_range_start = 0;
    my $toc_name = "";
    ⟨Begin a new section TOC 19⟩;
    for ($i=0; $i<$no_lines; $i++) {
        if (($line_category[$i] == $DEFINITIONS_LCAT) ||
            ($line_category[$i] == $PURPOSE_LCAT)) {
            $ornament = "\\P";
            $par_number_counter = 0;                          Start counting paras from 1
        }
        if ($line_category[$i] == $BAR_LCAT) {
            $ornament = "\\S";
            $par_number_counter = 0;                          Start counting paras from 1 again
            ⟨Complete any named paragraph range now half-done in the TOC 21⟩;
        }
        if ($line_category[$i] == $PARAGRAPH_START_LCAT) {
            $par_number_counter++;
            if ($line_operand[$i] == 2) {                     weight 2, so a section heading
                if ($line_sec[$i] > 0)                        i.e., if this isn't the first section
                    ⟨Complete the TOC from the section just ended 22⟩;
                ⟨Begin a new section TOC 19⟩;
            }
            $section_no_pars[$line_sec[$i]]++;
            if ($line_operand[$i] == 1)                       weight 1: a named @p or @pp paragraph
                ⟨Begin a new named paragraph range in the TOC 20⟩;
        }
        $line_paragraph_ornament[$i] = $ornament;
        $line_paragraph_number[$i] = $par_number_counter;
        if ($line_category[$i] == $MACRO_DEFINITION_LCAT)
            ⟨Record details of this CWEB macro 18⟩;
        $line_csn[$i] = $section_sigil[$line_sec[$i]].
            '.$'.$line_paragraph_ornament[$i].'$'.
            $line_paragraph_number[$i];
    }
    ⟨Complete the TOC from the very last section 23⟩;
}
```

§**18.** We now record the four hashes of data about the macro just found:

⟨Record details of this CWEB macro 18⟩ ≡
```
my $j = $i;
my $name = $line_operand[$i];
$cweb_macros_paragraph_number{$name} = $par_number_counter;
$cweb_macros_start{$name} = $j+1;
$cweb_macros_surplus_bit{$name} = $line_operand_2[$j];
$j++;
while (($j<$no_lines) && ($line_category[$j] == $CODE_BODY_LCAT)) {
    $line_occurs_in_CWEB_macro_definition[$j] = 1;
    $j++;
}
$cweb_macros_end{$name} = $j;
```

This code is used in §17.

§**19.** The table of contents (TOC) for a section is abbreviated, and we generate it sequentially, with the following mildly convoluted logic. The idea is to end up with something like "¶2-5. Blue cheese; ¶6. Danish blue; ¶7-11. Gouda; §1-3. Slicing the cheese; §4. Wrapping in greaseproof paper." We start writing a TOC entry when hitting a new named paragraph, and write just the "¶7" part; only when we hit the next named paragraph, or the bar dividing the section, or the end of the section, or the end of the whole web, do we complete it by adding "-11. Gouda".

⟨Begin a new section TOC 19⟩ ≡
```
$toc = ""; $toc_range_start = 0;
```

This code is used in §17.

§**20.** The first of the four ways we might have to complete a part-made entry.

⟨Begin a new named paragraph range in the TOC 20⟩ ≡
```
$toc = complete_toc_chunk($toc, $toc_range_start, $par_number_counter-1, $toc_name);
$toc_range_start = $par_number_counter; $toc_name = $line_operand_2[$i];
if ($toc eq "") { $toc = $section_sigil[$line_sec[$i]]."."; }
else { $toc .= "; "; }
$toc .= '$'.$ornament.'$'.$par_number_counter;
```

This code is used in §17.

§**21.** This one is when we hit the bar.

⟨Complete any named paragraph range now half-done in the TOC 21⟩ ≡
```
$toc = complete_toc_chunk($toc, $toc_range_start, $par_number_counter, $toc_name);
$toc_range_start = 0;
```

This code is used in §17.

§**22.** And here we run into the end of section.

⟨Complete the TOC from the section just ended 22⟩ ≡
```
$toc = complete_toc_chunk($toc, $toc_range_start, $par_number_counter-1, $toc_name);
set_toc_for_section($line_sec[$i]-1, $toc);
```

This code is used in §17.

§**23.** And here, the end of the whole web.

⟨Complete the TOC from the very last section 23⟩ ≡

```
    $toc = complete_toc_chunk($toc, $toc_range_start, $par_number_counter, $toc_name);
    set_toc_for_section($line_sec[$no_lines-1], $toc);
```

This code is used in §17.

§**24.** Note the TEX tie ~ to prevent line breaks between the paragraph number and names, and that we use a number range only when there are multiple paragraphs.

```
sub complete_toc_chunk {
    my $toc = $_[0];
    my $toc_range_start = $_[1];
    my $pnum = $_[2];
    my $toc_name = $_[3];
    if ($toc_range_start != 0) {
        if ($pnum != $toc_range_start) { $toc .= "-".$pnum; }
        $toc .= "~".$toc_name;
    }
    return $toc;
}
```

§**25.** Once we have determined the table of contents for a section, we call the following:

```
sub set_toc_for_section {
    my $sect_no = $_[0];
    my $toc = $_[1];
    $section_toc[$sect_no] = $toc;
    return;                                          delete this line for a debugging trace, viz...
    if ($toc eq "") {
        print "Warning: ", $section_sigil[$sect_no], " has no TOC.\n";
    } else {
        print $section_sigil[$sect_no], " has TOC ", $toc, "\n";
    }
}
```

Identifiers 2/ident

Purpose

To find the identifier names of functions and structures, and monitor in which sections of the program they are used; and so to police the accuracy of declarations at the head of each section.

Definitions

¶1. In some languages, we can detect function and/or structure definitions, and also the places where they are used: we use this to provide a little bit of syntax-colouring in the weaver, and to provide little footnotes to sections explaining cross-section function calls.

We also check the section's declared `@Interface`, if the web is set up with "Strict Usage Rules" to "On" – though the default is "Off".

¶2. Yet further section arrays are created here:

(S9) `$section_errors[]` will hopefully be the empty text, but otherwise accumulates any errors we find in the section's declared interface to other sections. (In languages or projects not using interfaces, this is always empty and has no effect.)

(S10) `$section_declared_a_service[]` is set to indicate that the section's `@Interface` says it provides functions to the entire web and therefore doesn't want to have to detail all its client sections. (Again, this has no effect in a web not using interface declarations.)

(S11) `$section_declared_structure_interface[]` is a concatenation of the structure ownership part of the `@Interface` for a section, with blank lines removed.

(S12) `$section_I6_template_identifiers[]` is colon-separated list of Inform 6 routines invoked from the template interpreter, and is always blank unless the language is C-for-Inform.

(S13) `$section_declared_used_by_block[]` is a block of used-by declarations from the section's `@Interface`, but rejigged into section list form (see below).

(S14) `$section_declared_uses_block[]` is the same, but for the "uses" part of the `@Interface`.

(S15) `$section_correct_uses_block[]` is what (S13) ought to be, based on the actual function calls observed in the web source.

(S16) `$section_correct_used_by_block[]` is what (S14) ought to be, based on the actual function calls observed in the web source.

(S17) `$section_correct_structures_block[]` is what (S11) ought to be, based on the actual structure usage observed in the web source.

¶3. In order to detect function definitions and structure elements, we need to know all of the types in use, and in particular we make a hash called `$base_types{}` whose keys are the base type names available to the code in the web: for instance, `int` and `char` are valid base types, and so, for convenience's sake, is `void`. These are only base types, so types resulting from qualification or pointer constructions, like `unsigned int` or `char *`, do not count.

¶4. As we go, we also accumulate data on functions, which are stored in hashes indexed by the C identifier of the function name:

(F1) `$functions_line{}` is the line number where the function declaration begins.
(F2) `$functions_declared_scope{}` is the declared scope, if any, of the function in "row of stars" notation, that is:
  (0) `""` means the scope is section-wide (which if we are checking strictly means the function may only be used in the current section);
  (1) `"*"` is illegal and never happens;
  (2) `"**"` means the scope is chapter-wide;
  (3) `"***"` means the scope is web-wide;
  (4) `"****"` is used only in C-for-Inform mode, and means the scope extends to allowing calls from the Inform template interpreter, i.e., under the control of data files outside the web;
  (5) `"*****"` means the function is `main()`, where execution begins.
(F3) `$functions_return_type{}` is the C return type of the function, e.g., `"void"` or `"unsigned char **"`;
(F4) `$functions_arglist{}` is the argument list in the declaration of the function, e.g., `"int x, int y"`;
(F5) `$functions_usage_count{}` is the number of times the function is referred to throughout the web (including in its definition, so this is always at least 1);
(F6) `$functions_usage_section_list{}` records which other sections call the function, if any do; the value is a section list concatenating string entries in the form `:C-L`, where `C` is the chapter number and `L` is the leafname of the section;
(F7) `$functions_usage_concisely_described{}` is essentially the same data, but in the more concise form of `:S` where `S` is the section number – so, for instance, `:4:13` means it is found in sections 4 and 13;
(F8) `$functions_usage_verbosely_described{}` is a verbose form of the same thing, used in comments in the tangled code (and thus unnecessary, really, but it was useful during the debugging of `inweb`).

¶5. Where exactly are functions used? We track this with `$function_usage_hash{}`, whose keys are not function names but are pairs in the form

```
function_name+Section Name.w
```

Thus the existence of a key means that the function with this name is used in the section with this name: note that `+` is not legal in a C identifier. Again, definition counts as usage, so every function is listed in this hash under its own section name.

¶6. We also want to track structures, their members, and the usage of these. For convenience of looping over all structures, a hash `$structures{}` is created with no useful value but such that its list of keys is exactly the set of structures. (The list of keys of, say, (T1) below would do just as well, but that would make the loops look a bit misleading.)

Structures then use the following hashes, whose keys are their typedef names:

(T1) `$structure_declaration_line{}` is the line number on which the `typedef` for this structure is made.
(T2) `$structure_declaration_end{}` is the final line number of its `typedef`, which is always $\geq$ (T1).
(T3) `$structure_CREATE_in_section{}`, which is used in C-for-Inform mode only, records the second number in which the `CREATE(whatever)` macro is used for the structure `whatever`. (In NI, each individual structure is allowed to be created in only one section – a self-imposed rule to improve the encapsulation of the code.)
(T4) `$structure_owner{}` is the section number of the section deemed to be the "owner" of this structure, a concept used to help `inweb` to enforce some encapsulation of code around related data.
(T5) `$structure_usage_section_list{}` is a list of leafnames of the sections which access this structure's members, – not those which merely use pointers to them, which doesn't break any encapsulation rules.
(T6) `$structure_ownership_summary{}` is similar, but in a more compressed form and using sigils; this helps the weaver to print neat footnotes under structure definitions.
(T7) `$structure_incorporates{}` is a list of structures incorporated into this one, that is, which are used as members of the current one. (Pointers do not count.)

(T8) `$structure_declaration_tangled{}` is a flag indicating that the structure's `typedef` block has now been tangled.

¶7. And members use the following hashes, whose keys are the member names:

(M1) The hash `$member_usage_section_list{}` is a section list of each distinct section using the member, which is indexed by its name. A section list is a convenient string representation of a set of sections: for instance,

```
"-Mushrooms and Toadstools.w:-Badgers.w:"
```

Each entry in the list begins with a dash and ends with a colon, and the latter character is not permitted in a section leafname, so this is an unambiguous representation.

(M2) `$member_structures{}` is a list of all structures using members with this name, with a ':' placed after each structure name: for instance, `"parse_node:tree_fern:"`.

¶8. The following global variable keeps track of which `typedef struct` definition we are currently scanning:

```
$current_struct = "";
```

---

§**1. More detailed parsing for C-like languages.** This will be another distinct phase within `inweb`: something which happens once, towards the middle of the run, though this time it happens only for projects written in a C-like language. But first we have to step back in time a little and look at something which happened during the Parser phase of `inweb`: the discovery of the types.

§**2.** The hash `$base_types{}` is used simply as a set of keys: we initialise it to the names of some built-in types, then add more from any `typedef struct` definitions we find in the web.

```
sub create_base_types_hash {
    my $i;
    $base_types{"char"} = 1;
    $base_types{"int"} = 1;
    $base_types{"float"} = 1;
    $base_types{"void"} = 1;
    $base_types{"FILE"} = 1;
    for ($i=0; $i<$no_lines; $i++) {
        if ($line_text[$i] =~ m/^typedef\s+struct\s+(\S+)\s+/) {
            $base_types{$1} = 1;
        }
    }
}
```

§**3. Parsing a section interface.** Most of what we want to do is to go through the web and look to see how functions and structures are being used, as we shall see. In a web with Strict Usage Rules set to "On", the results of that parsing have to exactly match the various declarations made in the sections. In such a web, each section has to contain an `@Interface:` block like the following: it is required to declare

(a) any functions it provides to the Inform 6 template interpreter (this is only used in C-for-Inform mode, of course);
(b) any other sections which call its functions (those it is "used by");
(c) any other sections whose functions it calls (those it "uses");
(d) any structures it creates, and whether they are accessed only in this section ("private") or from other sections (in which case it goes on to say exactly which ones).

```
-- Defines {-callv:index_sounds}
-- Used by Chapter 5/Literal Productions.w
-- Used by Chapter 10/Bibliographic Data.w
-- Used by Chapter 10/Figures.w
-- Uses Chapter 1/Platform-Specific Definitions.w
-- Uses Chapter 2/Sound Durations.w
-- Uses Chapter 10/Figures.w
-- Owns struct blorb_sound (private)
```

§**4.** The Parser section offers us the chance to grab and parse any such line of interface specification by calling this routine on each possible line. If we recognise the line as being an interface syntax, we return 1; if not, we return 0. That whole process will be complete *before* the call to `parse_C_like_sections`, so the hashes we create here will all be in existence in time to be used by `parse_C_like_sections` and its subroutines.

The scanner is used not only to spot parts of the `@Interface`, but also to notice any typedefs of structures which fly by in the code. In those cases we return 0, though, as we don't want to make the Parser categorise those lines as part of an interface.

```
sub scan_line_for_interface {
    my $i = $_[0];
    my $l = $line_text[$i];
    my $secname = $section_leafname[$line_sec[$i]];
    ⟨Try a structure definition part of the interface 5⟩;
    if ($web_language == $C_FOR_INFORM_LANGUAGE)
        ⟨Try an Inform template interpreter part of the interface 10⟩;
    ⟨Try a uses or used by part of the interface 6⟩;
    if ($current_struct ne "")
        ⟨Spot the declaration of any structure member in the current typedef 9⟩;
    ⟨Notice the beginning of a typedef structure definition 7⟩;
    ⟨Notice the end of a typedef structure definition 8⟩;
    return 0;
}
```

§**5.** We do little more than bottle this up for later:

⟨Try a structure definition part of the interface 5⟩ ≡
```
if ($l =~ m/^\-\-\s+Owns struct (\S+)\s+/) {
    $section_declared_structure_interface[$line_sec[$i]] .= $l."\n";
    $structure_declared_owner{$1} = $secname;
    return 1;
}
if ($l =~ m/^\s+\!\- shared with/) {
    $section_declared_structure_interface[$line_sec[$i]] .= $l."\n";
    return 1;
}
```

This code is used in §4.

§**6.**

⟨Try a uses or used by part of the interface 6⟩ ≡
```
if ($l =~ m/^\-\-\s+Used by Chapter (\d+)\/(.*?)\s*$/) {
    $section_declared_used_by_block[$line_sec[$i]] .= ':'.$1.'-'.$2;
    return 1;
}
if ($l =~ m/^\-\-\s+Uses Chapter (\d+)\/(.*?)\s*$/) {
    $section_declared_uses_block[$line_sec[$i]] .= ':'.$1.'-'.$2;
    return 1;
}
if ($l =~ m/^\-\-\s+Service\: used widely\s*$/) {
    $section_declared_a_service[$line_sec[$i]] = 1;
    return 1;
}
```

This code is used in §4.

§**7.** We exclude any structures whose names are made with the C preprocessor concatenation operator `##`, as anything with macros creating multiple structures is going to be waaaaay too hard for us to analyse. (The Memory section of `NI` does this.)

⟨Notice the beginning of a typedef structure definition 7⟩ ≡
```
if ($l =~ m/typedef\s+struct\s+(\S+)/) {
    $candidate = $1;
    if (not ($candidate =~ m/\#\#/)) {
        $structure_declaration_line{$candidate} = $i;
        $current_struct = $candidate;
        $structures{$candidate} = 1;
    }
}
```

This code is used in §4.

§**8.** The definition is required by us to end at its first close brace:

⟨Notice the end of a typedef structure definition 8⟩ ≡
```
if (($current_struct ne "") && ($l =~ m/^\}/)) {
    $structure_declaration_end{$current_struct} = $i;
    $current_struct = "";
}
```
This code is used in §4.

§**9.** When one structure contains another one – as opposed to merely a pointer to another one – we need to take note, as this will affect the order in which we have to tangle their definitions.

⟨Spot the declaration of any structure member in the current typedef 9⟩ ≡
```
if ($l =~ m/^\s*struct\s+([A-Za-z_][A-Za-z0-9_]*)\s+(\**)(.*?)\s*\;/) {
    The member is either another structure or a pointer to one
    if ($2 eq "") {                                    An actual incorporation of another structure
        $structure_incorporates{$current_struct} .= '->'.$1;
    }
    member_detected($current_struct, $1, $2, $3, 1);
} elsif ($l =~ m/^\s*([A-Za-z_][A-Za-z0-9_]*)\s+()\(\*(.*?)\)\(.*\)\;/) {
    The member is a function pointer
    member_detected($current_struct, $1, $2, $3, 0);
} elsif ($l =~ m/^\s*([A-Za-z_][A-Za-z0-9_]*)\s+(\**)(.*?)\s*\;/) {
    The member has a base type which is not a structure
    member_detected($current_struct, $1, $2, $3, 0);
}
```
This code is used in §4.

§**10.** For the format of Inform template commands, see the I6 template files. These declarations basically refer to Inform 6 identifier names used in compilation.

⟨Try an Inform template interpreter part of the interface 10⟩ ≡
```
if ($l =~ m/^\-\-\s+Defines \{-callv*\:(.*?)\}\s*$/) {
    $id = $1; $id =~ s/::/__/g;
    $section_I6_template_identifiers[$line_sec[$i]] .= $id.':';
    return 1;
}
if ($l =~ m/^\-\-\s+Defines \{-array\:(.*?)\}\s*$/) {
    $id = $1; $id =~ s/::/__/g;
    $section_I6_template_identifiers[$line_sec[$i]] .= $id.'_array:';
    return 1;
}
if ($l =~ m/^\-\-\s+Defines \{-routine\:(.*?)\}\s*$/) {
    $id = $1; $id =~ s/::/__/g;
    $section_I6_template_identifiers[$line_sec[$i]] .= $id.'_routine:';
    return 1;
}
```
This code is used in §4.

§**11.** When a new member is detected, we need to register it in a dictionary of all structure members used in the web. These are defined in C with lines like

```
int one, two[10];
```

which would lead to the function call

```
member_detected("what_have_you", "int", "", "one, two[10]", 0);
```

being made. We need to uncouple a list such as `"one, two[10]"` and remove the punctuation and array extents to obtain the actual member names present: in this case `one` and `two`.

```
sub member_detected {
    my $structure = $_[0];
    my $return_type = $_[1];
    my $return_type_pointer_stars = $_[2];
    my $member_name = $_[3];
    my $vouched_for = $_[4];
    ⟨Recursively uncouple the member list 12⟩;
    $member_name =~ s/\[.*$//;
    if (exists($blacklisted_members{$member_name})) { return; }
    if ((exists $base_types{$return_type}) || ($vouched_for == 1)) {
        $member_structures{$member_name} .= $structure.":";
        $member_types{$member_name} .= " ".$return_type.$return_type_pointer_stars;
    } else {
        inweb_error("member '".$member_name."' of structure '".$structure.
            "' has unknown type '".$return_type."'");
    }
}
```

§**12.**

⟨Recursively uncouple the member list 12⟩ ≡

```
    if ($member_name =~ m/^\s*(.*?)\,\s*(.*?)\s*$/) {
        my $left_chunk = $1;
        my $right_chunk = $2;
        member_detected($structure, $return_type, $return_type_pointer_stars,
            $left_chunk, $vouched_for);
        member_detected($structure, $return_type, $return_type_pointer_stars,
            $right_chunk, $vouched_for);
        return;
    }
```

This code is used in §11.

§13. The parsing-C-like-sections stage. We can now get on with the main narrative of this section: what happens when `inweb` parses a section of a C-like-language web to look at its functions, its structures, and (if Strict Usage Rules is "On") also its `@Interface` declarations. The check comes in two parts, with any errors reported in a unified way at the end:

```
sub parse_C_like_sections {
    my $i;
    for ($i=0; $i<$no_sections; $i++) { $section_errors[$no_sections] = ""; }
    ⟨Stage I: police functions and section usage 14⟩;
    ⟨Stage II: police structures and member usage 15⟩;
    report_any_interface_errors_found();
}
```

§14.

⟨Stage I: police functions and section usage 14⟩ ≡

```
    find_function_definitions();
    scan_identifiers_in_source();
    find_actual_section_usage();
    check_interface_declarations_for_uses_and_used_by();
```

This code is used in §13.

§15.

⟨Stage II: police structures and member usage 15⟩ ≡

```
    find_structure_ownership();
    establish_structure_usage();
    check_interface_declarations_for_structures();
    if ($web_language == $C_FOR_INFORM_LANGUAGE) { check_uniqueness_of_structure_members(); }
```

This code is used in §13.

§16. Recognising C function definitions. A function has to be declared inside code, or inside an `@d` definition:

```
sub find_function_definitions {
    my $i;
    for ($i=0; $i<$no_lines; $i++) {
        if (($line_category[$i] == $CODE_BODY_LCAT)
            || ($line_category[$i] == $BEGIN_DEFINITION_LCAT)
            || ($line_category[$i] == $CONT_DEFINITION_LCAT))
            ⟨Look for a function definition on this line 17⟩;
    }
}
```

§**17.** We recognise a C function as being a line which starts with an optional comment in the form `/**/`, `/***/`, `/****/` or `/*****/`, and then (or instead) takes the form `type identifier(args...`, where type is either one of the base types found already or else a pointer to a type.

⟨Look for a function definition on this line 17⟩ ≡

```
my $look_for_identifiers = $line_text[$i];
my $stars_in_comment = "";
my $type_qualifiers = "";
if ($look_for_identifiers =~ m/^\s*\/(\*+)\/\s*(.*)$/) {
    $stars_in_comment = $1; $look_for_identifiers = $2;
}
if ($look_for_identifiers =~ m/^\s*(signed|unsigned|long)\s+(.*)$/) {
    $type_qualifiers = $1; $look_for_identifiers = $2;
}
if ($look_for_identifiers =~
    m/^\s*([A-Za-z_][A-Za-z0-9_]*)\s+(\**)(([A-Za-z_][A-Za-z0-9_]*|\:\:)+)\s*\((.*)$/) {
    my $return_type = $1;
    my $return_type_pointer_stars = $2;
    my $fname = $3;
    my $arguments = $5;
    if ((not(exists $blacklisted_functions{$fname})) &&
        ($bibliographic_data{"Namespaces"} eq "On")) {
        if ($stars_in_comment ne "") {
            inweb_error("with Namespaces on, $fname should not be marked /*...*/");
        }
        if (($fname =~ m/::/) && ($stars_in_comment eq "")) {
            $stars_in_comment = "***";
        }
    }
    $fname =~ s/::/__/g;
    if ((exists $base_types{$return_type}) &&
        (not(exists $blacklisted_functions{$fname}))) {
        ⟨Deal with type qualifiers and constructors 18⟩;
        ⟨Accept this as a new function definition 19⟩;
    }
}
```

This code is used in §16.

§**18.** Combine the three parts of the return type:

⟨Deal with type qualifiers and constructors 18⟩ ≡

```
if ($return_type_pointer_stars ne "") {
    $return_type .= " ".$return_type_pointer_stars;
}
if ($type_qualifiers ne "") {
    $return_type = $type_qualifiers." ".$return_type;
}
```

This code is used in §17.

§**19.** In the following, we merge the text of up to the next 9 subsequent lines of code, in order to have enough of the function to be pretty sure that we've got the whole of its arguments: then we peel this off to obtain the argument list in the definition.

⟨Accept this as a new function definition 19⟩ ≡

```
$functions_line{$fname} = $i;
$functions_return_type{$fname} = $return_type;
$functions_declared_scope{$fname} = $stars_in_comment;

my $idash;
for ($idash = $i+1;
    (($idash<$no_lines) && ($idash-$i<10));
    $idash++) {
    $arguments .= ' '.$line_text[$idash];
}
if ($arguments =~ m/^(.*?)\)\s+\{/) {
    $functions_arglist{$fname} = $1;
} else {
    inweb_error("unable to find spec: $fname (args: '$arguments')");
}
```

This code is used in §17.

§**20. Scanning for function calls and accesses of structure members.** In other words, here we look for where things are actually used, rather than where they are declared or defined. We skip material in the appendices, as these might be configuration files likely to cause false positives.

```
sub scan_identifiers_in_source {
    my $i;
    for ($i=0; $i<$no_lines; $i++) {
        if (($line_category[$i] == $CODE_BODY_LCAT)
            || ($line_category[$i] == $BEGIN_DEFINITION_LCAT)
            || ($line_category[$i] == $CONT_DEFINITION_LCAT)) {
            my $current_section_leafname = $section_leafname[$line_sec[$i]];
            my $current_section_number = $line_sec[$i];
            my $look_for_identifiers = $line_text[$i];
            if ($section_sigil[$line_sec[$i]] =~ m/^[A-O]/) { next; }
            ⟨Remove any C comments from the line 21⟩;
            if ($web_language == $C_FOR_INFORM_LANGUAGE)
                ⟨Detect some of NI's fruitier macros 25⟩;
            ⟨Detect use of C's dot operator for structure member access 22⟩;
            ⟨Detect use of C's arrow operator for structure member access 23⟩;
            while ($look_for_identifiers =~ m/^.*?([A-Za-z_]([A-Za-z0-9_]|::)*)(.*?)$/) {
                $identifier = $1;
                $look_for_identifiers = $3;
                $identifier =~ s/::/__/g;
                ⟨Is this identifier a function whose name we recognise? 24⟩;
            }
        }
    }
}
```

§**21.** This would be fooled by e.g. `a="/*"; call_me(); b="*/";`, but we'll take the risk. None of what we do in this section is mission-critical in that its failure would cause the program to mis-compile: the worst case is that we don't always print the right footnotes in the woven form, which would be sad, but not the end of the world.

⟨Remove any C comments from the line 21⟩ ≡
```
$look_for_identifiers =~ s/\/\*(.*?)\*\///g;
```
This code is used in §20.

§**22.** For instance, spot `parse_node.word_ref1` as an access of member `word_ref1`.

⟨Detect use of C's dot operator for structure member access 22⟩ ≡
```
while ($look_for_identifiers =~ m/^(.*?)\.([A-Za-z_][A-Za-z0-9_]*)(.*?)$/) {
    $look_for_identifiers = $1.$3;
    note_usage_of_structure_member($2, $i);
}
```
This code is used in §20.

§**23.** And almost identically but for `->`, with pointers to structures:

⟨Detect use of C's arrow operator for structure member access 23⟩ ≡
```
while ($look_for_identifiers =~ m/^(.*?)\-\>([A-Za-z_][A-Za-z0-9_]*)(.*?)$/) {
    $look_for_identifiers = $1.$3;
    note_usage_of_structure_member($2, $i);
}
```
This code is used in §20.

§**24.** Since we know all the function names by this point...

define `$debug_identifier_detection 0`

⟨Is this identifier a function whose name we recognise? 24⟩ ≡
```
if (exists $functions_line{$identifier}) {
    if ($debug_identifier_detection == 1) { print STDERR "ID: ", $identifier, "\n"; }
    if ($identifier eq "main") { next; }                    which of course isn't called in the web
    my $section_defining_function =
        $section_leafname[$line_sec[$functions_line{$identifier}]];
    $functions_usage_count{$identifier}++;
    $function_usage_hash{$identifier.'+'.$section_chap[$line_sec[$i]]} ++;
    if ($current_section_leafname ne $section_defining_function) {
        $functions_usage_section_list{$identifier} .=
            ":".$section_chap[$line_sec[$i]]."-".$current_section_leafname;
        $functions_usage_concisely_described{$identifier} .=
            ":".$current_section_number;
        $functions_usage_verbosely_described{$identifier} .=
            language_comment("called by Chapter ".
                $section_chap[$line_sec[$i]]."/".$current_section_leafname.
                " line ".$line_source_file_line[$i]);
    }
    if ($debug_identifier_detection == 1) {
        print STDERR "Added fn to code area $section_chap[$line_sec[$i]]\n";
    }
}
```
This code is used in §20.

§**25.** For more on what these macros do, see the Inform source. `CREATE(T)` is a macro creating a structure of type `T`, which is defined with:

```
@d CREATE(type_name) ...
```

This definition shouldn't count as a creation of anything, but if `T` is anything else then it does. The sole point of this check is to warn us if we've accidentally created instances of the same structure in more than one section of Inform's web.

Macros such as `ALLOW_ARRAY(scene_data)` are used in Inform to set up what is legal when reading I6 template code: for instance, this would enable

```
{-array:scene_data}
```

the effect being that Inform's `compile_scene_data_array` function would be called whenever this token in an I6 template was expanded. (We have to take note of that here since the use of a macro disguises that a function call is being made.)

⟨Detect some of NI's fruitier macros 25⟩ ≡

```
    if ($look_for_identifiers =~ m/CREATE\((.*?)\)/) {
        if ($1 ne "type_name") {
            if ((exists ($structure_CREATE_in_section{$1})) &&
                ($structure_CREATE_in_section{$1} ne $section_leafname[$line_sec[$i]])) {
                add_error_to_section($current_section_number,
                    "Use of CREATE($1) in multiple sections");
            }
            $structure_CREATE_in_section{$1} = $section_leafname[$line_sec[$i]];
        }
    }
    while ($look_for_identifiers =~ m/^(.*?)ALLOW_([A-Z]+)\(([A-Za-z_0-9:_]*)\)(.*)$/) {
        $identifier = $3;
        $metalanguage_type = $2;
        $look_for_identifiers = $1.$4;
        $identifier =~ s/::/__/g;
        if (($metalanguage_type eq "CALL") ||
            ($metalanguage_type eq "CALLV")) {
            $dot_i6_identifiers{$identifier} = 1;
        }
        if ($metalanguage_type eq "ARRAY") {
            $dot_i6_identifiers{$identifier."_array"} = 1;
        }
        if ($metalanguage_type eq "ROUTINE") {
            $dot_i6_identifiers{$identifier."_routine"} = 1;
        }
    }
```

This code is used in §20.

§**26.** Keep track of each distinct section in which a given structure member is used:

```
sub note_usage_of_structure_member {
    my $member_name = $_[0];
    my $line_number = $_[1];
    my $found_in = "-".$section_leafname[$line_sec[$line_number]].":";
    if ($member_usage_section_list{$member_name} =~ m/$found_in/) { return; }
    $member_usage_section_list{$member_name} .= $found_in;
}
```

§**27. Determine which sections call which other sections.** We now know all of the functions, and where all of the function calls occur. This means we can determine (i) the actual scope of a function and (ii) how the "used by" and "uses" parts of a section's `@Interface` ought to read, if it were going to have them.

```
sub find_actual_section_usage {
    my $sn;
    for ($sn=0; $sn<$no_sections; $sn++) {
        my %usages = ();
        ⟨Check that the section's declared I6 template interface is correct 32⟩;
        foreach $f (sort keys %functions_line) {
            if ($section_leafname[$line_sec[$functions_line{$f}]] eq $section_leafname[$sn]) {
                ⟨Check that this function is properly declared, and add sections calling it to the usages hash 28⟩;
            }
        }
        if ($section_declared_a_service[$sn] == 1) { %usages = (); }
        ⟨Translate the usages hash into an unsorted form of the correct used-by block 29⟩;
    }
    for ($sn=0; $sn<$no_sections; $sn++)
        ⟨Deduce the correct uses block from the collection of all correct used-by blocks 30⟩;
    for ($sn=0; $sn<$no_sections; $sn++)
        ⟨Tidy up and finalise the correct uses and used-by blocks 31⟩;
}
```

§**28.** A neat trick here is that we knock the function's own chapter out of its usage list with a search and replace, and then look for chapter markers in what remains: if there are any, then the usage list must have contained sections from other chapters.

⟨Check that this function is properly declared, and add sections calling it to the usages hash 28⟩ ≡

```
    my $owning_chapter = $section_chap[$line_sec[$functions_line{$f}]];
    my $cp = $functions_usage_section_list{$f};
    $cp =~ s/\:$owning_chapter\-//g;
    my $actual_scope = "";
    if ($functions_usage_section_list{$f} ne "") { $actual_scope = "**"; }
    if ($f eq "main") { $actual_scope = "*****"; }
    elsif (exists $dot_i6_identifiers{$f}) {
        $actual_scope = "****";
        check_section_declares_template_I6($sn, $f);
    } elsif ($cp =~ m/\:\d+\-/) { $actual_scope = "***"; }
    check_function_declared_correct_scope($sn, $f, $actual_scope);
    $cp = $functions_usage_section_list{$f};
    while ($cp =~ m/^(\:\d+\-[^:]*)(.*)$/) {
        $usages{$1}++;
        $cp = $2;
    }
```

This code is used in §27.

§**29.** The usages hash is really just used as a set of keys here, and we concatenate it into a single string. (The list is unsorted but at least can't contain duplicates.) We tack a colon onto the end for convenience of the next search, but after that we'll remove it again.

⟨Translate the usages hash into an unsorted form of the correct used-by block 29⟩ ≡

```
$section_correct_used_by_block[$sn] = "";
foreach $u (sort keys %usages) {
    $section_correct_used_by_block[$sn] .= $u;
}
$section_correct_used_by_block[$sn] .= ":";
```

This code is used in §27.

§**30.** The uses blocks form a sort of transpose of the used-by blocks, if we think of these data structures as matrices rather than arrays: and as with all matrix-esque calculations, we end up with quadratic running time in the dimensions of the matrix. That is, if there are s sections, then the following takes $O(s^2)$ time, but since in practice $s < 200$ this is quite bearable.

⟨Deduce the correct uses block from the collection of all correct used-by blocks 30⟩ ≡

```
$section_correct_uses_block[$sn] = "";
my $j;
for ($j=0; $j<$no_sections; $j++) {
    if ($section_correct_used_by_block[$j] =~ m/\-$section_leafname[$sn]\:/) {
        if ($section_declared_a_service[$j] != 1) {
            $section_correct_uses_block[$sn]
                .= ":".$section_chap[$j]."-".$section_leafname[$j];
        }
    }
}
```

This code is used in §27.

§**31.** Note the promised removal of the terminating colons in the used-by blocks:

⟨Tidy up and finalise the correct uses and used-by blocks 31⟩ ≡

```
$section_correct_used_by_block[$sn] =~ s/\:$//;
$section_correct_used_by_block[$sn] =
    section_list_sort($section_correct_used_by_block[$sn]);
$section_correct_uses_block[$sn] =
    section_list_sort($section_correct_uses_block[$sn]);
```

This code is used in §27.

§**32.** If a section in a C-for-Inform web which uses an `@Interface` is going to name any I6 template things, well, they'd better be right: even in Weak mode we're not going to forgive any error in this.

⟨Check that the section's declared I6 template interface is correct 32⟩ ≡

```
my $a = $section_I6_template_identifiers[$sn];
while ($a =~ m/^(.*?)\:(.*)$/) {
    $a = $2;
    if (not(exists $dot_i6_identifiers{$1})) {
        add_error_to_section($sn, "Bad scope: $1 isn't a {-} command as claimed");
    }
    if ($section_leafname[$line_sec[$functions_line{$1}]] ne $section_leafname[$sn]) {
        add_error_to_section($sn, "Bad scope: $1 isn't defined in this section");
    }
}
```

This code is used in §27.

§**33. Structures.** We now come to the second part of the checking code for C-like sections: the part which verifies that structures and their members are used "correctly". We don't come to the party with nothing in hand, though: we have already found all the `typedef` structure declarations, so we know what the names of the structures are and on what line they are created; and we have kept an eye out, if in C-for-Inform mode, for any use of the `CREATE(S)` memory management macro as applied to a structure `S`.

§**34.** First: which section rightfully "owns" a structure? First, anyone with the power to `CREATE` has dominion; if nobody does, then anyone who claims the right in his `@Interface` is allowed to have it; and if nobody even claims the structure, well then, it belongs to whichever section declared its `typedef`.

```
sub find_structure_ownership {
    my $struc;
    foreach $struc (keys %structures) {
        if (exists $structure_CREATE_in_section{$struc}) {
            $structure_owner{$struc} = $structure_CREATE_in_section{$struc};
        } elsif (exists $structure_declared_owner{$struc}) {
            $structure_owner{$struc} =
                $structure_declared_owner{$struc};
        } else {
            $structure_owner{$struc} =
                $section_leafname[$line_sec[$structure_declaration_line{$struc}]];
        }
    }
}
```

§**35.** Second: which sections use which members, and by extension, which structures? Are any members entirely unused and consequently redundant?

```
sub establish_structure_usage {
    my $member;
    ⟨Check that any -analyse-structure command line setting wasn't mistyped 36⟩;
    foreach $member (sort keys %member_structures) {
        if (exists $member_usage_section_list{$member}) {
            my $msul = $member_usage_section_list{$member};
            while ($msul =~ m/^\-(.*?)\:(.*)$/) {
                my $observed_in = $1; $msul = $2;
                my $owners = $member_structures{$member};
                while ($owners =~ m/^(.*?)\:(.*)$/) {
                    my $owner = $1; $owners = $2;
                    my $susl_chunk = "-".$observed_in.":";
                    ⟨In -analyse-structure mode, throw an error for each member used in a foreign section 37⟩;
                    if (not ($structure_usage_section_list{$owner} =~ m/$susl_chunk/)) {
                        $structure_usage_section_list{$owner} .= $susl_chunk;
                    }
                }
            }
        } else {
            member_declared_but_not_used($member);
        }
    }
    ⟨Compose the correct structure declaration to appear in the Interface of each section 38⟩;
}
```

§**36.** As crude as this tool is, it was surprisingly helpful during a consolidation period when the code of `NI` was being tidied up into a more encapsulated form.

⟨Check that any -analyse-structure command line setting wasn't mistyped 36⟩ ≡

```
if ($analyse_structure_setting ne "") {
    if (not(exists($structures{$analyse_structure_setting}))) {
        inweb_fatal_error("no such structure: ".$analyse_structure_setting);
    }
    print "The structure $analyse_structure_setting is owned by ",
        $structure_owner{$analyse_structure_setting}, "\n";
    print "Members of the structure used from other sections are as follows:\n";
}
```

This code is used in §35.

§**37.** Perhaps these aren't really errors, but it does make for a tidy end to the process after we're done.

⟨In -analyse-structure mode, throw an error for each member used in a foreign section 37⟩ ≡

```
if (($analyse_structure_setting ne "") &&
    ($analyse_structure_setting eq $owner)) {
    if ($observed_in ne $structure_owner{$owner}) {
        inweb_error($member.": Chapter ".
            $section_chap[$section_number_from_leafname{$observed_in}].
            "/".$observed_in);
    }
}
```

This code is used in §35.

§**38.** The following assumes that structures are not used in the Appendices, which of course is true since these do not contain code.

⟨Compose the correct structure declaration to appear in the Interface of each section 38⟩ ≡

```
my $sec_number;
for ($sec_number=0; $sec_number<$no_sections; $sec_number++) {
    my $declaration = "";
    my $struc;
    foreach $struc (sort keys %structures) {
        if ($structure_owner{$struc} eq $section_leafname[$sec_number]) {
            my $structure_billing = "";
            ⟨Work out the billing for this structure 39⟩;
            $declaration .= $structure_billing;
        }
    }
    $section_correct_structures_block[$sec_number] = $declaration;
}
```

This code is used in §35.

§**39.** What we declare for the structure is: whether private or public, that is, open for other sections to access its members; whether it is `typedefd` by a section other than its rightful owner (which is allowed but deprecated, in terms of `NI`); and if it is public, then which other sections have action.

⟨Work out the billing for this structure 39⟩ ≡

```
my %usage_list = ();
my %shorter_usage_list = ();
my $structure_used_externally = 0;
my $deviant_ownership_note = "";
if ($section_leafname[$line_sec[$structure_declaration_line{$struc}]]
    ne $structure_owner{$struc}) {
    $deviant_ownership_note =
        ": typedef in $section_leafname[$line_sec[$structure_declaration_line{$struc}]]";
}
⟨Fill the usage list hash with the names of external sections accessing structure 40⟩;
if ($structure_used_externally == 0) {
    $structure_billing = "-- Owns struct $struc (private$deviant_ownership_note)\n";
    $private_structures++;
} else {
    $structure_billing = "-- Owns struct $struc (public$deviant_ownership_note)\n";
    $shared_structures++;
    foreach $k (sort keys %usage_list) {
        $structure_billing .= $usage_list{$k};
        $structure_ownership_summary{$struc} .= $shorter_usage_list{$k} . " ";
    }
}
```

This code is used in §38.

§**40.**

⟨Fill the usage list hash with the names of external sections accessing structure 40⟩ ≡

```
my $susl = $structure_usage_section_list{$struc};
while ($susl =~ m/^\-(.*?)\:(.*)$/) {
    my $client_section = $1; $susl = $2;
    if ($client_section ne $section_leafname[$sec_number]) {
        $structure_sharings++;
        $structure_used_externally = 1;
        $usage_list{sprintf("%05d", $section_number_from_leafname{$client_section})}
            .= "   !- shared with Chapter ".
            $section_chap[$section_number_from_leafname{$client_section}].
            "/".$client_section."\n";
        $shorter_usage_list{sprintf("%05d", $section_number_from_leafname{$client_section})}
            .= $section_sigil[$section_number_from_leafname{$client_section}];
    }
}
```

This code is used in §39.

§**41. Hacking with section list strings.** We take a section list string to pieces, sort it and reassemble it in sorted order. Don't look so suspicious: it works.

```
sub section_list_sort {
    my $list = $_[0];
    my %lines_unpacked = ();
    my $this_line;
    my $account = "";
    while ($list =~ m/^\:(\d+)\-([^:]*)(.*)$/) {
        $this_line = $prefix. "Chapter ". $1. "/". $2. "\n";
        $lines_unpacked{sprintf("%04d", $section_number_from_leafname{$2})} = ":".$1."-".$2;
        $list = $3;
    }
    foreach $this_line (sort keys %lines_unpacked) {
        $account .= $lines_unpacked{$this_line};
    }
    return $account;
}
```

§**42.** To unpack a section list is to reconstruct it as a sequence of lines in sorted order and with a slightly different format.

```
sub section_list_unpack {
    my $prefix = $_[0];
    my $list = $_[1];
    my %lines_unpacked = ();
    my $this_line;
    my $account = "";
    while ($list =~ m/^\:(\d+)\-([^:]*)(.*)$/) {
        $this_line = $prefix. "Chapter ". $1. "/". $2. "\n";
        $lines_unpacked{sprintf("%04d", $section_number_from_leafname{$2})} = $this_line;
        $list = $3;
    }
    foreach $this_line (sort keys %lines_unpacked) {
        $account .= $lines_unpacked{$this_line};
    }
    return $account;
}
```

§43. Applying the Strict Usage Rules of a vengeful god.

```
sub check_interface_declarations_for_uses_and_used_by {
    my $sn;
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    if ($bibliographic_data{"Declare Section Usage"} eq "Off") { return; }
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_declared_used_by_block[$sn] ne
            $section_correct_used_by_block[$sn]) {
            $em = "Interface error:\n"
                .section_list_unpack(
                    "%    Says used by ", $section_declared_used_by_block[$sn])
                .section_list_unpack(
                    "-- Used by ", $section_correct_used_by_block[$sn]);
            add_error_to_section($sn, $em);
        }
        if ($section_declared_uses_block[$sn] ne
            $section_correct_uses_block[$sn]) {
            $em = "Interface error:\n"
                .section_list_unpack(
                    "%    Says uses ", $section_declared_uses_block[$sn])
                .section_list_unpack(
                    "-- Uses ", $section_correct_uses_block[$sn]);
            add_error_to_section($sn, $em);
        }
    }
}
```

§**44.** And similarly for the structures part of the `@Interface:...`

```
sub check_interface_declarations_for_structures {
    my $sn;
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_declared_structure_interface[$sn] ne
            $section_correct_structures_block[$sn]) {
            add_error_to_section($sn,
                "Incorrect structure billing: should be...\n".
                $section_correct_structures_block[$sn].
                "...and not...\n".
                $section_declared_structure_interface[$sn]);
        }
    }
}
```

§**45.** If we're being Strict, functions must also be marked with their correct scopes:

```
sub check_function_declared_correct_scope {
    my $sn = $_[0];
    my $f = $_[1];
    my $actual_scope = $_[2];
    if ($bibliographic_data{"Namespaces"} eq "On") {
        if ($f ne "main") {
            if (($actual_scope ne "") && ($functions_declared_scope{$f} eq "")) {
                add_error_to_section($sn,
                    "Begin externally called, function $f should belong to a :: namespace");
                return;
            }
            if (($actual_scope eq "") && ($functions_declared_scope{$f} ne "")) {
                add_error_to_section($sn,
                    "Begin internally called, function $f must not belong to a :: namespace");
                return;
            }
        }
        $functions_declared_scope{$f} = $actual_scope;
        return;
    }
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    if ($actual_scope ne $functions_declared_scope{$f}) {
        add_error_to_section($sn,
            "Bad scope: $f should be /".$actual_scope."/ not /".
                $functions_declared_scope{$f}."/");
    }
    if ($f =~ m/^([A-Z].*__)(.*?)$/) {
        my $declared = $1;
        my $within = $section_namespace[$sn];
        $within =~ s/::/__/g;
        if ($within eq "") {
            $fcc = restore_quadpoint($f);
            add_error_to_section($sn,
                "Bad scope: $fcc declared outside of any namespace");
        } elsif (not ($declared =~ m/^$within/)) {
            $fcc = restore_quadpoint($f); $wcc = restore_quadpoint($within);
            add_error_to_section($sn,
                "Bad scope: $fcc not allowed inside the section's namespace $wcc");
        }
    }
}
sub restore_quadpoint {
    my $f = $_[0];
    $f =~ s/__/::/g;
    return $f;
}
```

§**46.** The following is applied only in C-for-Inform mode; in ordinary C, no such functions occur.

```
sub check_section_declares_template_I6 {
    my $sn = $_[0];
    my $f = $_[1];
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    f = s/::/g;
    if (not ($section_I6_template_identifiers[$sn] =~ m/$f/)) {
        add_error_to_section($sn,
            "Bad scope: $f isn't declared:\n-- Defines {-callv:".$f."}\n");
    }
}
```

§**47.** It is, of course, a Sin to declare structures with member names that are never actually used. Unless of course we are in a messy program which uses them in concealed ways because of macro expansion – hence the exemption clause.

```
sub member_declared_but_not_used {
    my $member = $_[0];
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    if (exists($members_allowed_to_be_unused{$member})) { return; }
    inweb_error("structure(s) '".$member_structures{$member}."' has or have member '".
        $member."' declared but not used");
}
```

§**48.** `inweb` prefers a tidy setup where the same member name is not used in multiple structures, and in strict mode, it insists of this: that is, where `p->drivel` identifies the type of `p` since `drivel` is a member in only one possible structure. The following routine polices that rule.

```
sub check_uniqueness_of_structure_members {
    my $member;
    if ($bibliographic_data{"Strict Usage Rules"} eq "Off") { return; }
    foreach $member (sort keys %member_structures) {
        my $owner_count = 0;
        my $x = $member_structures{$member};
        while ($x =~ m/^(.*?)\:(.*)$/) { $x = $2; $owner_count++; }
        if ($owner_count > 1) {
            inweb_error("element '".$member."' belongs to multiple structures: ".
                $member_structures{$member});
        }
    }
}
```

§**49. Interface errors.** When an interface error comes to light, we call:

```
sub add_error_to_section {
    my $section_number = $_[0];
    my $error_text = $_[1];
    if ($error_text eq "") { print "*** Bad error text ***\n"; exit(1); }
    $section_errors[$section_number] .= $error_text."\n";
}
```

§**50.** So that at the end of the checking process, we can report and take action:

```
sub report_any_interface_errors_found {
    my $infractions = 0;
    my $sn;
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_errors[$sn] ne "") {
            inweb_error("interface errors on $section_sigil[$sn] ($section_leafname[$sn])");
            print STDERR $section_errors[$sn];
            $infractions++;
        }
    }
    if ($infractions > 0) { inweb_fatal_error("halting because of section errors"); }
}
```

3 Outputs

`3/anal`: *The Analyser.w* Miscellaneous useful (or anyway, formerly useful) checks to carry out on the source code.

`3/swarm`: *The Swarm.w* To feed multiple output requests to the weaver, and to present weaver results, and update indexes or contents pages.

`3/weave`: *The Weaver.w* To weave a portion of the code into instructions for TeX.

`3/bnf`: *Backus-Naur Form.w* To weave any collated BNF grammar from the web into a nicely typeset form.

`3/tang`: *The Tangler.w* To write a portion of the code in a compilable form.

`3/plan`: *Programming Languages.w* To characterise the relevant differences in behaviour between the various programming languages we support: for instance, how comments are represented.

`3/cfori`: *C for Inform.w* To provide a convenient extension to C syntax for the C-for-Inform language, which is likely never to be used for any program other than the Inform 7 compiler.

The Analyser 3/anal

Purpose

Miscellaneous useful (or anyway, formerly useful) checks to carry out on the source code.

§**1. Dot files for dependency graphs.** The target `0` as always means the entire web, so that we have a dependency graph of each chapter; otherwise we must specify a particular chapter (not a section, not an appendix).

```
sub compile_graphs {
    my $sigil = $_[0];
    if ($sigil eq "0") {
        my $ch;
        for ($ch=0; $ch<$no_chapters; $ch++) { compile_graph($ch); }
    } elsif ($sigil =~ m/^\d+$/) {
        compile_graph(eval($sigil));
    } else {
        inweb_fatal_error("can't compile dependency graph(s) for target $sigil");
    }
    print "Dot files written\n";
}
```

§**2.** Thus the following compiles the graph for a single chapter, converting it on request into a PNG image.

```
sub compile_graph {
    my $ch = $_[0];
    my $dotfile = compile_chapter_graph($ch);
    my $pngfile = pathname_to_png_of_dot_file($ch);
    if ($convert_graphs_switch == 1) {
        system($dot_utility_path." -Tpng \"".$dotfile."\" -o \"".$pngfile."\"");
    }
}
```

§**3.** The dot files are the sources of the graph illustrations: they need to be compiled by the `dot` utility to turn them into actual images. We keep the dot files in the `Tangled` folder, as they are essentially generated from code, but the images they turn into are in the `Figures` folder, since the expectation is that they'll be used as illustrations in the woven form of the web.

```
sub pathname_to_dot_file {
    my $cn = $_[0];
    return $web_setting."Tangled/Chapter-".$cn."-Dependencies.dot";
}
sub pathname_to_png_of_dot_file {
    my $cn = $_[0];
    return $web_setting."Figures/Chapter-".$cn."-Dependencies.png";
}
```

§**4.** This is all pretty straightforward: see the `dot` user manual for details of the format.

```
sub compile_chapter_graph {
    my $cn = $_[0];
    my $i;
    $dotname = pathname_to_dot_file($cn);
    open DOTFILE, ">".$dotname or die "Can't open dot file $dotname for output";
    ⟨Start a directed graph 5⟩;
    ⟨Declare the vertices 7⟩;
    ⟨Declare the edges 9⟩;
    ⟨End the directed graph 6⟩;
    close DOTFILE;
    return $dotname;
}
```

§**5.** The start is always the same:

⟨Start a directed graph 5⟩ ≡

```
    print DOTFILE "digraph L0 {\n";
    print DOTFILE "    size = \"8,8\";\n";
    print DOTFILE "    ordering = out;\n";
    print DOTFILE "    compound = true;\n";
    print DOTFILE "    node [shape = box];\n";
```

This code is used in §4.

§**6.** And so is the end:

⟨End the directed graph 6⟩ ≡

```
    print DOTFILE "}\n";
```

This code is used in §4.

§**7.** In between, we start with vertex declarations: one for each section in the chapter, plus one for the template interpreter.

⟨Declare the vertices 7⟩ ≡

```
    my $i;
    $current_class = -1; $i6_controller = -1;
    for ($i=0; $i<$no_sections; $i++) {
        if ($section_chap[$i] != $cn) { next; }
        ⟨Declare a vertex for this section 8⟩;
    }
    ⟨Make an orchid-coloured circle to represent the template interpreter 12⟩;
    if ($current_class >= 0) { print DOTFILE "}\n"; }
```

This code is used in §4.

§**8.** We draw a subgraph box around all vertices in a given equivalence class, which is neat if we're drawing a big multi-chapter graph, but in fact we aren't doing that so instead `$equivalence_class[$i]` is always 0 for now, and the effect is largely decorative – though note that the special vertex for the I6T interpreter, in the case of graphing Inform, lies outside the box (and rightly, since it makes function calls from out of the sky).

⟨Declare a vertex for this section 8⟩ ≡
```
$caption = $section_sigil[$i];
if ($caption eq "14/meta") { $i6_controller = $i; }
if ($section_sigil[$i] ne "") { $caption = $section_sigil[$i]; }
if ($current_class != $equivalence_class[$i]) {
    if ($current_class >= 0) { print DOTFILE "}\n"; }
    $current_class = $equivalence_class[$i];
    print DOTFILE "subgraph cluster", $cn, " {\n";
    print DOTFILE "label=\"Chapter ", $cn, "\";\n";
}
print DOTFILE "    n", $i, " [label=\"", $caption, "\"];\n";
if (($section_I6_template_identifiers[$line_sec[$i]] ne "") ||
    ($i6_controller == $i)) {
    $section_I6_template_identifiers[$line_sec[$i]] = "";
    if ($i6_controller != $i) { $dot_i6_calls{$i} = "i6"; }
}
```

This code is used in §7.

§**9.**

⟨Declare the edges 9⟩ ≡
```
my $i;
for ($i=0; $i<$no_sections; $i++) {
    if ($section_chap[$i] != $cn) { next; }
    ⟨Declare the directed edges outbound from this vertex 10⟩;
}
```

This code is used in §4.

§**10.**

⟨Declare the directed edges outbound from this vertex 10⟩ ≡
```
my $x = $section_correct_uses_block[$i];
my $no_outbound_calls = 0;
while ($x =~ m/^\:(.*?)\-(.*?\.w)(.*)$/) {
    $x = $3;
    $to_section = $section_number_from_leafname{$2};
    $from_chapter = $section_chap[$i];
    $to_chapter = $section_chap[$to_section];
    if ($from_chapter == $to_chapter) {
        $uses_this[$no_outbound_calls] = $to_section;
        $uses_this_section[$no_outbound_calls] = 1;
        $no_outbound_calls++;
    } else {
        $uses_this[$no_outbound_calls] = $to_chapter;
        $uses_this_section[$no_outbound_calls] = 0;
        $no_outbound_calls++;
    }
}
```

⟨Make black arrows for all the calls out of this section into others on the graph 11⟩;
⟨Make an arrow representing a template interpreter invocation if necessary 13⟩;

This code is used in §9.

§**11.** This makes lines like `n3 -> {n2, n4, n5}`, meaning arrows run from node 3 to each of nodes 2, 4 and 5.

⟨Make black arrows for all the calls out of this section into others on the graph 11⟩ ≡

```
if ($no_outbound_calls > 0) {
    print DOTFILE "    n", $i, " -> ";
}
if ($no_outbound_calls > 1) {
    print DOTFILE "{ ";
}
for ($z=0; $z<$no_outbound_calls; $z++) {
    if ($uses_this_section[$z] == 1) {
        print DOTFILE "n", $uses_this[$z], " ";
    }
}
if ($no_outbound_calls > 1) {
    print DOTFILE "} ";
}
if ($no_outbound_calls > 0) {
    print DOTFILE "\n";
}
```

This code is used in §10.

§**12.** For use with `NI` only:

⟨Make an orchid-coloured circle to represent the template interpreter 12⟩ ≡

```
print DOTFILE "    i6 [shape=circle] [color=orchid] [label=\".i6\"];\n";
```

This code is used in §7.

§**13.** Orchids come in lots of colours, but this one is quite a pleasing pastel purple, anyway.

⟨Make an arrow representing a template interpreter invocation if necessary 13⟩ ≡

```
if ($dot_i6_calls{$i} ne "") {
    print DOTFILE "    ", $dot_i6_calls{$i}, " -> n", $i, " [color=orchid];\n";
}
```

This code is used in §10.

§**14.** And so that this does, somehow, form part of the call graph itself:

⟨On the section owning the template interpreter, make an arrow pointing to it 14⟩ ≡

```
my $i;
for ($i=0; $i<$no_sections; $i++) {
    if ($section_chap[$i] != $cn) { next; }
    if ($section_sigil[$i] eq "14/meta") {
        print DOTFILE "    n", $i6_controller, " -> i6 [color=orchid];\n";
    }
}
```

This code is used in §.

§15. The section catalogue. Quite a useful snapshot of the sections, and also of the data structures used in a big C-like project.

```
sub catalogue_the_sections {
    my $sigil = $_[0];
    my $functions_too = $_[1];
    my $only = -1;
    my $cn = -1;
    if ($sigil eq "0") { $only = -1; }
    elsif ($sigil =~ m/^\d+$/) { $only = eval($sigil); }
    else { inweb_fatal_error("can't catalogue target $sigil"); }
    for ($i=0; $i<$no_sections; $i++) {
        if (($only != -1) && ($only != $section_chap[$i])) { next; }
        ⟨Produce dividing bars between chapters 16⟩;
        ⟨Produce catalogue line for this section 17⟩;
        if ($functions_too == 1) ⟨Produce list of functions owned by this section 19⟩
        else ⟨Produce list of data structures owned by this section 18⟩;
    }
}
```

§16.

⟨Produce dividing bars between chapters 16⟩ ≡

```
    if ($cn != $section_chap[$i]) {
        if ($cn >= 0) { print sprintf("      %-9s  %-50s  \n", "--------", "--------"); }
        $cn = $section_chap[$i];
    }
```

This code is used in §15.

§17.

⟨Produce catalogue line for this section 17⟩ ≡

```
    if ($cn != 0) { $main_title = "Chapter ".$cn."/"; }
    else { $main_title = ""; }
    $main_title .= $section_leafname[$i];
    print sprintf("%4d  %-9s  %-50s  ",
        $section_extent[$i], $section_sigil[$i], $main_title);
    if ($functions_too == 1) {
        print $section_namespace[$i];
    } else {
        print $section_namespace[$i], " ";
        if ($functions_usage_count{"compare_word"} > 0) {
            print sprintf("CW:%3d   ", $functions_usage_count{"compare_word"});
        }
        foreach $struc (sort keys %structures) {
            if ($structure_owner{$struc} eq $section_leafname[$i]) {
                print $struc, "  ";
            }
        }
    }
    print "\n";
```

This code is used in §15.

§**18.** This does nothing for non-C-like languages, since the hash is empty.

⟨Produce list of data structures owned by this section 18⟩ ≡
```
    foreach $struc (sort keys %structures) {
        if ($structure_owner{$struc} eq $section_leafname[$i]) {
            if ($structure_ownership_summary{$struc} ne "") {
                print sprintf("      %-9s  %-50s  ", "", "");
                print $struc, ": ", $structure_ownership_summary{$struc}, "\n";
            }
        }
    }
```

This code is used in §15.

§**19.** And here we have the function catalogue:

⟨Produce list of functions owned by this section 19⟩ ≡
```
    foreach $f (sort keys %functions_line) {
        if ($f =~ m/__/) {
            if ($section_leafname[$line_sec[$functions_line{$f}]] eq $section_leafname[$i]) {
                $f =~ s/__/::/g;
                print sprintf("      %-9s  %-50s -> \n", "", $f);
            }
        }
    }
```

This code is used in §15.

§**20.** The void of eternity. Or what have you. Presuming that `void *` pointers are the road to a dusty death, we catalogue them here. (This is a leftover from the days when the main Inform source used `void *` pointers now and then, and the code was added here to help purge them from Inform by reporting them.)

```
sub catalogue_void_pointers {
    my $sigil = $_[0];
    if ($sigil ne "0") { inweb_fatal_error("can't catalogue voids for target $sigil"); }
    my $el;
    foreach $el (sort keys %member_types) {
        if ($member_types{$el} =~ m/ void\*/) {
            print $el, " in ", $member_structures{$el}, ": ", $member_types{$el}, "\n";
        }
    }
}
```

The Swarm 3/swarm

Purpose

To feed multiple output requests to the weaver, and to present weaver results, and update indexes or contents pages.

§**1. Swarming.** Let us hope this is a glittering cloud, like the swarm of locusts in the title of Chapter 25 of Laura Ingalls Wilder's *On the Banks of Plum Creek*: at any rate it does make quite a deal of individual PDFs.

Each individual weave returns an ID number, its "weave number", which can be used to look up the individual results. This is stored in the arrays:

(C5) `$chapter_weave_number[]` is either the ID of the completed weave of this individual chapter, or -1 if no weave was made from it.

(S18) `$section_weave_number[]` is either the ID of the completed weave of this individual section, or -1 if no weave was made from it.

```
sub weave_swarm {
    my $i;
    for ($i=0; $i < $no_sections; $i++) { $section_weave_number[$j] = -1; }
    for ($i=0; $i < $no_chapters; $i++) { $chapter_weave_number[$j] = -1; }
    if ($swarm_mode >= $SWARM_SECTIONS) {
        for ($i=0; $i < $no_sections; $i++) {
            if (($only_setting eq "") || ($section_sigil[$i] =~ m/^$only_setting\//)) {
                $section_weave_number[$i] = weave_sigil($section_sigil[$i], 0);
            }
        }
    }
    if (($web_is_chaptered == 1) && ($swarm_mode >= $SWARM_CHAPTERS)) {
        for ($i=0; $i < $no_chapters; $i++) {
            if ($only_setting ne "") {
                if ($chapter_sigil[$i] ne $only_setting) { next; }
            }
            $chapter_weave_number[$i] = weave_sigil($chapter_sigil[$i], 0);
            if ($only_setting ne "") {
                $complete_web_weave_number = $chapter_weave_number[$i];
                $complete_PDF_leafname = $chapter_woven_pdf_leafname[$i];
            }
        }
    }
    if (($swarm_mode >= $SWARM_CHAPTERS) && ($only_setting eq "")) {
        $complete_web_weave_number = weave_sigil("0", 0);
    }
    weave_index_templates();
}
```

§**2.** After every swarm, we rebuild all the indexes specified in the Index Template list for the web:

```
sub weave_index_templates {
    ⟨Weave one or more index files 3⟩;
    ⟨Copy in one or more additional files to accompany the index 4⟩;
}
```

§**3.**

⟨Weave one or more index files 3⟩ ≡

```
    my $temp_list;
    my $leaf = "";
    if (exists ($bibliographic_data{"Index Template"})) {
        $temp_list = $bibliographic_data{"Index Template"};
    } else {
        if ($web_is_chaptered == 1) {
            $temp_list = $path_to_inweb_setting.'inweb/Materials/chaptered-index.html';
        } else {
            $temp_list = $path_to_inweb_setting.'inweb/Materials/unchaptered-index.html';
        }
        $leaf = "index.html";
    }
    while ($temp_list ne "") {
        my $this_temp;
        if ($temp_list =~ m/^(.*?)\s*\,\s*(.*)$/) {
            $temp_list = $2; $this_temp = $1;
        } else {
            $this_temp = $temp_list; $temp_list = "";
        }
        if ($leaf eq "") {
            $leaf = $this_temp;
            if ($leaf =~ m/^.*\/(.*?)$/) { $leaf = $1; }
        }
        print "Weaving index file: Woven/$leaf\n";
        weave_contents_from_template($this_temp, $leaf);
        $leaf = "";
    }
```

This code is used in §2.

§**4.** The idea here is that an HTML index may need some binary image files to go with it, for instance.

⟨Copy in one or more additional files to accompany the index 4⟩ ≡

```
my $copy_list = $path_to_inweb_setting.'inweb/Materials/download.gif'.
    ', '.$path_to_inweb_setting.'inweb/Materials/lemons.jpg';
if (exists ($bibliographic_data{"Index Extras"})) {
    $temp_list = $bibliographic_data{"Index Extras"};
}
my $path_to_copy_binaries = $web_setting."Woven";
while ($copy_list ne "") {
    my $this_file;
    if ($copy_list =~ m/^(.*?)\s*\,\s*(.*)$/) {
        $copy_list = $2; $this_file = $1;
    } else {
        $this_file = $copy_list; $copy_list = "";
    }
    my $leaf = $this_file;
    if ($leaf =~ m/^.*\/(.*?)$/) { $leaf = $1; }
    print "Copying additional index file: Woven/$leaf\n";
    system("cp '".$this_file."' '".$path_to_copy_binaries."'");
}
```

This code is used in §2.

§**5.** The following is where an individual weave task begins, whether it comes from the swarm, or has been specified at the command line (in which case the call comes from Program Control).

```
sub weave_sigil {
    my $request = $_[0];                                    The sigil of the target to be weaved
    my $open_afterwards = $_[1];                            Open the PDF in the host OS's viewer
    my $cover_sheet_flag = 0;                               Shall we have one?
    my $weave_section_match = '';                           Reg exp for sigil to match
    my $tex_file_leafname;                                  What to call the resulting TeX file
    my $path_to_loom = $web_setting."Woven/";
    ⟨Translate the request sigil into details of what to weave 6⟩;
    weave_source($path_to_loom.$tex_file_leafname, $cover_sheet_flag, $weave_section_match);
    if ($lines_woven == 0) {
        inweb_fatal_error("empty weave request: $request");
    }
    my $wtn = run_woven_source_through_tex($path_to_loom.$tex_file_leafname,
        $open_afterwards, $cover_sheet_flag, $weave_section_match);
    ⟨Report on the outcome of the weave to the console 7⟩;
    return $wtn;
}
```

§**6.** From the sigil, we determine where to put the resulting TEX file, whether to make a PDF from it (always, at present), whether to open that PDF in the operating system (depends on the -open-pdf switch being used), whether to include a cover sheet (yes unless it's for just a single section), and which lines of the web should contribute to the source. We represent this last by supplying a regular expression which the sigil of the section owning the line should match: for instance, requiring this to match `11/` catches all of the lines in Chapter 11.

⟨Translate the request sigil into details of what to weave 6⟩ ≡

```
if ($request eq "0") {
    $weave_section_match = '.*';
    $booklet_title = "Complete Program";
    $tex_file_leafname = "Complete.tex";
    $cover_sheet_flag = 1;
} elsif ($request =~ m/^\d+$/) {
    my $cn = eval($request);
    $weave_section_match = '^'.$cn.'\/';
    $booklet_title = "Chapter ".$cn;
    $tex_file_leafname = "Chapter-".$cn.".tex";
    $cover_sheet_flag = 1;
} elsif ($request =~ m/^[A-O]$/) {
    my $cn = eval($request);
    $weave_section_match = '^'.$cn.'\/';
    $booklet_title = "Appendix ".$request;
    $tex_file_leafname = "Appendix-".$request.".tex";
    $cover_sheet_flag = 1;
} elsif ($request =~ m/^P$/) {
    my $cn = eval($request);
    $weave_section_match = '^'.$cn.'\/';
    $booklet_title = "Preliminaries";
    $tex_file_leafname = "Preliminaries.tex";
    $cover_sheet_flag = 1;
} elsif ($request =~ m/^(\S+?)\/(\S+)$/) {
    my $cn = eval($request);
    $weave_section_match = '^'.$1.'\/'.$2.'$';
    $booklet_title = $request;
    my $srequest = $request;
    $srequest =~ s/\//-/;
    $tex_file_leafname = $srequest.".tex";
    $cover_sheet_flag = 0;
} else {
    inweb_fatal_error("unknown weave request: $request");
}
```

This code is used in §5.

§**7.** Each weave results in a compressed one-line printed report:

⟨Report on the outcome of the weave to the console 7⟩ ≡

```
print "[", $request, ": ", $page_count[$wtn], "pp ",
    $pdf_size[$wtn]/1024, "K";
if ($overfull_hbox_count[$wtn] > 0) {
    print ", ", $overfull_hbox_count[$wtn], " overfull hbox(es)";
}
if ($tex_error_count[$wtn] > 0) {
    print ", ", $tex_error_count[$wtn], " error(s)";
}
print "]\n";
```

This code is used in §5.

§**8. Running TeX.** Although we are running `pdftex`, a modern variant of TEX, rather than the original, they are very similar as command-line tools; the difference is that the output is a PDF file rather than a DVI file, Knuth's original stab at the same basic idea.

In particular, we call it in "scrollmode" so that any errors whizz by rather than interrupting or halting the session. Because of that, we spool the output onto a console file which we can then read in and parse to find the number of errors actually generated. Prime among errors is the "overfull hbox error", a defect of TEX resulting from its inability to adjust letter spacing, so that it requires us to adjust the copy to fit the margins of the page properly. (In practice we get this here by having code lines which are too wide to display.)

```
sub run_woven_source_through_tex {
    my $tex_filename = $_[0];
    my $open_PDF = $_[1];
    my $with_cover_sheet = $_[2];
    my $weave_section_match = $_[3];

    my $path_to_tex = "";
    my $tex_leafname = "";
    my $console_filename = "";
    my $console_leafname = "";
    my $log_filename = "";
    my $pdf_filename = "";
    ⟨Work out these filenames and leafnames 9⟩;

    my $serious_error_count = 0;

    ⟨Call TeX and transcribe its output into a console file 10⟩;
    ⟨Read back the console file and parse it for error messages 11⟩;
    ⟨Remove the now redundant TeX, console and log files, to reduce clutter 12⟩;

    if ($open_PDF == 1) ⟨Try to open the PDF file in the host operating system 13⟩;
    return $no_weave_targets++;                    Return the weave ID number for this run of TEX
}
```

§**9.**

⟨Work out these filenames and leafnames 9⟩ ≡

```
$tex_leafname = $tex_filename;
if ($tex_leafname =~ m/^(.*)\/(.*?)$/) { $path_to_tex = $1; $tex_leafname = $2; }
$console_leafname = $tex_leafname; $console_filename = $tex_filename;
$console_leafname =~ s/\.tex$/.console/; $console_filename =~ s/\.tex$/.console/;
$log_filename = $tex_filename; $log_filename =~ s/\.tex$/.log/;
$pdf_filename = $tex_filename; $pdf_filename =~ s/\.tex$/.pdf/;
```

This code is used in §8.

§**10.**

⟨Call TeX and transcribe its output into a console file 10⟩ ≡

```
my $set_cwd = "";
if ($path_to_tex ne "") { $set_cwd = "cd \"".$path_to_tex."\"; "; }
$command = $set_cwd.$pdftex_configuration." -interaction=scrollmode \"".$tex_leafname."\" "
    .">\"".$console_leafname."\"";
system($command);
```

This code is used in §8.

§**11.** TEX helpfully reports the size and page count of what it produces, and we're not too proud to scrape that information out of the console file, besides the error messages (which begin with an exclamation mark in column 1).

⟨Read back the console file and parse it for error messages 11⟩ ≡

```
$overfull_hbox_count[$no_weave_targets] = 0;
$tex_error_count[$no_weave_targets] = 0;
$page_count[$no_weave_targets] = 0;
$pdf_size[$no_weave_targets] = 0;
open(CONSOLE, $console_filename) or die "no console file?";
while ($csl = <CONSOLE>) {
    if ($csl =~ m/Output written .*? \((\d+) page.*?(\d+) byte/) {
        $page_count[$no_weave_targets] = eval($1); $pdf_size[$no_weave_targets] = eval($2);
    }
    if ($csl =~ m/verfull \\hbox/) {
        $overfull_hbox_count[$no_weave_targets]++;
    } else {
        if ($csl =~ m/^\!/) {
            $tex_error_count[$no_weave_targets]++;
            $serious_error_count++;
        }
    }
}
close (CONSOLE);
```

This code is used in §8.

§**12.** The log file we never wanted, but TEX produced it anyway; it's really a verbose form of its console output. Now it can go. So can the console file and even the TEX source, since that was mechanically generated from the web, and so is of no lasting value. The one exception is that we keep the console file in the event of serious errors, since otherwise it's impossible for the user to find out what those errors were.

⟨Remove the now redundant TeX, console and log files, to reduce clutter 12⟩ ≡

```
if ($serious_error_count == 0) { system("rm \"".$console_filename."\""); }
system("rm \"".$log_filename."\"");
system("rm \"".$tex_filename."\"");
```

This code is used in §8.

§**13.** We often want to see the PDF immediately, so:

⟨Try to open the PDF file in the host operating system 13⟩ ≡

```
if ($open_command_configuration eq "") {
    inweb_error("no way to open PDF (see configuration file)", $pdf_filename);
} else {
    system($open_command_configuration." \"".$pdf_filename."\"");
}
```

This code is used in §8.

§**14. The Contents Interpreter.** This is a little meta-language all of its very own, with a stack for holding nested repeat loops, and a program counter and – well, and nothing else to speak of, in fact, except for the slightly unusual way that loop variables provide context by changing the subject of what is discussed rather than by being accessed directly.

define $TRACE_CI_EXECUTION 0 *For debugging*

```
sub weave_contents_from_template {
    my $path_to_template = $_[0];
    my $contents_page_leafname = $_[1];
    my $no_tlines = 0;
    ⟨Read in the source file containing the contents page template 15⟩;
    ⟨Open the contents page file to be constructed 16⟩;
    my $lpos = 0;                                   This is our program counter: a line number in the template
    $stack_pointer = 0;                             And this is our stack pointer for tracking of loops
    CYCLE: while ($lpos < $no_tlines) {
        my $tl = $tlines[$lpos++];                  Fetch the line at the program counter and advance
        $tl =~ m/^(.*?)\s*$/; $tl = $1;             Strip trailing spaces
        if ($TRACE_CI_EXECUTION == 1)
            ⟨Print line and contents of repeat stack 17⟩;
        if ($tl =~ m/^\s*\[\[(.*?)\]\]\s*$/) {
            my $command = $1;
            ⟨Deal with a Select command 18⟩;
            ⟨Deal with a Repeat command 19⟩;
            ⟨Deal with a Repeat End command 20⟩;
            print "Still here, ", $command, "\n";
        }
        ⟨Skip line if inside an empty loop 21⟩;
        ⟨Make substitutions of square-bracketed variables in line 24⟩;
        print CONTS $tl, "\n";                      Copy the now finished line to the output
    }
    close (CONTS);
}
```

§15. File handling.

⟨Read in the source file containing the contents page template 15⟩ ≡

```
if (not(open(TEMPL, $path_to_template))) {
    print "inweb: warning: unable to generate index because can't find template at ",
        $path_to_template, "\n";
    return;
}
while ($tl = <TEMPL>) {
    $tlines[$no_tlines++] = $tl;
}
close (TEMPL);
if ($TRACE_CI_EXECUTION == 1) {
    print "Read template <", $path_to_template, ">: ", $no_tlines, " line(s)\n";
}
```

This code is used in §14.

§16.

⟨Open the contents page file to be constructed 16⟩ ≡

```
my $path_to_contents = $web_setting."Woven/".$contents_page_leafname;
if (not(open(CONTS, '>'.$path_to_contents))) {
    print "inweb: warning: unable to generate index because can't open to write ",
        $path_to_contents, "\n";
    return;
}
```

This code is used in §14.

§17. The repeat stack and loops.

⟨Print line and contents of repeat stack 17⟩ ≡

```
my $j;
print sprintf("%04d: %tl\nStack:", $lpos-1, $tl);
for ($j=0; $j<$stack_pointer; $j++) {
    print " ", $j, ": ", $repeat_stack_variable[$j], "/", $repeat_stack_threshold[$j], " ",
        $repeat_stack_level[$j];
}
print "\n";
```

This code is used in §14.

§**18.** We start the direct commands with Select, which is implemented as a one-iteration loop in which the loop variable has the given section or chapter as its value during the sole iteration.

⟨Deal with a Select command 18⟩ ≡

```
if ($command =~ m/^Select (.*)$/) {
    my $sigil = $1;
    my $j;
    for ($j = 0; $j<$no_sections; $j++) {
        if ($section_sigil[$j] eq $sigil) {
            start_CI_loop("Section", $j, $j, $lpos);
            next CYCLE;
        }
    }
    for ($j = 0; $j<$no_chapters; $j++) {
        if ($chapter_sigil[$j] eq $sigil) {
            start_CI_loop("Chapter", $j, $j, $lpos);
            next CYCLE;
        }
    }
    inweb_error_at("don't recognise the chapter or section abbreviation $sigil",
        $path_to_template, $lpos);
    next;
}
```

This code is used in §14.

§**19.** Next, a genuine loop beginning:

⟨Deal with a Repeat command 19⟩ ≡

```
if ($command =~ m/^Repeat (Chapter|Section)$/) {
    my $from;
    my $to;
    my $lev = $1;
    if ($lev eq "Chapter") {
        $from = 0;
        $to = $no_chapters-1;
        if ($only_setting) {
            my $j;
            for ($j = 0; $j<$no_chapters; $j++) {
                if ($chapter_sigil[$j] eq $only_setting) {
                    start_CI_loop("Chapter", $j, $j, $lpos);
                    next CYCLE;
                }
            }
        }
    } elsif ($lev eq "Section") {
        my $within_chapter = heading_topmost_on_stack("Chapter");
        if ($within_chapter == -1) {
            $from = 0;
            $to = $no_sections-1;
        } else {
            $from = -1;
            my $sn;
            for ($sn = 0; $sn < $no_sections; $sn++) {
```

```
                if ($section_chap[$sn] == $within_chapter) {
                    if ($from == -1) { $from = $sn; }
                    $to = $sn;
                }
            }
        }
    } else {
        inweb_error_at("don't know how to repeat $lev: only Chapter or Section",
            $path_to_template, $lpos);
    }
    if ($from >= 0) { start_CI_loop($lev, $from, $to, $lpos); }
    next CYCLE;
}
```

This code is used in §14.

§**20.** And at the other bookend:

⟨Deal with a Repeat End command 20⟩ ≡

```
if ($command =~ m/^End (Repeat|Select)$/) {
    if ($stack_pointer <= 0) {
        inweb_error_at("stack underflow on contents template", $path_to_template, $lpos);
    }
    $repeat_stack_variable[$stack_pointer-1]++;
    if ($repeat_stack_variable[$stack_pointer-1] >=
        $repeat_stack_threshold[$stack_pointer-1]) {
        end_CI_loop();
    } else {
        $lpos = $repeat_stack_startpos[$stack_pointer-1];                    Back round loop
    }
    next CYCLE;
}
```

This code is used in §14.

§**21.** It can happen that a section loop, at least, is empty:

⟨Skip line if inside an empty loop 21⟩ ≡

```
my $rstl;
for ($rstl = $stack_pointer-1; $rstl >= 0; $rstl--) {
    if ($repeat_stack_variable[$stack_pointer-1] >=
        $repeat_stack_threshold[$stack_pointer-1]) { next CYCLE; }
}
```

This code is used in §14.

§**22.** If called with level `"Chapter"`, this returns the topmost chapter number on the stack; and similarly for `"Section"`.

```
sub heading_topmost_on_stack {
    my $level = $_[0];
    my $rstl;
    for ($rstl = $stack_pointer-1; $rstl >= 0; $rstl--) {
        if ($repeat_stack_level[$rstl] eq $level) {
            return $repeat_stack_variable[$rstl];
        }
    }
    return -1;
}
```

§**23.** This is the code for starting a loop, which stacks up the details, and similarly for ending it by popping them again:

```
sub start_CI_loop {
    my $level = $_[0];
    my $start_point = $_[1];
    my $end_point = $_[2];
    my $first_line_of_body = $_[3];
    $repeat_stack_level[$stack_pointer] = $level;
    $repeat_stack_variable[$stack_pointer] = $start_point;
    $repeat_stack_threshold[$stack_pointer] = $end_point+1;
    $repeat_stack_startpos[$stack_pointer++] = $first_line_of_body;
}
sub end_CI_loop {
    $stack_pointer--;
}
```

§**24. Variable substitutions.** We can now forget about this tiny stack machine: the one task left is to take a line from the template, and make substitutions of variables into its square-bracketed parts.

⟨Make substitutions of square-bracketed variables in line 24⟩ ≡

```
    while ($tl =~ m/^(.*?)\[\[(.*?)\]\](.*?)\s*$/) {
        my $left = $1; my $right = $3; my $subs = $2;
        if (exists($bibliographic_data{$subs})) {
            ⟨Substitute any bibliographic datum named 25⟩;
        } elsif ($subs =~ m/^(Chapter|Section|Complete) (.*)$/) {
            my $lev = $1;
            my $detail = $2;
            my $heading_number;
            if ($lev eq "Complete") {
                $heading_number = $complete_web_weave_number;
            } else {
                $heading_number = heading_topmost_on_stack($lev);
                if ($heading_number == -1) {
                    inweb_error_at("no $lev is currently selected",
                        $path_to_template, $lpos);
                }
            }
```

```
            if ($lev eq "Complete") {
                ⟨Substitute a detail about the complete PDF 26⟩;
            } elsif ($lev eq "Chapter") {
                ⟨Substitute a detail about the currently selected Chapter 27⟩;
            } else {
                ⟨Substitute a detail about the currently selected Section 28⟩;
            }
        } else {
            $subs = '<b>'.$subs.'</b>';
        }
        $tl = $left.$subs.$right;
    }
```

This code is used in §14.

§**25.** This is why, for instance, `[[Author]]` is replaced by the author's name:

⟨Substitute any bibliographic datum named 25⟩ ≡

```
    $subs = $bibliographic_data{$subs};
```

This code is used in §24.

§**26.** We store little about the complete-web-in-one-file PDF:

⟨Substitute a detail about the complete PDF 26⟩ ≡

```
    if ($detail eq "PDF Size") {
        $subs = ($pdf_size[$complete_web_weave_number]/1024)."KB";
    } elsif ($detail eq "Extent") {
        $subs = ($page_count[$complete_web_weave_number])."pp";
    } elsif ($detail eq "Leafname") {
        $subs = $complete_PDF_leafname;
    } else {
        $subs = $detail.' for complete web';
    }
```

This code is used in §24.

§**27.** And this is why [[Chapter Leafname]] turns into `$chapter_woven_pdf_leafname[$c]` for the current chapter number `$c`. Extent, PDF Size and Errors can only be determined if the chapter in question has already been woven on this run of `inweb`: hence the dashes if not.

⟨Substitute a detail about the currently selected Chapter 27⟩ ≡

```
    if ($detail eq "Title") {
        $subs = $chapter_title[$heading_number];
    } elsif ($detail eq "Purpose") {
        $subs = $chapter_rubric[$heading_number];
    } elsif ($detail eq "Leafname") {
        $subs = $chapter_woven_pdf_leafname[$heading_number];
    } elsif ($detail eq "Extent") {
        my $wtn = $chapter_weave_number[$heading_number];
        if ($wtn == -1) { $subs = "--"; } else {
            $subs = $page_count[$wtn]."pp";
        }
    } elsif ($detail eq "PDF Size") {
        my $wtn = $chapter_weave_number[$heading_number];
```

```
        if ($wtn == -1) { $subs = "--"; } else {
            $subs = ($pdf_size[$wtn]/1024)."KB";
        }
    } elsif ($detail eq "Errors") {
        my $wtn = $chapter_weave_number[$heading_number];
        if ($wtn == -1) { $subs = ""; } else {
            $subs = "";
            if ($overfull_hbox_count[$wtn] > 0) {
                $subs = $overfull_hbox_count[$wtn]." overfull hbox ";
            }
            if ($tex_error_count[$wtn] > 0) {
                $subs .= $tex_error_count[$wtn]." TeX error";
            }
        }
    } else {
        $subs = $detail.' for '.$lev.' '.$heading_number;
    }
```

This code is used in §24.

§**28.** And this, finally, is a very similar construction for Sections.

⟨Substitute a detail about the currently selected Section 28⟩ ≡

```
    if ($detail eq "Title") {
        $subs = $section_leafname[$heading_number];
        $subs =~ s/\.w$//;
    } elsif ($detail eq "Purpose") {
        $subs = $section_purpose[$heading_number];
    } elsif ($detail eq "Leafname") {
        $subs = $section_sigil[$heading_number];
        $subs =~ s/\//-/; $subs = $subs.'.pdf';
    } elsif ($detail eq "Code") {
        $subs = $section_sigil[$heading_number];
    } elsif ($detail eq "Lines") {
        $subs = $section_extent[$heading_number];
    } elsif ($detail eq "Source") {
        $subs = $section_pathname_relative_to_web[$heading_number];
    } elsif ($detail eq "Paragraphs") {
        $subs = $section_no_pars[$heading_number];
    } elsif ($detail eq "Mean") {
        my $denom = $section_no_pars[$heading_number];
        if ($denom == 0) { $denom = 1; }
        $subs = $section_extent[$heading_number]/$denom;
    } elsif ($detail eq "Extent") {
        my $wtn = $section_weave_number[$heading_number];
        if ($wtn == -1) { $subs = "--"; } else {
            $subs = $page_count[$wtn]."pp";
        }
    } elsif ($detail eq "PDF Size") {
        my $wtn = $section_weave_number[$heading_number];
        if ($wtn == -1) { $subs = "--"; } else {
            $subs = ($pdf_size[$wtn]/1024)."KB";
        }
    } elsif ($detail eq "Errors") {
```

```
        my $wtn = $section_weave_number[$heading_number];
        if ($wtn == -1) { $subs = ""; } else {
            $subs = "";
            if ($overfull_hbox_count[$wtn] > 0) {
                $subs = $overfull_hbox_count[$wtn]." overfull hbox ";
            }
            if ($tex_error_count[$wtn] > 0) {
                $subs .= $tex_error_count[$wtn]." TeX error";
            }
        }
    } else {
        $subs = $detail.' for '.$lev.' '.$heading_number;
    }
```

This code is used in §24.

The Weaver 3/weave

Purpose

To weave a portion of the code into instructions for TeX.

§**1. The Master Weaver.** Here's what has happened so far, on a `-weave` run of `inweb`: on any other sort of run, of course, we would never be in this section of code. The web was read completely into memory, and then fully parsed, with all of the arrays and hashes populated. A request was then made either to swarm a mass of individual weaves, or to make just a single weave, with the target in each case being identified by its sigil. A further decoding layer then translated each sigil into rather more basic details of what to weave and where to put the result: and so we arrive at the front door of the routine `weave_source` below.

```
sub weave_source {
    my $filename = $_[0];
    my $with_cover_sheet = $_[1];                    Should we add a cover sheet at the front?
    my $weave_section_match = $_[2];                 Regular expression which sigils for the lines must match
    open(WEAVEOUT, ">".$filename) or die "inweb: can't open weave file '$filename' for output";
    print WEAVEOUT "% Weave ", $filename, " generated by ", $INWEB_BUILD, "\n";
    ⟨Incorporate suitable TEX macro definitions into the woven output 3⟩;
    if ($with_cover_sheet == 1) { weave_cover_sheet(); }
    ⟨Start the weaver with a clean slate 2⟩;
    OUTLOOP:
    for ($i=0; $i<$no_lines; $i++) {
        ⟨Skip material from any sections which are not part of this target 4⟩;
        $lines_woven++;                              In principle, then, what remains is in the target; but...
        ⟨Material between ...and so on... markers is not visible 5⟩;
        ⟨Grammar and index entries are collated elsewhere, not woven in situ 6⟩;
        ⟨Respond to any commands aimed at the weaver, and otherwise skip commands 7⟩;
        Some of the more baroque front matter of a section...
        ⟨Weave the Purpose marker as a little heading 9⟩;
        ⟨If we need to work in a section table of contents and this is a blank line, do it now 10⟩;
        ⟨Deal with the Interface passage 11⟩;
        ⟨Weave the Definitions marker as a little heading 12⟩;
        ⟨Weave the section bar as a horizontal rule 13⟩;
        The crucial junction point between modes...
        ⟨Deal with the marker for the start of a new paragraph, section or chapter 32⟩;
        With all exotica dealt with, we now just have material to weave verbatim...
        my $matter = $line_text[$i];
        if ($line_is_comment[$i] == 1) ⟨Weave verbatim matter in commentary style 14⟩
        else ⟨Weave verbatim matter in code style 19⟩;
    }
    ⟨Complete any started but not-fully-woven paragraph 36⟩;
    print WEAVEOUT "% End of weave: ", $lines_woven, " lines from ", $no_lines, "\n";
    print WEAVEOUT '\end', "\n";
    close(WEAVEOUT);
}
```

§**2.** We can now begin on a clean page, by initialising the state of the weaver. It's convenient for these to be global variables since the weaver is not recursively called, and it avoids some nuisance over scope caused by the braces implicit in the `inweb` macro below:

⟨Start the weaver with a clean slate 2⟩ ≡

```
$within_TeX_beginlines = 0;                 Currently setting copy between \beginlines and \endlines?
$weaving_suspended = 0;
$interface_table_pending = 0;
$functions_in_par = ""; $structs_in_par = "";
$current_macro_definition = "";
$next_heading_without_vertical_skip = 0;
$show_section_toc_soon = 0;                 Is a table of contents for the section imminent?
$horizontal_rule_just_drawn = 0;
$chaptermark = ""; $sectionmark = "";
begin_making_pdf_links();                   Be ready to set PDF hyperlinks from now on
$lines_woven = 0;                           Number of lines in the target to be woven
```

This code is used in §1.

§**3. The TEX macros.** We don't use TEX's `\input` mechanism for macros because it is so prone to failures when searching directories (especially those with spaces in the names) and then locking TEX into a repeated prompt for help from `stdin` which is rather hard to escape from.

Instead we paste the entire text of our macros file into the woven TEX:

⟨Incorporate suitable TEX macro definitions into the woven output 3⟩ ≡

```
my $ml;
open(MACROS, $path_to_inweb_setting.'inweb/Materials/inweb-macros.tex')
    or die "inweb: can't open file of TeX macros";
while ($ml = <MACROS>) {
    $ml =~ m/^(.*)\s*$/; $ml = $1;
    print WEAVEOUT $ml, "\n";
}
close MACROS;
```

This code is used in §1.

§**4. Reasons to skip things.** We skip any material in files not chosen at the command line (or by the swarmer) for weave output:

⟨Skip material from any sections which are not part of this target 4⟩ ≡

```
if ($weave_section_match ne "") {
    if (not ($section_sigil[$line_sec[$i]] =~ m/$weave_section_match/)) { next OUTLOOP; }
}
```

This code is used in §1.

§**5.** We skip material between "...and so on..." markers as being even more tedious than the rest of the program:

⟨Material between ...and so on... markers is not visible 5⟩ ≡
```
if ($line_category[$i] == $TOGGLE_WEAVING_LCAT) {
    if ($weaving_suspended == 0) {
        print WEAVEOUT "\\smallskip\\par\\noindent";
        print WEAVEOUT "{\\ttninepoint\\it ...and so on...}\\smallskip\n";
        $weaving_suspended = 1;
        next OUTLOOP;
    }
    $weaving_suspended = 0;
    next OUTLOOP;
}
if ($weaving_suspended == 1) { next OUTLOOP; }
```
This code is used in §1.

§**6.** And we skip some material used only for compiling tables:

⟨Grammar and index entries are collated elsewhere, not woven in situ 6⟩ ≡
```
if ($line_category[$i] == $GRAMMAR_LCAT) { next OUTLOOP; }
if ($line_category[$i] == $GRAMMAR_BODY_LCAT) { next OUTLOOP; }
if ($line_category[$i] == $INDEX_ENTRY_LCAT) { next OUTLOOP; }
```
This code is used in §1.

§**7.** And lastly we ignore commands, or act on them if they happen to be aimed at us; but we don't weave them into the output, that's for sure.

⟨Respond to any commands aimed at the weaver, and otherwise skip commands 7⟩ ≡
```
if ($line_category[$i] == $COMMAND_LCAT) {
    my $argument = $line_operand_2[$i];
    if ($line_operand[$i] == $PAGEBREAK_CMD) {
        print WEAVEOUT "\\vfill\\eject\n";
    }
    if ($line_operand[$i] == $BNF_GRAMMAR_CMD) {
        weave_bnf_grammar($argument);
        print WEAVEOUT "\\smallbreak\n";
        print WEAVEOUT "\\hrule\\smallbreak\n";
    }
    if ($line_operand[$i] == $THEMATIC_INDEX_CMD) {
        weave_thematic_index($argument);
        print WEAVEOUT "\\smallbreak\n";
        print WEAVEOUT "\\hrule\\smallbreak\n";
    }
    if ($line_operand[$i] == $FIGURE_CMD) {
        ⟨Weave a figure 8⟩;
    }
    Otherwise assume it was a tangler command, and ignore it here
    next OUTLOOP;
}
```
This code is used in §1.

§**8.** TEX itself has an almost defiant lack of support for anything pictorial, which is one reason it didn't live up to its hope of being the definitive basis for typography; even today the loose confederation of TEX-like programs and extensions lack standard approaches. Here we're going to use `pdftex` features, having nothing better. All we're trying for is to insert a picture, scaled to a given width, into the text at the current position.

⟨Weave a figure 8⟩ ≡

```
my $figname = $argument;
my $width = "";
if ($figname =~ m/^(\d+)cm\: (.*)$/) {
    $figname = $2;
    $width = " width ".$1."cm";
}
print WEAVEOUT "\\pdfximage".$width."{../Figures/".$figname, "}\n";
print WEAVEOUT "\\smallskip\\noindent",
    "\\hbox to\\hsize{\\hfill\\pdfrefximage \\pdflastximage\\hfill}",
    "\\smallskip\n";
```

This code is used in §7.

§**9. Headings.** The purpose is set with a little heading. Its operand is that part of the purpose-text which is on the opening line; the rest follows on subsequent lines until the next blank.

⟨Weave the Purpose marker as a little heading 9⟩ ≡

```
if ($line_category[$i] == $PURPOSE_LCAT) {
    print WEAVEOUT "\\smallskip\\par\\noindent{\\it Purpose}\\smallskip\\noindent\n";
    print WEAVEOUT $line_operand[$i], "\n";
    $show_section_toc_soon = 1;
    $horizontal_rule_just_drawn = 0;
    next OUTLOOP;
}
```

This code is used in §1.

§**10.** This normally appears just after the Purpose subheading:

⟨If we need to work in a section table of contents and this is a blank line, do it now 10⟩ ≡

```
if (($show_section_toc_soon == 1) && ($line_text[$i] =~ m/^\s*$/)) {
    print WEAVEOUT "\\medskip";
    $show_section_toc_soon = 0;
    if ($section_toc[$line_sec[$i]] ne "") {
        print WEAVEOUT "\\smallskip\\hrule\\smallskip\\par\\noindent{\\usagefont ",
            $section_toc[$line_sec[$i]], "}\\par\\medskip\\hrule\\bigskip\n";
        $horizontal_rule_just_drawn = 1;
    } else {
        $horizontal_rule_just_drawn = 0;
    }
}
```

This code is used in §1.

§**11.** After which we have the Interface, except that this is skipped entirely, as far as weaving is concerned, unless (a) there's a body to it, and (b) we are in C-for-Inform mode, in which case the body is ignored anyway but a table of I6 template invocations of the section's functions is set instead:

⟨Deal with the Interface passage 11⟩ ≡
```
    if ($line_category[$i] == $INTERFACE_LCAT) {
        $interface_table_pending = 1;
        next OUTLOOP;
    }
    if ($line_category[$i] == $INTERFACE_BODY_LCAT) {
        if ($interface_table_pending) {
            $interface_table_pending = 0;
            if ($web_language == $C_FOR_INFORM_LANGUAGE) {
                $horizontal_rule_just_drawn = 0;
                weave_interface_table_for_section($line_sec[$i]);
            }
        }
        next OUTLOOP;
    }
```

This code is used in §1.

§**12.** And another little heading...

⟨Weave the Definitions marker as a little heading 12⟩ ≡
```
    if ($line_category[$i] == $DEFINITIONS_LCAT) {
        print WEAVEOUT "\\smallskip\\par\\noindent{\\it Definitions}\\smallskip\\noindent\n";
        $next_heading_without_vertical_skip = 1;
        $horizontal_rule_just_drawn = 0;
        next OUTLOOP;
    }
```

This code is used in §1.

§**13.** ...with the section bar to follow. The bar line completes any half-finished paragraph and is set as a horizontal rule:

⟨Weave the section bar as a horizontal rule 13⟩ ≡
```
    if ($line_category[$i] == $BAR_LCAT) {
        ⟨Complete any started but not-fully-woven paragraph 36⟩;
        $functions_in_par = "";
        $structs_in_par = "";
        $current_macro_definition = "";
        $within_TeX_beginlines = 0;
        $next_heading_without_vertical_skip = 1;
        if ($horizontal_rule_just_drawn == 0) {
            print WEAVEOUT "\\par\\medskip\\noindent\\hrule\\medskip\\noindent\n";
        }
        next OUTLOOP;
    }
```

This code is used in §1.

§14. Commentary matter. Typographically this is a fairly simple business, since the commentary is already by definition written in TEX format: it's almost the case that we only have to transcribe it. But not quite! This is where we implement the convenient additions `inweb` makes to TEX syntax.

⟨Weave verbatim matter in commentary style 14⟩ ≡
```
⟨Weave displayed source in its own special style 15⟩;
⟨Weave a blank line as a thin vertical skip and paragraph break 16⟩;
⟨Weave bracketed list indications at start of line into indentation 17⟩;
⟨Weave tabbed code material as a new indented paragraph 18⟩;
print WEAVEOUT $matter, "\n";
next OUTLOOP;
```

This code is used in §1.

§15. Displayed source is the material marked with >> arrows in column 1.

⟨Weave displayed source in its own special style 15⟩ ≡
```
if ($line_category[$i] == $SOURCE_DISPLAY_LCAT) {
    print WEAVEOUT "\\quotesource{", $line_operand[$i], "}\n";
    next OUTLOOP;
}
```

This code is used in §14.

§16. Our style is to use paragraphs without initial-line indentation, so we add a vertical skip between them to show the division more clearly.

⟨Weave a blank line as a thin vertical skip and paragraph break 16⟩ ≡
```
if ($matter =~ m/^\s*$/) {
    print WEAVEOUT "\\smallskip\\par\\noindent%\n";
    next OUTLOOP;
}
```

This code is used in §14.

§17. Here our extension is simply to provide a tidier way to use TEX's standard `\item` and `\itemitem` macros for indented list items.

⟨Weave bracketed list indications at start of line into indentation 17⟩ ≡
```
if ($matter =~ m/^\(\.\.\.\)\s+(.*?)$/) {                    continue single
    $matter = '\item{}'.$1;
} elsif ($matter =~ m/^\(-\.\.\.\)\s+(.*?)$/) {              continue double
    $matter = '\itemitem{}'.$1;
} elsif ($matter =~ m/^\(([a-z0-9A-Z\.]+)\)\s+(.*?)$/) {     begin single
    $matter = '\item{('.$1.')}'.$2;
} elsif ($matter =~ m/^\(-([a-z0-9A-Z\.]+)\)\s+(.*?)$/) {    begin double
    $matter = '\itemitem{('.$1.')}'.$2;
}
```

This code is used in §14.

§**18.** Finally, matter encased in vertical strokes one tab stop in from column 1 in the source is set indented in code style.

⟨Weave tabbed code material as a new indented paragraph 18⟩ ≡
```
if ($matter =~ m/^\t\|/) {
    $matter = '\par\noindent\qquad '.$matter;
}
```
This code is used in §14.

§**19. Code-like matter.** Even though `inweb`'s approach, unlike `CWEB`'s, is to respect the layout of the original, this is still quite typographically complex: commentary and macro usage is rendered differently.

⟨Weave verbatim matter in code style 19⟩ ≡
```
⟨Enter beginlines/endlines mode if necessary 20⟩;
⟨Weave a blank line as a thin vertical skip 21⟩;
my $tab_stops_of_indentation = 0;
⟨Convert leading space in line matter to a number of tab stops 22⟩;
⟨Weave a suitable horizontal advance for that many tab stops 23⟩;
my $concluding_comment = "";
⟨Extract any comment matter ending the line to be set in italic 24⟩;
⟨Encase the code matter within vertical strokes 25⟩;
⟨Give constant definition lines slightly fancier openings 26⟩;
⟨Detect any structure or function definitions being woven in this paragraph 27⟩;
⟨Typeset the CWEB-style macros with cute highlighting and PDF links 28⟩;
⟨Insert continuation line breaks to cope with very long lines 29⟩;
print WEAVEOUT $matter, $concluding_comment, "\n";
next OUTLOOP;
```
This code is used in §1.

§**20.** Code is typeset between the `\beginlines` and `\endlines` macros in TEX, so:

⟨Enter beginlines/endlines mode if necessary 20⟩ ≡
```
if ($within_TeX_beginlines == 0) {
    print WEAVEOUT "\\beginlines\n"; $within_TeX_beginlines = 1;
}
```
This code is used in §19.

§**21.** A blank line is typeset as a thin vertical skip (no TEX paragraph break is needed):

⟨Weave a blank line as a thin vertical skip 21⟩ ≡
```
if ($matter =~ m/^\s*$/) {
    print WEAVEOUT "\\smallskip\n";
    next OUTLOOP;
}
```
This code is used in §19.

§**22.** Examine the white space at the start of the code line, and count the number of tab steps of indentation, rating 1 tab = 4 spaces:

⟨Convert leading space in line matter to a number of tab stops 22⟩ ≡
```
my $spaces_in = 0;
while ($matter =~ m/^(\s)(.*)$/) {
    $matter = $2;
    $whitespace = $1;
    if ($whitespace eq "\t") {
        $spaces_in = 0;
        $tab_stops_of_indentation++;
    } else {
        $spaces_in++;
        if ($spaces_in == 4) {
            $tab_stops_of_indentation++;
            $spaces_in = 0;
        }
    }
}
```

This code is used in §19.

§**23.** We actually use horizontal spaces rather than risk using TEX's messy alignment system:

⟨Weave a suitable horizontal advance for that many tab stops 23⟩ ≡
```
my $i;
for ($i=0; $i<$tab_stops_of_indentation; $i++) {
    print WEAVEOUT "\\qquad";
}
```

This code is used in §19.

§**24.** Comments which run to the end of a line are set in italic type. If the only item on their lines, they are presented at the code tab stop; otherwise, they are set flush right.

⟨Extract any comment matter ending the line to be set in italic 24⟩ ≡
```
if (line_ends_with_comment($matter)) {
    if ($part_before_comment eq "") {
        $matter = $part_before_comment; my $commentary = $part_within_comment;
        $concluding_comment = "{\\ttninepoint\\it ".$commentary."}";
    } else {
        $matter = $part_before_comment; my $commentary = $part_within_comment;
        if ($commentary =~ m/^C\d+\S+$/) {
            $commentary = "Test with |".$commentary.".txt|";
        }
        $concluding_comment = "\\hfill\\quad {\\ttninepoint\\it ".$commentary."}";
    }
}
```

This code is used in §19.

§**25.** Code is typeset by TEX within vertical strokes; these switch a sort of typewriter-type verbatim mode on and off. To get an actual stroke, we must escape from code mode, escape it using a backslash `|`, then re-enter code mode once again:

⟨Encase the code matter within vertical strokes 25⟩ ≡
```
$matter =~ s/\|/\|\\\|\|/g;
$matter = '|'.$matter.'|';
```
This code is used in §19.

§**26.** Set the `@d` definition escape very slightly more fancily (remembering that we are now encased in verticals):

⟨Give constant definition lines slightly fancier openings 26⟩ ≡
```
$matter =~ s/^\|\@d /{\\ninebf define}| /;
```
This code is used in §19.

§**27.** We note any structure typedefs, and also any functions which are called from outside this section, whose names we typeset in red. (We do this so that the endnotes can be added at the foot of the paragraph.)

⟨Detect any structure or function definitions being woven in this paragraph 27⟩ ≡
```
if ($matter =~ m/^\|\s*typedef\s+struct\s+(.*?)\s+\{/) {
    $structs_in_par .= $1.",";
}
if ($matter =~ m/^\|\/(\*+)\/\s*(.*?\**)(\S+?)\((.*)$/) {
    my $fstars = $1;
    my $ftype = $2;
    my $fname = $3;
    my $frest = $4;
    $matter = '|'.$ftype.'|\pdfliteral direct{0 1 1 0 k}|'.$fname.'|\special{PDF:0 g}|'.
        '('.$frest;
    $functions_in_par .= $fname.",";
}
if (($tab_stops_of_indentation == 0) &&
    ($matter =~ m/^\|(\S.*?\**)(\S+?)\((.*)$/)) {
    my $ftype = $1;
    my $fname = $2;
    my $frest = $3;
    my $unamended = $fname;
    $fname =~ s/::/__/g;
    if (exists($functions_line{$fname})) {
        $matter = '|'.$ftype.'|\pdfliteral direct{0 1 1 0 k}|'.$unamended.
            '|\special{PDF:0 g}|'.'('.$frest;
        $functions_in_par .= $fname.",";
    }
}
```
This code is used in §19.

§**28.** Any usage of angle-macros is highlighted in several cute ways: first, we make use of colour and we drop in the paragraph number of the definition of the macro in small type –

⟨Typeset the CWEB-style macros with cute highlighting and PDF links 28⟩ ≡
```
while ($matter =~ m/^(.*?)\@\<(.*?)\@\>(.*)$/) {
    my $left = $1;
    my $right = $3;
    my $href = $2.' {\sevenss '.$cweb_macros_paragraph_number{$2}.'}';
    my $angled_sname = $2;
    my $xrefcol = '\pdfliteral direct{1 1 0 0 k}';
    my $blackcol = '\special{PDF:0 g}';
    my $tweaked = '$\langle${\xreffont'.$xrefcol.$href.'}'.$blackcol.'$\rangle$';
    if ($right =~ m/^\s*\=(.*)$/) ⟨This is where the angle-macro is defined 30⟩
    else ⟨This is a reference to an angle-macro, not its definition 31⟩;
    $matter = $left.'|'.$tweaked.'|'.$right;
}
```
This code is used in §19.

§**29.** Overlong lines of code are liable to cause overfull hbox errors, the bane of all TEX users.

⟨Insert continuation line breaks to cope with very long lines 29⟩ ≡
```
my $count = 0;
my $code_matter = 0;
my $done = "";
my $number_of_splits = 0;
my $last_char = "";
while ($matter =~ m/^(.)(.*?)$/) {
    my $next_char = $1; $matter = $2;
    if (($next_char eq "|") && ($last_char ne "\\")) { $code_matter = 1 - $code_matter; }
    elsif ($code_matter == 1) {
        $count++;
        if ($count > 99) {
            $done = $done."| \n\\quad ...| "; $number_of_splits++; $count = 10;
        }
    }
    $done = $done.$next_char;
    $last_char = $next_char;
}
$matter = $done;
```
This code is used in §19.

§**30.** We mark this as a hyperlink destination (using `pdftex` extensions again):

⟨This is where the angle-macro is defined 30⟩ ≡
```
$right = $1;
$tweaked = '\pdfdest num '.link_for_par_name($angled_sname).' fit '.$tweaked;
$tweaked = $tweaked.' $\equiv$ ';
$current_macro_definition = $angled_sname;
```
This code is used in §28.

§**31.** And this is a link:

⟨This is a reference to an angle-macro, not its definition 31⟩ ≡

```
my $new_p = $line_paragraph_number[$i];
if (not($angled_paragraph_usage{$angled_sname} =~ m/$new_p,$/)) {
    $angled_paragraph_usage{$angled_sname} .= $new_p.',';
}
$tweaked =
    '\pdfstartlink attr{/C [0.9 0 0] /Border [0 0 0]} '.
    'goto num '.link_for_par_name($angled_sname).' '.
    $tweaked.'\pdfendlink';
```

This code is used in §28.

§32. How paragraphs begin.

⟨Deal with the marker for the start of a new paragraph, section or chapter 32⟩ ≡

```
if ($line_category[$i] == $PARAGRAPH_START_LCAT) {
    ⟨Complete any started but not-fully-woven paragraph 36⟩;
    ⟨If this is a paragraph break forced onto a new page, then throw a page 33⟩;
    my $weight = $line_operand[$i];
    my $para_title = $line_operand_2[$i];
    my $new_chap_num = $section_chap[$line_sec[$i]];
    my $secnum = $line_sec[$i];
    my $parnum = $line_paragraph_number[$i];
    my $ornament = $line_paragraph_ornament[$i];
    my $mark = "";
    ⟨Work out the next mark to place into the TEX vertical list 34⟩;
    $functions_in_par = "";
    $structs_in_par = "";
    $current_macro_definition = "";
    my $TeX_macro = "";
    ⟨Choose which TEX macro to use in order to typeset the new paragraph heading 35⟩;
    print WEAVEOUT "\\".$TeX_macro, "{", $parnum, "}{", $para_title, "}{",
        $mark, "}{", $ornament, "}{", $section_sigil[$secnum], "}%\n";
    There's quite likely ordinary text on the line following the paragraph
    start indication, too, so we need to weave this out:
    if ($line_text[$i] ne "") { print WEAVEOUT $line_text[$i], "\n"; }
    Chapter headings get a chapter title page, or possibly pages, too:
    if ($weight == 3) {
        my $sigil = $chapter_sigil[$section_chap[$line_sec[$i]]];
        print WEAVEOUT $chapter_rubric{$sigil}, "\\medskip\n";
        my $sn;
        for ($sn=0; $sn<$no_sections; $sn++) {
            if (not($section_sigil[$sn] =~ m/^$sigil\//)) { next; }
            print WEAVEOUT "\\smallskip\\noindent |", $section_sigil[$sn], "|: ",
                "{\\it ", $section_leafname[$sn], "}\\quad\n";
            print WEAVEOUT $section_purpose[$sn];
        }
    }
    And that completes the new paragraph opening.
    next OUTLOOP;
}
```

This code is used in §1.

§33. A few paragraphs are marked `@pp` and forced to begin on a new page:

⟨If this is a paragraph break forced onto a new page, then throw a page 33⟩ ≡

```
if ($line_starts_paragraph_on_new_page[$i]) {
    print WEAVEOUT "\\vfill\\eject\n";
}
```

This code is used in §32.

§**34.** "Marks" are the vile contrivance by which TEX produces running heads on pages which follow the material on those pages: so that the running head for a page can show the paragraph sigil for the material which tops it, for instance.

The ornament has to be set in math mode, even in the mark. § and ¶ only work in math mode because they abbreviate characters found in math fonts but not regular ones, in TEX's slightly peculiar font encoding system.

define $DEBUG_MARKING 0

⟨Work out the next mark to place into the TEX vertical list 34⟩ ≡

```
if ($weight == 3) { $chaptermark = $para_title; $sectionmark = ""; }
if ($weight == 2) {
    $sectionmark = $para_title;
    if ($section_sigil[$secnum] ne "") { $chaptermark = $section_sigil[$secnum]; }
    if ($chaptermark ne "") { $sectionmark = " - ".$sectionmark; }
}
$mark = $chaptermark.$sectionmark."\\quad\$".$ornament."\$".$parnum;
if ($DEBUG_MARKING == 1) {
    print "Start par $i: weight $weight, title <$para_title>, ",
        "chap $new_chap_num, sec $secnum, par $parnum, ornament $ornament\n";
    print "Mark: $mark\n";
}
```

This code is used in §32.

§**35.** We want to have different heading styles for different weights, and TEX is horrible at using macro parameters as function arguments, so we don't want to pass the weight that way. Instead we use

```
\weavesection
\weavesections
\weavesectionss
\weavesectionsss
```

where the weight is the number of terminal ss, 0 to 3. (TEX macros, lamentably, are not allowed digits in their name.) In the cases 0 and 1, we also have variants `\nsweavesection` and `\nsweavesections` which are the same, but with the initial vertical spacing removed; these allow us to prevent unsightly excess white space in certain configurations of a section.

⟨Choose which TEX macro to use in order to typeset the new paragraph heading 35⟩ ≡

```
my $j;
$TeX_macro = "weavesection";
for ($j=0; $j<$weight; $j++) { $TeX_macro = $TeX_macro."s"; }
if (($next_heading_without_vertical_skip == 1) && ($weight < 2)) {
    $next_heading_without_vertical_skip = 0;
    $TeX_macro = "ns".$TeX_macro;
}
if ($weight == 3) {                                    a chapter heading: note the inch-high sigil...
    my $brief_title = $para_title;
    $brief_title =~ s/^.*?\: //;
    $para_title = "{\\sinchhigh ".$chapter_sigil[$section_chap[$line_sec[$i]]].
        "}\\quad ".$brief_title;
}
```

This code is used in §32.

§**36. How paragraphs end.** At the end of a paragraph, on the other hand, we do this:

⟨Complete any started but not-fully-woven paragraph 36⟩ ≡

```
if ($within_TeX_beginlines > 0) {
    print WEAVEOUT '\endlines', "\n"; $within_TeX_beginlines = 0;
}
show_endnotes_on_previous_paragraph($functions_in_par, $structs_in_par,
    $current_macro_definition);
```

This code is used in §1,13,32,1,13,32,1,13,32.

§**37.** The endnotes describe function calls from far away, or unexpected structure usage, or how CWEB-style code substitutions were made.

```
sub show_endnotes_on_previous_paragraph {
    my $functions_in_par = $_[0];
    my $structs_in_par = $_[1];
    my $current_macro_definition = $_[2];
    if ($current_macro_definition ne "")
        ⟨Weave endnote saying which other paragraphs use this one 38⟩
    else {
        my %fnames_done = ();
        while ($functions_in_par =~ m/^(\S+?)\,(.*)$/) {
            my $fname = $1;
            $functions_in_par = $2;
            if ($fnames_done{$fname}++ > 0) { next; }
            if ($fname eq "isdigit") { next; }
            ⟨Weave endnote saying where this function is called from 39⟩;
        }
        while ($structs_in_par =~ m/^(\S+?)\,(.*)$/) {
            my $sname = $1;
            my $sbilling;
            $structs_in_par = $2;
            if ($structure_ownership_summary{$sname} ne "") {
                $sbilling = $structure_ownership_summary{$sname};
                $sbilling =~ s/ /, /g;
                $sbilling =~ s/, $//;
                $sbilling =~ s/, (\S+)$/ and $1/;
                $sbilling = "shared with ".$sbilling;
            } else {
                $sbilling = "private to this section";
            }
            $sname =~ s/_/\\_/g;
            print WEAVEOUT "\\par\\noindent";
            print WEAVEOUT "\\penalty10000\n";
            $sname =~ s/\#/\\#/g;
            print WEAVEOUT "{\\usagefont The structure $sname is $sbilling.}\n";
        }
        print WEAVEOUT "\\smallskip\n";
    }
}
```

§**38.** We try hard to prevent a page break between paragraph and endnote (hence the high `\penalty` value). Sometimes this succeeds.

⟨Weave endnote saying which other paragraphs use this one 38⟩ ≡

```
my $used_in = $angled_paragraph_usage{$current_macro_definition};
$used_in =~ s/\,$//;
print WEAVEOUT "\\penalty1000\n";
print WEAVEOUT "\\noindent{\\usagefont This code is used in \$\\S\$",
    $used_in, ".}\\smallskip\n";
```

This code is used in §37.

§**39.** And similarly:

⟨Weave endnote saying where this function is called from 39⟩ ≡

```
my $scope = $functions_declared_scope{$fname};
my @usages;
my $no_sections_using = 0;
⟨Work out which sections to mention calls from 40⟩;
my $fname_with_underscores_escaped = $fname;
$fname_with_underscores_escaped =~ s/__/::/g;
$fname_with_underscores_escaped =~ s/_/\\_/g;
print WEAVEOUT "\\par\\noindent";
print WEAVEOUT "\\penalty10000\n";
print WEAVEOUT "{\\usagefont The function $fname_with_underscores_escaped is";
my $clause_begins = 0;
if ($scope eq "*****") {
    print WEAVEOUT " where execution begins"; $clause_begins = 1;
}
if ($scope eq "****") {
    print WEAVEOUT " invoked by a command in a |.i6t| template file"; $clause_begins = 1;
}
if (($clause_begins == 1) || (($scope eq "***") || ($scope eq "**"))) {
    if ($no_sections_using > 0) {
        if ($clause_begins) { print WEAVEOUT " and"; }
        print WEAVEOUT " called from ";
        my $x;
        for ($x=0; $x<$no_sections; $x++) {
            if ($usages[$x] == 1) {
                print WEAVEOUT $section_sigil[$x];
                if ($no_sections_using > 2) { print WEAVEOUT ", "; }
                if ($no_sections_using == 2) { print WEAVEOUT " and "; }
                $no_sections_using--;
            }
        }
    }
}
print WEAVEOUT ".}\n";
```

This code is used in §37.

§**40.** Loop through the concise string holding this information:

⟨Work out which sections to mention calls from 40⟩ ≡

```
my $x;
for ($x=0; $x<$no_sections; $x++) { $usages[$x] = 0; }
my $cp = $functions_usage_concisely_described{$fname};
while ($cp =~ m/^\:(\d+)(.*?)$/) { $usages[eval($1)] = 1; $cp = $2; }
for ($x=0; $x<$no_sections; $x++) { if ($usages[$x] == 1) { $no_sections_using++; } }
```

This code is used in §39.

§**41. The cover sheet.** This is added to the front of larger PDFs; whole chapters and the complete work. People will probably want to customise this, so it's implemented as a TEX file which we substitute bibliographic data into.

```
sub weave_cover_sheet {
    my $cover_sheet = $path_to_inweb_setting.'inweb/Materials/cover-sheet.tex';
    if (exists ($bibliographic_data{"Cover Sheet"})) {
        $cover_sheet = $web_setting.'Materials/'.$bibliographic_data{"Cover Sheet"};
    }
    $bibliographic_data{"Capitalized Title"} = $bibliographic_data{"Title"};
    $bibliographic_data{"Capitalized Title"} =~ tr/a-z/A-Z/;
    $bibliographic_data{"Booklet Title"} = $booklet_title;
    weave_cover_from($cover_sheet);
}
```

§**42.** Note that the substitution `[[Cover Sheet]]` embeds the whole default cover sheet; this is so that we can just add a continuation of that, if we want to.

```
sub weave_cover_from {
    my $cs_filename = $_[0];
    local *COVER;
    open(COVER, $cs_filename) or die "inweb: cannot find cover sheet file";
    my $csl;
    while ($csl = <COVER>) {
        $csl =~ m/^(.*?)\s*$/; $csl = $1;
        if ($csl =~ m/^\%/) { next; }                    a TeX comment begins with a percent sign
        while ($csl =~ m/^(.*?)\[\[(.*?)\]\](.*)$/) {
            print WEAVEOUT $1; my $expander = $2; $csl = $3;
            if ($expander eq "Cover Sheet") {
                my $insert_filename =
                    $path_to_inweb_setting.'inweb/Materials/cover-sheet.tex';
                if ($insert_filename ne $cs_filename) {
                    weave_cover_from($insert_filename);
                }
            } elsif (exists($bibliographic_data{$expander})) {
                print WEAVEOUT $bibliographic_data{$expander};
            } else {
                print WEAVEOUT $expander;
            }
        }
        print WEAVEOUT $csl, "\n";
    }
```

```
    close (COVER);
}
```

§**43. Table of I6 template interpreter invocations.** This is only used in C-for-Inform mode, and shows four-star functions – the ones called by the I6T interpreter.

```
sub weave_interface_table_for_section {
    my $sect_no = $_[0];
    my $j;
    my $red_text = "\|\\pdfliteral direct\{0 1 1 0 k\}\|";
    my $blue_text = "\|\\pdfliteral direct\{1 1 0 0 k\}\|";
    my $black_text = "\|\\special{PDF:0 g}\|";
    my $fname;
    my %functions_in_order = ();                          a device used to sort functions in definition order
    foreach $fname (keys %functions_line) {
        $functions_in_order{sprintf("%07d", $functions_line{$fname})} = $fname;
    }
    my $heading_made = 0;
    foreach $fnamed (sort keys %functions_in_order) {
        $fname = $functions_in_order{$fnamed};
        my $j = $functions_line{$fname};
        if ($line_sec[$j] != $sect_no) { next; }
        if ($functions_declared_scope{$fname} ne "****") { next; }
        if ($heading_made == 0) {
            $heading_made = 1;
            print WEAVEOUT "\\bigbreak\\par\\noindent",
                "{\\it Template interpreter commands}\\smallskip\n";
        }
        print WEAVEOUT "\\par\\noindent\n";
        print WEAVEOUT "{\\sevenss", $line_paragraph_number[$j], "}|  |\n";
        $spc = $functions_arglist{$fname};
        $spc =~ s/\,\s*/\,\n\t\t/g;
        $access = 'callv';
        if ($spc =~ m/FILE \*/) { $access = 'call'; }
        if ($fname =~ m/^(.*)_array$/) { $fname = $1; $access = 'array'; }
        if ($fname =~ m/^(.*)_routine$/) { $fname = $1; $access = 'routine'; }
        $fname =~ s/__/::/g;
        print WEAVEOUT '|{-', $access, ':', $blue_text, $fname, $black_text, "}|\n";
    }
    if ($heading_made == 1) {
        print WEAVEOUT "\\bigbreak\\noindent\n";
    }
}
```

§**44. Thematic Index.** Produced in response to explicit weaver commands (and otherwise never visible, so in a web which doesn't use indexing, this will never have any effect).

```
sub weave_thematic_index {
    my $wanted= $_[0];
    foreach $subject (sort keys %thematic_indices) {
        my $chunk;
        if (($wanted ne "Remainder") && ($wanted ne $subject)) { next; }
        if ($index_subject_done{$subject}) { next; }
        $index_subject_done{$subject} = 1;
        print WEAVEOUT "{\\bf{", $subject, "}}\\par\n\\smallbreak\\hrule\\smallbreak\n";
        print WEAVEOUT ;
        $chunk = $thematic_indices{$subject};
        while ($chunk =~ m/^(.*?)\.\.\.(\d+)\|(.*)$/) {
            my $entry = $1;
            my $at = $2;
            $chunk = $3;
            print WEAVEOUT "\\line{", $entry,
                "\\leaders\\hbox to 1em{\\hss.\\hss}\\hfill {\\xreffont ",
                $line_csn[eval($at)], "}}\n";
        }
    }
}
```

§**45. PDF links.** Hypertext links within a PDF document are now a pretty well-established part of the PDF specification: we don't set any that point outside it. TEX of course cannot generate such details, but `pdftex` and other more modern forms can.

```
sub begin_making_pdf_links {
    %link_number_for_par_name = ();
    $no_par_name_links = 1;
}
sub link_for_par_name {
    my $par_name = $_[0];
    if (not(exists $link_number_for_par_name{$par_name})) {
        $link_number_for_par_name{$par_name} = $no_par_name_links++;
    }
    return $link_number_for_par_name{$par_name};
}
```

Backus-Naur Form 3/bnf

Purpose

To weave any collated BNF grammar from the web into a nicely typeset form.

§**1. Parsing BNF.** See the manual for more on what these grammars are, and how they're notated in the web. They have no function in the compiled code, and are purely for documentation. It's a diversion rather than the main work of the weaver, but it may as well stay in `inweb`, though it won't be useful for many webs.

There are shameless Inform-specific hacks in this section, but they're not likely to do any harm.

When we read and parsed the web, we extracted the BNF grammar lines into an array. But they were otherwise raw, and we will have to manipulate them quite a bit to turn them into TEX.

First, a little bit of jargon. The tokens we represent in angle brackets are called "nonterminals", and the grammar lines which provide ways they can be made up are called "productions". Our grammar contains only lines in which the left hand side is a single nonterminal.

```
sub parse_bnf_grammar {
    ⟨Go through the raw lines of BNF grammar, stringing together, and spotting nonterminals 2⟩;
    ⟨Initialise the nonterminals discovered 6⟩;
    ⟨Work out which nonterminals depend on which other ones 7⟩;
    ⟨Elect certain nonterminals as being fundamental 8⟩;
    ⟨Make the fundamental nonterminals the roots of trees of those depending on them 9⟩;
    ⟨Any nonterminals remaining are placed at the top level 10⟩;
}
```

§**2.** Our basic plan is to use the hash `$productions{$nt}` to accumulate all productions which have the nonterminal `$nt` as their right-hand-side; in other words, all ways to make `$nt`.

⟨Go through the raw lines of BNF grammar, stringing together, and spotting nonterminals 2⟩ ≡

```
    my $i;
    my $l;
    my $nonterminal_being_defined;
    $nonterminal_being_defined = "";
    for ($i=0; $i<$bnf_grammar_lines; $i++) {
        $l = $bnf_grammar[$i];
        if ($l =~ m/^\s*\<(.*?)\>\s*(.*)$/) {
            $nonterminal_being_defined = $1; $l = $2;
            if (not(exists $productions{$nonterminal_being_defined}))
                ⟨Create new nonterminal 3⟩;
        }
        if ($l =~ m/^\s*$/) { next; }                                    skip blank line
        if ($l =~ m/^\s*\:\=\s+(.*)$/) ⟨A new production begins here 4⟩;
        if ($l =~ m/^\s*\.\.\.\s+(.*)$/) ⟨The previous production continues onto this line 5⟩;
    }
```

This code is used in §1.

§**3.** When we see a nonterminal with a name we haven't seen before, we make it a key in the `%productions` hash, and also set its TEX representation, including a footnote identifying where in the web it originates.

⟨Create new nonterminal 3⟩ ≡
```
$productions{$nonterminal_being_defined} = $i;
$production_tex{$nonterminal_being_defined}
    = "\\nonterminal{" . $nonterminal_being_defined . "}\$_{\\rm ".
        $section_sigil[$line_sec[$bnf_grammar_source[$i]]] . "}\$";
```
This code is used in §2.

§**4.** `$productions{$nt}` is a single string, but it holds a concatenated list of productions for `$nt`. We can divide this up into its entries again because each one ends in a sentinel ampersand, `&`. (In `inweb` BNF grammar, a literal ampersand would be written out: `AMPERSAND`. So this won't cause ambiguities.)

⟨A new production begins here 4⟩ ≡
```
my $rhs = $1;
if ($nonterminal_being_defined eq "") {
    inweb_error("BNF grammar for no obvious nonterminal\n", $bnf_grammar[$i]);
}
$no_productions_for{$nonterminal_being_defined}++;
$productions_for{$nonterminal_being_defined} .= $rhs."&";
next;
```
This code is used in §2.

§**5.** A grammar line beginning with an ellipsis marks (a) a continuation of the previous line, and also (b) a place to weave a line-break in the final TEX output – which is why we combine such a line with its predecessor, but leave the ellipsis token in.

⟨The previous production continues onto this line 5⟩ ≡
```
my $more_rhs = $1;
if ($nonterminal_being_defined eq "") {
    inweb_error("BNF grammar for no obvious nonterminal\n", $bnf_grammar[$i]);
}
my $existing_rhs = $productions_for{$nonterminal_being_defined};
$existing_rhs =~ s/\&$//;
$productions_for{$nonterminal_being_defined} = $existing_rhs." ... ".$more_rhs."&";
```
This code is used in §2.

§**6.** The scan is now complete, so we know all of the nonterminals. We do a little judicious respacing of the productions, to make square and curly braces tokens in their own rights (a convenience for later), and we strip out white space. We also initialise three more hashes:

(a) `$bnf_dependencies{$nt}` is a list of the other non-terminals which occur in productions leading to `$nt` – in effect, the things whose definitions we need to know in order to understand the definition of `$nt`.
(b) `$bnf_depth{$nt}` is used in weaving to print grammar in a hierarchical way. It will eventually be a positive integer, 1, 2, 3, ..., but will be 0 for nonterminals whose depth hasn't been determined. The idea is that if X depends on Y, and X is important enough to be worth giving a hierarchical display, then Y will have depth higher than X's.
(c) `$bnf_follows{$nt}` is either blank, or else the name of a nonterminal to which `$nt` "belongs" – meaning that it will be printed in the hierarchy underneath this owner. To a human reader, `$nt` follows this owner, but is indented further rightwards.

⟨Initialise the nonterminals discovered 6⟩ ≡

```
my $nt;
foreach $nt (sort keys %productions) {
    $bnf_dependencies{$nt} = "";
    $bnf_depth{$nt} = 0;
    $bnf_follows{$nt} = "";
    $productions_for{$nt} =~ s/\[/ \[ /g;
    $productions_for{$nt} =~ s/\]/ \] /g;
    $productions_for{$nt} =~ s/\{/ \{ /g;
    $productions_for{$nt} =~ s/\}/ \} /g;
    $productions_for{$nt} =~ s/\s+/ /g;
}
```

This code is used in §1.

§**7.** This calculates the `$bnf_dependencies{$nt}` hash (see above).

⟨Work out which nonterminals depend on which other ones 7⟩ ≡

```
my $nt;
foreach $nt (sort keys %productions) {
    my $lines = $productions_for{$nt};
    while ($lines =~ m/^(.*?)\&(.*)$/) {
        my $line = $1; $lines = $2;
        while ($line =~ m/^\s*(\S+)(.*?)$/) {
            my $tok = $1;
            $line = $2;
            if ($tok =~ m/^\<(.*)\>$/) {
                $tok = $1;
                if (exists $production_tex{$tok}) {
                    $bnf_dependencies{$tok} .= $nt . ",";
                }
            }
        }
    }
}
```

This code is used in §1.

§**8.** Some nonterminals are worth making a fuss over, and printing with little hierarchical grammars of their own. (This helps us to give structure to what would otherwise be a shapeless and difficult to read mass of productions.) Such nonterminals are called "fundamental" by `inweb`. The following is, pretty shamelessly, a list of the fundamental nonterminals in Inform.

⟨Elect certain nonterminals as being fundamental 8⟩ ≡

```
foreach $nt (sort keys %productions) {
    if (($bnf_dependencies{$nt} eq "")
        || ($nt =~ m/\-sentence$/)
        || ($nt eq "and")
        || ($nt eq "or")
        || ($nt eq "and-or")
        || ($nt eq "paragraph")
        || ($nt eq "sentence")
        || ($nt eq "condition")
        || ($nt eq "action-pattern")
        || ($nt eq "literal-value")
        || ($nt eq "type-expression")
        || ($nt eq "physical-description")
        || ($nt eq "value")) {
        $bnf_depth{$nt} = 1;
        $bnf_follows{$nt} = "";
    }
}
```

This code is used in §1.

§**9.** So at this point fundamental nonterminals have depth 1, while all other nonterminals have depth 0. The following decides, not very efficiently, which of the un-fundamental nonterminals need to be given increased depth so as to place them below their fundamental owners in the hierarchy.

⟨Make the fundamental nonterminals the roots of trees of those depending on them 9⟩ ≡

```
my $pass;
for ($pass = 1; $pass <= 2; $pass++) {
    my $changed = 1;
    while ($changed == 1) {
        $changed = 0;
        my $nt;
        foreach $nt (sort keys %productions) {
            if ($bnf_depth{$nt} == 0) {
                $scan = $bnf_dependencies{$nt};
                $min_depth = 10000000; $unfound = 0;
                while ($scan =~ m/^(.*?)\,(.*)$/) {
                    $scan = $2;
                    $used = $1;
                    if ($bnf_depth{$used} > 0) {
                        if ($min_depth > $bnf_depth{$used}) {
                            $min_depth = $bnf_depth{$used};
                            $min_depth_follows = $used;
                        }
                    } else { if ($pass == 1) { $unfound = 1; } }
                }
                if (($unfound == 0) && ($min_depth < 100000)) {
                    $bnf_depth{$nt} = $min_depth+1;
```

```
                    $bnf_follows{$nt} = $min_depth_follows;
                    $changed = 1;
                }
            }
        }
    }
}
```

This code is used in §1.

§**10.** Perhaps there were no fundamental nonterminals in the first place; or perhaps there were a few, but that they didn't cover the entire language. Either way, there could be nonterminals still having depth 0. We'll promote them to depth 1, so that they have a sensible margin setting when we come to weave.

⟨Any nonterminals remaining are placed at the top level 10⟩ ≡

```
my $nt;
foreach $nt (sort keys %productions) {
    if ($bnf_depth{$nt} == 0) {
        $bnf_depth{$nt} = 1;
    }
}
```

This code is used in §1.

§**11. Sequencing the output.** A large grammar with many nonterminals can be very difficult to read if listed in a jumbled-up order, so the following provides the ability to control how the thing comes out: we can output certain nonterminals of our choice first, and then ultimately follow up with the rest.

```
sub weave_bnf_grammar {
    my $toshow = $_[0];
    if ($bnf_already_parsed == 0) { parse_bnf_grammar(); $bnf_already_parsed = 1; }
    ⟨Handle a comma-separated list of commands 12⟩;
    ⟨Handle a command to show nonterminals including given text 13⟩;
    ⟨Handle a command to show a specific named nonterminal 14⟩;
    ⟨Handle a command to show all remaining nonterminals 15⟩;
    ⟨Finish up and check for errors 16⟩;
}
```

§**12.** A comma-separated list of commands is acted on from left to right:

⟨Handle a comma-separated list of commands 12⟩ ≡

```
if ($toshow =~ m/^\: (.*?)\,\s*(.*)$/) {
    my $a = $1;
    my $b = $2;
    weave_bnf_grammar(": ".$a);
    weave_bnf_grammar(": ".$b);
    return;
}
```

This code is used in §11.

§13. A dash indicates a partial match with the production name is enough. (For instance, to pick up every nonterminal with a name ending "-constant".)

⟨Handle a command to show nonterminals including given text 13⟩ ≡

```
if ($toshow =~ m/^\:\s*\-(.*)$/) {
    $tomatch = $1;
    foreach $nt (sort keys %productions) {
        if ($nt =~ m/\-$tomatch$/) {
            weave_nonterminal($nt);
        }
    }
    return;
}
```

This code is used in §11.

§14.

⟨Handle a command to show a specific named nonterminal 14⟩ ≡

```
if ($toshow =~ m/^\:\s*(.*)$/) {
    weave_nonterminal($1);
    return;
}
```

This code is used in §11.

§15.

⟨Handle a command to show all remaining nonterminals 15⟩ ≡

```
weave_nonterminal("value");
my $nt;
foreach $nt (sort keys %productions) {
    if ($bnf_depth{$nt} == 1) {
        weave_nonterminal($nt);
    }
}
```

This code is used in §11.

§16.

⟨Finish up and check for errors 16⟩ ≡

```
foreach $nt (sort keys %productions) {
    if ($production_done{$nt} == 0) {
        print "Error in weave of BNF grammar: omitted ", $production_tex{$nt}, "\n";
        print $nt, ": ", $bnf_depth{$nt}, " ", $bnf_follows{$nt}, " ",
            $bnf_dependencies{$nt}, "\n";
    }
}
foreach $nt (sort keys %unknown_productions) {
    print WEAVEOUT "\\medskip{\\bf Unknown production:} \\nonterminal{", $nt, "}\\par\n";
}
```

This code is used in §11.

§**17. Weaving a single production.** In other words, writing out the necessary TEX to display the grammar matching a single concept. Following each production are the ones used first in the course of defining it, and so on.

```
sub weave_nonterminal {
    my $nt = $_[0];
    if ($production_done{$nt} == 1) { return; }
    if ($bnf_depth{$nt} == 1) {
        print WEAVEOUT "\\smallbreak\\hrule\\smallbreak\n";
    } else {
        print WEAVEOUT "\\smallbreak\n";
    }
    ⟨Weave the actual nonterminal 19⟩;
    ⟨Weave grammars for any nonterminals in the hierarchy below this one 18⟩;
    $production_done{$nt} = 1;
}
```

§**18.**

⟨Weave grammars for any nonterminals in the hierarchy below this one 18⟩ ≡

```
    my $dep;
    foreach $dep (sort keys %productions) {
        if ($bnf_follows{$dep} eq $nt) {
            weave_nonterminal($dep);
        }
    }
```

This code is used in §17.

§**19.** There are various hacks here which are entirely to make the Inform grammar look prettier; I do not much repent.

⟨Weave the actual nonterminal 19⟩ ≡

```
    bnf_indent($bnf_depth{$nt} - 1);
    print WEAVEOUT $production_tex{$nt}, "\n";
    $lines = $productions_for{$nt};
    $joined_lines = 0;
    if ($nt eq "debugging-aspect") { $joined_lines = 1; $join_count = 3; }
    if ($nt eq "determiner") { $joined_lines = 1; $join_count = 2; }
    $lc = 0; $lcm = $join_count - 1;
    while ($lines =~ m/^(.*?)\&(.*)$/) {
        $lc++; $line = $1; $lines = $2;
        ⟨Get to the correct margin position for the production 20⟩;
        ⟨Weave out the tokens of the production 21⟩;
        print WEAVEOUT "\n";
    }
```

This code is used in §17.

§**20.**

⟨Get to the correct margin position for the production 20⟩ ≡
```
if ($joined_lines == 1) {
    $lcm++; if ($lcm == $join_count) { $lcm = 0; }
    if ($lc == 1) {
        print WEAVEOUT "\\par";
        bnf_indent($bnf_depth{$nt} - 1);
        print WEAVEOUT "\\qquad ::\$=\$ ";
    } else {
        if ($lcm == 0) {
            print WEAVEOUT "\\par";
            bnf_indent($bnf_depth{$nt} - 1);
            print WEAVEOUT "\\qquad \\phantom{::\$=\$} \\| ";
        } else {
            print WEAVEOUT "\\| ";
        }
    }
} else {
    print WEAVEOUT "\\par";
    bnf_indent($bnf_depth{$nt} - 1);
    print WEAVEOUT "\\qquad ::\$=\$ ";
}
```
This code is used in §19.

§**21.** The term "token" is used loosely here, and not in its technical context free grammar sense. A production at this point is a sequence of textual globs divided by a single space, such as this:

```
COMMA while <condition> [ SEMICOLON ]
```

The tokens are simply these globs, so here there are six tokens. `<condition>` here is a nonterminal.

⟨Weave out the tokens of the production 21⟩ ≡
```
$last_tok = "";
while ($line =~ m/^\s*(\S+)(.*?)$/) {
    $tok = $1;
    $line = $2;
    ⟨Insert inter-token space where appropriate 22⟩;
    $last_tok = $tok;
    ⟨Weave an ellipsis token 23⟩;
    ⟨Weave a nonterminal token 24⟩;
    ⟨Weave a literal text token 25⟩;
}
```
This code is used in §19.

§**22.**

⟨Insert inter-token space where appropriate 22⟩ ≡
```
if (($last_tok ne "[") && ($last_tok ne "{") && ($last_tok ne "")
    && ($tok ne "]") && ($tok ne "}")) {
    print WEAVEOUT " ";
}
```
This code is used in §21.

§**23.** An ellipsis is a continuation onto the next line:

⟨Weave an ellipsis token 23⟩ ≡
```
if ($tok eq "...") {
    print WEAVEOUT "\\par";
    bnf_indent($bnf_depth{$nt} - 1);
    print WEAVEOUT "\\qquad \\phantom{::\$=\$} ";
    next;
}
```

This code is used in §21.

§**24.**

⟨Weave a nonterminal token 24⟩ ≡
```
if ($tok =~ m/^\<(.*)\>$/) {
    $tok = $1;
    if (exists $production_tex{$tok}) {
        print WEAVEOUT $production_tex{$tok};
    } else {
        if ($tok =~ m/^(.*)-name$/) {
            print WEAVEOUT "{\\rm " . $1 . "}";
        } else {
            if ($tok =~ m/^(.*)-usage$/) {
                print WEAVEOUT "{\\it " . $tok . "}";
            } else {
                print WEAVEOUT "\\nonterminal{\\rm !!!!" . $tok . "!!!!}";
                $unknown_productions{$tok}++;
            }
        }
    }
    next;
}
```

This code is used in §21.

§**25.** And otherwise we have literal text in boldface, though the fact that curly braces have to be rendered in math mode complicates things a little.

⟨Weave a literal text token 25⟩ ≡
```
if ($tok eq "{") { $tok = "\$\\{\$"; }
if ($tok eq "}") { $tok = "\$\\}\$"; }
if ((length($tok) > 2) &&
    ($tok =~ m/^[A-Z\-][A-Z\-0-9]+$/)) {
    print WEAVEOUT "{\\eightbf ", $tok, "}";
    $lexical_productions{$tok}++;
} else {
    print WEAVEOUT "{\\bf ", $tok, "}";
}
```

This code is used in §21.

§**26.** Indentation, the crude way:

```
sub bnf_indent {
    my $dep = $_[0];
    my $d;
    for ($d=1; $d<=$dep; $d++) {
        print WEAVEOUT "\\quad";
    }
}
```

The Tangler 3/tang

Purpose

To write a portion of the code in a compilable form.

§**1. The Master Tangler.** Here's what has happened so far, on a `-tangle` run of `inweb`: on any other sort of run, of course, we would never be in this section of code. The web was read completely into memory, and then fully parsed, with all of the arrays and hashes populated. Program Control then sent us straight here for the tangling to begin:

```
sub tangle_source {
    my $target = $_[0];
    my $dest_file = $_[1];
    my $i;
    if ($target > 0) { print "Tangling independent target $target to $dest_file\n"; }
    language_set($tangle_target_language[$target]);
    if (language_tangles() == 0) {                                  for documentation webs, for instance
        inweb_fatal_error("can't tangle material in the language '".
            $bibliographic_data{"Language"}."'");
    }
    open(TANGLEOUT, ">".$dest_file) or die "inweb: can't open tangle file '$dest_file' for output";
    print TANGLEOUT language_shebang();
    if ($web_language != $I7_LANGUAGE) {
        print TANGLEOUT language_comment("Tangled output generated by inweb: do not edit");
    }
    for ($i=0; $i<$no_lines; $i++) {
        $line_suppress_tangling[$i] = 0;
        if ($line_category[$i] == $COMMAND_LCAT) {
            $line_suppress_tangling[$i] = 1;
        }
        if ($section_tangle_target[$line_sec[$i]] != $target) {
            $line_suppress_tangling[$i] = 1;
        }
    }
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE))
        ⟨Pull forward the inclusion of ANSI C libraries 2⟩;
    ⟨Tangle all the constant definitions in section order 3⟩;
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE))
        ⟨Tangle the structure definitions 4⟩;
    ⟨Tangle the paragraphs appearing above the bar in each section in turn 6⟩;
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE))
        ⟨Tangle predeclarations of C functions 7⟩;
    tangle_code(0, $no_lines-1, 0);
    close(TANGLEOUT);
}
```

§**2.** For C only: we need to include ANSI library files before declaring structures because otherwise `FILE` and suchlike types won't exist yet. It might seem reasonable to include all of the `#include` files right now, but that defeats any conditional compilation, which Inform (for instance) needs in order to make platform-specific details to handle directories without POSIX in Windows. So we'll just advance the common ANSI inclusions.

⟨Pull forward the inclusion of ANSI C libraries 2⟩ ≡

```
my $i;
for ($i=0; $i<$no_lines; $i++) {
    if ($line_text_raw[$i] =~ m/^\s*\#include\s+\<(.*?)\.h\>\s*$/) {
        $clib = $1;
        if (($clib eq "stdio") ||
            ($clib eq "ctype") ||
            ($clib eq "math") ||
            ($clib eq "stdarg") ||
            ($clib eq "stdlib") ||
            ($clib eq "string") ||
            ($clib eq "time")) {
            $line_now_tangled[$i] = 1;
            print TANGLEOUT $line_text_raw[$i], "\n";
        }
    }
}
```

This code is used in §1.

§**3.** This is the result of all those `@d` definitions, and the tricky part is that tangling them depends on how constants are defined in the current programming language; we also want to expand `[[Author]]`-like substitutions in them.

⟨Tangle all the constant definitions in section order 3⟩ ≡

```
my $i;
for ($i=0; $i<$no_lines; $i++) {
    if (($line_category[$i] == $BEGIN_DEFINITION_LCAT) &&
        ($target == $section_tangle_target[$line_sec[$i]])) {
        my $definition_fragment = $line_operand_2[$i];
        $line_suppress_tangling[$i] = 1;
        language_start_definition($line_operand[$i],
            expand_double_squares($definition_fragment, 0));
        $i++;
        while (($i<$no_lines) && ($line_category[$i] == $CONT_DEFINITION_LCAT)) {
            language_prolong_definition(expand_double_squares($line_text_raw[$i], 0));
            $line_suppress_tangling[$i] = 1;
            $i++;
        }
        language_end_definition();
        $i--;
    }
}
```

This code is used in §1.

§**4.** Structures must be declared in order such that if A incorporates at least one copy of B then B must be declared before A (since C compilers require this). To track this, we have already formed structures into a directed acyclic graph (DAG): a conceptual picture in which each structure forms a "vertex" and each incorporation of B within A makes an "edge" from A to B. (If it contains multiple copies of B, there will be multiple such edges, so properly speaking this is a multigraph.) As the following runs,

(a) The current vertices are those structure names `S` for which `$structure_declaration_tangled{S}` is still 0.
(b) The current edges outward from `S` are stored in `$structure_incorporates{S}` in the form of a list of destination vertices `R`, `T`, ..., as a string `->R->T...`. Each of these must be a current vertex.
(c) At any given time, either the DAG is empty or there is at least one vertex with no outward edges: for if not, then start from a vertex v_1; it must have an outward edge to $v_2 \neq v_1$ (for otherwise there a structure contains itself, which is impossible); v_2 must also have an outward edge to v_3, and so on. If it is never the case that $v_i = v_j$ for any $i \neq j$ then there are an infinite number of structures, which is impossible. So (suppose $i < j$) there exists a loop $v_i, v_{i+1}, v_{i+2}, ..., v_j, v_i$ showing that structure i incorporates an embedded copy of itself, which is again impossible.

The proof (c) gives us a fairly efficient way to find a vertex with no outward edges, but we won't do it that way: the number of vertices is small and the number of edges always likely to be small compared to that, so it is more efficient to sweep through the DAG repeatedly removing all vertices without outward edges.

⟨Tangle the structure definitions 4⟩ ≡

```
my $no_structs_left = 0;
foreach $struc (keys %structures) {
    my $j;
    $no_structs_left++;
    $structure_declaration_tangled{$struc} = 0;
}
while ($no_structs_left > 0) {
    my $changes_made = 0;
    foreach $struc (sort keys %structures) {
        if (($structure_incorporates{$struc} eq "") &&                    no outward edges
            ($structure_declaration_tangled{$struc} == 0))                but still in graph
            ⟨The structure is a vertex with no outward edges, so declare it now 5⟩;
    }
    if ($changes_made == 0) {
        inweb_error("cyclic error in structure dependencies");
        foreach $struc (sort keys %structures) {
            if ($structure_declaration_tangled{$struc} == 0) {
                inweb_error($struc." needs prior declaration of ".
                    $structure_incorporates{$struc});
            }
        }
        exit(1);
    }
}
```

This code is used in §1.

§**5.** Here we have a vertex in our DAG which has no outward edges, so we can make its declaration and then (i) delete all inward edges and (ii) delete the vertex itself, thus leaving the DAG strictly smaller and still a well-formed DAG.

⟨The structure is a vertex with no outward edges, so declare it now 5⟩ ≡

Tangle the `typedef` *declaration lines...*

```
my $j;
for ($j=$structure_declaration_line{$struc}; $j<=$structure_declaration_end{$struc}; $j++) {
    print TANGLEOUT $line_text_raw[$j], "\n";
    $line_suppress_tangling[$j] = 1;
}
```

...and keep the counts accurate:

```
$no_structs_left--; $changes_made++;
```

(i) Strike this structure name out of the dependency lists of all others...

```
foreach $n2 (sort keys %structures) {
    $structure_incorporates{$n2} =~ s/\-\>$struc\b//g;
}
```

(ii) Remove this structure from further consideration

```
$structure_declaration_tangled{$struc} = 1;
```

This code is used in §4.

§**6.** We have no need to loop through the sections; the lines already appear in the line list in section order. We let through everything between a `@Definitions:` heading and the bar into the tangler:

⟨Tangle the paragraphs appearing above the bar in each section in turn 6⟩ ≡

```
my $i;
for ($i=0; $i<$no_lines; $i++) {
    if ($line_category[$i] == $DEFINITIONS_LCAT) {
        $i++;
        my $md_from = $i;
        while (($i<$no_lines) && ($line_category[$i] != $BAR_LCAT)) { $i++; }
        my $md_to = $i;
        tangle_code($md_from, $md_to, 0);
        for ($i=$md_from; $i<$md_to; $i++) { $line_suppress_tangling[$i] = 1; }
        $i--;
    }
}
```

This code is used in §1.

§**7.** In a rather unnecessary way, we choose to predeclare the functions in a human-readable order, for the benefit of anyone checking the tangled source: so they are neatly arranged by chapter and section. It does mean that the running time of the following is $O(C \times S \times F)$, where C, S and F are the number of chapters, sections and functions respectively, and this is in principle cubic in the size of the web. In practice the coefficient is small, of course, and even on the largest webs yet constructed there's no appreciable delay.

⟨Tangle predeclarations of C functions 7⟩ ≡

```
my $chn;
for ($chn=0; $chn<$no_chapters; $chn++) {
    print TANGLEOUT "\n",
        language_comment("Functions in chapter ".$chapter_sigil[$chn]), "\n";
    my $sn;
    for ($sn=0; $sn<$no_sections; $sn++) {
        if ($section_chap[$sn] != $chn) { next; }
        print TANGLEOUT "\n", language_comment("Section ".$section_sigil[$sn]);
        foreach $fname (sort keys %functions_line) {
            $j = $functions_line{$fname};
            if ($line_sec[$j] != $sn) { next; }
            $spc = $functions_arglist{$fname};
            $spc =~ s/\,\s*/\,\n\t\t/g;
            print TANGLEOUT $functions_return_type{$fname}, " ", $fname, "(", $spc, ");\n";
            if (exists $dot_i6_identifiers{$fname}) {
                print TANGLEOUT language_comment("accessed via .i6 metalanguage");
            } else {
                print TANGLEOUT $functions_usage_verbosely_described{$fname};
            }
        }
    }
}
```

This code is used in §1.

§**8. The real work.** All of the above was merely a series of teasing preliminaries: here's the real thing, the single recursive routine which tangles the code.

```
sub tangle_code {
    my $from = $_[0];                                   Start line
    my $to = $_[1];                                     End line
    my $permit_macro = $_[2];                           Allow CWEB macro definition bodies through?
    my $tline;
    my $contiguous = 0;
    for ($tline=$from; $tline<=$to; $tline++) {
        if (($permit_macro == 0) && ($line_occurs_in_CWEB_macro_definition[$tline] == 1)) {
            $contiguous = 0; next;
        }
        if ($line_suppress_tangling[$tline] == 1) { $contiguous = 0; next; }
        if (not(($line_category[$tline] == $CODE_BODY_LCAT))) { $contiguous = 0; next; }
        if ($line_text_raw[$tline] =~ m/^\s*\@c/) { $contiguous = 0; next; }
        if ($line_text_raw[$tline] =~ m/^(.*?)\@\<(.*?)\@\>\s*(.*?)$/)
            ⟨Expand the CWEB macro used on this line 10⟩;
        ⟨Place a comment indicating original source file position if necessary 9⟩;
        $expanded = expand_double_squares($line_text_raw[$tline], 1);
        if ($expanded =~ m/^\?\:\s*(.*)$/) {
            inweb_error_at_program_line("C-for-Inform error ($1)", $tline);
        }
        tangle_out($expanded);
    }
}
```

§**9.** If order for C compilers to report C syntax errors on the correct line, despite rearranging by automatic tools, C conventionally recognises the preprocessor directive `#line` to tell it that a contiguous extract follows from the given file; we generate this automatically. (In its usual zany way, Perl recognises the exact same syntax, thus in principle overloading its comment notation `#`.)

⟨Place a comment indicating original source file position if necessary 9⟩ ≡

```
    if (($contiguous == 0) &&
        (($web_language == $C_LANGUAGE) ||
            ($web_language == $C_FOR_INFORM_LANGUAGE) ||
            ($web_language == $PERL_LANGUAGE))) {
        $contiguous = 1;
        tangle_out("#line ".$line_source_file_line[$tline].
            " \"".$section_pathname_relative_to_web[$line_sec[$tline]]."\"");
    }
```

This code is used in §8,10,8,10,8.

§**10.** The tricky point here is that CWEB is stream-of-characters rather than line-oriented, which makes it awkward declaring to the C preprocessor exactly where the material came from without inserting a lot of line breaks: because the C preprocessor believes each line in the final C code must have a single file and line as its point of origin, and that just isn't true if the original CWEB code read, say,

```
if (black == white) { @<Observe the universe's predicament@>; exit(1); }
```

Happily C is pretty relaxed about treating newlines as just more white space, so the rather spurious newlines cause no trouble.

Because we work on lines, not a stream, it isn't very convenient for us to make more than one macro substitution on the same line. We could fix this if we wanted to, but in practice it just doesn't naturally arise, because they tend to have rather long descriptive names.

A point of difference with CWEB is that we automatically encase the expanded macro in braces, for C or Perl, at any rate. This is convenient since it allows us to treat a CWEB macro as a single statement; and also usefully prevents us from some of the abusive things done in *TEX: The Program*, where Knuth uses CWEB macros to combine bits of top-level global variable definitions and other such outside-of-functions hackery (to be fair, Knuth was forced into this by the deficiencies of Pascal as it then stood).

⟨Expand the CWEB macro used on this line 10⟩ ≡

```
my $prologue = $1; local $name = $2; local $residue = $3;
if ($residue =~ m/^(.*?)\@\<(.*?)\@\>(.*)$/) {
    inweb_error_at_program_line("only one <...> macro can be used on any single line",
        $tline);
}
⟨Place a comment indicating original source file position if necessary 9⟩;
tangle_out(expand_double_squares($prologue, 1));
if (not(exists $cweb_macros_start{$name})) {
    inweb_error_at_program_line("no such <...> macro as '".$name."'", $tline);
} else {
    if (($web_language == $C_LANGUAGE) ||
        ($web_language == $C_FOR_INFORM_LANGUAGE) ||
        ($web_language == $PERL_LANGUAGE)) { print TANGLEOUT " { "; }
    tangle_out(expand_double_squares($cweb_macros_surplus_bit{$name}, 1));
    tangle_code($cweb_macros_start{$name}, $cweb_macros_end{$name}, 1);
    if (($web_language == $C_LANGUAGE) ||
        ($web_language == $C_FOR_INFORM_LANGUAGE) ||
        ($web_language == $PERL_LANGUAGE)) { print TANGLEOUT " } "; }
}
$contiguous = 0;
print TANGLEOUT "\n";
if ($residue ne "") {
    ⟨Place a comment indicating original source file position if necessary 9⟩;
    tangle_out(expand_double_squares($residue, 1));
}
next;
```

This code is used in §8.

§**11.**

```
sub tangle_out {
    my $line_of_code = $_[0];
    if ($web_language == $C_FOR_INFORM_LANGUAGE) {
        $line_of_code =~ s/\:\:/__/g;
    }
    print TANGLEOUT $line_of_code, "\n";
}
```

§**12. Substituting bibliographic data.** We expand material in double square brackets if either

(a) we recognise it as the name of a piece of bibliographic data, or
(b) it is one of the C-for-Inform language extensions and in a code context where this would make sense.

Otherwise, we leave well alone.

```
sub expand_double_squares {
    my $line = $_[0];
    my $with_extensions = $_[1];
    my $fore;
    my $aft;
    my $comm;
    my $expanded;
    my $safety_check = 0;
    my $so_far = "";
    while ($line =~ m/^(.+?)\[\[(.*?)\]\](.*)$/) {
        $fore = $1; $line = $3; $comm = $2;
        $so_far .= $fore;
        if (exists ($bibliographic_data{$comm})) {
            $so_far .= $bibliographic_data{$comm};
        } elsif (($with_extensions == 1) && ($web_language == $C_FOR_INFORM_LANGUAGE)) {
            $expanded = expand_double_squared_command($comm);
            if ($expanded =~ m/^\?\: /) { return $expanded; }
            $so_far .= $expanded;
            if ($safety_check++ == 100) {
                return '?: expander gone into infinite regress';
            }
        } else {
            $so_far .= '[['.$comm.']]';
        }
    }
    $so_far .= $line;
    return $so_far;
}
```

Programming Languages 3/plan

Purpose

To characterise the relevant differences in behaviour between the various programming languages we support: for instance, how comments are represented.

Definitions

¶1. At any given time, there's a current programming language, and it is always one of the following possibilities:

```
define $C_LANGUAGE 1                          C or C++
define $C_FOR_INFORM_LANGUAGE 2               Ditto, but with NI extensions
define $PERL_LANGUAGE 3                       Perl
define $I6_LANGUAGE 4                         Inform 6
define $I7_LANGUAGE 5                         Inform 7 extension
define $PLAIN_LANGUAGE 6                      Plain text
define $NO_LANGUAGE 100                       Nothing to tangle, by fiat

$web_language = $C_LANGUAGE;                  The default
$tangled_extension = ".c";
```

---

§**1.** Setting the current language from a textual form:

```
sub language_set {
    my $lname = $_[0];
    $web_language = -1;
    if ($lname eq "C") { $web_language = $C_LANGUAGE; $tangled_extension = ".c"; }
    if ($lname eq "C++") { $web_language = $C_LANGUAGE; $tangled_extension = ".cpp"; }
    if ($lname eq "C for Inform") {
        $web_language = $C_FOR_INFORM_LANGUAGE; $tangled_extension = ".c";
        ⟨Flag some identifiers as unusual in this form of C 2⟩;
    }
    if ($lname eq "Perl") { $web_language = $PERL_LANGUAGE;  $tangled_extension = ".pl"; }
    if ($lname eq "Inform 6") { $web_language = $I6_LANGUAGE;  $tangled_extension = ".i6"; }
    if ($lname eq "Inform 7") { $web_language = $I7_LANGUAGE;  $tangled_extension = ".i7x"; }
    if ($lname eq "Plain Text") { $web_language = $PLAIN_LANGUAGE;  $tangled_extension = ".txt"; }
    if ($lname eq "None") { $web_language = $NO_LANGUAGE;  $tangled_extension = ""; }
    if ($web_language == -1) {
        inweb_fatal_error("unsupported programming language ".$lname);
    }
}
```

§**2.** The term "blacklisting" is a little melodramatic. A function which has been blacklisted is allowed to be defined more than once in the source code; Inform needs this for `isdigit`, of all things, because the Windows library's implementation is deficient.

Similarly, a member name which has been blacklisted is allowed to be present in more than one structure. This of course is perfectly legal in C, but the Inform source code tries to avoid doing it, since it leads to confusion and makes it harder to nail down where the structure is used. A few exemptions are made for members deliberately used in common across wide ranges of structures.

⟨Flag some identifiers as unusual in this form of C 2⟩ ≡

```
    $blacklisted_functions{"isdigit"} = 1;
    $blacklisted_members{"word_ref1"} = 1;
    $blacklisted_members{"word_ref2"} = 1;
    $blacklisted_members{"next"} = 1;
    $blacklisted_members{"down"} = 1;
    $blacklisted_members{"allocation_id"} = 1;
    $members_allowed_to_be_unused{"handling_routine"} = 1;
```

This code is used in §1.

§**3.** The file extension generally used by files of code in this language:

```
sub language_file_extension { return $tangled_extension; }
```

§**4.** Any compulsory heading that must occur on line 1, really:

```
sub language_shebang {
    my $text = $_[0];
    if ($web_language == $PERL_LANGUAGE) { return "#!/usr/bin/perl\n\n"; }
    return "";
}
```

§**5.** Now a routine to write a comment. Languages without comment should write nothing.

```
sub language_comment {
    my $text = $_[0];
    if ($web_language == $C_LANGUAGE) { return "/* ".$text." */\n"; }
    if ($web_language == $C_FOR_INFORM_LANGUAGE) { return "/* ".$text." */\n"; }
    if ($web_language == $PERL_LANGUAGE) { return "# ".$text."\n"; }
    if ($web_language == $I6_LANGUAGE) { return "! ".$text."\n"; }
    if ($web_language == $I7_LANGUAGE) { return "[".$text."]\n"; }
    return "";
}
```

§**6.** And we need to spot and so on... comments:

```
sub language_and_so_on {
    my $text = $_[0];
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        if ($text =~ m/^\s*\/\* ...and so on... \*\/\s*/) { return 1; }
    }
    if ($web_language == $I6_LANGUAGE) {
        if ($text =~ m/^\s*\!\s*...and so on...\s*$/) { return 1; }
    }
    if ($web_language == $I7_LANGUAGE) {
        if ($text =~ m/^\s*\[\s*...and so on...\s*\]\s*$/) { return 1; }
    }
    return 0;
}
```

§**7.** And in general to see when a line ends in a comment:

```
sub line_ends_with_comment {
    my $matter = $_[0];
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        if ($matter =~ m/^(.*)\/\*\s*(.*?)\s*\*\/\s*$/) {
            $part_before_comment = $1; $part_within_comment = $2; return 1;
        }
    }
    if ($web_language == $PERL_LANGUAGE) {
        if ($matter =~ m/^\#\s+(.*?)\s*$/) {
            $part_before_comment = ""; $part_within_comment = $1; return 1;
        }
        if ($matter =~ m/^(.*)\s+\#\s+(.*?)\s*$/) {
            $part_before_comment = $1; $part_within_comment = $2; return 1;
        }
    }
    return 0;
}
```

§**8.** To place page breaks strategically within code:

```
sub language_pagebreak_comment {
    my $l = $_[0];
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        if ($l =~ m/^\s*\/\* PAGEBREAK \*\/\s*$/) { return 1; }
    }
    if ($web_language == $PERL_LANGUAGE) {
        if ($l =~ m/^\s*\#\s+PAGEBREAK\s*$/) { return 1; }
    }
    if ($web_language == $I6_LANGUAGE) {
        if ($l =~ m/^\s*\!\s+PAGEBREAK\s*$/) { return 1; }
    }
    if ($web_language == $I7_LANGUAGE) {
        if ($l =~ m/^\s*\[\s*PAGEBREAK\s*\]\s*$/) { return 1; }
    }
    return 0;
}
```

§**9.** Does the currently selected language allow tangling? (Yes, unless this is purely a documentation project.)

```
sub language_tangles {
    if ($web_language == $NO_LANGUAGE) { return 0; }
    return 1;
}
```

§**10.** The following routines handle the `@d` escape, writing a definition of the constant named `$term` as the value given. If the value spans multiple lines, the first-line part is supplied to `language_start_definition` and then subsequent lines are fed in order to `language_prolong_definition`. At the end, `language_end_definition` is called.

```
sub language_start_definition {
    my $term = $_[0];
    my $startval = $_[1];
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        if ($web_language == $C_FOR_INFORM_LANGUAGE) { $startval =~ s/::/__/g; }
        print TANGLEOUT "#define ", $term, " ", $startval;
        return;
    }
    if ($web_language == $PERL_LANGUAGE) {
        print TANGLEOUT $term, " = ", $startval;
        return;
    }
    inweb_error("programming language $web_language does not support \@d");
}
sub language_prolong_definition {
    my $more = $_[0];
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        if ($web_language == $C_FOR_INFORM_LANGUAGE) { $more =~ s/::/__/g; }
        print TANGLEOUT "\\\n    ", $more;
        return;
```

```
    }
    inweb_error("programming language $web_language does not support multiline \@d");
}
sub language_end_definition {
    if (($web_language == $C_LANGUAGE) || ($web_language == $C_FOR_INFORM_LANGUAGE)) {
        print TANGLEOUT "\n";
    }
    if ($web_language == $PERL_LANGUAGE) {
        print TANGLEOUT "\n;\n";
    }
}
```

Purpose

To provide a convenient extension to C syntax for the C-for-Inform language, which is likely never to be used for any program other than the Inform 7 compiler.

§**1. Unit testing.** The following provides a simple test of the extended syntax, and is run by using the `-test-extensions` switch at the command line. It gives some examples of what we'll be compiling below, anyway.

```
sub full_test_double_squares {
    test_ds("[[w1, w2 == ...]]");
    test_ds("[[w1, w2 == golly]]");
    test_ds("[[word w1 == zowie]]");
    test_ds("[[word w1 == wow/huh/zowie]]");
    test_ds("[[w1, w2 == by the pool]]");
    test_ds("[[w1, w2 == moreover ***]]");
    test_ds("[[w1, w2 == *** by the pool]]");
    test_ds("[[w1, w2 == by the pool ***]]");
    test_ds("[[w1, w2 == golly gosh/moses mrs smith]]");
    test_ds("[[w1, w2 == so mr ### we meet again]]");
    test_ds("[[w1, w2 == hello there ...]]");
    test_ds("[[w1, w2 == ... the memory of water]]");
    test_ds("[[w1, w2 <-- something]]");
    test_ds("[[w1, w2 == ... nantucket ...]]");
    test_ds("[[w1, w2 == ... when ... : w]]");
    test_ds("[[w1, w2 == hot porridge ... when ... : w]]");
    test_ds("[[w1, w2 == ... when ... breakfast is served]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served : w]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served : w, x]]");
    test_ds("[[w1, w2 == ... when ... and ... breakfast is served : w, x, y]]");
    test_ds("[[w1, w2 == ... when ... breakfast is served : w, z, x]]");
    test_ds("[[w1, w2 == bacon ... when ... breakfast is served : w]]");
    test_ds("[[w1, w2 == and now ... --> now1, now2]]");
    test_ds("[[w1, w2 == ... until then --> then1, then2]]");
    test_ds("[[w1, w2 == ... when ... : w --> w1, w2 ... when1, when2]]");
    test_ds("[[w1, w2 == ... when ... : w --> when1, when2 ... w1, w2]]");
    test_ds("[[w1, w2 == ... when ... : w --> when1, w2 ... when2, w1]]");
    test_ds("[[w1, w2 == ... OPENBRACKET for ... only CLOSEBRACKET --> x1, x2 ... vm1, vm2]]");
}

sub test_ds {
    my $original = $_[0];
    print "inweb: testing expansion ", $original, "\n\n";
    print expand_double_squares(" ".$original), "\n\n";
}
```

§**2. The expander.** We are given what might, or might not, be a valid command. If we can make sense of it, we return its expansion; if not, we return an error message with a query ? in column 1.

```
sub expand_double_squared_command {
    my $comm = $_[0];
    my $operator = 0;
    my $lhs;
    my $rhs;
    my $w1;
    my $w2;
    my $single_word_flag = 0;
    ⟨Identify the command as either a test or an assignment 3⟩;
    ⟨Parse the left-hand side into C expressions for a word range 4⟩;
    if ($operator == 1) {
        if ($single_word_flag == 1) ⟨Expand a single word test 6⟩
        else ⟨Expand a word range test 7⟩;
    }
    if ($operator == 2) ⟨Expand an assignment command 5⟩;
    return '?: This should not happen';
}
```

§**3.** We split the line as RHS, operator, LHS; there are only two valid operators.

⟨Identify the command as either a test or an assignment 3⟩ ≡

```
    if ($comm =~ m/^(.*?)\s*\=\=\s*(.*?)\s*$/) {
        $operator = 1; $lhs = $1; $rhs = $2;
    } else {
        if ($comm =~ m/^(.*?)\s*\<\-\-\s*(.*?)\s*$/) {
            $operator = 2; $lhs = $1; $rhs = $2;
        }
    }
    if ($operator == 0) {
        return '?: [[ ... ]] with neither <-- nor == operators';
    }
```

This code is used in §2.

§**4.**

⟨Parse the left-hand side into C expressions for a word range 4⟩ ≡

```
    if ($lhs =~ m/^\s*(\S+?)\s*\,\s*(\S+)\s*$/) {
        $w1 = $1; $w2 = $2;
    } else {
        if ($lhs =~ m/^\s*(\S+)\s*$/) {
            $w1 = $1."->word_ref1"; $w2 = $1."->word_ref2";
        } else {
            if ($lhs =~ m/^\s*word\s+(\S+)\s*$/) {
                $w1 = $1; $w2 = $w1; $single_word_flag = 1;
            } else {
                return '?: [[ ... ]] without comma-separated word pair';
            }
        }
    }
```

This code is used in §2.

§**5.** There are three basic commands, of which one is easy:

⟨Expand an assignment command 5⟩ ≡
```
if ($single_word_flag == 1) {
    return '?: [[ ... ]] cannot assign a single word variable';
}
return $w1 . " = " . $rhs . "->word_ref1; ". $w2 . " = " . $rhs . "->word_ref2;";
```
This code is used in §2.

§**6.** And one is nearly easy:

⟨Expand a single word test 6⟩ ≡
```
if ($rhs =~ m/^\s*(\S+?)\s*$/) {
    my $list = $1;
    my $cond = '((' .$w1.'>=0) && (';
    my $ct = 1;
    while ($list =~ m/^([^\/]+)(.*)$/) {
        if ($ct > 1) { $cond .= ' || '; }
        $ct++;
        $cond .= compare_word_p($w1, 0, $1);
        $list = $2; $list =~ s/^\///;
    }
    $cond .= '))';
    return $cond;
}
return '?: [[ ... ]] cannot test a single word variable to other than a single word';
```
This code is used in §2.

§**7.** While the third is the big one. We are now comparing a word range against what may be a complicated pattern.

⟨Expand a word range test 7⟩ ≡
```
my $write_positions = "";
my $write_ranges = "";
⟨Split off optional portions of the pattern indication what positions or ranges to write 8⟩;
my $right_unanchored = 0;
my $left_unanchored = 0;
⟨Determine whether the pattern is anchored to the right or left edges 9⟩;
my $nw = 0;
⟨Split the pattern into an array of words 10⟩;
my $nr = 0;
⟨Divide this up into ranges with ellipses between 11⟩;
⟨Work out how these ranges are anchored 12⟩;
if ($nr == 0) { return "(FALSE)"; }
my $cond = "(";
⟨Work out a C condition to test the pattern 13⟩;
$cond .= ")";
if ($write_ranges ne "") { return '?: [[ ... ]] has too many write pairs of variables'; }
if ($write_positions ne "") { return '?: [[ ... ]] has too many : variables'; }
return $cond;
```
This code is used in §2.

§**8.**

⟨Split off optional portions of the pattern indication what positions or ranges to write 8⟩ ≡

```
if ($rhs =~ m/^(.*)\s*\-\-\>\s*(.*?)\s*$/) {
    $write_ranges = $2; $rhs = $1;
}
if ($rhs =~ m/^(.*)\s*\:\s*(.*?)\s*$/) {
    $rhs = $1; $write_positions = $2;
}
if (($rhs.' '.$write_positions) =~ m/\-\-\>/) { return '?: [[ ... ]] has too many -->s'; }
$rhs =~ m/^\s*(.*?)\s*$/; $rhs = $1;
$write_positions =~ m/^\s*(.*?)\s*$/; $write_positions = $1;
$write_ranges =~ m/^\s*(.*?)\s*$/; $write_ranges = $1;
```

This code is used in §7.

§**9.** If either end of the pattern is `***`, this means "match all words here right up to the edge"; we'll call the pattern "unanchored", because the word position where the match will start or finish is somewhere loose inside the word range we're matching.

⟨Determine whether the pattern is anchored to the right or left edges 9⟩ ≡

```
if ($rhs =~ m/^(.*)\s+\*\*\*\s*$/) {
    $rhs = $1;
    $right_unanchored = 1;
}
if ($rhs =~ m/^\*\*\*\s+(.*)$/) {
    $rhs = $1;
    $left_unanchored = 1;
}
```

This code is used in §7.

§**10.**

⟨Split the pattern into an array of words 10⟩ ≡

```
while ($rhs =~ m/^(\S*?)\s+(.*)$/) {
    $words[$nw++] = $1; $rhs = $2;
}
$words[$nw++] = $rhs;
```

This code is used in §7.

§**11.**

⟨Divide this up into ranges with ellipses between 11⟩ ≡

```
my $i;
my $range_start = -1;
for ($i=0; $i < $nw; $i++) {
    if ($words[$i] eq '...') {
        if ($range_start >= 0) {
            $range_from[$nr] = $range_start; $range_to[$nr++] = $i-1;
            $range_start = -1;
        }
    } else {
        if ($range_start == -1) { $range_start = $i; }
    }
}
if ($range_start >= 0) {
    $range_from[$nr] = $range_start; $range_to[$nr++] = $nw-1;
}
```

This code is used in §7.

§**12.**

⟨Work out how these ranges are anchored 12⟩ ≡

```
my $i;
for ($i=0; $i<$nr; $i++) {
    $range_length[$i] = $range_to[$i]-$range_from[$i]+1;
    $range_anchor[$i] = -1;
    $range_anchor_direction[$i] = 0;
    $range_from_fixed[$i] = 0;
    $range_to_fixed[$i] = 0;
    if (($range_from[$i] == 0) && ($left_unanchored == 0)) {
        $range_anchor[$i] = 0;
        $range_anchor_direction[$i] = 1;
        $range_from_fixed[$i] = 1;
    }
    if (($range_to[$i] == $nw-1) && ($right_unanchored == 0)) {
        $range_anchor[$i] = $nw-1;
        $range_anchor_direction[$i] = -1;
        $range_to_fixed[$i] = 1;
    }
}
```

This code is used in §7.

§**13.**

⟨Work out a C condition to test the pattern 13⟩ ≡

```
⟨Begin with a check that the word range is valid and includes enough words 14⟩;
my $i;
for ($i=0; $i<$nr; $i++) { $range_done[$i] = 0; }
$left_inset = 0;
$right_inset = 0;
$left_extent = 0;
$right_extent = 0;
⟨First check all of the ranges with fixed endpoints 15⟩;
⟨Then check all of the ranges with floating endpoints 16⟩;
for ($i=0; $i<$nr; $i++) {
    if ($range_done[$i] == 0) { return '?: [[ ... ]] has intractable range'; }
}
⟨Add always-true conditions with the side effect of writing the word ranges 17⟩;
```

This code is used in §7.

§**14.** Word ranges inside the Inform compiler are pairs of values (w_1, w_2) where either $w_1 = w_2 = -1$, meaning "no text", or $0 \le w_1 \le w_2 < N$, where N is the number of words read in by the lexer. Here we are testing a word range where w_1 is represented by the C expression in `$w1`, and similarly w_2 by `$w2`. So we first check that $w_1 \ge 0$, and then that not only is $w_2 \ge w_1$, but also that it's larger by sufficient to include all of the words we're trying to match. Thus

```
[[x1, x2 == unseeded ... grapes]]
```

will certainly fail if the word range contains fewer than three words.

⟨Begin with a check that the word range is valid and includes enough words 14⟩ ≡

```
$cond .= "(". $w1 . ">=0)";
my $need = 0;                                   minimum number of words needed
my $exact = 1;                                  is this the exact number of words we must have?
for ($i=0; $i<$nr; $i++) {
    $need = $need + $range_length[$i];
    if ($range_from_fixed[$i] == 0) { $exact = 0; }
    if ($range_to_fixed[$i] == 0) { $exact = 0; }
}
for ($i=0; $i<$nw; $i++) {
    if ($words[$i] eq '...') { $need++; }
}
$need--;
if ($exact == 1) { $op = "=="; } else { $op = ">="; }
$cond .= " && (". $w2 . $op . var_offset($w1, $need) . ")";
```

This code is used in §13.

§**15.** Suppose we have a pattern such as

```
[[x1, x2 == peeled or ... perhaps ... unpeeled mangos]]
```

This gives us three ranges to match, of lengths 2, 1, 2. We start by testing the two end ranges, because we can do so quickly – if there's going to be a match, we know exactly at what word positions they must occur. Thus, "peeled" must be at `x1`, "mangos" must be at `x2`, and so on. The reason for doing this is that it will be slower to find "perhaps", whose position is ambiguous, so we don't want to look unless the pattern otherwise matches – this enables us to reject non-matching word ranges more quickly.

⟨First check all of the ranges with fixed endpoints 15⟩ ≡

```
my $j;
for ($i=0; $i<$nr; $i++) {
    if ($range_done[$i] == 1) { next; }
    if (($range_from[$i] == 0) && ($range_from_fixed[$i] == 1)) {
        for ($j=0; $j<$range_length[$i]; $j++) {
            $cond .= compare_word($w1, $j, $words[$j]);
        }
        $range_done[$i] = 1;
        $left_extent = $range_length[$i];
        $left_inset += $range_length[$i] - 1;
        next;
    }
    if (($range_to[$i] == $nw-1) && ($range_to_fixed[$i] == 1)) {
        for ($j=$range_length[$i]-1; $j>=0; $j--) {
            $cond .= compare_word($w2, 0-$j, $words[$nw-1-$j]);
        }
        $range_done[$i] = 1;
        $right_inset += $range_length[$i] - 1;
        $right_extent = $range_length[$i];
        next;
    }
}
```

This code is used in §13.

§**16.** A floating range can consist of only a single word, at present. (An occasional nuisance when coding Inform, but basically an acceptable compromise.) We detect this with a call to Inform's routine `is_word_intermediate`, which determines whether a given word occurs at a position p such that $w_1 < p < w_2$, and returns the minimum value of p if it does, -1 if it does not. If there's only one floating range like this, we can just check the return value to see if it's non-negative. But if there are two or more floating ranges, thus:

```
[[x1, x2 == peeled and ... seeded ... sliced ... in syrup : i, j]]
```

then we need to store the position of "seeded", since we must not only check that it exists but also use that as the left end of the range of words inside which we look for "sliced". To do this, we extract the name of a C variable to store the information in, finding it in the `$write_positions` part of the pattern. Thus on a successful match, the example text would store the word position of "seeded" in `i`, and of "sliced" in `j`.

⟨Then check all of the ranges with floating endpoints 16⟩ ≡

```
my $left_var = $w1;
my $right_var = $w2;
$unwritten_var_exists = 0;
$no_written_vars = 0;
my $i;
for ($i=0; $i<$nr; $i++) {
```

```
        if ($range_done[$i] == 1) { next; }
        if ($range_length[$i] == 1) {
            $cond .= " && (";
            if ($write_positions ne "") {
                if ($write_positions =~ m/^\s*(.*?)\s*\,\s*(.*)$/) {
                    $write_var = $1; $write_positions = $2;
                } else { $write_var = $write_positions; $write_positions = ""; }
                $cond .= "(".$write_var."=";
            } else {
                if ($unwritten_var_exists == 1) {
                    return '?: [[ ... ]] has insufficient : variables';
                }
                $unwritten_var_exists = 1;
            }
            if ($write_var eq "") { $cond.= "("; }
            $cond .=
                "Text__is_word_intermediate(".$words[$range_from[$i]]."_V,"
                . var_offset($left_var, $left_inset)
                . ","
                . var_offset($right_var, 0-$right_inset).("))";
            if ($write_var ne "") {
                $cond .= ",(".$write_var.">=0))";
                $left_var = $write_var;
                $written_vars[$no_written_vars++] = $write_var;
            } else {
                $cond.= " >= 0)";
            }
            $range_done[$i] = 1;
            next;
        }
    }
```

This code is used in §13.

§**17.** That's it for testing the condition: now for the tricky part, storing the word ranges matched on a successful outcome. Note that nothing is written unless the test has succeeded. To see the difficulty here, consider

```
[[w1, w2 == ... OPENBRACKET ... CLOSEBRACKET : i --> w1, w2 ... p1, p2]]
```

which looks for a bracketed clause at the end of the word range and splits it off into (p_1, p_2). We are clearly going to want to set p_2 to w_2, but to the old value of w_2, not the new one, which is going to be just before the open-bracket. So it is important to assign these variables in the right order, so that we don't change w_2 before using it.

⟨Add always-true conditions with the side effect of writing the word ranges 17⟩ ≡

```
$no_assignments = 0;
⟨Work out which variables need to be assigned, and to what 18⟩;
for ($i=0; $i<$no_assignments; $i++) { $assignment_done[$i] = 0; }
⟨Iteratively make all possible safe assignments until all have been assigned 19⟩;
```

This code is used in §13.

§18. Let's call the variables to be written $v_1, v_2, v_3, ..., v_n$, where n will always be an even number. (In the example above, $n = 4$.)

In the following, we generate triples (v_i, v_k, x) to represent that an assignment $v_i \mapsto v_k + x$. Each triple is set up by a call to the routine `assign_set_var`.

⟨Work out which variables need to be assigned, and to what 18⟩ ≡

```
my $j = 0;
my $i;
for ($i=0; $i<$nw; $i++) {
    if ($words[$i] eq '...') {
        if ($j == 0) {
            $from_var = $w1;
            $from_offset = $left_extent;
        } else {
            $from_var = $written_vars[$j-1];
            $from_offset = 1;
        }
        if ($j<$no_written_vars) {
            $to_var = $written_vars[$j];
            $to_offset = -1;
            $j++;
        } else {
            $to_var = $w2;
            $to_offset = 0-$right_extent;
        }
        if ($write_ranges =~ m/^\s*(\S+)\s*\,\s*(\S+)\s*(.*?)\s*$/) {
            $write_from_var = $1;
            $write_to_var = $2;
            $write_ranges = $3;
            assign_set_var($write_from_var, $from_var, $from_offset);
            assign_set_var($write_to_var, $to_var, $to_offset);
            if ($write_ranges ne "") {
                if ($write_ranges =~ m/^\s*\.\.\.\s*(.*)$/) {
                    $write_ranges = $1;
                } else {
                    return '?: [[ ... ]] pairs of write vars should be divided by ...';
                }
            }
        } else {
            if ($write_ranges ne "") {
                return '?: [[ ... ]] write vars should be in pairs divided by ...';
            }
        }
    }
}
```

This code is used in §17.

§19. Now we have a mass of triples (v_i, v_k, x), each representing that an assignment $v_i \mapsto v_k + x$ must be made. But we can't make this assignment until v_i no longer appears in any unassigned triple (v_j, v_i, y), because to do so would invalidate the value of v_i used in that assignment.

We solve this iteratively, at each stage looking for the safe triples to assign. For each variable occurring in the middle position, we calculate `$necessity_count{$v}` as the number of other variables whose assignment is to `$v` plus or minus some offset. Thus, it's safe to assign to `$v` if and only if the necessity count is 0.

A case we have to be careful about is (v_i, v_i, x), where the assignment we'll make has the effect of adding x to v_i. This would be impossible ever to carry out if we regarded it as a dependency of v_i upon itself, which is why such a triple doesn't contribute to the necessity count of v_i.

⟨Iteratively make all possible safe assignments until all have been assigned 19⟩ ≡

```
my $no_done = 0;
my $safety_check = 0;                                    in case of a bug in this code, really
while ($no_done < $no_assignments) {
    ⟨Calculate the necessity count for each unassigned variable 20⟩;
    ⟨Assign all unassigned variables with necessity count 0 21⟩;
    if ($safety_check++ == 1000) ⟨Panic! Something has gone terribly wrong 22⟩;
}
```

This code is used in §17.

§20.

⟨Calculate the necessity count for each unassigned variable 20⟩ ≡

```
my $i;
for ($i=0; $i<$no_assignments; $i++) {
    $necessity_count{$var_to_read[$i]} = 0;
    $necessity_count{$var_to_assign[$i]} = 0;
}
for ($i=0; $i<$no_assignments; $i++) {
    if ($assignment_done[$i] == 0) {
        if ($var_to_read[$i] ne $var_to_assign[$i]) {
            $necessity_count{$var_to_read[$i]}++;
        }
    }
}
```

This code is used in §19.

§21.

⟨Assign all unassigned variables with necessity count 0 21⟩ ≡

```
my $i;
for ($i=0; $i<$no_assignments; $i++) {
    if (($assignment_done[$i] == 0) && ($necessity_count{$var_to_assign[$i]} == 0)) {
        $cond .= set_var($var_to_assign[$i], $var_to_read[$i], $var_to_offset[$i]);
        $assignment_done[$i] = 1;
        $no_done++;
    }
}
```

This code is used in §19.

§**22.** Well, perhaps this is overdramatising it. A simple case here might be

```
[[w1, w2 == possibly ... --> w2, w1]]
```

where we have to make the assignments $w_2 \mapsto w_1 + 1$, $w_1 \mapsto w_2$. Each variable depends on the other, and the log-jam cannot be broken without use of some additional storage (which C offers no convenient way to allocate, within a condition). No doubt with more labour we could contrive a way around this, but it's not worth it: such cases never in practice arose in the whole history of writing Inform.

⟨Panic! Something has gone terribly wrong 22⟩ ≡

```
my $i;
for ($i=0; $i<$no_assignments; $i++) {
    if ($assignment_done[$i] == 0) {
        inweb_error("  ".$var_to_assign[$i]." = ".$var_to_read[$i]." + ".$var_to_offset[$i]);
    }
}
return '?: [[ ... ]] no safe way to write these pairs of variables';
```

This code is used in §19.

§**23.** And that's it for the main routine. We needed two routines for dealing with these variable-assignment triples; first, to create one:

```
sub assign_set_var {
    my $write_var = $_[0];
    my $from_var = $_[1];
    my $from_offset = $_[2];
    if (($from_offset == 0) && ($write_var eq $from_var)) { return ""; }
    $var_to_assign[$no_assignments] = $write_var;
    $var_to_read[$no_assignments] = $from_var;
    $var_to_offset[$no_assignments] = $from_offset;
    $no_assignments++;
}
```

§**24.** And then to compile the assignment asked for:

```
sub set_var {
    my $write_var = $_[0];
    my $from_var = $_[1];
    my $from_offset = $_[2];
    if (($from_offset == 0) && ($write_var eq $from_var)) { return ""; }
    return " && (".$write_var."=".var_offset($from_var, $from_offset).")";
}
```

§**25.** These both need the following useful routine, which compiles a C expression consisting of a variable number plus a constant offset. For legibility, we compile to, say, `q1-5` rather than `q1+-5` in the case of a negative offset.

```
sub var_offset {
    my $var = $_[0];
    my $offset = $_[1];
    if ($offset == 0) { return $var; }
    if ($offset > 0) { return $var."+".$offset; }
    return $var."-".(0-$offset);
}
```

§**26.** Finally, a routine to generate the condition for testing a single word. Here we know we want to test the word whose number is stored in `$var` plus a constant `$with`; for instance, word `w2-1`, the penultimate word in the range being tested. And we want to test that this word is the one in `$with`.

The slight complication here is that a word in the form

```
apple/pear/peach
```

means "any of apple, pear or peach", so that we have to compile a compound condition.

```
sub compare_word {
    my $var = $_[0];
    my $offset = $_[1];
    my $with = $_[2];
    my $this;
    my $rest;
    my $cond;
    $cond = " && ";
    if ($with =~ m/\//) {
        $cond .= "(";
        while ($with =~ m/^(.*?)\/(.*)$/) {
            $this = $1; $with = $2;
            $cond .= compare_word_p($var, $offset, $this);
            $cond .= " || ";
        }
        $cond .= compare_word_p($var, $offset, $with);
        $cond .= ")";
    } else {
        $cond .= compare_word_p($var, $offset, $with);
    }
    return $cond;
}
```

§**27.** So it's actually *this* routine which generates the test of a single word. There's one final language feature to implement: the special word `###` matches any single word.

```
sub compare_word_p {
    my $var = $_[0];
    my $offset = $_[1];
    my $with = $_[2];
    if ($with eq '###') { return "(TRUE)"; }
    return "(compare_word(" . var_offset($var, $offset) . ", " . $with . "_V))";
}
```