INWEB

The Program

Preliminaries

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 `#defines`) 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.