FunnelWeb User's Manual

Ross N. Williams

V1.0 for FunnelWeb V3.0

May 1992

Contents

Preface

When, in 1986, I first read Donald Knuth's technical report on Web[**Knuth83**], and tried Web out, I was simultaneously excited by Knuth's idea of literate programming, and disappointed by his implementation of it. I was excited because I could sense the potential for the literate style to transform the state of mind of the programmer, but was disappointed by Web's rigidity and lack of practicality, which seemed to betray this potential. The Web I used was Pascal-specific, TEX-specific, and applied too many constraints to the programming process. In particular, it insisted on taking control of the program text, mangling the code in the Pascal output files, and imposing its own rather rigid ideas about indenting in the TEX output. All this, combined with the complexity of the tool, led me to come to perceive Web as problem rather than solution.

Despite all this, I was well and truly hooked on the idea of literate programming. The inevitable result was that I designed and implemented my own version of Web — FunnelWeb!

FunnelWeb is not the most sophisticated literate programming tool available, but it is an extremely *practical* tool, striving for simplicity and portability in all areas. Not only is FunnelWeb language-independent, and to some extent typesetter independent, but its implementation also stresses portability, being written in C, and currently operating on four major platforms (Sun, Vax, PC, Mac). FunnelWeb allows the programmer total control over the output file, making it suitable for use with all sorts of format-sensitive languages. It also allows control over its own source code, which has been released under a GNU license. FunnelWeb is quite solid, having to pass a regression testing suite of over 200 tests before being released. Finally, FunnelWeb is well documented by this manual which provides a tutorial, advanced hints, and a language definition.

I would like to dedicate FunnelWeb and this manual to Donald Knuth and his literate programming tool Web. Although this manual is somewhat critical of some aspects of Web, this criticism is really a product of differing design goals. Knuth designed a paradigm (literate programming) and a tool (Web) aimed at the highest pitch of program presentation and typesetting. FunnelWeb aims lower, relaxing constraints, and making compromises in order to achieve simplicity, flexibility, and portability. The result is a practical tool which I hope will attract more people to the literate style.

Ross N. Williams
Adelaide, Australia
May 1992

Acknowledgements

Many thanks to **David Hulse** (dave@cs.adelaide.edu.au) for translating the original version of FunnelWeb (FunnelWeb V1) from Ada into C and getting it to work on Unix and a PC. The C code written by David (FunnelWeb V2) formed the basis of FunnelWeb V3, but was entirely rewritten during the intensive refinement and feature-injection period leading up to this release (FunnelWeb V3 is about three times the size of FunnelWeb V2). Nevertheless, without this important first translation step, I would probably not have found the motivation to develop FunnelWeb to its present state.

Thanks go to **Simon Hackett** (simon@internode.com.au) of Internode Systems Pty Ltd for the use of his Sun, Mac, and PC, for assistance in porting FunnelWeb to the Sun and PC, and for helpful discussions.

Thanks go to **Jeremy Begg** (jeremy@vsm.com.au) of VSM Software Services for the use of his VAX, and for assistance with the VMS-specific code.

Thanks to **Barry Dwyer** (dwyer@cs.adelaide.edu.au) and **Roger Brissenden** (rjb@koala.harvard.edu) for trying out FunnelWeb Version 1 in 1987 and providing valuable feedback.

Thanks to Donald Knuth for establishing the idea of literate programming in the first place.

Ross N. Williams
Adelaide, Australia
May 1992

Presentation Notes

References: All references are set in bold and are cited in square brackets in the form [<*firstauthor*><*year*>]. All references cited in the text appear in the reference list and the index.

Special terms: New or important terminology has been set in bold face and appears in the index. A glossary appears as an appendix.

Typesetting: This document was prepared by the author using Andrew Trevorrow's (`akt150@cscgpo.anu.edu.au`) implementation (OzTeX) of the TEX/LATEX[**Knuth84**] [**Lamport86**] typesetting system running on a Macintosh-SE.

Graphics: All diagrams have been constructed out of text using the LATEX `verbatim` environment so as to allow this document to be disseminated electronically and printed using LATEX without access to the author's drawing tools.

Known typesetting problems: While every attempt has been made to give a good presentation within the time available, some shortcuts have had to be taken. In particular, the author has been unable to work out how to get LATEX to suppress blank pages at the start of chapters.

Chapter 1

A Tutorial Introduction

1.1 What is Literate Programming?

A traditional computer program consists of a text file containing program code. Scattered in amongst the program code are comments which describe the various parts of the code.

In **literate programming** the emphasis is reversed. Instead of writing code containing documentation, the literate programmer writes documentation containing code. No longer does the English commentary injected into a program have to be hidden in comment delimiters at the top of the file, or under procedure headings, or at the end of lines. Instead, it is wrenched into the daylight and made the main focus. The "program" then becomes primarily a document directed at humans, with the code being herded between "code delimiters" from where it can be extracted and shuffled out sideways to the language system by literate programming tools.

The effect of this simple shift of emphasis can be so profound as to change one's whole approach to programming. Under the literate programming paradigm, the central activity of programming becomes that of conveying meaning to other intelligent beings rather than merely convincing the computer to behave in a particular way. It is the difference between performing and exposing a magic trick.

In order to program in a literate style, particular tools are required. The traditional approach (used in the FunnelWeb system) is to have some sort of text-file-in/text-file-out utility that reads a literate program (containing a program commentary peppered with scraps of program text) and writes out a file containing all the program code and a file containing typesetter commands representing the entire input document, documentation, code, and all (**Figure 1**).

Given the coming age of hypertext systems, this is probably not the best approach. However, it does mesh beautifully with current text files and command line interfaces, the expectation of linear presentations in the documents we read, and the particular requirements of current programming languages and typesetting systems. It is certainly not a bad approach.

With this structure in place, the literate programming system can provide far more than just a reversal of the priority of comments and code. In its full blown form, a good literate programming facility can provide total support for the essential thrust of literate programming, which is that computer programs should be written more for the human reader than for the compiler. In particular, a literate programming system can provide:

> **Re-ordering of code:** Programming languages often force the programmer to give the various parts of a computer program in a particular order. For example, the Pascal programming language[**BSI82**] imposes the ordering: constants, types, variables, procedures, code. Pascal also requires that procedures appear in an order consistent with the partial ordering imposed by the static call graph (but forward declarations allow this to be bypassed). In contrast, the literate style requires that the programmer be free to present the computer program in any order whatsoever. The facility to

```
         +-----------------------------------------+
         | File containing the program description |
         | peppered with scraps of program code.   |
         | This is what the programmer works on.   |
         |          (e.g. sloth.web)               |
         +-----------------------------------------+
                             |
                             v
               o---------------------------o
               | Literate Programming Tool |
               o---------------------------o
                             |
         +-------------------+-------------------+
         |                                       |
         v                                       v
+------------------+           +----------------------------------+
|   Traditional    |           | Documentation file suitable for  |
| Computer Program |           | input into a typesetting program |
|  (e.g. sloth.c)  |           |        (e.g. sloth.tex)          |
+------------------+           +----------------------------------+
```

Literate programming tools could be organized in a number of ways. However, to fit in with current file and command line based environments, most tools conform to the traditional architecture shown here in which the user feeds in a file containing a literate program, and the literate programming utility generates program files and a documentation file.

Figure 1: Traditional architecture of literate programming tools.

do this is implemented in literate programming tools by providing text *macros* that can be defined and used in any order.

Typeset code and documentation: Traditionally program listings are dull affairs consisting of pages of fan-form paper imprinted with meandering coastlines of structured text in a boring font. In contrast, literate programming systems are capable of producing documentation that is superior in two ways. First, because most of the documentation text is fed straight to the typesetter, the programmer can make use of all the power of the underlying typesetter, resulting in documentation that has the same presentation as an ordinary typeset document. Second, because the literate programming utility sees all the code, it can use its knowledge of the programming language and the features of the typesetting language to typeset the program code as if it were appearing in a technical journal. It is the difference between:

```
while sloth<walrus loop
   sloth:=sloth+1;
end loop
```

and

while *sloth*$<$*walrus* **loop**
    sloth$\leftarrow$*sloth*$+1$;
end loop

Unfortunately, while FunnelWeb provides full typesetting of the documentation, it typesets all of its code in the style of the first of these two examples. To typeset in the style of the second requires knowledge of the programming language, and the current version of FunnelWeb is programming language independent. At a later stage, it is possible that FunnelWeb will be modified to read in a file containing information about the target programming language to be used to assist in typesetting the code properly.

Cross referencing: Because the literate tool sees all the code and documentation, it is able to generate extensive cross referencing information in the typeset documentation. This makes the printed program document more easy to navigate and partially compensates for the lack of an automatic searching facility when reading printed documentation.

In the end, the details don't matter. The most significant benefit that literate programming offers is *its capacity to transform the state of mind of the programmer*. It is now legend that the act of explaining something can transform one's understanding of it. This is one of the justifications behind the powerful combination of research and teaching in universities[**Rosovsky90**]. Similarly, by constantly explaining the unfolding program code in English to an imaginary reader, the programmer transforms his perception of the code, laying it open, prone, to the critical eye.

The result of this exposure is a higher quality of programming. When exposed to the harsh light of the literate eye, bugs crawl out, special cases vanish, and sloppy code evaporates. As a rule literate programs take longer to write than ordinary programs, but the total development time is the same or less because the time taken to write and document the program carefully is compensated for by a reduced debugging and maintenance time. Thus literate programming does not merely assist in the preparation of documentation, but also makes significant contributes to the process of programming itself. In practice this has turned out to be a contribution far more important than the mere capacity to produce typeset documentation.

For more information on literate programming, the reader is directed to Knuth's early founding work [**Knuth83**] and [**Knuth84**]. For more recent information refer to [**Smith91**], which provides a comprehensive bibliography up to 1990.

1.2 What is FunnelWeb?

FunnelWeb is a particular literate programming system that is implemented by a single C program. FunnelWeb takes as input a single `.fw` **input file** and writes one or more **product files** and a **documentation file** (**Figure 2**).

```
                        +-----------+
                        | sloth.fw  |
                        +-----------+
                              |
                              v
                    o-------------------o
                    | FUNNELWEB Program |
                    o-------------------o
                              |
           +------------------+----------------+
           |                                   |
           v                                   v
+-----------------------------+   +--------------------------------+
| Product File (e.g. sloth.c) |   | Documentation File (sloth.tex) |
+-----------------------------+   +--------------------------------+
```

FunnelWeb follows the traditional architecture of literate programming tools.

Figure 2: Architecture of FunnelWeb.

In literate programming systems, it is usual to refer to the product file as a "program file". However, as FunnelWeb is a general tool that can be used to prepare all sorts of text files that are not computer programs, the more generic term "product file" was chosen. Product files should be carefully distinguished from the term **output files** which refers to all of the output files produced by FunnelWeb.

FunnelWeb is distinguished by the following characteristics:

Simplicity: A governing design goal of FunnelWeb is to provide a *simple* tool that could be easily learnt and completely mastered. This manual is thick because it is comprehensive and lingers on the ways in which FunnelWeb can be used. The tool itself is quite simple.

Reliability: Another design goal is to provide a tool that will protect the user as much as possible from silly errors. Macro preprocessors are notorious for causing obscure errors. Every attempt has been made in FunnelWeb to keep the syntax robust. For example, in FunnelWeb the syntax of macro calls has been purposely designed to be highly visible so that the reader is always aware when the macro facility is being invoked.

Language and Typesetter Independence: Unlike Knuth's original Web system which was specific to the Pascal programming language[**BSI82**] and the TEX typesetting language[**Knuth84**], FunnelWeb strives to be language and typesetter independent. The current version of FunnelWeb is completely language independent, but is still somewhat dependent on the TEX typesetter language.

Portability: FunnelWeb has been written in the C programming language with great emphasis on portability. FunnelWeb currently runs on the Sun, VAX, IBM PC, and Mac.

Controllable: FunnelWeb is an extremely controllable tool. To protect users' investment in source files constructed in the FunnelWeb macro language, the C source code to FunnelWeb has been released under GNU license. This means that it will always be available to everyone. Furthermore, license has been granted for the FunnelWeb User's Manual and FunnelWeb Hacker's Manual to be copied freely so long as they are not modified. All this means that FunnelWeb is not going to disappear suddenly.

A Production Tool: Above all, FunnelWeb has been designed to be a production tool and every effort has been made to ensure that it will operate effectively in a professional environment. FunnelWeb is "open" and portable. There is a comprehensive user manual. Its error messages are comprehensive. It is fast. Finally, it has been designed with the experience of three years of using FunnelWeb V1.

For more information on the history and design of FunnelWeb, see the *FunnelWeb Hacker's Manual*.

1.3 The Name FunnelWeb

The name "FunnelWeb" was chosen because it contains the name "WEB", which is the name of Knuth's system. It was also chosen because it has a distinctly Australian flavour.

Funnel-web spiders are found in Northern and Eastern Australia. They are about three to four centimetres long and are very poisonous. The Sydney Funnel-web spider (*Atrax robustus*), common in Sydney, has caused the most trouble and has been responsible for several deaths. Funnel-web spiders love to crawl into temporarily discarded shoes where they later react in a hostile manner to an unsuspecting foot. They are known to hang on once they sink their fangs in. Funnel-web spiders derive their name from the shape of their webs which are horizontally-aligned narrowing tubes, open at one end[**ANZE**].

The Funnel-web spider, like the tiger snake and the white pointer shark, is secretly regarded by Australians as a kind of national treasure.

> **F** is for Funnel-web
> Our furry-legged foe.
> He sleeps in your slipper
> And breakfasts on toe.
> — One verse from *A Megastar's Mantras: Things that Mean a Lot to Me*,
> by Dame Edna Everage[**Humphries91**].

1.4 A FunnelWeb Tutorial

The remainder of this chapter consists of an introductory tutorial on FunnelWeb. Ideally you should have a working version of FunnelWeb in front of you when reading this chapter so that you can try out the examples yourself. There is no need to try all the examples so long as you type in enough to feel comfortable with what you are reading. There is no harm in reading this chapter offline too, so long as you get to the computer within a couple of days to reinforce what you have read.

For best effect, you should create a new, temporary, empty directory in which to experiment with FunnelWeb. That way, it will be more obvious when FunnelWeb creates an output file. You can either type in the examples in this chapter directly, or copy and paste them from the LaTeX source file for this chapter or the FunnelWeb test suite. The source file for this chapter should be available in a file such as `/fwdir/userman/ch1.tex`. The test suite should be available in a directory such as `/fwdir/tests/`. The test files called `ex01.fw` through `ex16.fw` contain the examples in this chapter. The test files called `hi01.fw` through `hi10.fw` contain the examples in the next chapter.

If you do not yet have an installed copy of FunnelWeb, refer to Chapter 4 for full details on how to obtain and install a copy of FunnelWeb. If you are not sure if you have an installed copy, try invoking FunnelWeb by giving the command "`fw`". If this yields an error such as "command not found" then you do not have a properly installed version of FunnelWeb.

1.5 A Hello World Document

Just as one starts the process of learning a new programming language with a "Hello World" program, when learning FunnelWeb, you can start with a "Hello World" document! And here it is! Edit a text file called `hello.fw` and put the following text in it. (Note: The second character is the letter "Oh", not the digit "Zero").

```
@O@<hello.txt@>@{Hello World@+@}
```

To "run" this "program", invoke FunnelWeb using the "`fw`" command as follows.

```
fw hello
```

If this command doesn't work, then chances are that FunnelWeb has not been installed on your machine. Refer to Chapter 4 for full details on how to obtain and install a copy of FunnelWeb.

There should be no errors. If there are, have a look at the listing file `hello.lis`, which should contain an explanation of the error, and compare the area in the file where the error occurred with the text above. If there are no errors, you will find that the following two files have been created.

```
hello.lis   - The LISTING file.
hello.txt   - The PRODUCT file.
```

Take a look at `hello.txt`. It should contain a single line with the text `Hello World`. Let's take another look at the input file.

```
@O@<hello.txt@>@{Hello World@+@}
```

The whole structure of the input file is controlled by "`@`", called the **special character**, which introduces **special sequences**. A scanner's-eye view of the command line looks like this:

```
@O  @<  "hello.txt"  @>  @{  "Hello World"  @+  @}
```

The `@` character controls everything. In this file we have six different special sequences that together form a single macro definition. The `@<` and `@>` delimit the name of the macro. The `@O` signals the start of the macro definition and indicates that the macro is to be connected to a product file with the same name as the macro (This is is why we got a product file when we ran FunnelWeb). The `@{` and `@}` delimit the body of the macro. Finally, the `@+` instructs that an end of line sequence should be inserted at that point in the product file.

If you think this syntax looks messy, then you're right. It *is* messy. FunnelWeb *could* have employed a "simpler" notation in which more of the `@` sequences were eliminated. For example:

```
Warning: This example is NOT legal FunnelWeb.

#hello.txt{Hello World+}
```

However, if such a syntax were used, the user (you!) would have to remember that `#` starts a new macro. You would also have to remember that the characters } and `+` cannot be used in a macro body without a fuss. And so on. FunnelWeb is messier, but provides one simple rule: *Nothing special happens unless the special character* `@` *appears.*

This means that in FunnelWeb, you can look at large blocks of text in the confidence that (unlike for the C pre-processor) there are no macro calls hidden in there. If there were, there would be an `@` character![1]

Let's take another look at the hello world program.

[1] The only exception to this rule occurs where the user has explicitly changed the special character using the `@=` special sequence.

```
@O@<hello.txt@>@{Hello World@+@}
```

In its current form, it consists of a single macro definition. This definition, while completely valid on its own, only represents half the power of FunnelWeb. In fact you could say that it is a "Hello Northern Hemisphere Program". To turn it into a proper FunnelWeb "Hello World" program, we need to add some documentation!

A FunnelWeb input file consists of a sequence of macro definitions surrounded by a sea of documentation which is just ordinary text. Modify your hello world document so that it looks like this:

```
This hello world document was created by -insert your name here-.

@O@<hello.txt@>@{Hello World@+@}

It writes out a file called hello.txt containing the string ``Hello World''.
```

Now run it through FunnelWeb, but this time, add a `+t` to the command line.

```
fw hello +t
```

If all goes well, you should find that you now have

```
hello.lis   - A LISTING       file.
hello.tex   - A DOCUMENTATION file (in TeX format).
hello.txt   - A PRODUCT       file.
```

Take a look at `hello.txt`. You will find that it is identical to the `hello.txt` of the previous run. Only macro definitions affect the product files that FunnelWeb produces (as a result of `@O` macro definitions). The surrounding documentation has *no* effect. In contrast, the new file, `hello.tex` (have a look at it now) which was created as a result of your adding the `+t` option contains a fairly full representation of the input file. Whereas `hello.txt` is the *product file* of FunnelWeb, `hello.tex` is the *documentation file*. Try typesetting the documentation file now using the TeX typesetting program. Then print it. The following commands are an example of the sort of commands you will have to give to do this.

```
tex hello                    ! Typeset the documentation.
lpr -Pcslw -d hello.dvi      ! Print the typeset documentation.
```

The result should be a single page containing the two lines of documentation along with a typeset representation of the macro. At this point, you have exercised the two main aspects of FunnelWeb. Starting with an input file containing macros (or in this case macro) and documentation, you have successfully generated a product file based on the macros, and a documentation file, based on the entire document. Congratulations!

The remainder of this tutorial is divided into two parts, which focus on FunnelWeb's macro facilities and its typesetting facilities. By tradition, the generation of program files from a literate text is called **Tangling**, and the generation of typeset documentation is called **Weaving**.[2]

[2] In FunnelWeb, these two functions are aspects of a single computer program. However, in Knuth's WEB system, the two functions are embodied in two separate computer programs called Tangle and Weave, presumably because, as everyone knows, "it takes two to Tangle".

1.6 Macro Facilities

The hello world program of the previous section exercised both the macro expansion (product-file) aspect of FunnelWeb, and the typesetting (documentation file) aspect of FunnelWeb. This section contains an exposition of the macro facilities, and totally ignores the documentation side. This is partly to increase the focus of the tutorial, and partly because documentation is usually bulky and would take too long for the reader to type in to make the tutorial effective.

1.6.1 Simple Macros

The original "Hello World" program consisted of a single macro definition.

```
@O@<hello.txt@>@{Hello World@+@}
```

In fact, this is a rather exceptional macro, as it causes its expansion to be written to a product file. The `@O` (for **O**utput) signals this. In FunnelWeb, most macros are defined using `@$`. This results in a macro that does not generate a product file, but which can be called in other macros (including `@O` macros). Let us expand the hello world program to include some other macros.

```
@O@<hello.txt@>@{@<Greetings@>@+@}

@$@<H@>==@{Hello@}
@$@<W@>==@{World@}

@$@<Greetings@>==@{@<H@> @<W@>@}
```

Type in the file and run it through FunnelWeb using the command:

```
fw hello
```

The product file (result.out) should look like this:

```
Hello World
```

This short program illustrates some of the features of ordinary macros in FunnelWeb. Consider the `@O` macro. Instead of containing straight text ("Hello World"), it now contains the macro call `@<Greetings@>`. A FunnelWeb macro can be called from within the body of another macro just by giving the macro name delimited in `@<` and `@>`.

At the bottom of the file is the definition of the `@<Greetings@>` macro. The definition is similar to the definition of `hello.txt` except that it starts with `@$` to indicate that no product file is desired from this macro (directly). It also employs the optional `==` syntax which has no semantic impact, but can be used to make definitions clearer. The body of the `@<Greetings@>` macro consists of calls to the `H` and `W` macros which are defined immediately above.

Note that the macros are not constrained to be defined in any particular order. One of the main features of literate programming tools is that they allow the different parts of the text document being developed (usually a computer program) to be layed out in any order. So long as there is a definition somewhere in the input file for every macro call, FunnelWeb will sort it all out.

In fact, FunnelWeb's macro facility is very simple. Unlike many macro preprocessors which allow macros to define other macros, FunnelWeb completely finishes parsing and analysing the macros in the input file before it starts expanding them into product files. Other preprocessors allow macros to be redefined like variables (as in, say, TeX) taking on many different values as the macro preprocessor travels through the input file. In contrast, FunnelWeb has no concept of "different times" and treats the input as one huge static orderless, timeless, collection of definitions. In FunnelWeb, there is only ever one time, and so there can only ever be one value/definition for each macro.

1.6.2 Number of Times Called

So far we have seen only tiny, degenerate input files. The next example moves up to the level of "trivial", but starts to convey the flavour of the way FunnelWeb can be used in practice. Normally, there would be documentation text appearing between the macros, but this has been omitted so as to keep the focus on the macros themselves. Although the next example is much longer than the previous example, the only new construct is `@-` which can appear only at the end of a line, and suppresses it, preventing it from appearing in the text. The `@-` construct allows the text of a macro to be aligned at the left margin, rather than having the first line hanging at the end of the `@{`. FunnelWeb could have been set up so that this end of line marker was suppressed. However, it would have been a special case that would have broken the very memorable rule "the text of a macro is the text appearing between the `@{` and `@}`".

Type the following text into the file `hello.fw` and run it through FunnelWeb. The file contains some intentional errors so be sure to type it in exactly and worry only if FunnelWeb *doesn't* generate some errors.

```
@O@<hello.c@>==@{@-
@<Include Files@>
@<Include Files@>
@<Main Program@>
@}

@$@<Main Program@>==@{@-
main()
{
 doit();
}
@}

@$@<Subroutine@>==@{@-
void doit()
{
 int i;
 for (i=0;i<10;i++)
   {
    @<Print@>
    @<Print@>
   }
}@}

@$@<Print@>==@{@-
printf("Hello World!");
printf("\n");@}

@$@<Scan@>==@{scanf@}

@$@<Include Files@>==@{@-
#include <stdio.h>
#include <stdlib.h>@}
```

What happened? Well, if you haven't typed the file in properly, you may get some miscellaneous syntax errors. Fix these before continuing. If the file has been correctly typed, you should be faced with some error messages to do with the number of times some of the macros are called.

By default, FunnelWeb insists that each macro defined is invoked exactly once. However, the file above defines macros that are used more than once and a macro that is not used at all. Let us examine the errors.

First, we see that FunnelWeb has alerted us to the fact that the `Include Files` macro has been called twice. Once alerted to this, a quick look at the program convinces us that calling the macro twice is a mistake, and that one of the calls should be eliminated.

Second, we note that FunnelWeb has alerted us to the fact that the `@<subroutine@>` macro is never called. Again, a quick look at the program tells us that this is a mistake (and a very common one in the use of FunnelWeb), and that a call to the `@<subroutine@>` macro should be inserted just above the call to the `@<Main Program@>` macro in the definition of `@<hello.c@>`.

These two cases demonstrate why these checks have been placed in FunnelWeb. It is nearly always acceptable for a macro to be called once. However, if a macro is not called at all, or called more than once, this is often a sign that the user has made a mistake.

These checks have a dark side too. In addition to the errors mentioned above, FunnelWeb has generated two similar errors that do not help us.

First, we are alerted to the fact that the `@<print@>` macro has been called twice. Clearly, in this case, this is not a problem, and so here FunnelWeb's fussiness is a nuisance.

Second, we are alerted to the fact that the `@<scan@>` macro has never been called. Like the `@<print@>` macro, this macro was defined as a notational convenience, and clearly it does not matter here if it is not used. Again, FunnelWeb is being a nuisance.

The four cases above demonstrate the light and dark side of FunnelWeb's insistence that each macro be called exactly once. To resolve the conflict without reducing the strength of the checking, FunnelWeb provides two special sequences `@Z` (for **Z**ero) and `@M` (for **M**any) that can be attached to macro definitions. Presence of the `@Z` tag allows the designated macro to be called zero times. Presence of the `@M` tag allows the designated macro to be called more than once. A single macro may carry both tags. It is always true that all macros are allowed to be called exactly once.

Here is the revised program with the errors fixed, by eliminating or adding macro calls, or by adding tags. Try processing the file now. There should be no errors.

```
@O@<hello.c@>==@{@-
@<Include Files@>
@<Function@>
@<Main Program@>
@}

@$@<Main Program@>==@{@-
main()
{
 doit();
}
@}

@$@<Function@>==@{@-
void doit()
{
 int i;
 for (i=0;i<10;i++)
   {
    @<Print@>
    @<Print@>
   }
}@}

@$@<Print@>@M==@{@-
printf("Hello World!");
printf("\n");@}
```

```
@$@<Scan@>@Z==@{scanf@}

@$@<Include Files@>==@{@-
#include <stdio.h>
#include <stdlib.h>@}
```

1.6.3 Indentation

The body of the `print` macro of the previous example contains two lines of text. A literal substitution of this macro's body in its context would result in:

```
   {
    printf("Hello World!");
printf("\n");
    printf("Hello World!");
printf("\n");
   }
```

But instead, it comes out as (have a look at this part of `hello.c` now):

```
   {
    printf("Hello World!");
    printf("\n");
    printf("Hello World!");
    printf("\n");
   }
```

The explanation is that FunnelWeb indents each line of multiline macros by the level of indentation at the point of call. This means that, as in the case above, program texts, which are usually highly indented, come out looking "right".

In other circumstances, where the model of the text is one dimensional, FunnelWeb's indentation could become an impediment or even a danger. In these cases, it can be switched off by including the FunnelWeb **pragma** line

```
@p indentation = none
```

anywhere in the input file.

One of the design goals of FunnelWeb is to allow the user total control over the product files. This contrasts with the approach of Knuth's WEB system **[Knuth83]** (upon which FunnelWeb is based), which mangles the input text at the Pascal program syntax level, truncating identifiers, converting the text to upper case, and paragraphing text. Here is an example of part of a Pascal program produced by WEB (from page 14 of **[Knuth83]**):

```
IF R=0 THEN XREF[P]:=XREFPTR ELSE XMEM[R].XLINKFIELD:=XREFPTR;END;{:51}
{58:}FUNCTION IDLOOKUP(T:EIGHTBITS):NAMEPOINTER;LABEL 31;
VAR I:0..LONGBUFSIZE;H:0..HASHSIZE;K:0..MAXBYTES;W:0..1;
L:0..LONGBUFSIZE;P:NAMEPOINTER;BEGIN L:=IDLOC-IDFIRST;{59:}
H:=BUFFER[IDFIRST];I=IDFIRST+1;
WHILE I<IDLOC DO BEGIN H:=(H+H+BUFFER[I])MOD HASHSIZE;I=I+1;END{:59};
```

Knuth's theory is that the program generated by a literate programming system should be treated as object code and hence should look like object code too. While this may be an admirable approach in the long run, the present programming environment is one of faulty compilers and buggy tools. The FunnelWeb view is that, in this environment, the programmer needs all the help he can get and that therefore he should be allowed total control over the product file. Another reason for FunnelWeb's providing total control over the product file, is that FunnelWeb is intended to be target language independent, and so even if Knuth's view were adopted, it would not be clear what a legitimate transformation of the text could be.

1.6.4 Additive Macros

Sometimes it is convenient to build up the definition of a macro in stages throughout the input file. In FunnelWeb, this can be done using an **additive macro**. An additive macro is identical to an ordinary macro except that

1. It has `+=` instead of `==`.
2. It can be defined in one or more parts throughout the input file. The definition of the macro is the concatenation of all the parts in the order in which they appear.

The following example shows how additive macros can be used to scatter and regroup information, in this case assisting in the lucid construction of a data abstraction in a language (Pascal) that does not support them explicitly.

```
@!******************************

@O@<prog.pas@>==@{@-
program adt(input,output);
@<Types@>
@<Variables@>
@<Procedures@>
begin startproc; end.
@}

@!******************************

@$@<Types@>+=@{@-
type buffer_type =
   record
   length : integer;
   buf : array[1..100] of char;
   end;
@}

@$@<Variables@>+=@{@-
bigbuf : buffer_type;
@}

@$@<Procedures@>+=@{@-
procedure buf_init (var b : buffer_type               ) {Body of buf_init}
procedure buf_add  (var b : buffer_type;     ch : char) {Body of buf_add}
procedure buf_get  (var b : buffer_type; var ch : char) {Body of buf_get}
@}

@!******************************
```

```
@$@<Types@>+=@{@-
type complex_type = record r,i : real; end;
@}

@$@<Procedures@>+=@{@-
procedure cm_set (var c: complex_type; a,b: real)         {Body of cm_set}
procedure cm_add (a,b: complex_type; var c: complex_type) {Body of cm_add}
{Other procedures and functions}
@}

@!******************************

{...more pieces of program...}

@!******************************
```

It is important to remember that the definition of each macro does not change throughout the input file. FunnelWeb parses the entire input file and assembles all the macro definitions before it even starts to expand macros. As a result, each additive macro can only have one definition, and that definition is the concatenation of all its parts.

The example above shows how additive macros can be used to rearrange the presentation of a computer program in the order in which the user wishes to discuss it rather than the order in which the compiler requires that it be consumed. It is easy, however, to abuse the feature of additive macros. In many cases, the same effect can be obtained more clearly by replacing each part of an additive macro in-situ using uniquely named non-additive macros, and then collect them together as a group at the point where the additive macro is called. Doing this is more work, and is more error prone, but can result in a clearer exposition. The following program illustrates this alternative approach.

```
@!******************************

@O@<prog.pas@>==@{@-
program adt(input,output);
@<Types@>
@<Variables@>
@<Procedures@>
begin startproc; end.
@}

@$@<Types@>==@{@-
@<Buffer type@>
@<Complex type@>
@}

@$@<Variables@>==@{@-
@<Buffer variable@>
@}

@$@<Procedures@>==@{@-
@<Buffer procedures@>
@<Complex procedures@>
@}

@!******************************

@$@<Buffer type@>==@{@-
```

```
type buffer_type = record
                   length : integer;
                   buf : array[1..100] of char;
                   end;
@}

@$@<Buffer variable@>==@{@-
bigbuf : buffer_type;
@}

@$@<Buffer procedures@>==@{@-
procedure buf_init(var b : buffer_type) {Body of buf_init}
procedure buf_add(var b : buffer_type; ch : char) {Body of buf_add}
procedure buf_get(var b : buffer_type; var ch : char) {Body of buf_get}
@}

@!******************************

@$@<Complex type@>==@{@-
type complex_type = record r,i : real; end;
@}

@$@<Complex procedures@>+=@{@-
procedure cm_set(var c: complex_type; a,b : real)  {Body of cm_set}
procedure cm_add(a,b : complex_type; var c: complex_type) {Body of cm_add}
{Other procedures and functions}
@}

@!******************************

{...more pieces of program...}

@!******************************
```

One of advantages of FunnelWeb (and literate programming in general) is that (as shown above) it allows the user to lay out the program in whatever order is desired with near total independence from the ordering requirements of the target programming language.

Additive macros are allowed to be tagged with `@Z` and `@M` just as other macros can, but the tags must appear only on the first definition of the macro. Additive macros cannot be connected directly to product files.

1.6.5 Parameterized Macros

No self-respecting macro preprocessor would be complete without some form of macro parameterization, and FunnelWeb is no exception. FunnelWeb allows each macro to have from zero to nine formal parameters named `@1`, `@2`, `@3`, `@4`, `@5`, `@6`, `@7`, `@8`, and `@9`.

To define a macro with one or more parameters, insert a formal parameter list just after the macro name in the macro definition. Because macro parameters have fixed names (`@1`...`@9`), there is no need to specify the names of formal parameters in the formal parameter list. All that need be conveyed is how many parameters the macro has. Here is an example of the definition of a macro having three parameters:

```
@$@<While loop@>@(@3@)@M==@{@-
@1
while (@2)
```

```
  {
   @3
  }
@}
```

To call a parameterized macro, an actual parameter list must be supplied that contains exactly the same number of actual parameters as there are formal parameters in the definition of the macro being called. An actual parameter list is delimited by `@(` and `@)`, and parameters are *separated* by "`@,`". The actual parameters themselves are general FunnelWeb expressions (see Chapter 3 for the exact syntax) and can be inserted into the list directly or can be delimited by `@"` so as to allow some white space to assist in formatting the actual parameters. Here are some examples of calls of the `While loop` macro defined above.

```
@! First form of actual parameters without whitespace and double quotes.
@<While loop@>@(x=1;@,x<=10@,printf("X=%u\n",x);@)

@! Second form of actual parameters. The double quotes allow non-active
@! whitespace that helps to lay out the actual parameters neatly.
@! This call is functionally identical to the one above.
@<While loop@>@(
   @"x:=1;@" @,
   @"x<=10@" @,
   @"printf("X=%u\n",x);@" @)

@! The two forms can be mixed in a single call.
@<While loop@>@(x=1;@,x<=10@,
              @"printf("X=%u\n",x);@" @)
```

A few rules about parameterized macros are worth mentioning. Macros that do not have any parameters must have no formal or actual parameter lists. Additive macros can have parameters, but the formal parameter list must appear in the first definition part only.

Here is another example of the use of parameterized macros. This time, parameters and macro calls are used in a FunnelWeb input file that constructs an $O(n)$ representation of a song whose full size is $O(n^2)$ in the number n of unique lines.

```
@O@<Twelve_bugs.txt@>==@{@-
The Twelve Bugs of Christmas
----------------------------
@<Verse@>@(@"first@"    @,@<1@>@)
@<Verse@>@(@"second@"   @,@<2@>@)
@<Verse@>@(@"third@"    @,@<3@>@)
@<Verse@>@(@"fourth@"   @,@<4@>@)
@<Verse@>@(@"fifth@"    @,@<5@>@)
@<Verse@>@(@"sixth@"    @,@<6@>@)
@<Verse@>@(@"seventh@"  @,@<7@>@)
@<Verse@>@(@"eighth@"   @,@<8@>@)
@<Verse@>@(@"ninth@"    @,@<9@>@)
@<Verse@>@(@"tenth@"    @,@<A@>@)
@<Verse@>@(@"eleventh@" @,@<B@>@)
@<Verse@>@(@"twelfth@"  @,@<C@>@)

This song appeared in the internet newsgroup rec.humor.funny on 24-Dec-1991.
It was contributed by Pat Scannell (scannell@@darkstar.ma30.bull.com).
@}
```

```
@$@<Verse@>@(@2@)@M==@{@-
For the @1 bug of Christmas, my manager said to me
     @2
@}

@$@<1@>@M==@{See if they can do it again.@}
@$@<2@>@M==@{Ask them how they did it and@+@<1@>@}
@$@<3@>@M==@{Try to reproduce it@+@<2@>@}
@$@<4@>@M==@{Run with the debugger@+@<3@>@}
@$@<5@>@M==@{Ask for a dump@+@<4@>@}
@$@<6@>@M==@{Reinstall the software@+@<5@>@}
@$@<7@>@M==@{Say they need an upgrade@+@<6@>@}
@$@<8@>@M==@{Find a way around it@+@<7@>@}
@$@<9@>@M==@{Blame it on the hardware@+@<8@>@}
@$@<A@>@M==@{Change the documentation@+@<9@>@}
@$@<B@>@M==@{Say it's not supported@+@<A@>@}
@$@<C@>@M==@{Tell them it's a feature@+@<B@>@}
```

1.6.6 Macro Expansion

One of the strengths of FunnelWeb is that, when writing product files, it does not attempt to evaluate any text expression (e.g. text block, parameter, macro call) in memory and then write the result out. Instead, it always writes out what it is expanding dynamically and directly. This means that the user need not fear defining macros that expand to huge amounts of text and then calling those macros in other macros, or passing those huge macros as parameters to other macros. In all cases, FunnelWeb expands directly to the product file, and there can be no danger in running out of memory during expansion (except for running out of stack space and other marginally used resources in pathological cases).

The only thing to remember in this regard is that FunnelWeb always stores the entire *input* file and all included files, in their entirety in memory, for the duration of the run.

Here is an example, that illustrates how robust FunnelWeb is:

```
@! FunnelWeb copes well with the following macro definitions.
@! (Providing that it has a little over ten megabytes of memory).

@O@<woppa.txt@>==@{@<Quote@>@(@<Humungeous@>@)@+@}

@$@<Quote@>@(@1@)==@{"@1"@}

@$@<Humungeous@>==@{@-
...Ten Megabytes of Text...
@}
```

1.6.7 Include Files

FunnelWeb provides a nested include file facility that can be used for a number of purposes. When FunnelWeb runs into a single line containing the special sequence `@i` followed by a blank, followed by a file name, it reads in the designated file and replaces the line containing the command (including the end of line marker at the end of the line) with the entire contents of the designated file. For example, if there was a file called `camera.txt` containing the two lines:

```
'Cos I shoot with a camera instead of a gun.
The animals flock to be petted and fed,
```

and another file called `poem.fw` containing the following four lines

```
I like to go shooting, it's a whole lot of fun,
@i camera.txt
Cos they know my camera isn't loaded with lead.
- RNW, 04-Jan-1991.
```

Then, if FunnelWeb were to process `poem.fw`, the result would be as if FunnelWeb had read in:

```
I like to go shooting, it's a whole lot of fun,
'Cos I shoot with a camera instead of a gun.
The animals flock to be petted and fed,
'Cos they know my camera isn't loaded with lead.
- RNW, 04-Jan-1991.
```

FunnelWeb expands include files before it starts scanning and parsing the included text. The result is that include files can contain anything that can be found in a FunnelWeb file. The following example illustrates the level at which the include mechanism operates. If `main.fw` contains

```
@O@<output.dat@>==@{@-
@i inc.fw
This is the text of the sloth macro.
@}
```

and inc.fw contains

```
@<Sloth@>
@}

@$@<Sloth@>==@{@-
```

Then if FunnelWeb were applied to `main.fw`, it would see:

```
@O@<output.dat@>==@{@-
@<Sloth@>
@}

@$@<Sloth@>==@{@-
This is the text of the sloth macro.
@}
```

which it would process in the normal manner. The only special sequence processing that takes place at a level lower than include files is the processing of the `<special>=<newspecial>` sequence which changes the special character.

A few other facts about include files are worth mentioning here. Include files inherit the directory specification supplied using the `+I` command line option. The special character is saved at the start of each include file and restored to its previous value at the end of each include file. Include files can be nested up to ten levels. Recursive included files will always cause an infinite recursion as there is no bottoming out mechanism available. Include files must contain an integer number of lines (i.e. the last line must be terminated with an end of line marker). Once FunnelWeb has seen "`@i `" at the start of a line, it will grab the rest of the line raw and treat it as a file name. There is no place on the line for things like FunnelWeb comments (see later) or extraneous text.

Include files can be used for many purposes, but are particularly useful for hauling in macro libraries.

1.7 Typesetting Facilities

The first half of this tutorial focuses solely on the macro facilities of FunnelWeb (which it more or less covers completely). As a result, the example documents you have seen so far have been gross distortions of "normal" FunnelWeb documents which often contain as much documentation as code.[3] While there are applications where FunnelWeb can be used solely as a macro preprocessor, most applications will use its typesetting facilities as well.

This section restores the balance in this tutorial by presenting FunnelWeb's typesetting facilities.

1.7.1 Overview

The macro definitions discussed in the macro tutorial completely define the contents of the product files that FunnelWeb will generate. These macro definitions can be arranged in any order and nothing external to them can affect the contents of the product files. The macros can be thought of as a group of self-contained islands.

Although FunnelWeb will can process the macros all on their own, the full power of FunnelWeb is realized only when the macros are surrounded by a sea of documentation. This sea can take two forms: directives and free text. Some of the directives control things such as the maximum input line length. However, most of them are typesetting directives that affect the printed documentation. Thus a FunnelWeb document can be viewed as a sequence of **macro definitions**, **directives**, and **free text**.

Unlike the product files which consist of unscrambled macro calls, the documentation file is more or less a direct representation of the input file. Each part of the input file appears in the documentation file in the order in which it appears in the input file. However, each different kind of part is typeset[4] in a different manner. Macros are typeset in a particular style, with the macro body appearing in `tt font` (see some FunnelWeb printed documentation for an example). Typesetter directives have specific defined effects (more later). Free text is typeset exactly as it is, except that each block of text between blank lines is filled and justified as a paragraph.

The following example demonstrates how all this works. Type in the following as `example.fw` and run it through FunnelWeb with the command "`fw example +t`". The "`+t`" instructs FunnelWeb to generate a documentation file called `example.tex`. Run the file through TeX and print it. Examine the files `example.out` and `example.tex`.

```
You are reading some free text before the macro. Free text can consist
of any text (not containing the FunnelWeb special character) including
typesetter commands
such as $, %, #, and \TeX{} which
will be typeset to appear exactly as they do in the input file!
Look out! Here comes a macro!

@O@<example.out@>==@{@-
This text is part of
a macro definition.
@}

This is free text following the macro. This sentence contains
two @{inline@} typesetter @/directives@/.
Now here is a non-inline typesetting directive.
```

[3] As an example, the author used FunnelWeb to develop a largish computer program and found that on average his style of using FunnelWeb resulted in about 30% documentation and 70% macros (code) (measured by numbers of lines).

[4] Here the term "typeset" is used loosely to refer to FunnelWeb's generation of typesetter commands for each construct in the input file. Strictly, the term should be used only to describe the actions of a typesetter program (e.g. TeX).

```
@t new_page

This sentence will appear on the next page.
```

At the top of the `example.tex` documentation file will be a set of TeX macro definitions. The TeX code corresponding to the input above appears at the end of the file. It should look something like this.

```
You are reading some free text before the macro. Free text can consist
of any text (not containing the FunnelWeb special character) including
typesetter commands
such as \$, \%, \#, and $\backslash$TeX$\{$$\}$ which
will be typeset to appear exactly as they do in the input file!
Look out! Here comes a macro!

\fwbeginmacro
\fwfilename{example.out}{1}\fwequals \fwodef \fwbtx[This text is part of
a macro definition.
]fwetx=%
\fwcdef
\fwbeginmacronotes
\fwisafile{This macro is attached to an output file.}
\fwendmacronotes
\fwendmacro

This is free text following the macro. This sentence contains
two \fwlit{inline} typesetter \fwemp{directives}.
Now here is a non-inline typesetting directive.

\fwnewpage

This sentence will appear on the next page.
```

The following points explain the `example.tex` file.

You don't have to know TeX: If you don't know TeX, don't pay too much attention to this section. You don't need to know TeX to use FunnelWeb.

In order: FunnelWeb has merely transformed the input. It hasn't rearranged it.

Free text: Most of the free text has been simply copied over. The TeX typesetter justifies and fills all paragraphs fed to it by default, so most of the text has just been copied verbatim.

TeX codes: The characters and sequences which TeX treats as special have been neutralized in the documentation file. For example, "`$`" has become "`\$`". By default, FunnelWeb allows the user to write any text as free text and not have to worry about accidentally invoking typesetter features.

fw sequences: The `fw` sequences (e.g. `\fwbeginmacro`) invoke TeX macros defined earlier in the documentation file (and not shown here).

The macro: The macro is typeset using a set of predefined TeX macros. See the printed documentation to see what this looks like on paper.

Typesetter directives: Unlike the TeX command sequences (which were neutralized), the FunnelWeb typesetter directives turn into TeX macro calls. For example, "`@{inline@}`" became "`\fwlit{inline}`".

In summary, FunnelWeb produces typeset documentation that transforms, but does not reorder, the input file. Macros are typeset in a specific style. FunnelWeb typesetter directives have particular well-defined effects. Free text is filled and justified, but will otherwise appear in the printed documentation exactly as it appears in the input file.

1.7.2 Typesetter Independence

Although the current version of FunnelWeb can only generate documentation files in TeX form, it encourages typesetter independence by neutralizing all TeX control sequences before writing them out. The result is that you don't have worry about upsetting or depending on TeX by accidentally including some special character or sequence. By default your input file is **typesetter independent**.

This scheme differs from other literate programming tools, including all earlier versions of FunnelWeb, which copy their free text directly to the documentation file, the justification being that the programmer can use the full power of the typesetter language to describe the program. The disadvantages of doing this are first that the programmer is required to know the typesetting language and second that the input file becomes typesetter dependent. FunnelWeb avoids these problems by knobbling the free text be default.

However, FunnelWeb does provide a trapdoor for those who want their free text to be fed directly to TeX. To open the trapdoor, simply include the following pragma somewhere in your input file.

```
@p typesetter = tex
```

See Section 3.11.2 for more information.

FunnelWeb leaves the degree to which the user wishes to bind a particular document to a particular typesetter up to the user. In some cases, the extra typesetting power may compensate for the lack of portability. However, as a rule, it is best to avoid typesetter-specific commands, so as to allow your input files to be formatted at a later date for different typesetters. FunnelWeb includes a number of its own typesetter commands so as to support typesetter-independent input files. The following sections describe some of these commands. In particular, the next section describes the most powerful FunnelWeb typesetting directives which allow the user to structure the document hierarchically.

1.7.3 Hierarchical Structure

The tree structure is one of the most effective structuring tools that exists, deriving its power from the principal of divide and conquor. So effective is it that the internal organization of most technical books are tree structures which are concisely summarized in the table of contents. In contrast, computer programs are usually presented as flat sequences of text to be consumed by an anonymous compiler.

In order to bring program documentation up to the structural sophistication commonplace in technical books, FunnelWeb provides five levels of section headings implemented by the five special sequences `@A`, `@B`, `@C`, `@D`, and `@E`. These must always appear at the start of a line. `@A` is the highest level section (e.g. like LaTeX's `\chapter`) and `@E` is the lowest level section (e.g. like LaTeX's `\subsubsubsection`). Section headings can appear anywhere in the free text of a FunnelWeb input file (i.e. anywhere except inside a macro definition).

Each section heading in a FunnelWeb document has an associated name. The name of a section can be provided explicitly by supplying it delimited by `@<` and `@>` immediately after the section sequence (e.g. `@A`), or implicitly by not providing an explicit name, in which case the section takes the name of the first macro defined between the section header in question and the following section header. An error is generated if a section has not been given an explicit name and does not contain any macro definitions. Here are some example headings:

```
@A@<Feed the Penguins and Save the World@>
@B@<Feed the Penguins@>
@C@<Feed the little penguins@>
@C@<Feed the big penguins@>
@B@<Save the World@>
@C@<Save Europe@>
@C@<Save Africa@>

@C This heading hasn't been given an explicit name, but will inherit the
name \p{Save the rest of the world} from the macro definition below.

@$@<Save the rest of the world@>@Z==@{...@}
```

The feature of having unnamed sections inherit the name of the first macro defined within their scope is present because a common style of writing in FunnelWeb is to have one section per macro definition. Because, under this style, each section describes a single macro, it usually turns out that the macro name makes a good name for the section too. The inheritance mechanism prevents duplication of the name.

Apart from the requirement that each section have an explicit or implicit name and that its special sequence appear at the start of a line, the only other restriction on section headings is that a section heading at level n cannot appear immediately after a section heading at level $n-1$ or less. In other words, the hierarchy cannot be broken. For example, an `@C` cannot appear after an `@A` heading unless there is an intervening `@B` heading.

```
@A@<The Top Heading@>
@C@<Level C here is not allowed after an A and will cause an error@>
```

This rule extends to the start of the file; if there are any headings at all, the first one must be an `@A` heading. The following file, while short, is in error.

```
This FunnelWeb input file is in error because its first section heading
is at level C rather than level A.
@C@<2@>
```

1.7.4 Understanding the Printed Documentation

Type in the following file, and use FunnelWeb and TEX to generate the corresponding printed documentation.

```
@A@<Table of Contents@>

@t table_of_contents

@A@<Macros for Moral Support@>

The following macro contain comments that provide moral support in the
output code.

@$@<Programmer's Cheer@>@M==@{
-- Shift to the left!
-- Shift to the right!
-- Pop up, push down!
-- Byte! Byte! Byte!
-- (From "The New Hacker's Dictionary").
```

```
@}

The next macro is similar but is distributed throughout the program.
@$@<Hacker's Cheer@>+=@{@+-- Pointer to the left@+@}

@A@<An Extremely Imperative Stack Abstraction@>

@B@<Define the Stack@>
@$@<Hacker's Cheer@>+=@{-- Pointer to the right@+@}
@$@<Stack Type@>@Z==@{type stack = record ... end;@}

@B@<Push the Stack@>
@$@<Hacker's Cheer@>+=@{-- Hack that code@+@}
@$@<Push Procedure@>@Z==@{@-
procedure push(var b:stack; v:value); @<Programmer's Cheer@> {...}@}

@B@<Pop the Stack@>
@$@<Hacker's Cheer@>+=@{-- Tight! Tight! Tight!@+@}
@$@<Pop Procedure@>@Z==@{@-
procedure pop(var b:stack); @<Programmer's Cheer@> {...}@}

@B@<Rough the Stack Up a Bit@>
@$@<Hacker's Cheer@>+=@{-- (RNW, 04-Jan-1991).@+@}
@$@<Rough Procedure@>@Z==@{@-
procedure rough(var b:stack); @<Hacker's Cheer@> {...}@}

@O@<dummy.txt@>==@{dummy@+@}
```

An examination of the printed documentation reveals a lot about how FunnelWeb's presentation works.

First, notice how the `@t` typesetter directive at the top of the file has caused a table of contents to appear. This is one of FunnelWeb's typesetting features and is discussed in a later section. The table of contents shows that the sections have been numbered hierarchically.

Now take a look at the typeset macro definitions. Most important are the numbers in square brackets that follow each macro name. As well as numbering the headings *hierarchically*, FunnelWeb *independently* numbers the macro definitions *sequentially*. The first macro definition (for "Programmer's Cheer") is numbered 1. The second (for "Hacker's Cheer") is numbered 2 and so on. Note that it is not macros that are numbered, but macro definitions. The distinction is necessary because some macros (such as the "Hacker's Cheer" macro) are additive. It is important to realize that there is no relationship between the numbers of the headings and the numbers of the macro definitions.

Now take a look at the notes beneath the body of each macro definition. All macro definitions are followed by a note indicating the definitions in which the macro is called. Additive macros have an additional list, listing the definitions in which they are defined.

Finally, take a look at the macro *call* of "Programmer's Cheer" in section 3.2 of the printed documentation. Macro calls are set in slanted roman (so that they can be distinguished from the `tt font` code) and are followed by the number of the defining macro definition. In this case, the macro was defined in definition 1. Further down, the call to the "Hacker's Cheer" macro indicates that the macro was defined in definition 2. In fact the macro is additive and definition 2 is just the first of many definitions. To list all definitions in a call to an additive macro would be unnecessarily messy.

1.7.5 Literals and Emphasis

When writing about program code, it is often desirable to be able to indicate that a particular word or phrase be typeset in the same manner as the code being discussed. For example, one might talk about the variable `topval` or the procedure `stack_pop` and wish for them to be typeset as they are in this sentence. This, of course, is simple to do using TeX macros, but use of the (more general) FunnelWeb typesetting directives to do the same work has the added benefit of keeping the document portable to other typesetters.

FunnelWeb provides two in-text type modification constructs: `@{...@}` and `@/...@/` where ... is raw text. The `@{...@}` construct sets the enclosed text in the same manner as the text of macro definitions is set. The `@/...@/` construct emphasises its enclosed text in some typesetter-dependent fashion. Typically the emphasised text is set in italics.

Here is an example of how these constructs might be used:

```
The following procedure @{put_sloth@} writes the @{sloth@} variable to
the output file. Note: @/The output file must be opened for writing
at this point or the program will crash!@/
```

1.7.6 Adding a Header Page

FunnelWeb provides a few typesetter-independent typesetting constructs which are specifically designed for the construction of header pages. These constructs are usually best placed at the top of your input file, but can be placed anywhere the document if desired to create header pages right through. The two main restrictions on these constructs is that the `@t` must start at the start of a line (which cannot contain comments), and that the constructs cannot appear inside a macro definition. Here is what the top of an input file might look like:

```
@t vskip 40 mm
@t title titlefont centre "Hairy Wombat"
@t title titlefont centre "Simulation"
@t vskip 10 mm
@t title smalltitlefont centre "A Program in Six Parts"
@t title smalltitlefont centre "Simulating the Life of Some Hairy Wombats"
@t vskip 20 mm
@t title normalfont left "By Zqitzypbuswapzra Ypongtatoslrtzz"
@t new_page
@t table_of_contents
@t new_page
```

The `@t` at the start of each line indicates that each entire line is a typesetter directive. The `vskip` directive instructs FunnelWeb to skip some vertical space (measured in millimetres). The `title` directive instructs FunnelWeb to position a string of text on a single line of its own. Options are provided for font and alignment. The first word after `title` is the font which can be one of (in decreasing order of size) `titlefont`, `smalltitlefont`, and `normalfont`. The second word after `title` is the desired alignment of the text. The options here are `left`, `right`, and `centre`. The `new_page` directive instructs FunnelWeb to skip to a new page. Finally, the `table_of_contents` directive instructs FunnelWeb to insert a table of contents at that point in the text.

1.7.7 Comments

A FunnelWeb comment commences with the `@!` sequence and continues up to, but not including, the end of line marker at the end of the line that the comment sequence is on. Comments can be placed on any line except `@i` include, `@p` pragma, and `@t` typesetter directive lines.

The text following the FunnelWeb comment sequence `@!` will not appear in the product files or the documentation file. It is only for the eyes of those who bother to look at the original `.fw` input file. Typically FunnelWeb comments are used to describe the way in which particular FunnelWeb constructs are being used. Example:

```
@! This macro is really revolting. Please forgive me. I had to do it!
@$@<Revolt Me@>==@{@-
@#X@(@#Y@(@#Z@,@"@#Z@"@)=6@,Teapot@,@"@#Q@(45@)@"@,Tiger@)@}
```

1.8 A Complete Example

To finish off the chapter, a complete example of a FunnelWeb input file is presented. Although unrealistically short, it gives a better idea of what a typical FunnelWeb `.fw` file looks like.

```
@!---------------------------------------!
@!  Start of FunnelWeb Example .fw File  !
@!---------------------------------------!

@t vskip 40 mm
@t title titlefont centre "Powers:"
@t title titlefont centre "An Example of"
@t title titlefont centre "A Short"
@t title titlefont centre "FunnelWeb .fw File"
@t vskip 10 mm
@t title smalltitlefont centre "by Ross Williams"
@t title smalltitlefont centre "26 January 1992"
@t vskip 20 mm
@t table_of_contents

@A@<FunnelWeb Example Program@>

This  program writes  out each  of the  first @{p@}  powers of  the first  @{n@}
integers. These  constant parameters are located  here so that they  are easy to
change.

@$@<Constants@>==@{@-
n : constant natural := 10;     -- How many numbers? (Ans: [1,n]).
p : constant natural :=  5;     -- How many powers?  (Ans: [1,p]).@}

@B Here is  the outline of the  program. This FunnelWeb file  generates a single
Ada output file  called @{Power.ada@}. The main program consists  of a loop that
iterates once for each number to be written out.

@O@<Power.ada@>==@{@-
@<Pull in packages@>

procedure example is
   @<Constants@>
begin -- example
   for i in 1..n loop
      @<Write out the first p powers of i on a single line@>
   end loop;
end example;
@}
```

```
@B In this section,  we pull in the packages that this program  needs to run. In
fact, all we need is the IO package so that we can write out the results. To use
the IO package, we first of all need  to haul it in (@{with text_io@}) and then
we need to make all its identifiers visible at the top level (@{use text_io@}).

@$@<Pull in packages@>==@{with text_io; use text_io;@}

@B Here is  the bit that writes out  the first @{p@} powers of  @{i@}. The power
values  are  calculated  incrementally  in  @{ip@}  to  avoid  the  use  of  the
exponentiation operator.

@$@<Write out the first p powers of i on a single line@>==@{@-
declare
   ip : natural := 1;
begin
   for power in 1..p loop
      ip:=ip*i;
      put(natural'image(ip) & " ");
   end loop;
   new_line;
end;@}

@!----------------------------------------!
@!   End of FunnelWeb Example .fw File    !
@!----------------------------------------!
```

1.9 Summary

This chapter has provided an introduction to FunnelWeb and a tutorial that covers most of its features. FunnelWeb's functionality can be split into two parts: a macro preprocessor, and support for typesetting. The reader should be aware that the examples in this chapter, constructed as they were to demonstrate particular features of FunnelWeb, do not present a realistic picture of the best use of the tool. Only the final example of this chapter comes close. The reader should study this last example carefully and then write some real programs using FunnelWeb before proceeding to Chapter 2 which provides more advanced information. At this stage it does not particularly matter exactly how you use Funnelweb, as everyone develops their own style anyway. The important thing is to try it.

Chapter 2

FunnelWeb Hints

Whereas Chapter 1 provides an introduction to FunnelWeb and Chapter 3 a definition, *this* chapter contains hints about how FunnelWeb can be used. This chapter probably should not be read until the reader has already commenced using FunnelWeb, or at the very least, tried out some of the examples in Chapter 1. Those who find themselves using FunnelWeb frequently should read this chapter at some stage so as to ensure that they are getting the most out of it.

Most of the examples in this chapter have been placed in the FunnelWeb regression test suite which should be available in a directory called `/fwdir/tests/`. The files to examine are `hi01.fw` through `hi10.fw`.

2.1 Macro Names

When using FunnelWeb, the choice of macro names can be as important to the readability of a program as the choice of program identifiers, and it is important that the user know the range of options available.

Names are case sensitive and exact matching: Macro names are case sensitive and are matched exactly. The strings used as a macro name at the point of definition and call must be *identical* for the connection to be made.

Names can contain any printable character: FunnelWeb is less restrictive about its macro names than most programming languages are about their identifiers. A FunnelWeb macro name can contain any sequence of printable characters, including blanks and punctuation. Names can start and end with any character. Names cannot cross line boundaries. The following are all legal macro names:

```
@<This macro expands to some really bad code@>
@<@>
@<453 #$ %&# --===~~1">>>@>
@<<@>
@<<>@>
@<a b c d e f g@>
@<       !     @>
@<?? ...@>
@<"Who's been hacking MY program" said Father Bear.@>
@<Update the maximum and return for more data@>
```

Names must be no more than a maximum limit in length: Names can be no longer than a predefined maximum length. Currently this length cannot be modified.

Typically, macro names will consist of a short English phrase or sentence that describes the contents of the macro.

2.2 Quick Names

Sometimes a particular macro must be used extremely often. When this happens it is desirable to make the macro's name as short as possible. The shortest ordinary FunnelWeb macro name is the empty name "`@<@>`", which is four characters long. Single-character names are five characters long.

To cater for the cases where really short names are needed, FunnelWeb provides a **quick name** syntax that allows one-character macro names to be specified in two less characters. Quick names take the form of the special character, followed by a hash (`#`) followed by a single character. Examples:

```
@#A     @#|     @#&     @#m
```

This form of macro name has the same syntactic functionality as an ordinary name and can be substituted wherever an ordinary name can be. In fact quick names live in the same namespace as ordinary macro names. For example the quickname `@#A` is the *same name* (refers to the same macro) as the ordinary name `@<A@>`.

Because quick names look syntactically "open" (i.e. they do not have a closing`@>` as ordinary names do), it is best to avoid them except where a macro must be called very often.

2.3 FunnelWeb the Martinet

There are many ways in which a macro preprocessor can cause unexpected difficulties. FunnelWeb seeks to avoid many of these problems by performing a number of checks. This section describes some of the checks that FunnelWeb performs.

Trailing blanks in the input file: Trailing blanks are usually not dangerous, but FunnelWeb disallows them anyway. All trailing blanks in the *input* (`.fw` file) are flagged as errors by FunnelWeb. FunnelWeb does not flag trailing blanks in any of its output files.

Input line length: FunnelWeb has a maximum input line length. If FunnelWeb reads an input line longer than this length, it flags the line with an error message. The maximum length can be changed using a pragma (see Chapter 3).

Product file line length: FunnelWeb watches the length of output lines and all output lines longer than the limit are flagged with error messages. The maximum length can be changed using a pragma (see Chapter 3). That FunnelWeb polices output lines is very important. Some programs can behave very strangely if they get an input line that is too long (e.g. Fortran compilers can simply ignore text past a certain column!) and once FunnelWeb starts expanding macros using indentation, it is sometimes not obvious how wide the product file will be.

Control characters: The presence of control characters in a text file can result in some confusing behaviour downstream when the file is presented to various programs. Unfortunately, some text editors allow control characters to be inserted into the text rather too easily, and it is all too easy to be tripped up. FunnelWeb prevents these problems by flagging with diagnostics all non-end-of-line control characters detected in the input (`.fw`) file (even TABs). The result is that the user is guaranteed that product files generated from FunnelWeb contain no unintentional control characters. This said, FunnelWeb does allow the insertion of control characters in the output file by explicitly specifying them in the text using a `@^` control sequence.

Number of invocations: FunnelWeb checks the number of times that each macro is called and issues an error if the total is not one. The `@Z` (for zero) and `@M` (for many) macro attributes can be used to bypass these checks.

Recursion: Because FunnelWeb does not provide any conditional constructs, all recursively defined macros must, by definition, expand infinitely,[1] and are therefore unacceptable. FunnelWeb performs *static* checks to detect recursion, detecting it before macro expansion commences. The user need not fear that FunnelWeb will lock up or spew forth if a recursive macro is accidentally specified.

2.4 Fiddling With End of Lines

One of the fiddly aspects of programming with FunnelWeb is coping with end of lines. If you want your product file to be well indented without multiple blank lines or code run-ons, you have to spend a little time working out how the end of line markers get moved around.

The rule to remember is that, disregarding the effects of special sequences within a macro body, *the body of a macro consists of exactly the text between the opening* `@{` *and the closing* `@}`. This text includes end of line markers.

If for example you call a macro in a sequence of code...

```
while the_walrus_is_sleepy do
   begin
   writeln('zzzzzzz');
   @<Wake up the walrus@>
   writeln('Umpharumpha...');
   end;
```

where `<wake up the walrus>` is defined as follows

```
@$@<Wake up the walrus@>==@{
wake_up_the_walrus(the_walrus);
@}
```

then when `<Wake up the walrus>` is expanded you will get

```
while the_walrus_is_sleepy do
   begin
   writeln("zzzzzzz");

   wake_up_the_walrus(the_walrus);

   writeln("Umpharumpha...");
   end;
```

The blank lines were introduced by the end on line markers included in the definition of `<Wake up the walrus>`. A good solution to this problem is to suppress the end of line markers by defining the macro as follows

```
@$@<Wake up the walrus@>==@{@-
wake_up_the_walrus(the_walrus);@}
```

This is the usual form of macro definitions in FunnelWeb files.

In additive macros, this format does not work properly because the end of line that is suppressed by the trailing `@}` does not get replaced by the end of line at the end of the macro invocation. For example the definition

[1] A special case exists where there is recursion but no content. In this case, the expansion is finite (the empty string) even though the operation of expanding is infinite. FunnelWeb does not treat this case specially.

```
@$@<Wake up the walrus@>+=@{@-
wake_up_the_walrus_once(the_walrus);@}
```

later followed by

```
@$@<Wake up the walrus@>+=@{@-
wake_up_the_walrus_again(the_walrus);@}
```

is equivalent to the single definition

```
@$@<Wake up the walrus@>==@{@-
wake_up_the_walrus_once(the_walrus);wake_up_the_walrus_again(the_walrus);@}
```

Putting the trailing `@}` on a new line at the end of the macro (except for the last definition part) solves the problem.

```
@$@<Wake up the walrus@>+=@{@-
wake_up_the_walrus_once(the_walrus);
@}
```

later followed by

```
@$@<Wake up the walrus@>+=@{@-
wake_up_the_walrus_again(the_walrus);@}
```

is equivalent to the single definition

```
@$@<Wake up the walrus@>==@{@-
wake_up_the_walrus_once(the_walrus);
wake_up_the_walrus_again(the_walrus);@}
```

Managing end of line markers is tricky, but once you establish a convention for coping with them, the problem disappears into the background.

2.5 Fudging Conditionals

As a macro preprocessor, the facility that FunnelWeb most obviously lacks is a conditional facility (such as C's `#ifdef`). It might, therefore, come as a surprise to know that the first version of FunnelWeb actually had a built in conditional facility. The facility allowed the programmer to specify a construct that would select from one of a number of macro expressions depending on the value of a controlling macro expression.

In three years the construct was never used.

The reason was that conditional constructs could be fudged nearly as easily as they could be used. Because of this, the inbuilt conditional feature was removed in the current version of FunnelWeb. Not only did this simplify the program, but is also allowed recursive macros to be detected through static analysis rather than during macro expansion.

There are two basic ways to fudge a conditional. First, the comment facility of the target programming language may be employed. For example, in Ada, comments commence with "`--`" and terminate at the end of the line. Using this fact, it is easy to construct macros that can be called at the start of each target line and which turn on and off the lines so marked by defining the macro to be the empty string (ON) or the comment symbol (`--`) (OFF). For example:

```
@A@<Debug Macro@>

The following macro determines whether debug code will be included in the
program. All lines of debug code commence with a call to this macro and so
we can turn all that code on or off here by defining this macro to be either
empty or the single-line comment symbol (\p{--}). Note the use of a
quick macro name.

@$@#D@M==@{@}     @! Turns the debug code ON.
@! Use this definition to turn the debug code OFF: @$@#D==@{--@}

... then later in the file...

@$@<Sloth incrementing loop@>==@{@-
while sloth<walrus loop
   @#D assert(sloth<walrus,"AWK! sloth>=walrus!!!!!!!");
   @#D assert(timer<timermax,"AWK! timer>=timermax!!!");
   inc(sloth);
end loop@}
```

The other way to fudge a conditional is to define a macro with a single parameter. A call to the macro is then wrapped around all the conditional code in the program. The macro can then be defined to present or ignore the code of its argument. For example:

```
@A@<Debug Macro@>

The following macro determines whether debug code will be included in the
program. All debug code is wrapped by a call to this macro and so
we can turn all the debug code on or off here by defining this macro to be
either empty or its parameter.

@$@#D@(@1@)@M==@{@1@}     @! Turns the debug code ON.
@! Use this definition to turn the debug code OFF: @$@#D@(@1@)==@{@}

... then later in the file...

@$@<Sloth incrementing loop@>==@{@-
while sloth<walrus loop
   @#D@(assert(sloth<walrus,"AWK! sloth>=walrus!!!!!!!");
        assert(timer<timermax,"AWK! timer>=timermax!!!");@)
   inc(sloth);
end loop@}
```

In languages that allow multi-line comments (e.g. C with `/*` and `*/`), comments can be used to eliminate the conditioned code rather than absence. For example:

```
@$@#D@(@1@)@M==@{/* @1 */@}    @! Comments out the debug code
```

(Note: If this example were ever actually used, the programmer would have to be careful not to place comments in the argument code. Nested comments in C are non-portable.)

The parameterized macro idea can be generalized to support the choice of more than one mutually exclusive alternative. For example:

```
@A This module contains non-portable code that must execute on Hewlett
Packard, Sun, and DEC workstations. The following FunnelWeb macro is
```

```
defined to choose between these three. The first parameter is the HP code,
the second is the Sun code, and the third is the DEC code. Whichever
parameter constitutes the body of this macro determines which
machine the code is being targeted\note{Dictionary says only one t
in targeted.} for.

@$@<Machine specific code@>@(@3@)@M==@{@1@}  @! Configure for HP.

...then later in the file...

@<Machine specific code@>@(
@"get_command_line(comline)@"           @, @! HP.
@"scan_command_line(128,comline);@"     @, @! Sun.
@"dcl_get_command_line(comline,256);@"  @) @! DEC.
```

Of course, this could also be performed using three separate macros. The main advantage of using a single macro is that the mutual exclusivity is enforced. Also, because FunnelWeb ensures that the number of formal and actual parameters are the same, this method lessens the chance that a machine will be forgotten in some places.

2.6 Changing the Strength of Headings

FunnelWeb provides five heading levels: `@A`, `@B`, `@C`, `@D`, and `@E` to which it binds five different typographical strengths. These bindings are static; a level `@A` heading will always be typeset in a particular font size regardless of the size of the document. The font sizes have been preset to be "reasonable" for a range of document sizes, but may be inappropriate for very small or large documents.

FunnelWeb does not currently provide an "official" way (e.g. a pragma) to change the typesetting strength of headings. This feature might be added in later versions. Meanwhile, a hack is available that will do the job, providing that you do not mind the hack being TEX-specific and probably FunnelWeb-version specific.

Inside the set of TEX macro definitions that FunnelWeb writes at the top of every documentation file are five "library" definitions `fwliba`...`fwlibe` which provide five different typesetting strengths for headings. Near the end of the set of definitions, FunnelWeb binds these macros to five other macros `fwseca`...`fwsece` which are invoked directly in the generated TEX code to typeset the headings.

```
\def\fwseca#1#2{\fwliba{#1}{#2}}
\def\fwsecb#1#2{\fwlibb{#1}{#2}}
\def\fwsecc#1#2{\fwlibc{#1}{#2}}
\def\fwsecd#1#2{\fwlibd{#1}{#2}}
\def\fwsece#1#2{\fwlibe{#1}{#2}}
```

This means that the typesetting strength of the headings in a FunnelWeb document can be changed by redefining these macros at the top of a FunnelWeb document. For example:

```
@p typesetter = tex
\def\fwseca#1#2{\fwlibc{#1}{#2}}
```

would set `@A` headings at the same strength as the default strength of `@C` headings. The `typesetter` directive is necessary to ensure that the TEX control sequences get through to the documentation file unfiltered.

The following will tone down all headings by two levels (with the `@D` and `@E` levels being allocated the default `@E` typesetting strength because there is nothing weaker).

```
@p typesetter = tex
\def\fwseca#1#2{\fwlibc{#1}{#2}}
\def\fwsecb#1#2{\fwlibd{#1}{#2}}
\def\fwsecc#1#2{\fwlibe{#1}{#2}}
\def\fwsecd#1#2{\fwlibe{#1}{#2}}
\def\fwsece#1#2{\fwlibe{#1}{#2}}
```

These definitions affect only the headings that follow them, and so they should be placed at the top of the FunnelWeb input file.

2.7 Efficiency Notes

The following notes are worth keeping in mind when using FunnelWeb.

Memory: When FunnelWeb processes an input file, it reads the entire input file, and all the included files into memory.[2] This organization does not pose a constraint on machines with large memories, but could present a problem on the smaller machines such as the PC.

Speed: FunnelWeb is not a slow program. However, it is not particularly fast either. If the speed at which FunnelWeb runs is important to you, then the thing to keep in mind is that FunnelWeb has been optimized to deal efficiently with large slabs of text. FunnelWeb treats input files as a sequence of text slabs and special sequences (e.g. `@+`) and whenever it hits a special sequence, it has to stop and think. Thus, while a ten megabyte text slab would be manipulated as a single token, in a few milliseconds, a similar ten megabyte chunk filled with special sequences would take a lot longer. If FunnelWeb is running slowly, look to see if the input contains a high density of special sequences. This can sometimes happen if FunnelWeb is being used as a backend macro processor and its input is being generated automatically by some other program.

Macro expansion: When tangling (expanding macros), FunnelWeb never expands a macro expression into memory; it always writes it to the product file as it goes. This is a powerful fact, because it means that you can write macros containing an unlimited amount of text, and pass such macros as parameters to other macros without becoming concerned about overflowing some kind of buffer memory. In short, FunnelWeb does not impose any limits on the size of macro bodies or their expansions.

2.8 Interactive Mode

As well as having a command line interface with lots of options, FunnelWeb also provides a command language and a mode ("interactive mode") in which commands in the language can be typed interactively. The FunnelWeb command interpreter was created primarily to support regression testing, but can also be useful to FunnelWeb users.

FunnelWeb's command interpreter reads one command per line and can read a stream of commands either from a text file, or from the console. The interpreter can understand over twenty commands. See Chapter 3 for a full list. However, most of them were designed to support regression testing and will not be of use to the casual user.

The commands that are of greatest use to the casual user are:

```
!               - Comment. Ignores the whole line.
EXECUTE fn      - Execute the specified file.
```

[2] If a file is included n times, FunnelWeb keeps n copies in memory.

```
FW options      - Invoke FunnelWeb-proper once.
SET options     - Sets options.
SHOW            - Displays currently active options.
TRACE ON        - Turns command tracing ON.
QUIT            - Quits FunnelWeb.
```

To distinguish here between invocations of the FunnelWeb program and FunnelWeb runs inside the shell, we call the latter **FunnelWeb proper**. The "`FW`" command invokes FunnelWeb proper with the specified options which take the same syntax as they do on the command line. The only restriction is that none of the action options can be turned on except "`+F`" which must be turned on.

The "`SET`" command has the same syntax as the "`FW`" command except that it does not allow *any* action options to be specified. It's sole effect is to set default option values for the rest of the run.

The "`SHOW`" command displays the current default options.

By default, FunnelWeb does not echo the commands that it processes in a script. The "`TRACE ON`" command turns on such tracing.

These commands can be combined to streamline the use of FunnelWeb. For example, you might wish to create a script called `typeset.fws` to process a whole group of files.

```
trace on
!This script typesets the whole program.
! Set no listing file, no product files, but specify a documentation file
! and specify the directory into which it should be placed.
set -L -O +T/usr/ross/typeset/
fw prog1
fw prog2
fw prog3
fw prog4
```

There are a few ways in which this script can be run. The simplest is simply to specify it in the "`+X`" option of a FunnelWeb invocation. FunnelWeb shellscripts default to "<current_directory>" and "`.fws`".

```
fw +xtypeset
```

The second alternative is to enter interactive mode.

```
fw +k
```

From there, you can execute the script using:

```
execute typeset
```

Interactive mode could be very useful to those with multiple-window workstations. The user could create a window containing an interactive session of FunnelWeb, and then switch between windows, editing, and executing FunnelWeb proper and other programs.

If you find yourself using the command interpreter a lot, be sure to read about the other commands that are available in Chapter 3.

2.9 Setting Up Default Options

If you do not like FunnelWeb's default settings for its command line options, there are a number of ways in which you can change them.

Define an "alias": Use your operating system "`alias`" facility to create an alias for FunnelWeb containing the desired options. FunnelWeb processes options from left to right, so you can override these defaults later if you wish.

Create a script called "`fwinit.fws`": When FunnelWeb starts up, it executes a script called "`fwinit.fws`" if such a script exists in the current directory. You can use this fact to set options before the run of FunnelWeb proper by creating such a script and placing a single "`set`" command in it containing the desired options. The main trouble with this approach is that the options in the `set` command will be processed *after* the command line options, which means that you won't be able to override them on the command line.

For example, you might be involved more with presenting programs than with running them, and want FunnelWeb to generate a documentation file by default, but not to produce listing or product files by default. In Unix you could do this with:

```
alias fw fw -L -O +T
```

2.10 FunnelWeb and Make

The Unix `Make` program allows a set of dependencies between a set of files to be described, and then uses these dependencies to control the way in which the files are created and updated. Typically, `Make` is used to control the process of transforming a collection of source code files to one or more executable files. As the use of FunnelWeb implies an extra stage to this process, it is natural to include the transformation of `.fw` files to source code files as part of the `Make` process. This is easy to do, but the user should be aware of one aspect of FunnelWeb which can cause problems.

It is often useful, when using FunnelWeb, to create a FunnelWeb `.fw` file that generates more than one product file. That is, a single `.fw` file may have many macro definitions connected to product files so that when the FunnelWeb `.fw` file is processed by FunnelWeb, several files are created. For example, this facility has been used to great effect to place the description of an Ada package's package specification file and package body file in the same FunnelWeb `.fw` file.

The use of multiple product files, however, provokes a problem with dependencies. Suppose for example that a FunnelWeb `prog.fw` produces two product files `proc.spec` (a package specification) and `prog.body` (a package body). If the package is accessed in the way that packages normally are, it will be quite common for the programmer to want to modify the package body without modifying the program specification. So the programmer will edit the `prog.fw` file to change the package body. The result of running this through FunnelWeb will be the desired new package body file. However, FunnelWeb will also produce a new package specification product file *even though it may be identical to the previous version!* The result is that the newly created (with a recent file date) specification package file could provoke a huge remake of much of the program in which it resides.

To solve the problem, FunnelWeb includes a command line option (`D` for Delete), which when turned on (using "`+D`") causes FunnelWeb to suppress product and documentation files that are identical to the previously existing versions of the same files. For example, if, during a FunnelWeb run, a macro was connected to a product file called `x.dat`, and the macro expanded to *exactly* the same text as is contained in `x.dat` then FunnelWeb would simply *never write the product file*, the file `x.dat` would be untouched and, as a result, no further `Make` propagations would take place.

FunnelWeb implements this feature by writing each product file to a temporary file with a temporary file name. It then compares the temporary file with the target file. If the two are identical, it deletes the temporary file. If the two are different it deletes the target file and renames the temporary file to the target file.

Use of the `D` facility means that the programmer need not be punished (by extra `Make` propagations) for describing more than one product file in the same FunnelWeb file.

2.11 The Dangers of FunnelWeb

Like many tools that are general and flexible, FunnelWeb can be used in a variety of ways, both good and bad. One of the original appeals of the literate approach to programming for Knuth, the inventor of literate programming, was that it allows the programmer to describe the target program bottom up, top down, size to side, or chaotically if desired. The flexibility that this style of programming leaves much room for bad documentation as well as good documentation. Years of experience with FunnelWeb has revealed the following stylistic pitfalls which the experienced FunnelWeb user should take care to avoid.[3]

> **Spaghetti organization:** By far the worst problem that arises in connection with the literate style occurs where the programmer has used the literate tool to completely scramble the program so that the program is described and layed out in an unordered, undisciplined "stream of consciousness". In such cases the programmer may be using the literate style as a crutch to avoid having to think about structuring the presentation.
>
> **Boring organization:** At the other extreme, a program may be organized in such a strict way that it is essentially laid out in the order most "desired" by the target programming language. For example, each macro might contain a single procedure, with all the macros being called by a macro connected to a file at the top. In many cases a boring structure may be entirely appropriate, but the programmer should be warned that it is easy to slip into such a normative style, largely forgetting the descriptive structural power that FunnelWeb provides.
>
> **Poor random access:** Using FunnelWeb, it is quite possible to write programs like novels — to be read from cover to cover. Sometimes the story is very exciting, with data structures making dashing triumphs and optimized code bringing the story to a satisfying conclusion. These programs can be works of art. Unfortunately, without careful construction, such "novel-programs" can become very hard to access randomly by (say) a maintenance programmer who wishes only to dive in and fix a specific problem. If the entire program is scrambled for sequential exposition, it can be hard to find the parts relating to a single function. Somehow a balance must be struck in the document between the needs of the sequential and of the random-access reader. This balance will depend on the intended use of the program.
>
> **Too-interdependent documentation:** Sometimes, when editing a program written using FunnelWeb, one knows how to modify the program, but one is unsure of how to update the surrounding documentation! The documentation may be woven into such a network of facts that it seems that changing a small piece of code could invalidate many pieces of documentation scattered throughout the document. The documentation becomes a big tar pit in which movement is impossible. For example, if you have talked about a particular data structure invariant throughout a document, changing that invariant in a small way could mean having to update all the documentation without touching much code. In such cases, the documentation is too interdependent. This could be symptomatic of an excessibly interconnected program, or of an excessively verbose or redundant documenting style. In any case, a balance must be struck between the conversational style that encourages redundancy (by mentioning things many times) and the normalized database approach where each

[3] The fact that these faults are listed here does not mean that the author has eliminated them in his own work. Rather, it is mainly the author's own mistakes that have resulted in this list being compiled. The author immediately confesses to several of the faults listed here, most notably that of Pavlov documentation.

fact is given at only one point, and the reader is left to figure out the implications throughout the document.

Pavlov documentation: By placing so much emphasis on the documentation, FunnelWeb naturally provides slots where documentation "should" go. For example, a FunnelWeb user may feel that there may be a rather unpleasant gap between a `@C` marker and the following macro. In many cases *no* commentary is needed and the zone is better left blank rather than being filled with the kind of uninformative waffle one often finds filling the slots of structured documentation written according to a military standards (e.g. MIL-STD-2167A).[4] The lesson is to add documentation only when it adds something. The lesson in Strunk and White[**Strunk79**] (p. 23) holds for program documentation as it does for other writing: "Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all his sentences short, or that he avoid all detail and treat his subjects only in outline, but that every word tell.".

Duplicate documentation: Where the programmer is generating product files that must exist on their own within the entire programming environment (e.g. the case of a programmer in a team who is using FunnelWeb for his own benefit but must generate (say) commented Ada specification package files) there is a tendency for the comments in the target code to duplicate the commentary in the FunnelWeb text. This may or may not be a problem, depending on the exact situation. However, if this is happening, it is certainly worth the programmer spending some time deciding if one or other of the FunnelWeb or inline-comment documentation should be discarded. In many cases, a mixture can be used, with the FunnelWeb documentation referring the reader to the inline comments where they are present. For example:

```
@A Here is the header comment for the list package specification.
The reader should read these comments carefully as they define a list.
There is no need to duplicate the comments in this text.

@$@<Specification package header comments@>==@{@-
-- LIST PACKAGE
-- ============
-- * A LIST consists of zero or more ITEMS.
-- * The items are numbered 1 to N where N is the number of items in the list.
-- * If the list is non-empty, item 1 is called the HEAD of the list.
-- * If the list is non-empty, item N is called the TAIL of the list.
-- ...
@}
```

Overdocumenting: Another evil that can arise when using FunnelWeb is to over-document the target program. In some of Knuth's earlier (e.g. 1984) examples of literate programming, each variable is given its own description and each piece of code has a detailed explanation. This level of analysis, while justified for tricky tracts of code, is probably not warranted for most of the code that constitutes most programs. Such over-commenting can even have the detrimental affect of obscuring the code, making it hard to understand because it is so scattered (see "spaghetti organization" earlier). It is up to the user to decide when a stretch of just a few lines of code should be pulled to bits and analysed and when it is clearer to leave it alone.

In the case where there are a few rather tricky lines of code, a detailed explanation may be appropriate. The following example contains a solution to a problem outlined in section 16.3 of the book "The Science of Programming" by David Gries[**Gries81**]
.

[4] Note: This is not a criticism of 2167A, only of the way it is sometimes used.

```
@C@<Calculation of the longest plateau in array b@>

This section contains a solution to a problem outlined in section 16.3 of
the book @/The Science of Programming@/ by David Gries[Gries81].

@D Given a sorted array @{b[1..N]@} of integers, we wish to determine the
@/length@/ of the longest run of identically valued elements in the array.
This problem is defined by the following precondition and postcondition.

@$@<Precondition@>==@{/* Pre: sorted(b). */@}
@$@<Postcondition@>==@{@-
/* Post: sorted(b) and p is the length of the longest run in b[1..N]. */@}

@D We approach a solution to the problem by deciding to try the approach of
scanning through the array one element at a time maintaining a useful
invariant through each iteration. A loop variable
array index @{i@} is created for this purpose. The bound function is
@{N-i@}. Here is the invariant.

@$@<Invariant@>==@{@-
/* Invariant: sorted(b) and 1<=i<=N and           */
/*            p is len of longest run in b[1..i]. */@}

@D Establishing the invariant above in the initial, degenerate case is easy.

@$@<Establish the plateau loop invariant initially@>==@{i=1; p=1;@}

@D At this stage, we have the following loop structure. Note that when both
the invariant and @{i != N@} are true, the postcondition holds and the loop
can terminate.

@$@<Set p to the length of the longest plateau in sorted array b[1..N]@>==@{@-
@<Precondition@>
@<Establish the plateau loop invariant initially@>
while (i != N)
  {
   @<Invariant@>
   @<Loop body@>
  }
@<Postcondition@>
@}

@D Now there remains only the loop body whose sole task is to increase @{i@}
(and so decrease the value of the bound function) while maintaining the
invariant. If @{p@} is the length of the longest run
seen so far (i.e. in b[1..i]), then, because the array is sorted,
the extension of our array range to
@{b[1..i+1]@} can only result in an increase in @{p@} if the new element
terminates a run of length @{p+1@}. The increase can be at most 1. Because
the array is sorted, we need
only compare the endpoints of this possible run to see if it exists. This
is performed as shown below.

@$@<Loop body@>==@{i++; if (b[i] != b[i-p]) p++;@}
```

Where the code is more obvious, it is often better to let the code speak for itself.

```
@C The following function compares two C~strings and returns TRUE iff they
are identical.

@$@<Function comp@>==@{@-
bool comp(p,q)
char *p,*q;
{
 while (TRUE)
   {
    if (*p != *q  ) return FALSE;
    if (*p == '\0') return TRUE;
    p++; q++;
   }
}
@}
```

2.12 Wholistic Debugging

Surprising though it may be, FunnelWeb has a key role to play in the *debugging* of programs. Long experience in programming has led me to the concept of **wholistic debugging**. When most programmers detect a bug, their first reaction seems to be to jump into the debugger where they often spend many hours stepping through endless stretches of code and generally wasting a lot of time.

In contrast, my first reaction when I detect a bug is to realize that *the code must not be in good enough shape if such a bug can arise.* The presence of the bug is taken as symptomatic of the lack of general health of the code. If that bug occurred, why not another? In response to this realization, my reaction is not to enter the debugger, but rather to return to the original code and tend it like a garden, adding more comments, reworking the grotty bits, adding assertions, and looking for faults. In many cases, the search for faults does not even centre on the specific bug that arose, but does tend to focus on the area of code where the bug is likely to be.

The result is often that the original bug is located more quickly than it would have been had the debugger been involved. But even if it isn't, there are other benefits. A programmer who enters the debugger may stay there for hours and still not find the bug. The result is frustration and no positive gain at all. In contrast, by tending to the code, the programmer is making forward progress at all times (the code is constantly improving) even if the bug is not immediately found. At the end of ten hours, the programmer can at least feel that the code is "ten hours better", whereas the debugger freak will likely feel defeated. All this makes code tending better psychologically as well as a more efficient approach to debugging.

I call this technique wholistic debugging, for it is like the difference between conventional and wholistic medicine. Go to a conventional doctor with a headache and he might send off for head X-rays, perform allergy tests and perform many other debugging activities. Go to a wholistic doctor with the same problem and he might look to see if you are fit, assess your mental health, and ask you if your marriage is working. Both approaches are appropriate at different times. In programming, the wholistic approach is not used enough.

2.13 Examples of FunnelWeb Applications

Despite (or perhaps because of) its flexibility and simplicity, FunnelWeb can be applied to quite a number of different text processing and documenting problems. This section describes some of the more interesting real problems that FunnelWeb has solved.

2.13.1 Analyzing the Monster Postscript Header File

During my Ph.D. candidature, I determined at one point that it would be very desirable to automatically insert diagrams from the *MacDraw* program on my Macintosh into TEX `insert`ions in my thesis. This would allow diagrams to float around with the text and be printed automatically rather than having to be printed separately and stuck in with real glue. On the face of it, the problem seemed inherently solvable as the Macintosh could generate PostScript for each diagram and this PostScript could presumably be inserted into the PostScript generated using TEX.

The only trouble was that the Macintosh PostScript code for the diagrams relied on an Apple PostScript header file. This meant that the header file had to be included at the start of the TEX PostScript if the inserted PostScript for the diagrams was to work. Unfortunately, merely including the header file at the top didn't work, and it turned out that a rather detailed analysis of some parts of the Apple header file was required in order to perform the necessary surgery on the header file to make it work. This analysis was severely aggravated by the fact that the PostScript header file was virtually unreadable. Basically it was about 50K of interwoven definitions, that looked as if it had been run through a word processor. There was no way that the code could be understood clearly without some kind of reformatting. Two other aspects of the problem further complicated the analysis:

- The definitions of interest (i.e. the ones causing the problems) were scattered throughout the file.
- Many definitions could not be moved. For one or more reasons (e.g. to keep a definition within the activation of a particular dictionary `begin` and `end`) it would have been unwise to move the definitions of interest to the same point in the file.

In fact the file was so messy and complicated that, as a rule, it had to be handled with kid gloves. It would have been unwise to re-arrange the definitions or to insert comments.

To my surprise, FunnelWeb provided an unexpected solution to the problem. First I replaced all occurrences of the `@` in the header file with `@@`. Second, I placed the entire header file in a FunnelWeb macro definition connected to a product file. I then processed the file and checked to make sure that the product file was identical to the original file. By doing all this I had placed the header file under FunnelWeb control. I then left the macro definition largely untouched, but replaced the PostScript definitions of interest with FunnelWeb macro calls, moving the actual PostScript definitions into FunnelWeb macro definitions at the end of the FunnelWeb file.

```
@O@<LaserHeader.ps@>==@{@-
Unreadable Postscript code
@<Print routine@>
Unreadable Postscript code
@<Zap routine@>
Unreadable Postscript code
@}

@A This routine looks as if it does this, but really is does that,
blah, blah blah.

@$@<Print routine@>==@{@-
/print { push pop pop push turn around and jump up and down and print it} def
@}

@A This routine zaps the...

@$@<Zap routine@>==@{@-
/zap { push pop pop push turn around and jump up and down and print it} def
@}
```

Use of FunnelWeb meant that I was able to pluck out the definitions of interest (a very small part of the whole file) and collect them as a group at the end of the file where they could be studied. Because each definition was safely contained in a macro, it was possible to write a detailed commentary of each routine without fear of affecting the final PostScript code in any way at all. Once this analysis was completed, it was possible to perform surgery on the offending PostScript definitions in an extremely controlled way. In particular, the FunnelWeb input file served as a repository for all the different versions of particular routines that were tried in order to get the definitions to work. A new (**Z**ero) macro was created for each version of each definition, and a commentary of how it performed added above it.

This case demonstrates that FunnelWeb is an extremely powerful tool for dissecting and documenting cryptic text files. Through the use of macros, particular parts of the file can be isolated and discussed without affecting the final product file in any way. In the example above, only a small part of the file was analysed, the rest being left as a blob, but in the general case, a cryptic text file could be inserted into FunnelWeb and then incrementally dissected (and possibly modified) until the result is a fully documented literate program. That this can be done without affecting the actual product file demonstrates the high degree of descriptive control that FunnelWeb provides.

2.13.2 Making Ada ADTs more A

Like many modern programming languages, Ada provides mechanisms for hiding information and structure. In particular, Ada provides a **package** facility that allows the programmer to declare objects in a package definition and define them in a corresponding package body. This works well for functions and procedures. However, in the case of types, implementation issues (in particular, the need to know the size of exported types) have led the designers of Ada to force the placement of private type definitions in the definition package rather than the implementation package. This means that some implementation details are present in the package definition for all to see. While not actually dangerous (the user of the package cannot make use of the information without recourse to "Chapter 13" of the Ada Language Reference Manual[**DOD83**]), this aspect of Ada is certainly unpleasant.

During the development of some Ada programs, FunnelWeb was used to solve this problem. Instead of creating a separate file for the package specification and package body, a single FunnelWeb file was created containing two sections, one for the each package part. The "private" part of the package specification was then moved (using a FunnelWeb macro definition) to the section describing the package body. Readers who wished only to read the package specification could read only the first part, which contained a fully documented description not containing the private definition.

2.13.3 Multiple Language Systems

With the prevalence of open systems and multi-vendor computing, it is often necessary to construct systems consisting of programs written in a number of different programming languages for a number of different systems. For example, a particular functionality might be implemented by a shellscript (invoked by the user) that calls a C program that makes a network connection to a Pascal program that queries a database. Quite often all these programs must conspire closely to execute their function. In the normal case, they must be written separately. FunnelWeb allows them to be written as a whole.

By creating a single FunnelWeb file that creates many product files in different languages, the programmer can describe the interaction between the different programs in any manner desired. Furthermore, because the different product files are all created in the same "text space" (i.e. in a single FunnelWeb file), it is easy for them to share information.

For example, in one real application FunnelWeb was used to create a system for printing files on a laser printer connected to a remote Vax Unix machine from a local Vax VMS machine. The system consisted of two files: a VMS DCL command procedure to run on the local node, and a Unix shellscript to run on the remote node. The user, by giving the print command, invoked the local VMS command procedure, which in turn fired up the remote Unix shellscript. The two scripts then cooperated to transfer the files to be printed and print them.

In addition to its usual documentation powers, FunnelWeb assisted in the creation of this system in two special ways. First, it allowed pieces of code from the two different command procedures to be partially interwoven in a description of their interaction. This is just not possible with comments. Second, it facilitated the use of shared information. For example, under some conditions, each file to be printed would be renamed and copied to the remote system using a particular constant filename (e.g. "`printfile.tmp`"). FunnelWeb allowed this constant filename to be included in a single macro definition which was invoked in the definition of each of the scripts. This ensured that the two scripts used the same name.

```
@A The following macro contains the temporary file name used to allow the
two shellscripts to transfer each file to be printed.

@$@<printfile@>@M==@{printme.txt@}

@A Here are the scripts for the local VMS node and the remote UNIX node.

@O@<vmscommandprocedure.com@>==@{@-
DCL commands
copy @<printfile@> unixnode::
DCL commands
@}

@O@<unixshellscript@>==@{@-
unix commands
print @<printfile@>
unix commands
@}
```

In the case of the printing system, the entire system was described and defined in a single FunnelWeb `.fw` file. In larger systems containing many FunnelWeb `.fw` files for many different modules in many different languages, the same trick can be pulled by placing FunnelWeb macro definitions for shared values into FunnelWeb include files. For example, a suite of implementations of network nodes, with each implementation being in a different programming language for a different target machine, could all share a table of configuration constants defined in macros in a FunnelWeb include file.

In summary, FunnelWeb's macro and include file mechanisms provide a simple way for programs written in different languages to share information. This reduces redundancy between the systems and hence the chance of inconsistencies arising.

2.13.4 The Case of the Small Function

Often, when programming, there is a need for a code abstraction facility that operates at the text level. If the statement "`a:=3;`" occurs often, it may be best simply to repeat it verbatim. If a sequence of one hundred statements is repeated often, it is normal to remove the code to a function and replace the occurrences by a function call. However, in between these two extremes are cases where a particular sequence of code is long enough and appears often enough to be troublesome, but which is bound so messily to its environment as to make a function call cumbersome.

For example, the following line of statements (referring to five variables declared *local* to a function) might appear ten times in a function:

```
a=b*3.14159; c=d % 256; e=e+1;
```

Now the "normal" rule of programming says that these statements should be placed in a procedure (also called a "function" in the C programming language used in this example), but here five local variables are used. Use of a procedure (function) would result in a procedure definition looking something like:

```
void frobit(a,b,c,d,e)
float *a,b;
int *c,d;
unsigned *e;
{*a=b << 8; *c=d % 256; *e=*e+1;}
```

and a procedure call something like

```
frobit(&a,b,&c,d,&e);
```

This might be workable in a language that allowed formal parameters to be specified to be bound only to particular variables. Similarly, it might be possible to avoid the parameter list in languages that support local procedures that can access non-local variables (such as Pascal). However, in our example here, in the C programming language, these options are not available, and so we must either create a function with five parameters, or use the C macro preprocessor (the best solution). FunnelWeb provides the same macro facility for languages that do not have a built-in preprocessor.

In particularly speed-stressed applications, the programmer may be reluctant to remove code to a procedure because of the procedure-call overhead. FunnelWeb macros can help there too.

In summary, there sometimes arises in programming situations where the cost of defining a procedure is higher than the benefits it will bestow. Common reasons for this are the run-time procedure overhead and the messy binding problems caused by removing target code from its target context. FunnelWeb can help in these situations by allowing the programmer to define a text macro. This avoids all the problems and provides an additional incentive for the programmer to describe the piece of code so isolated.

2.13.5 When Comments are Bad

In the "good old days" of small machine memories and interpreted BASIC, programmers would eliminate the "`REM`" statements (comments) from their BASIC programs so as to save space and increase execution speed. Whilst this was obviously an appalling programming practice, the small memories and slow microprocessors often made this tempting, if not necessary.

Thankfully, times have changed since then, and most code is now compiled rather than interpreted. However, from time to time one still runs into an environment or situation, or special-purpose language, where comments are either unavailable (no comment feature) or undesirable. Here FunnelWeb can be used to fully document the code without resulting in any comments in the final code at all. For example:

- Comments in frequently used `.h` header files in C programs can have a significant impact on compilation speed. Often such header files are fairly cryptic and really ought to be well commented, but their authors are reluctant to.
- Comments are undesirable in PostScript header files that must be transferred repeatedly along communications channels (e.g. the Apple Macintosh LaserWriter header file).
- Interpreted programs in embedded systems.
- Hand written machine code in hex dump form could be commented.
- A programmer may wish to annotate a text data file containing lists of numbers that is to be fed into a statistical program that does not provide any comment facility for its input file.

In all these situations, FunnelWeb allows full integrated documentation without any impact on the final code.

2.13.6 Documents That Share Text

FunnelWeb is very useful when preparing multiple documents that must share large slabs of identical text that are being constantly modified.

For example someone preparing two slightly different user manuals for two slightly different audiences might want the manuals to share large slabs of text, while still allowing differences between them. The following example shows how this can be done. The code is cluttered, but this clutter would not be a problem if the lumps of text were moderately large.

```
@O@<manual1.txt@>==@{@<M1@>@+@}
@O@<manual2.txt@>==@{@<M2@>@+@}

@$@<M1@>+=@{@<T1@>@}
@$@<M2@>+=@{@<T1@>@}
@$@<T1@>@M==@{First lump of text shared by both documents.@+@}

@$@<M1@>+=@{Text for first document@+@}
@$@<M2@>+=@{Text for second document@+@}

@$@<M1@>+=@{@<T2@>@}
@$@<M2@>+=@{@<T2@>@}
@$@<T2@>@M==@{Second lump of text shared by both documents.@+@}
```

An alternative approach, which might work better in situations where there are many small differences between the two documents rather than a few large ones, is to define a macro with two arguments, one for each product file document. Write the document from top to bottom, but place all stretches that differ between the two documents in a macro call.

```
@! Set the definition of @#D to
@!    @1 to create the shareholders report.
@!    @2 to create the customers report.
@$@#D@(@2@)@M==@{@1@}

@O@<report.txt@>==@{@-
1992 ANNUAL REPORT TO @#D@(Shareholders@,Customers@)
======================@#D@(============@,=========@)
This has been a very good year for The Very Big Corporation of America.
With your help, we have been able to successfully
@#D@(@"screw the customers for every cent they have@"@,
     @"knock the shareholders into submission to bring you lower prices@"@).
With gross earnings approaching six trillion dollars, we have been able to
@#D@(@"increase dividends@"@,
     @"lower prices@"@).
We expect to have an even better year next year.
@}
```

One application where text sharing can be particularly useful is in the preparation of computer documentation containing examples. For example, a book describing a new programming language might be full of examples of small programs written in the language which the user might want to try without having to type them all in. The "default" approach of keeping a copy of the examples in the text of the book and another copy in separate files is cumbersome and error prone, because both files have to be updated whenever an example is changed. A more sophisticated approach is to store each example in a separate file, and then use the "include file" facility of the word processor to include each example in the text. This is a better solution, but suffers from a few drawbacks. First, when editing the book in a word processor, the examples in the book will not be directly accessible or visible. To see an example, the writer would have to open the file containing the example in a separate window. This could become tedious if the text contained many examples, as many texts do. Furthermore, there is a risk that some example files will be included in the wrong place. Second, because the book is dependent on the included files, the book will end up consisting of a directory of a hundred or more files instead of just a few.

An alternative solution is to construct a single FunnelWeb `.fw` file that, when processed, produces both the book file and the example files. This solution assumes that the book consists of a text file containing commands for a typesetter such as TEX.

```
@O@<Book.tex@>==@{@#B@}

@$@#B+=@{@-
The first step to learning the object oriented AdaCgol++ language is to examine
a hello world program.

\start{verbatim}
@<Ex1@>
\finish{verbatim}
@}

@$@<Ex1@>==@{read iopack@+Enter !World~! !Hello~! ex pr flu X[1]@}
@O@<Ex1.c@>==@{@<Ex1@>@}

@$@#B+=@{@-
To understand the program, think of the execution state as a plate of cheese...
@}
```

Most of the file will consist of part definitions of the additive macro `@#B`. The definition is "broken" to allow a macro definition, wherever an example appears.

The example above is a little messy because FunnelWeb does not allow macros connected to product files to be called, and it does not have text expressions that write to an product file as well as evaluating to text. Nevertheless, it presents a fairly clean solution to the problem of keeping the example programs in a computing text up to date.

2.13.7 Generics

It is well known that generics in programming languages are closely aligned with textual substitution. In fact, a good way to understand the generic facility of a new programming language is to ask oneself the question "In what way does this generic facility differ from simple text substitution?" The differences, if any, typically have to do with the difference in scoping between textual and intelligent substitution and whether the generic code is shared or copied by the implementation. In most cases the differences are quite minor.

Because generic facilities are so closely aligned with text substitution, it is possible to use FunnelWeb's parameterized macros to provide generics in programming languages that do not support generics. Simply write a FunnelWeb macro whose parameters are the parameters of the generic and whose body is the generic object.

The following FunnelWeb file gives an example of a fully worked Vax Pascal generic set package implemented using FunnelWeb parameterized macros. The package was written by Barry Dwyer of the Computer Science Department of the University of Adelaide in 1987 and was emailed to me on 11 November 1987. The generic package provides a set abstraction implemented using linked lists. Note the clever use of the instantiation parameters in type, function, and procedure names.

```
@$@<Generic Set Module@>@(@2@)==@{@-
```

```
@! @1 is the base type, @2 is the set type.
[inherit ('@1'), environment ('@2')]

module @2;

type  @2 = ^@2Record;
      @2Record = record
         Member: @1;
         Next: @2;
         end;

procedure Null@2 (var Result: @2);
begin new (Result);
Result^.Member := (- MaxInt)::@1;
Result^.Next := nil end;

function IsNull@2 (S: @2): boolean;
begin IsNull@2 := S^.Member::integer = - MaxInt end;

procedure ForEach@1 (S: @2; procedure DoIt (i: @1));
var   ThisS, NextS: @2;
begin ThisS := S;
while ThisS^.Member::integer <> - MaxInt do
   begin NextS := ThisS^.Next;
   DoIt (ThisS^.Member);
   ThisS := NextS end;
end;

function First@1 (S: @2): @1;
begin First@1 := S^.Member end;

function Is@1InSet (i: @1; S: @2): boolean;
   procedure TestEquals (j: @1);
   begin if Equal@1 (i, j) then Is@1InSet := true; end;
begin Is@1InSet := false; ForEach@1 (S, TestEquals); end;

function Includes@2 (S1, S2: @2): boolean;
var Result: boolean;
   procedure TestIfInS1 (i: @1);
   begin if Result then if not Is@1InSet (i, S1) then Result := false; end;
begin Result := true;
ForEach@1 (S2, TestIfInS1);
Includes@2 := Result end;

function Disjoint@2s (S1, S2: @2): boolean;
var Result: boolean;
   procedure TestIfInS1 (i: @1);
   begin if Result then if Is@1InSet (i, S1) then Result := false; end;
begin Result := true;
ForEach@1 (S2, TestIfInS1);
Disjoint@2s := Result end;

function Equal@2 (S1, S2: @2): boolean;
begin
Equal@2 := Includes@2 (S1, S2) and Includes@2 (S2, S1);
end;
```

```
procedure Insert@1 (i: @1; var S: @2);
var   This, Pred, Succ: @2;
begin
if not Is@1InSet (i, S) then
   begin
   Pred := nil; Succ := S;
   while Succ^.Member::integer > i::integer do begin
      Pred := Succ; Succ := Succ^.Next end;
   if Succ^.Member::integer < i::integer then begin
      new (This); This^.Next := Succ; This^.Member := i;
      if Pred <> nil then Pred^.Next := This else S := This;
      end;
   end;
end;

procedure Insert@1s (S1: @2; var S2: @2);
var   This, Pred, Succ: @2;
   procedure Add@1 (i: @1);
   begin Insert@1 (i, S2) end;
begin
ForEach@1 (S1, Add@1);
end;

procedure Remove@1 (i: @1; var S: @2);
var   Pred, This: @2;
begin
Pred := nil; This := S;
while not Equal@1 (This^.Member, i) do begin
   Pred := This; This := This^.Next end;
if Pred <> nil then Pred^.Next := This^.Next else S := This^.Next;
Dispose (This);
end;

procedure Dispose@2 (var S: @2);
var   Old: @2;
begin
while S <> nil do begin Old := S; S := S^.Next; Dispose (Old) end;
end;

end.
@}

@O@<NaryTreeSet.pas@>==@{@-
  @<Generic Set Module@>@(@"NaryTree@"@,@"NaryTreeSet@"@)@}
@O@<NaryTreeSetSet.pas@>==@{@-
  @<Generic Set Module@>@(@"NaryTreeSet@"@,@"NaryTreeSetSet@"@)@}
```

A great advantage of the approach reflected in the above example is that it allows the programmer to construct a generic object in a language that does not supply generics, *with complete typesafety.* This contrasts to the approach that might be used in a language such as C where the programmer might choose to construct a "generic" package by parameterizing a package with pointers to `void`. The resulting package is powerful but extremely untypesafe. Such a generic list package is used in the code of FunnelWeb itself and caused no end of problems, as the compiler had no way of telling if pointers to the correctly typed object were being handed to the correct list-object/function combination.

The major disadvantage of the text generic approach is that it causes the code of the generic object to be duplicated once for each instantiation. Depending on the number and size of the instantiations, this may or may not be acceptable.

Where the duplication of code is unacceptable, a hybrid approach may be taken. As in the C example, the programmer could write a single generic package using pointers to `void` or some other untypesafe mechanism. Then the programmer creates a FunnelWeb generic package whose functions do nothing more than call the functions of the untypesafe package, and whose types do nothing more than contain the types of the untypesafe package. This solution involves the use of untypesafe programming, but this is a one-off and if done carefully and correctly, the result can be a typesafe generic package involving minimal code duplication.

2.14 Summary

This chapter has described some of the finer aspects of the use of FunnelWeb. Throughout, the power and danger of FunnelWeb as a general text-rearranging preprocessor has been emphasised. FunnelWeb can be used both to make programs more readable or more obscure. It is up to the programmer to ensure that FunnelWeb is used properly.

Chapter 3

FunnelWeb Definition

3.1 Introduction

This purpose of this chapter is to provide a complete and consistent definition of the FunnelWeb input language and the behaviour of the FunnelWeb program. Usually, a chapter such as this is called a "reference manual", but this chapter is intended to go further by actually defining the language and program. This chapter takes precedence over all other chapters and all implementations of FunnelWeb. If an implementation contradicts this chapter, then the implementation is wrong.

This is the chapter that you should turn if you find yourself asking a specific question about a specific aspect of FunnelWeb. In many cases it will be convenient to access this chapter through the index.

3.2 Notation

A particular variant of EBNF (Extended Bachus Naur Form) will be used to describe the FunnelWeb syntax. In this variant, literal strings are delimited by double quotes (e.g. `"string"`), optional constructs by square brackets (e.g. `[optional]`), and constructs repeated zero or more times by braces (e.g. `{zeroormore}`). Constructs to be repeated a fixed number of times are enclosed in braces followed by a decimal number indicating the number of times to be repeated (e.g. `{sixtimes}6`). Constructs to be repeated one or more times are enclosed in braces and followed by a `+` (e.g. `{oneormore}+`). The traditional BNF "`::=`" is replaced by the visually simpler "`=`". The traditional BNF angle brackets are abandoned.

Although FunnelWeb allows the special character to be changed using the construct "<special>=", use of "<special>" to refer to FunnelWeb's special character is cumbersome and abstract. To simplify the presentation, the default special character "`@`" is used throughout this chapter to represent the special character.

3.3 Terminology

A specific terminology has arisen for dealing with FunnelWeb. Some particularly useful examples are:

Journal file: An output file containing a copy of the output sent to the user's console during an invocation of FunnelWeb. In other systems, this file is sometimes called a "log file".

Product file: An output file, generated by the Tangle component of FunnelWeb, that contains the expansion of the macros in the input file.[1]

[1] Other names considered for this were: generated file, expanded file, result file, program file, and tangle file.

A complete list of all the special FunnelWeb terminology appears in the glossary. Be sure to refer to it if any of the terms used are unclear.

3.4 An Architectural Overview

An understanding of the internals of FunnelWeb assists with understanding its operation (**Figure 3**).

During a single run, FunnelWeb reads and processes a single input file called the **input file** or the **FunnelWeb file**. The file is processed by passing it through a series of stages called **phases**. The result is that some **output files** are generated. A **journal file** is generated containing a copy of the messages that appear on the console during the FunnelWeb run. A **listing file** is created containing a summary of the run, including any error messages. A **documentation** file is generated containing typesetter commands that when fed into a typesetter program will result in printed documentation. Finally, one or more **product files** are generated containing the result of unscrambling the macro definitions of the input file.

These files need not all be generated on any particular FunnelWeb run. Whether each output file appears is controlled by command line options.

```
         .fw Input File (FunnelWeb file)
                      V
               +---------+              \
               | Scanner |              |
               +---------+              |
                      V                 |
               +--------+               |
               | Parser |               |
               +--------+               |
                      V                 |
               +----------+             >-------+------------+
               | Analyser |             |       |            |
               +----------+             |       |            |
                      V                 |       |            |
         +-----------+------------+     |       V            V
         V                        V     |       |            |
     +--------+               +-------+ |       |            |
     | Tangle |               | Weave | |       |            |
     +--------+               +-------+ /       |            |
         |                        |             |            |
         V                        V             V            V
   Product Files         Documentation File  Listing File  Journal File
```

FunnelWeb processes each input file in a sequence of phases. If an error occurs during a phase, no subsequent phases are executed.

Figure 3: FunnelWeb's processing phases.

The phases are briefly described below.

> **The Scanner** reads the input file, expands and reads in include files, scans the input stream, processes pragmas and typesetter directives, and parses all the FunnelWeb special sequences. The result is a list of tokens that is handed to the parser.
>
> **The Parser** reads the scanner's token list and parses it, constructing a document list and a macro table. which are passed to later phases.

The Analyser examines the macro table generated by the parser and performs a number of checks of the macro structures that the parser could not make on its single pass. For example, the analyser detects and flags unused macros and recursive macros. The analyser forms the final stage of FunnelWeb's front-end processing.

Tangle expands certain macros in the macro table to generate one or more product files.

Weave uses the document list to generate a documentation file.

A single run through these phases constitutes a single invocation of **FunnelWeb proper**. Most invocations of the **FunnelWeb program** will consist only of a single execution of FunnelWeb proper. However, FunnelWeb also provides a command shell that provides many useful commands, including a command to invoke FunnelWeb proper. Discussion of the command shell is deferred until Section 3.15.

3.5 Diagnostics

During execution, FunnelWeb proceeds cautiously with each of its phases, only proceeding with the next phase if the previous phase has been successful. This means that, when debugging a FunnelWeb file, you may find that the number of errors *increases* after you fix some of them, as you will be exposing yourself to the next FunnelWeb phase.

FunnelWeb employs five levels of diagnostics at different levels of severity. Severity is defined in terms of the level of activity at which the diagnostic causes FunnelWeb to abort.

Warning: A warning does not cause FunnelWeb to terminate or curtail its operation in any way, but serves merely to warn the user of particular conditions that might be symptomatic of deeper problems.

Error: An error causes FunnelWeb to terminate processing of the current input file at the end of the current phase. For example, if an error occurs during scanning, FunnelWeb will continue scanning (and possibly generate further scanning diagnostics), but will not invoke the parser.

Severe Error: A severe error (or "severe" for short) is the same as an error except that FunnelWeb terminates the current phase immediately.

Fatal Error: A fatal error causes FunnelWeb not only to terminate the current phase and run immediately, but also to terminate total FunnelWeb processing immediately. A severe error will not cause a FunnelWeb script to terminate, but a fatal error will. A fatal error causes FunnelWeb to return control to the operating system.

Assertion Error: An assertion error occurs if FunnelWeb detects an internal inconsistency, in which case FunnelWeb terminates immediately and ungracefully. Such an error can occur only if there are bugs in FunnelWeb. With luck, such errors will be extremely rare.

FunnelWeb indicates the level of severity of each diagnostic that it issues by starting each diagnostic either with the full name of the severity level or with just the first letter of the severity level followed by a colon.

FunnelWeb conveys the presence or absence of diagnostics at the operating system level by returning `EXIT_SUCCESS` status if no diagnostics occurred during the run and `EXIT_FAILURE` status if one or more diagnostics (including warnings) occurred during the run.[2]

3.6 Typesetter Independence

One of the design goals of FunnelWeb was to provide a *target-language* independent literate programming system. This goal has been achieved simply by treating the text written to the product

[2]From the symbols of the ANSI standard C library `stdlib.h`. See **[Kernighan88]**, p.252.

file as homogeneous and typesetting it in `tt font`. A secondary goal was to provide a *typesetter* independent literate programming system. By this is meant that it be possible to create FunnelWeb input files that do not contain typesetter-specific commands. To a lesser extent this goal has also been achieved.

The difficulty with providing typesetter-independent typesetting is that each desired typesetting feature must be recreated in a typesetter-independent FunnelWeb typesetting construct that FunnelWeb can translate into whatever typesetting language is being targeted by Weave. Taken to the extreme, this would result in FunnelWeb providing the full syntactic and semantic power of TEX, but with a more generic, FunnelWeb-specific syntax. This was unfeasible in the time available, and undesirable as well.

The compromise struck in the FunnelWeb design is to provide a set of primitive typesetter-independent typesetting features that are implemented by FunnelWeb. These are the **typesetter directives**. If the user is prepared to restrict to these directives, then the user's FunnelWeb document will be both target-language and typesetter independent. However, if the user wishes to use the more sophisticated features of the target typesetting system, the user can specify the typesetter in a "`typesetter`" pragma and then place typesetter commands in the free text of the FunnelWeb document where they will be passed verbatim to the documentation file. The choice of the trade-off between typesetter independence and typesetting power is left to the user.

This said, experience with FunnelWeb V1 over a three year period, indicates that the typesetting facilities provided by FunnelWeb are sufficient for most documentation.

3.7 Command Line Interface

3.7.1 Invoking FunnelWeb

When a user invokes FunnelWeb at the operating system command level, the user must provide a command line instructing FunnelWeb what to do. Typically an operating system command line consists of a *verb* indicating that a particular program should be run, followed by a list of options. For example:

```
$ rename file1 file2
```

In this case, the verb is `rename` and the command line options are `file1 file2`. The entire command line begins with the `$` and ends with the `2`.

Operating systems differ greatly in the depth with which they process their command lines, ranging from systems that simply pass the entire command line string to the invoked program (e.g. MSDOS) through to systems that perform complete command line parsing (e.g. VMS). Syntax conventions vary considerably.

So as to achieve maximum portability and consistency of invocation across different platforms, FunnelWeb reads its command line as a raw string and performs all its own parsing. This is portable because, at the very least, all operating systems allow invoked programs access to the raw command line.

The command verb used to invoke FunnelWeb should be "`fw`".

```
FunnelWeb_verb =  "fw"
```

If this verb is not available, some alternatives are "`funweb`", "`fun`", and "`funnelweb`". The verbs `web` or `fweb` should be avoided as they are the names of other literate programming systems.

3.7.2 Command Line Arguments

Following the verb is the body of the command line which FunnelWeb parses into zero or more **arguments** separated by runs of one or more blanks.

```
FunnelWeb_command_line = FunnelWeb_verb {{" "}+ argument}
```

Because some operating systems convert their command line to upper case before handing it to the invoked program, FunnelWeb has been constructed so as to be *insensitive* to the case of its command line arguments. However, when dealing internally with arguments, FunnelWeb *preserves* the case of its command line arguments so that it will be able to operate with operating systems (such as Unix) whose file names are case dependent.

A valid FunnelWeb argument consists of a **sign**, an identifying **letter**, and an optional **string** with no spaces separating them.

```
argument  = sign id_letter [non_blank_string]
sign      = "+" | "-" | "="
id_letter = "B" | "C" | "D" | "F" | "H" |"I" | "J" | "K" |
            "L" | "O" | "Q" | "S" | "T" | "W" | "X"
```

In addition there is a special form of argument that does not begin with a sign.

```
argument = non_blank_string_not_beginning_with_+_=_or_-
```

This form is exactly equivalent to the same string with "`+F`" prepended to it.

The semantic effect of these arguments is defined in terms of **options** which are the internal parameters of FunnelWeb and which correspond closely with the set of legal command line arguments. FunnelWeb has a predefined set of options each identified by an identifying letter having two attributes: a *string*, and a *boolean*. The boolean determines whether an option is turned *on* or *off*. The string contains additional information depending on the option.

When FunnelWeb starts up, its options have predefined default values. FunnelWeb then parses its command line sequentially from left to right executing the effect of each argument on the argument's corresponding option. The sign and the string components of the argument are processed *independently*. A sign of `+` turns the option on. A sign of `-` turns the option off. A sign of `=` leaves the option's boolean attribute unchanged. The argument string replaces the string of the corresponding option, unless the argument string is empty, in which case the option string is not changed.

Because FunnelWeb processes its command line arguments from left to right, a later argument can cancel the effect of an earlier one. For example `fw +t -t` will result in the `t` option ending up *off*. This allows users to set up their own default arguments by defining a symbol in their operating system's command language. For example, a Unix user who wants FunnelWeb to delete all identical output files and create a documentation file on each run with a default `.typ` extension could simply place the following definition in their "`.login`" file.

```
alias fw fw +d +t.typ
```

These default options can then later be easily overridden on the command line.

3.7.3 Options

FunnelWeb's options are internal parameters which can be modified by corresponding arguments on FunnelWeb's command line. A description of each argument and option follows.

B1...B6: Tracedumps: These six options have been provided to assist in the debugging and testing of FunnelWeb. They determine which of six possible trace dumps are to be written to the listing file. Only the boolean attributes of these options are ever used. The six dumps are identified by the digits `1..6` as follows:

1. Dump a hexdump of each mapped input and include file.
2. Dump the global line list created by the scanner.
3. Dump the token list created by the scanner.
4. Dump the macro table created by the parser.
5. Dump the document list created by the parser.
6. Dump a table summarizing CPU and real time usage.

Because these options are so closely related, a hack has been pulled to enable them to all to be controlled by the `B` argument. The string argument to the `B` argument determines which of the six options are to be affected by the sign. Examples: `+B134` turns on options `B1`, `B3`, and `B4`. `-B1` turns off option `B1`. `Default: -B123456`.

B7: Determinism: If the `B7` option is turned on, FunnelWeb suppresses the output of anything non-deterministic, or machine dependent. This assists in regression testing. Only the boolean attribute is used in this option. This option is controlled by the `B7` argument which falls under the same argument syntax as the other `B` options. Examples: `+B7`, `-B7`. `Default: -B7`.

C: Listing File Context: The `C` option is always turned on and cannot be turned off. Its only attribute is a number which determines the number of lines of context that the lister will place around lines flagged with diagnostics in the listing file (if a listing file is written). A value of 100 indicates infinite context which means that the entire listing file will be written out if a single diagnostic occurs. The value of this number can be specified by specifying it as a string of decimal digits to the `+C` argument. Examples: `+C100`, `+C10`. `Default: +C2`.

D: Delete Identical Output Files: Only the boolean attribute of this option is used. When turned on, the option causes the suppression (deletion) of product files and documentation files (but not listing or journal files) that are identical to the currently existing files of the same name. For example, if FunnelWeb is instructed to generate `stack.h` as an product file, and the text to be written to `stack.h` is identical to the currently existing `stack.h`, then FunnelWeb will simply not write any product file, leaving the currently existing `stack.h` as it is (and in particular leaving the file's date attribute the same). This prevents unnecessary `make` propagations. For example, in a C program, if `stack.fw` is a FunnelWeb input file that generates `stack.h` and `stack.c`, a modification to `stack.fw` that affects `stack.c` but does not affect `stack.h` will not provoke the recompilation of modules that `#include stack.h`, so long as the intervening FunnelWeb run has `+D` set. Examples: `-D`, `+D`. `Default: -D`.

F: FunnelWeb Input File: If this option is turned on, FunnelWeb processes the input file whose name is specified by the option string. Examples: `+Fsloth.fw`, `+Fwalrus`, `-F`. `Default: -F`.

H: Display Help Message: If this option is turned on, FunnelWeb displays the message specified by the argument string. Each message has a name. The main help message is called "`menu`" and contains a list of the other help messages. Examples: `+Hregistration`, `+Hoptions`. `Default: -Hmenu`.

I: Include default file specification: This option is always turned on and cannot be turned off. Its string attribute is used as the default file specification for include files. Usually this option is used to specify a directory from which include files should be obtained. Examples: `=I/usr/dave/includes/`. `Default: +I`.

J: Journal File: If this option is turned on, FunnelWeb generates a journal file. A journal file contains a log of all the console input and output to FunnelWeb during a single invocation of the FunnelWeb program (Note: The `Q` option does not affect this.). The journal file is particularly useful for examining what happened during a FunnelWeb shell run. The string attribute is the name of the journal file. Examples: `+Jjournfile`, `-J`. `Default: -J`.

K: Keyboard: If this option is turned on, FunnelWeb enters an interactive mode in which the user can enter FunnelWeb shell commands interactively. The string attribute is unused. Examples: `+K`, `-K`. `Default: -K`.

L: Listing File: If this option is turned on, FunnelWeb generates a listing file containing a summary of a run on FunnelWeb proper. The string argument is the name of the listing file to be created. Examples: `+L`, `-L`, `+Llisting.lis`. `Default: -L`.

O: Product Files: If this option is turned on, FunnelWeb generates a product file for each macro in the input file that is bound to an output file. The string attribute contributes to the name of the product files. This option is controlled by the `O` argument because product files used to be called "**O**utput files"). Examples: `-O`, `+O/usr/dave/product/`. `Default: +O`.

Q: Quiet: If this option is turned on, FunnelWeb suppresses all output to the screen (standard output) unless one or more errors occur, in which case a single line summarizing the errors is sent to standard output at the end of the run. If this option is turned off, FunnelWeb writes to the console in its normal garrulous way. The string attribute is unused in this option. Examples: `-Q`, `+Q`. `Default: -Q`.

S: Screen: If this option is turned on, FunnelWeb writes all diagnostics to the screen (standard output) as well as to the listing file. By default, they are sent only to the listing file. This option has a single numerical attribute that can be specified as a decimal string in the string component of the `S` argument. The number is the number of lines of context that should surround each diagnostic sent to the screen. Examples: `-S`, `+S6`, `+S0`. `Default: -S`.

T: Documentation file: If this option[3] is turned on, FunnelWeb generates a documentation file in TeX format. The string argument contributes to the name of the documentation file to be created. By default this option is turned off, as experience has shown that most FunnelWeb runs are made during program development; documentation runs occur far more rarely. Examples: `-T`, `+Tsloth.tex`. `Default: -T`.

W: Width of Product Files: If this option is turned on, a limit is placed on the length of lines in product files generated during the run. Lines that breach the limit are flagged with error messages. This option has a single numerical attribute that can be specified as a decimal string in the string component of the `W` argument. The number is the specified maximum width. This option is one of two limits that are placed on the width of product files. The other limit is an attribute of the input file that defaults to 80 characters, but can be raised or lowered using an output line length pragma. The width that is enforced is the lower of this value and the value of the `W` option (if turned on). Examples: `-W`, `+W100`. `Default: -W80`.

X: Execute: If this option is turned on, FunnelWeb executes the FunnelWeb shell script file specified by the string attribute. Examples: `+Xmaster`, `-X`. `Default: -X`.

[3] This option is controlled by the `T` command line argument because documentation files used to be called typesetter files.

3.8 File Name Inheritance

During a single run of FunnelWeb, FunnelWeb can produce many different output files. As it would be very tedious to have to specify the name of each of these files explicitly each time FunnelWeb is run, FunnelWeb provides a system of defaults that allows the user to specify the minimum required to successfully complete the run. To do this FunnelWeb allows file specifications to inherit fields from one another.

FunnelWeb structures filenames into three fields which are inherited independently. The fields are: **directory**, **name**, and **extension**. On systems having other fields (e.g. *network node*, *device name*), the extra fields are considered to be part of the directory field. Version numbers are ignored. A field can inherit a value if its current value is the empty string.

The following table gives the full inheritance scheme used in FunnelWeb.

Script	Input	Include	Journal	List	Document	Product
		`@i`				`@o`
`+x`	`+f`	`+i`	`+j`	`+l`	`+t`	`+o`
"`.fws`"	"`.fw`"	"`.fwi`"	"`.jrn`"	"`.lis`"	"`.tex`"	
		`+f`	`+f`	`+f`	`+f`	
DefDir	Defdir	Defdir	Defdir	Defdir	Defdir	Defdir

The table is arranged with items of highest priority at the top. The "`+<letter>`" cells refer to the file specification supplied in the given command line argument. "`+F`" is the name of the input file. "Defdir" refers to the default directory specification provided by the operating system. Empty cells do not contribute.

The following example shows how the table is used. Suppose that the user invoked FunnelWeb as follows:

```
fw /usr/ross/work/sloth.fw +twalrus
```

To work out what the documentation file should be called, FunnelWeb starts with the empty string and then works down the Document column of the table. The top entry is empty so we ignore it and proceed to the second entry which consists of "`+T`". The user specified the string "`walrus`" as the value of this option, and as our current (empty) string does not have a name field, we insert the string "`walrus`" into the name field, resulting in the string "`walrus`". Moving down to the next row, we encounter the constant string "`.tex`". This string consists of an empty directory and name field, but a "`.tex`" file extension. As our current string "`walrus`", does not already have a file extension (i.e. the file extension field of our current string is empty), we add in "`.tex`", resulting in the string "`walrus.tex`". Next we encounter the "`+F`" field which is the input filename "`/usr/ross/work/sloth.fw`" consisting of a directory field "`/usr/ross/work/`", a name field "`sloth`", and a file extension field "`.fw`". Our "`walrus.tex`" string already has name and file extension fields, but its directory field is empty, and so we add in the directory field from the input file specification, resulting in the string "`/usr/ross/work/walrus.tex`". Finally, we hit the default directory specification, which is (say) "`/usr/ross/play/`". However, as the directory field of our walrus string is already full, it has no effect.

In general, there is no need to remember the exact details of FunnelWeb's filename inheritance. The important thing is to know that it exists, and to use it.

3.9 FunnelWeb Startup

FunnelWeb's command line options can be divided into two groups. **Action options** instruct FunnelWeb to performs some sort of independent action such as processing a file. **Ordinary options** merely modify the way in which FunnelWeb executes the actions.

The four action options are: `+F`, `+K`, `+X`, and `+H`. For FunnelWeb to be successfully invoked, at least one action option must be specified. If zero action options are specified, FunnelWeb terminates with failure status. If more than one action option is specified, FunnelWeb performs the specified actions in a predefined order.

Assuming that the user has specified at least one action, the order in which actions are executed is as follows:

Initialization script: FunnelWeb starts by looking in the current directory for a file called "`fwinit.fws`". If it doesn't find one, it doesn't raise any error. If it does find one, it executes it as a FunnelWeb shellscript. Initialization scripts are useful for setting up FunnelWeb options (e.g. using the "`set`" command without having to type them each time).

Execute argument script: If a shellscript has been specified using the "`+X`" option, FunnelWeb executes it.

Process input file: If the user has specified an input file using the "`+F`" option, then this is processed next (by FunnelWeb proper).

Display help message: If the user requested, using the "`+H`" option, that a help message be displayed, the message is displayed at this time.

Interactive mode: If the user specified the "`+K`" option, FunnelWeb enters interactive (keyboard) mode.

FunnelWeb processes these actions in the above order regardless of the order in which they appear on the command line.

It may be hard to see how some of these actions might be combined. Nevertheless, FunnelWeb allows this. For example, a user might wish to process a batch of files as specified in a script ("`+Xscript.fws`"), be reminded of the interactive commands available ("`+Hcommand`"), and then enter interactive mode so as to be able to reprocess files for which FunnelWeb reported errors (after correcting the errors in a different workstation window).

3.10 Scanner

The scanner reads in the input file and produces a list of tokens which it hands onto the parser. In addition, some input constructs may cause the scanner to modify some of FunnelWeb's options.

3.10.1 Basic Input File Processing

In order to read in an input file or include file, the scanner calls a submodule called the **mapper** that reads a file in and creates a contiguous copy of it in memory. The scanner then performs three checks on the file, the first (file termination) of which is performed before scanning commences, and the other two of which take place during scanning before each line is scanned.

File Termination: The first check the scanner makes is whether the file is terminated properly. A file is considered to be properly terminated if it either contains no lines, or if the last line in the file is terminated by an end-of-line marker. If the scanner detects that an input file is not properly terminated, it adds an end-of-line marker itself (to the copy in memory only).

Unprintable Characters: The second check the scanner makes is for unprintable characters (ASCII 0–31 and 127–255 (except for EOL(10))) which it flags as errors and replaces by question marks.

Line Lengths: The third check the scanner makes is input line length. When FunnelWeb starts up, a default maximum input line length of 80 is set. This can be changed dynamically during scanning using a `@p maximum_input_line_length` pragma. If the number of characters on a line (not including the end of line marker) exceeds this limit, FunnelWeb generates an error.

3.10.2 Special Sequences

The scanner scans the input file from top to bottom, left to right, treating the input as ordinary text (to be handed directly to the parser as a text token) unless it encounters the **special character**[4] which introduces a **special sequence**. Thus, the scanner partitions the input file into ordinary text and special sequences.

```
input_file = {ordinary_text | special_sequence}
```

Upon startup, the special character is `@`, but it can be changed using the <special>`=`<new_special> special sequence. Rather than using <special> whenever the special character appears, this document uses the default special character "`@`" to represent the current special character. More importantly, FunnelWeb's error messages all use the default special character in their error messages even if the special character has been changed.

An occurrence of the special character in the input file introduces a special sequence. The kind of special sequence is determined by the character following the special character. Only printable characters can follow the special character.

The following list gives all the possible characters that can follow the special character, and the legality of each sequence. The first column gives the ASCII number of each ASCII character. The second column gives the special sequence for that character. The next column contains one of three characters: "`-`" means that the sequence is illegal. "`S`" indicates that the sequence is a **simple sequence** (with no attributes or side effects) that appears exactly as shown and is converted directly into a token and fed to the parser. Finally, "`C`" indicates that the special sequence is complex, possibly having a following syntax or producing funny side effects.

```
ASC  SEQ  COMMENT
-----------------
000       \
016        | Unprintable characters and hence illegal specials.
031       /
032  @    - Illegal (space).
033  @!   C Comment.
034  @"   S Parameter delimeter.
035  @#   C Short name sequence.
036  @$   S Start of macro definition.
037  @%   - Illegal.
038  @&   - Illegal.
039  @'   - Illegal.
040  @(   S Open parameter list.
041  @)   S Close parameter list.
042  @*   - Illegal.
043  @+   C Insert newline.
044  @,   S Parameter separator.
045  @-   C Suppress end of line marker.
046  @.   - Illegal.
047  @/   S Open or close emphasised text.
048  @0   - Illegal.
049  @1   S Formal parameter 1.
050  @2   S Formal parameter 2.
051  @3   S Formal parameter 3.
052  @4   S Formal parameter 4.
```

[4] This sort of character is often referred to as the "escape character" or the "control character" in other systems. However, as there is great potential to confuse these names with the "escape" character (ASCII 27) and ASCII "control" characters, the term "special" has been chosen instead. This results in the terms *special character* and *special sequence*.

```
053  @5   S Formal parameter 5.
054  @6   S Formal parameter 6.
055  @7   S Formal parameter 7.
056  @8   S Formal parameter 8.
057  @9   S Formal parameter 9.
058  @:   - Illegal.
059  @;   - Illegal.
060  @<   S Open macro name.
061  @=   C Set special character.
062  @>   S Close macro name.
063  @?   - Illegal. Reserved for future use.
064  @@   C Insert special character into text.
065  @A   S New section (level 1).
066  @B   S New section (level 2).
067  @C   S New section (level 3).
068  @D   S New section (level 4).
069  @E   S New section (level 5).
070  @F   - Illegal.
071  @G   - Illegal.
072  @H   - Illegal.
073  @I   C Include file.
074  @J   - Illegal.
075  @K   - Illegal.
076  @L   - Illegal.
077  @M   S Tag macro as being allowed to be called many times.
078  @N   - Illegal.
079  @O   S New macro attached to product file. Has to be at start of line.
080  @P   C Pragma.
081  @Q   - Illegal.
082  @R   - Illegal.
083  @S   - Illegal.
084  @T   C Typesetter directive.
085  @U   - Illegal.
086  @V   - Illegal.
087  @W   - Illegal.
088  @X   - Illegal.
089  @Y   - Illegal.
090  @Z   S Tags macro as being allowed to be called zero times.
091  @[   - Illegal. Reserved for future use.
092  @\   - Illegal.
093  @]   - Illegal. Reserved for future use.
094  @^   C Insert control character into text
095  @_   - Illegal.
096  @`   - Illegal.
097  @a   \
109  @m    | Identical to @A..@Z.
122  @z   /
123  @{   S Open macro body/Open literal directive.
124  @|   - Illegal.
125  @}   S Close macro body/Close literal directive.
126  @~   - Illegal.
127 to 255 are not standard printable ASCII characters and are illegal.
```

The most important thing to remember about the scanner is that *nothing happens unless the special character is seen.* There are no funny sequences that will cause strange things to happen. The best way to view a FunnelWeb document at the scanner level is as a body of text punctuated by special sequences that serve to structure the text at a higher level.

The remaining description of the scanner consists of a detailed description of the effect of each complex special sequence.

3.10.3 Setting the Special Character

The special character can be set using the sequence <special>=<newspecialchar>. For example, `@=#` would change the special character to a hash (`#`) character. The special character may be set to any printable ASCII character except the blank character (i.e. any character in the ASCII range $[33, 126]$). In normal use, it should not be necessary to change the special character of FunnelWeb, and it is probably best to avoid changing the special character so as not to confuse FunnelWeb readers conditioned to the `@` character. However, the feature is very useful where the text being prepared contains many `@` characters (e.g. a list of internet electronic mail addresses).

3.10.4 Inserting the Special Character into the Text

The special sequence <special>`@` inserts the special character into the text as if it were not special at all. The `@` of this sequence has nothing to do with the current special character. If the current special character is `P` then the sequence `P@` will insert a `P` into the text. Example: `@@#@=#@#@#=@@@` translates to `@#@#@`.

3.10.5 Inserting Arbitrary Characters into the Text

While FunnelWeb does not tolerate unprintable characters in the input file (except for the end of line character), it does allow the user to specify that unprintable characters appear in the product file. The `@^` sequence inserts a single character of the user's choosing into the text. The character can be specified by giving its ASCII number in one of four bases: binary, octal, decimal, and hexadecimal. Here is the syntax:

```
control_sequence = "@^" char_spec
char_spec        = binary | octal | decimal | hexadecimal
binary           = ("b" | "B")             "(" {binary_digit}8  ")"
octal            = ("o" | "O" | "q" | "Q") "(" {octal_digit}3   ")"
decimal          = ("d" | "D")             "(" {decimal_digit}3 ")"
hexadecimal      = ("h" | "H" | "x" | "X") "(" {hex_digit}2     ")"
binary_digit     = "0" | "1"
octal_digit      = binary_digit | "2" | "3" | "4" | "5" | "6" | "7"
decimal_digit    = octal_digit | "8" | "9"
hex_digit        = decimal_digit | "A" | "B" | "C" | "D" | "E" | "F"
                                 | "a" | "b" | "c" | "d" | "e" | "f"
```

Example:

```
@! Unix Make requires that productions commence with tab characters.
@^D(009)prog.o <- prog.c
```

Note that the decimal "`9`" is expressed with leading zeros as "`009`". FunnelWeb requires a fixed number of digits for each base. Eight digits for base two, three digits for base ten, three digits for base eight and two digits for base sixteen.

FunnelWeb treats the character resulting from a `@^` sequence as ordinary text in every sense. If your input file contains many instances of a particular control character, you can package it up in a macro like any other text. In particular, quick names can be used to great effect:

```
@! Unix "Make" requires that productions commence with tab characters.
@! So we define a macro with a quick name as a tab character.
$@#T@{@^D(009)@}
@! And use it in our productions.
@#Tprog.o <- prog.c
@#Ta.out <- prog.o
```

Warning: If you insert a Unix newline character (decimal 10) into the text, FunnelWeb will treat this as an end of line sequence regardless of what the character sequence for end of line is on the machine upon which it is running. Unix EOL is FunnelWeb's internal representation for end of line. Thus, in the current version of FunnelWeb, inserting character 10 into the text is impossible unless this also happens to be the character used by the operating system to mark the end of line.

3.10.6 Comments

When FunnelWeb encounters the `@!` sequence during its left-to-right scan of the line, it throws away the rest of the line (including the EOL) without analysing it further. Comments can appear in any line except "`@i`", "`@t`", and "`@p`" lines.

FunnelWeb comments can be used to insert comments into your input file that will neither appear in the product files nor in the documentation file, but will be solely for the benefit of those reading and editing the input file directly. Example:

```
@! I have used a quick macro for this definition as it will be used often.
@$@#C@{--@}
```

Because comments are defined to include the end-of-line marker, care must be taken when they are being added or removed within the text of macro bodies. For example the text fragment

```
for (i=0;i<MAXVAL;i++)      @! Print out a[0..MAXVAL-1].
   printf("%u\n",a[i]);
```

will expand to

```
for (i=0;i<MAXVAL;i++)         printf("%u\n",a[i]);
```

This problem really has no solution; if FunnelWeb comments were defined to omit the end of line marker, the expanded text would contain trailing blanks! As it is, FunnelWeb comments are designed to support single line comments which can be inserted and removed as a line without causing trouble. For example:

```
@! Print out a[0..MAXVAL-1].
for (i=0;i<MAXVAL;i++)
   printf("%u\n",a[i]);
```

If you want a comment construct that does not enclose the end of line marker, combine the insert end of line construct `@+` with the comment construct `@!` as in

```
for (i=0;i<MAXVAL;i++)      @+@! Print out a[0..MAXVAL-1].
   printf("%u\n",a[i]);
```

FunnelWeb comments should really only be used to comment the FunnelWeb constructs being used in the input file. Comments on the target code are best placed in comments in the target language or in the documenting text surrounding the macro definitions. In the example above, a C comment would have been more appropriate.

3.10.7 Quick Names

FunnelWeb provides a **quick name** syntax as an alternative, for macros whose name consists of a single character, to the angle bracket syntax usually used (e.g. `@<Sloth@>`). A quick name sequence consists of `@#`x where x, the name of the macro, can be any printable character except space.

```
quick_name = "@#" non_space_printable
```

The result is identical to the equivalent ordinary name syntax, but is shorter. For example, `@#X` is equivalent to `@<X@>`. This shorter way of writing one-character macro names is more convenient where a macro must be used very often. For example, the macro calls in the following fragment of an Ada program are a little clumsy.

```
@! Define @<D@> as "" to turn on debug code and "--" to turn it off.
@$@<D@>@{--@}
@<D@>assert(b>3);
@<D@>if x>7 then write("error") end if
```

The calls can be shortened using the alternative syntax.

```
@! Define @#| as "" to turn on debug code and "--" to turn it off.
@$@#|@{--@}
@#|assert(b>3);
@#|if x>7 then write("error") end if
```

3.10.8 Inserting End of Line Markers

An end of line marker/character can be inserted into the text using the `@+` sequence. This is exactly equivalent to a real end of line in the text at the point where it occurs. While this feature may sound rather useless, it is very useful for laying out the input file. For example, the following input data for a database program

```
Animal = Kangaroo
Size   = Medium
Speed  = Fast

Animal = Sloth
Size   = Medium
Speed  = Slow

Animal = Walrus
Size   = Big
Speed  = Medium
```

can be converted into

```
Animal = Kangaroo  @+Size = Medium  @+Speed = Fast    @+
Animal = Sloth     @+Size = Medium  @+Speed = Slow    @+
Animal = Walrus    @+Size = Big     @+Speed = Medium  @+
```

which is easier to read, and more easily allows comparisons between records.

3.10.9 Suppressing End of Line Markers

End of line markers can be suppressed by the `@-` sequence. A single occurrence of a `@-` sequence serves to suppress only the end of line marker following it and must appear *exactly* before the end of line marker to be suppressed. No trailing spaces, `@!` comments, or any other characters are permitted between a `@-` sequence and the end of line that it is supposed to suppress. The `@-` sequence is useful for constructing long output lines without them having to appear in the input. It can also be used in the same way as the `@+` was used in the previous section to assist in exposing the structure of output text without affecting the output text itself. Finally, it is invaluable for suppressing the EOL after the opening macro text `@{` construct. For example:

```
@$@<Walrus@>@{@-
I am the walrus!@}
```

is equivalent to

```
@$@<Walrus@>@{I am the walrus!@}
```

The comment construct (`@!`) can also be used to suppress end of lines. However, the `@-` construct should be preferred for this purpose as it makes explicit the programmer's intent to suppress the end of line.

3.10.10 Include Files

FunnelWeb provides an include file facility with a maximum depth of 10. When FunnelWeb sees a line of the form `@i <filename>`, it replaces the entire line (including the EOL) with the contents of the specified include file. FunnelWeb's include file facility is intended to operate at the line level. If the last line of the include file is not terminated by an EOL, FunnelWeb issues a warning and interts one (in the copy in memory).

The `@i` construct is illegal if it appears anywhere except at the start of a line. The construct must be followed by a single blank. The file name is defined to be everything between the blank and the end of the line (no comments (`@!`) please!). Example: If the input file is

```
"Uh Oh, It's the Fuzz. We're busted!" said Baby Bear.
@i mr_plod.txt
"Quick! Flush the stash down the dunny and let's split." said Father Bear.
```

and there is a file called `mr_plod.txt` containing

```
"'Ello, 'Ello, 'Ello! What's all this 'ere then?" Mr Plod exclaimed.
```

then the scanner translates the input file into

```
"Uh Oh, It's the Fuzz. We're busted!" said Baby Bear.
"'Ello, 'Ello, 'Ello! What's all this 'ere then?" Mr Plod exclaimed.
"Quick! Flush the stash down the dunny and let's split." said Father Bear.
```

As a point of terminology, FunnelWeb calls the original input file the **input file** and calls include files and their included files **include files**.

The include file construct operates at a very low level. An include line can appear anywhere in the input file regardless of the context of the surrounding lines.

FunnelWeb sets the special character to the default (`@`) at the start of each include file and restores it to its previous value at the end of the include file. This allows macro libraries to be constructed and included that are independent of the prevailing special character at the point of inclusion. The same goes for the input line length limit which is reset to the default value at the start of each include file and restored to its previous value afterwards.

3.10.11 Pragmas

Most tools have to support some essential, but rather inelegant features. In FunnelWeb these messy bits have all been stuffed into the scanner's **pragma** (for *pragma*tic) construct.

A pragma consists of a single line of input (including the EOL) commencing with `@p`. This must be followed by a single space, and then the pragma verb. This must be followed by a sequence of zero or more arguments separated by one or more spaces. Four pragmas are available

```
pragma = pragma_ident | pragma_mill | pragma_moll | pragma_typesetter
```

The following syntax definitions assist in defining the pragmas.

```
s         = {" "}+
ps        = ("@p" | "@P") " "
number    = { decimal_digit }+
numorinf = number | "infinity"
```

The arguments to pragmas are case-sensitive and must be specified in lower case.

Pragmas are processed and consumed entirely by the scanner. The parser never sees them and so they can play no part in the parser level syntax. As a result, pragma lines can appear anywhere in the entire input file regardless of the surrounding context (e.g. even in the middle of a macro definition). The sole effect of a pragma is to modify some internal parameter of FunnelWeb.

The following sections describe the four FunnelWeb pragmas.

3.10.11.1 Indentation

When FunnelWeb expands a macro, it can do so in two ways. First it can treat the text it is processing as a one-dimensional stream of text, and merely insert the body of the macro in place of the macro call. Second, it can treat the text of the macro as a two dimensional object and indent each line of the macro body by the amount that the macro call itself was indented. Consider the following macros.

```
@$@<Loop Structure@>@{@-
i=1;
while (i<=N)
   @<Loop body@>
endwhile
@}

@$@<Loop body@>@{@-
a[i]:=0;
i:=i+1;@}
```

Under the regime of **no indentation** the loop structure macro expands to:

```
i=1;
while (i<=N)
   a[i]:=0;
i:=i+1;
endwhile
```

Under the regime of **blank indentation** the loop structure macro expands to:

```
i=1;
while (i<=N)
   a[i]:=0;
   i:=i+1;
endwhile
```

The `indentation` pragma determines which of these two regimes will be used to expand the macros when constructing the product files. The syntax of the pragma is:

```
pragma_ident = ps "indentation" s "=" s ("blank" | "none")
```

Its two forms look like this:

```
@p indentation = blank
@p indentation = none
```

In the current version of FunnelWeb, the indentation regime is an attribute that is attached to an entire run of Tangle; it is not possible to bind it to particular product files or to particular macros. As a result, it doesn't matter where indentation pragmas occur in the input file or how many there are so long as they are all the same. By default FunnelWeb uses blank indentation.

3.10.11.2 Maximum Input Line Length

FunnelWeb generates an error for each input line that exceeds a certain maximum number of characters. At the start of the processing of each input file and each include file, this maximum is set to a default value of 80. However, the maximum can be changed using a maximum input line length pragma.

```
pragma_mill = ps "maximum_input_line_length" s "=" s numorinf
```

The maximum input line length can be varied *dynamically* throughout the input file. Each maximum input line length pragma's scope covers the line following the pragma through to and including the next maximum input line length pragma, but not covering any intervening include files. At the start of an include file, FunnelWeb resets the maximum input line length to the default value. It restores it to its previous value at the end of the include file.

This pragma is useful for detecting text that has strayed off the right side of the screen when editing. If you use FunnelWeb, and set the maximum input line length to be the width of your editing window, you will never be caught by, for example, off-screen opening comment symbols. You can also be sure that your source text can be printed raw, if necessary, without lines wrapping around.

3.10.11.3 Maximum Output File Line Length

As well as keeping an eye on input line lengths, FunnelWeb also keeps an eye on the line lengths of product files and flags all lines longer than a certain limit with error messages. Unlike the maximum input line length, which can vary dynamically throughout the input file, the maximum product file line length remains fixed throughout the generation of all the product files. The maximum product file line length pragma allows this value to be set. If there is more than one such pragma in an input file, the pragmas must all specify the same value.

```
pragma_moll = ps "maximum_output_line_length" s "=" s numorinf
```

The default value is 80 characters.

This pragma is only one of two constraints on the length of the lines of the product files. The `+W` command line option also contributes. The actual value that FunnelWeb uses is the minimum of the limits specified in the command line and pragmas.

FunnelWeb does not monitor the length of the lines of its other output files (journal file, listing file, documentation file).

3.10.11.4 Typesetter

The `typesetter` pragma allows the user to specify whether the input file is supposed to be typesetter-independent, or whether it contains commands in a particular typesetter language. The pragma has the following syntax.

```
pragma_typesetter = ps "typesetter" s "=" s ("none" | "tex")
```

The two forms of the pragma look like this.

```
@ typesetter = none
@ typesetter = tex
```

A source file can contain more than one typesetter pragma, but they must all specify the same value. The default is `none`. The typesetter setting affects two things:

> **Handling of free text:** If the typesetter is not `none`, Weave writes the free text *directly* to the documentation file without changing it whatsoever. This means that if (say) `\centerline` appears in the input file, it will copied directly to the documentation file. If the typesetter is `none`, Weave intercepts any characters or sequences that might have a special meaning to the target typesetter and replaces them with typesetter commands to typeset the sequences so that they will appear as they do in the input. For example, if the typesetter is `none` and the target typesetter is TeX, then if `$` (the TeX "mathematics mode" character) appears in the input file, it will be be written to the documentation file as `\$`.
>
> **Restrictions on the target typesetter:** At a later date, different weave modules might be incorporated into FunnelWeb to cater for a variety of different typesetters. If this happens, it will be important to ensure that typesetter-specific source files (i.e. `typesetter` $\neq$ `none`) are not processed with different target typesetters. For example, a user might innocently attempt to generate a `troff` documentation file from a FunnelWeb source file containing a `typesetter = tex` (and by implication TeX control sequences). The pragma could also be useful for catching typesetter clashes in source and include files. The setting `none` is special because it is guaranteed to work with any future target typesetter.

The aim of all this is to ensure that any typesetter dependency is correctly proclaimed. Because `none` is the default typesetter, a user who creates a source file without a `typesetter = x` pragma will soon find that the control sequences they are inserting into the source document are appearing verbatim in the printed documentation! In order to activate these sequences, they will be forced to add a `typesetter` pragma, thus making the dependency explicit.

It may seem strange to place the `typesetter` setting facility within a pragma (`@p`) when there is a separate typesetting construct (`@t`). This has been done to sustain the rule of thumb that says that pragmas do not participate in the parser-level syntax, but typesetter directives do.

3.10.12 Freestanding Typesetter Directives

FunnelWeb provides two kinds of typesetter directive to assist the user to produce documentation. These are **inline** and **freestanding**. Unlike pragmas, each of these categories of directive participates in the parser-level syntax and can appear only in certain contexts (see the parser section). Inline directives are designed to be used within paragraphs to alter the look of the enclosed text. Freestanding typesetter directives are designed to appear on lines of their own and have a bigger typographical impact.

The syntax of freestanding typesetter directives is almost identical to that of pragmas. All the same syntax rules apply (except that the actual keywords are different). The following subsections describe the four typesetter directives available.

```
ftd = ftd_newpage | ftd_toc | ftd_vskip | ftd_title
ts  = "@t "
```

3.10.12.1 New Page

The new page pragma is a typesetting pragma with the following syntax.

```
ftd_newpage = ts "new_page"
```

It only form looks like this.

```
@t new_page
```

Its sole effect is to cause a "skip to a new page" command to be inserted into the documentation file. The new page command is such that if the typesetter is already at the top of a page, it will skip to the top of the next page.

3.10.12.2 Table of Contents

The new page pragma is a typesetting pragma with the following syntax.

```
ftd_toc = ts "table_of_contents"
```

It only form looks like this.

```
@t table_of_contents
```

Its sole effect is to instruct Weave to insert a table of contents at this point in the printed documentation. This pragma does not skip to a top of a new page first.

3.10.12.3 Vertical Skip

The vertical skip pragma is a typesetting pragma that instructs Weave to insert a specified amount of vertical space into the documentation. The pragma has the following syntax.

```
ftd_vskip = ts "vskip" s number s "mm"
```

For example:

```
@t vskip 26 mm
```

3.10.12.4 Title

The title pragma is a typesetting pragma with the following syntax.

```
ftd_title = ts "title" s font s alignment text
font      = "normalfont" | "titlefont" | "smalltitlefont"
alignment = "left" | "centre" | "right"
text      = """" {printable_char} """"
```

It's effect is to instruct Weave to insert a single line into the printed documentation containing the specified text set in the specified font and aligned in the specified manner. The double quotes delimiting the text are for show only; if you want to put a double quote in the string, you don't need to double them.

Here is an example of the pragma.

```
@t title smalltitlefont centre "How to Flip a Bit"
```

3.10.13 Scanner/Parser Interface

If the scanner terminates without any errors, control is passed to the parser. The parser parses the token list generated by the scanner. The token list consists of text scraps, freestanding typesetter directives, and special sequence tokens.

The user should bear in mind that *the scanner finishes running before the parser starts running.* This means that the scanner cannot be influenced in any way by higher order structures such as the parser might parse. For example, it is impossible to write a FunnelWeb macro to include a file, or insert a `vskip` pragma into the input text.

3.11 Parser

By the time the parser starts, the scanner has completely terminated. At this point, it is not possible for any more files to be included, and special characters are no longer present to confuse things. All that remains is a list of **text tokens**, **special tokens**, and **typesetter directive tokens**. Text tokens consist entirely of sequences of printable characters and end of line markers. Special tokens represent the special sequences that the scanner found in the input file. Typesetter directive tokens represent the freestanding typesetter directives that the scanner encountered. The parser consumes the token list and builds a macro table that is later used to generate product files. It also constructs a document list that is used to generate the documentation file.

The syntax rules appearing in the following sections refer to the token list.

3.11.1 High Level Structure

At the highest level, the FunnelWeb parser parses the input file (token list) into a sequence of text scraps, macro definitions, and typesetter directives.

```
input_file = {text | macro | directive}
```

All three of these kinds of components contribute to the documentation file, but only macro definitions contribute to the product files. If all the free text and directives were removed from a FunnelWeb input file, the product files would not be affected.

3.11.2 Free Text

Free text is any text that is not part of a macro definition or a directive. A scrap of free text consists of a sequence of items drawn from the following list: non-special printable characters, insert-eol special sequences, insert special character special sequences, insert arbitrary character special sequence.

```
free_text      = ordinary_text
ordinary_text  = {ordinary_char | eol | text_special}+
text_special   = "@+" | "@@" | "@^" char_spec
ordinary_char  = " ".."~"-special
```

An example of some rather messy free text is as follows:

```
This@@ is a very@+ messy
@^D(009)chunk of text indeed.
But FunnelWeb still views it as
a single chunk of text.
```

FunnelWeb never sees two text chunks next to each other in the input; they are always merged into a single text token.

The free text in an input file does not affect the product files. However, by default, it appears in the printed documentation exactly as it is given in the input file, except that it is filled and justified into paragraphs.

Any printable character or particular sequence of characters may appear in the free text of a document. FunnelWeb ensures that they will appear exactly as given in the input file, even if they happen to be escape characters or commands in the target typesetter. However, FunnelWeb also provides a special mode that allows this censoring to be overridden. See Section for more information.

3.11.3 Typesetter Directives

FunnelWeb provides a variety of typesetter directives to assist the user to typeset the document in a typesetter-independent way. These are divided into **freestanding typesetter directives** (ftd) and **inline typesetter directives** (itd). The internal syntax of the freestanding typesetter directives has already been discussed in the scanner section. The following syntax rule defines the context in which these constructs can appear.

```
directive = ftd | itd
itd       = section | literal | emphasis
```

The remainder of this section describes the inline typesetter directives.

3.11.3.1 Section

The section directive provides a way for the user to structure the program and documentation into a hierarchical tree structure, just as in most large documents. A section construct consists of a case-insensitive identifying letter, which determines the absolute level of the section in the document, and an optional section name, which has exactly the same syntax as a macro name.

```
section   = "@" levelchar [name]
levelchar = "A" | "B" | "C" | "D" | "E" |
            "a" | "b" | "c" | "d" | "e"
```

The section construct is not quite "inline" as it must appear only at the start of a line. However, unlike the "`@i`", "`@p`", and "`@t`" constructs, it does not consume the remainder of the line (although it would be silly to place anything on the same line anyway).

FunnelWeb provides five levels of sections, ranging from the highest level of `A` (like a LaTeX chapter) to the lowest level of `E` (like a LaTeX subsubsubsection). FunnelWeb input files need not contain any sections at all, but if they do, the first section must be at level `A`, and following sections must not skip hierarchical levels (e.g. an `@D` cannot follow an `@C`). FunnelWeb generates an error if a level is skipped.

All section *must* have names associated with them, but for convenience, the section name is optional if the section contains one or more macro definitions (i.e. at least one macro definition appears between the section construct in question and the next section construct in the input file.). In this case, the section *inherits* the name of the first macro defined in the section. This feature streamlines the input file, avoiding duplicate name inconsistencies.

Any sequence of printable characters can be used in the section name, even the target typesetter's escape sequence (e.g. in TeX, "\").

The following example demonstrates the section construct.

```
@A@<Life Simulation@>

This is the main simulation module for planet earth, simulated down to the
molecular level. This is a REALLY big program. I mean really big. I mean,
if you thought the X-Windows source code was big, you're in for a shock...

@B We start by looking at the code for six legged stick insects as they
form a good example of a typical object-oriented animal implementation.

@$@<Six Legged Stick Insects@>@{@-
slsi.creep; slsi.crawl; slsi.creep;@}
```

In the above example, the name for the level A section is provided explicitly, while the name for the level B section will be inherited from the macro name.

3.11.3.2 Literal Directive

Experience has shown that one of the most common typesetting requirement is that of being able to typeset small program fragments in the middle of the documenting free text. Typically there is a frequent need to refer to program identifiers, and it assists the reader to have such identifiers typeset in the same manner as the program text in the macro definition. FunnelWeb V1 defined a TeX macro for this (called `p`) that simply typeset its argument in `tt font`. This proved so useful, that the facility has been made typesetter-independent in FunnelWeb V3.

To specify that some text be typeset in `tt font`, enclose the text in curly brace special sequences as follows.

```
literal = "@{" ordinary_text "@}"
```

As in macro names, section names, and macro bodies, the text contained within the literal construct is protected by FunnelWeb from any non-literal interpretation by the typesetter and the user is free to enclose *any* text covered by the definition `ordinary_text`. FunnelWeb guarantees that, no matter what the text is, it will be typeset in `tt font` exactly as it appears. However, the text will be filled and justified into a paragraph as usual.

Here is an example of the use of the construct:

```
@C The @{WOMBAT@} (Waste Of Money, Brains, And Time) function
calls the @{kangaroo@} input function which has been
known to cause keybounce.
This keybounce can be dampened using the @{wet_sloth@} subsystem.
```

3.11.3.3 Emphasis Directive

The emphasis directive is very similar to the literal directive except that it causes its argument to be typeset in an emphasised manner (e.g. italics). Like the literal directive, the emphasis directive protects its text argument.

```
emphasise = "@/" ordinary_text "@/"
```

Example:

```
@C What you @/really@/ need, of course, is a @/great@/, @/big@/,
network with packets just flying @/everywhere@/.
This section implements an interface
to such a @/humungeous@/ network.
```

3.11.4 Macros

The third category of construct appearing at the highest syntactic level in a FunnelWeb input file is the macro definition. A macro definition binds a unique **macro name** to a **macro body** containing an **expression** consisting of text, calls to other macros, and formal parameters. The syntax for a macro definition is as follows:

```
macro = ("@O" | "@$") name [formal_parameter_list]
        ["@Z"] ["@M"] ["==" | "+="] "@{" expression "@}"
```

The complexity of the macro definition syntax is mostly to enable the user to attach various attributes to the macro. If the user chooses `@O`, then the macro cannot be called, but is instead attached to a product file. If the user chooses `@$`, then the macro is an ordinary macro definition that is not attached to a file.

By default, a non-file macro must be invoked exactly once by one other macro. Macros that aren't are flagged with errors by the FunnelWeb analyser. However, if the user uses the `@Z` sequence in the macro definition, the macro is then permitted to be invoked zero times, as well as once. Similarly, if the user uses the `@M` sequence in the macro definition, the macro is permitted to be called many times as well as once. If both `@Z` and `@M` are present then the macro is permitted to be invoked zero, one, or many times.

The purpose of enforcing the default "exactly one call" rule is to flag pieces of code that the user may have defined in a macro but not hooked into the rest of the program. Experience shows that this is a common error. Similarly, it can be dangerous to multiply invoke a macro intended to be invoked only once. For example, it may be dangerous to invoke a scrap of non-idempotent initialization code in two different parts of the main function of a program! However, FunnelWeb will not generate an error if a macro without `@M` is called by another macro that is called more than once.

If the text string `==` (or nothing) follows the macro name, the expression that follows is the entire text of the macro body. If the text string `+=` follows the macro name, then more than one such definition is allowed (but not required) in the document and the body of the macro consists of the concatenation of all such expressions in the order in which they occur in the input file. Such a macro is said to be additive and is **additively defined**. Thus a macro body can either be defined in one place using one definition (using `==`) or it can be *distributed* throughout the input file in a sequence of one or more macro definitions (using `+=`). If neither `==` and `+=` are present, FunnelWeb assumes a default of `==`.

Macros attached to product files cannot be additively defined. Additively defined macros can have parameter lists and `@Z` and `@M` attributes, but these must be specified only in the first definition of the macro. However, `+=` must appear in each definition.

3.11.4.1 Names

Names are used to identify macros and sections. A name consists of a sequence of from zero to 80 printable characters, including the blank character. End of line characters are not permitted in names. Names are case sensitive; two different macros are permitted to have names that differ in case only. Like free text, names are typeset by FunnelWeb and are safe from misinterpretation by the target typesetter. For example, it is quite acceptable to use the macro name `@<\medskip@>` even if the target typesetter is TEX.

```
name      = "@<" name_text "@>"
name_text = {ordinary_char | text_special}
```

3.11.4.2 Formal Parameter Lists

FunnelWeb allows macros to have up to nine macro parameters, named `@1`, `@2`, ..., `@9`. If a macro does not have a formal parameter list, it is defined to have no parameters, and an actual parameter list must not appear at the point of call. If a macro has a formal parameter list, it is defined to have one or more parameters, and a corresponding actual parameter must be supplied for each formal parameter, at the point of call.

Because FunnelWeb parameters have predictable names, the only information that a formal parameter list need convey is *how many* parameters a macro has. For this reason a formal parameter list takes the form of the highest numbered formal parameter desired, enclosed in parentheses sequences.

```
formal_parameter_list = "@(" formal_parameter "@)".
formal_parameter = "@1" | "@2" | "@3" | "@4" | "@5" |
                   "@6" | "@7" | "@8" | "@9"
```

3.11.5 Expressions

Expressions are FunnelWeb's most powerful form of expressing a text string. Macro bodies are defined as expressions. Actual parameters consist of expressions.

An expression consists of a sequence of zero or more expression elements. An expression element can be ordinary text, a macro call, or a formal parameter of the macro *definition* in which the formal parameter occurs.

```
expression = {ordinary_text | macro_call | formal_parameter}
```

3.11.6 Macro Calls

A macro call consists of a name optionally followed by an actual parameter list. The number of parameters in the actual parameter list must be the same as the number of formal parameters specified in the definition of the macro. If the macro has no formal parameter list, its call must have no actual parameter list.

```
macro_call            = name [actual_parameter_list]
actual_parameter_list = "@(" actpar { "@," actpar } "@)"
actpar                = expression |
                        ( whitespace "@"" expression "@"" whitespace )
whitespace            = {" " | eol}
```

FunnelWeb allows parameters to be passed directly, or delimited by special double quotes. Each form is useful under different circumstances. Direct specification is useful where the parameters are short and can be all placed on one line. Double quoted parameters allow whitespace on either side (that is not considered part of the parameter) and are useful for laying out rather messy parameters. Here are examples of the two forms.

```
@<Generic Loop@>@(
   @"x:=1;@"  @,
   @"x<=10;@" @,
   @"print "x=%u, x^2=%u",x,x*x;
   x:=x+1;@+@"
@)

@<Colours@>@(red@,green@,blue@,yellow@)
```

As shown, the two forms may be mixed within the same parameter list.

Experience has shown that the vast majority of macros have no parameters.

3.11.7 Formal Parameters

Formal parameters can appear in the expressions forming macro bodies in accordance with the syntax rules defined above. A formal parameter expands to the text of the expansion of its corresponding actual parameter. There is nothing preventing a formal parameter being provided as part of an expression that forms an actual parameter. In that happens, the formal parameter is bound to the actual parameter of the calling macro, not the called macro. After the following definitions,

```
@$@<One@>@(@1@)=@{A walrus in @1 is a walrus in vain.@}
@$@<Two@>@(@1@)=@{@<One@>@(S@1n@)@}
```

the call

```
@<Two@>@(pai@)
```

will result in the expansion

```
A walrus in Spain is a walrus in vain.
```

3.11.8 Macros are Static

In FunnelWeb, the actions of *macro definition* and *macro expansion* occur during two separate phases (parser and tangle) and cannot be interleaved. As a result, the FunnelWeb macro facility is completely static. It is not possible for one macro to define another while the first macro is being expanded; each must be defined statically. It is not possible to define a macro to even assist in the definition of other macros. Because the scanner, parser, analyser, and tangler phases are all invoked sequentially, there is no room for feedback of definitions between different levels (e.g. the user cannot define a macro for the `vskip` pragma).

This lack of power is fully intentional. By totally excluding the more incomprehensible ways in which a general purpose macro preprocessor can be used, FunnelWeb provides definite guarantees to the reader of its input files:

- FunnelWeb guarantees that a piece of text does not contain a macro call unless it contains the special character followed by < or #.
- FunnelWeb allows calls to be made to macros that are defined later in the input file.

3.12 Analyser

The effect of the parser is to construct a macro table containing a representation of all the macros defined within the document, and a document list which contains a complete representation of the entire document. If there are no error diagnostics (or worse) at the end of the parser run, FunnelWeb invokes the analyser which tests for the following conditions and flags them with errors if they arise.

- No macros defined in the input file.
- No macros connected to output files.
- Call of an undefined macro.
- Call having the wrong number of parameters.
- Call of a macro that is connected to an output file.
- No calls made to a macro without the `@Z` option.
- More than one call made to a macro without the `@M` option.
- Directly or indirectly recursively defined macros.
- Unnamed sections that contain no macro definitions.

FunnelWeb performs a static analysis to detect recursion. Unfortunately, the recursion detection algorithm flags all macros that have an infinite expansion rather than just all macros with a recursive definition. If A calls B, and B calls C, and C calls B, then FunnelWeb will flag A as well as B and C. It is hoped that this problem will be fixed in a later version.

Because FunnelWeb does not provide any kind of conditional feature, the prevention of recursion does not represent a curtailment of expressive power.

Macros may be invoked recursively, but may not be recursive. Thus:

```
@<Sloth@>@(@<Sloth@>@(Walrus@)@)      @! LEGAL   recursive invocation.

@$@<Teapot@>==@{@<Teapot@>@}          @! ILLEGAL recursive definition.
```

3.13 Tangle

If the scanner, parser, and analyser have successfully (i.e. with no errors, severe errors, or fatal errors) completed, and the Tangle option (`+O`) is turned on (it is by default), then the Tangle component of FunnelWeb is invoked to generate the product files specified in the `@O` macros of the input file.

The operation of Tangle is very simple. Each `@O` macro is expanded and written to a file of the same name. As there are a finite number of macros, and the analyser guarantees that the macro structure is non-recursive, Tangle is guaranteed to terminate.

Three remaining points are worth discussing.

1. Tangle expands macros using blank indentation unless the user has specified otherwise in an indentation pragma in the input file (see Section 3.10.11.1).

2. Tangle keeps track of the length of the lines that it is writing and issues an error if any line of any product file that it generates is longer than the maximum. The maximum is the minimum of a value defaulted or specified in the input file (Section 3.10.11.3), and the value (if any) provided by the `+W` command line argument (Section 3.7.3).

3. It is worth the user obtaining some understanding of the resources that FunnelWeb requires to perform its task.

When FunnelWeb's scanner executes, it reads each file into memory where it is kept for the duration of the run. Thus, there must be room in memory for the entire input file, including all include files. While this approach may seem expensive in memory, it is almost necessary in order to support forward references. To merely scan the input file, recording the macro names, but leaving the text on disk, would require many random access disk seeks.

In contrast, FunnelWeb never builds an internal representation of the product file. Instead, each piece of output is written immediately to the product file. This means that as long as the input file fits in memory, the product file can be arbitrarily large. It also means that users need not fear to define or call macros that they know will expand to megabytes of text. Nor need they fear placing a call to such a macro as part of an actual parameter. FunnelWeb does not ever expand actual parameters internally. In fact, it does not expand them until it hits the corresponding formal parameter during its expansion of the called macro. At that point, it looks up the *expression* (not the expansion of the expression) for the corresponding actual parameter, and starts expanding it.

3.14 Weave

If the scanner, parser, and analyser have successfully (i.e. with no errors, severe errors, or fatal errors) completed, and the Weave option (`+T`) is turned on (it is *off* by default), then the Weave component of FunnelWeb is invoked to generate a text file in the format of a particular typesetter. The result, when fed through the particular typesetter and printed, is a fully typeset representation of the entire input file complete with cross referencing information.

3.14.1 Target Typesetter

Currently, FunnelWeb produces documentation files in the format of only one typesetter — TEX. However, the Weave package of FunnelWeb is fairly small, and it is hoped that it can be rewritten so as to provide a collection of typesetter modules from which the user will be able to choose using a command line argument.

3.14.2 Cross Reference Numbering

When FunnelWeb produces its typeset documentation, it *numbers* each section and each macro definition and cross references the macro definitions. The exact scheme used has been carefully thought out. However, as it can be a little confusing to the beginner, it is explained here in full.

The most important thing is that there is *no relation* between the macro numbering and the section numbering. In Knuth's Web there are only section numbers. In FunnelWeb, the numbering of sections and macros is separated.

In FunnelWeb, *sections* are numbered hierarchically in ascending order. For example, the second level-C section of the third level-B section of the first level-A section is numbered "1.3.2". In contrast, *macro definitions* are numbered sequentially in ascending order. For example, the first macro definition is number 1, the second is number 2, and so on. Note that it is *macro definitions* that are numbered, not *macros*. This distinction is necessary because additive macros (i.e. the ones with `+=`) can be defined by a collection of partial definitions scattered throughout the input file. A single additive macro may be defined in definitions 5, 67, 128, and 153.

3.15 FunnelWeb Shell

3.15.1 Introduction

One of the goals of FunnelWeb is that it must be extremely portable. Huge efforts, desperate actions, and great sacrifices were made in the name of portability. For example, FunnelWeb is written in C.

An equally important goal was that of correctness and reliability. To this end, it was determined that a large automated suite of test programs be prepared to assist in regression testing. Preparing the test suite was tedious, but achievable. Automating it portably was more difficult.

The difficulty faced was that if FunnelWeb was implemented in the form of a utility that could be invoked from the operating system command language, the only way to set up regression testing was in the command language of the operating system of the target machine (shellscripts for UNIX, DCL for VMS, batch files for MSDOS, and *nothing* on the Macintosh). The huge variation in these command languages led to the conclusion that either the automation of regression testing would have to be rewritten on each target machine, or a small command language would have to be created within FunnelWeb. In the end, the twin goals of portability and regression testing were considered so important that a small command shell was constructed inside FunnelWeb. This is called the **FunnelWeb command shell**, or just "the shell" for short.

By default, when FunnelWeb is invoked, it does not enter its shell. If just given the name of an input file, it will simple process the input file in the normal manner and then terminate. To instruct FunnelWeb to invoke its shell, the `+K` or `+X` command line option must be specified when FunnelWeb is invoked from the operating system. It is also invoked upon startup if the file **`fwinit.fws`** exists.

Most FunnelWeb users will never need to use the shell and need not even know about it. There are four main uses of the shell:

1. As a tool to support automated regression testing.
2. As a development tool on machines that do not have a built in shell (e.g. the Macintosh). The shell can be used to process whole groups of files automatically.
3. As a convenience. A user working on a multi-tasking, multi-window workstation may wish to keep an interactive session of FunnelWeb going in one window rather than having to run up the utility each time it is required.
4. As a convenient vehicle for enclosing utilities. The FunnelWeb shell contains useful general purpose commands such as the differences command `diff`.

3.15.2 Return Statuses

The hierarchy of diagnostics described in Section 3.5 is also used in the shell commands. Each shell command returns a status which can affect further processing.

> **Success** status is the normal command return status.
>
> **Warning** status is returned if some minor problem arose with the execution of the command.
>
> **Error** status is returned if a significant problem arises during the execution of the command. However, unlike a severe error, it does *not* cause termination of the enclosing shellscript.
>
> **Severe error** status is returned if a problem arises during the execution of the command that prevents the command from delivering on its "promise". A severe error causes FunnelWeb to abort the script (and any stacked scripts) to the interactive level. (However, the `tolerate` command allows this to be temporarily overridden).

Fatal error status is returned if a problem arises that is so serious that execution of FunnelWeb cannot continue. A fatal error causes FunnelWeb to abort to the operating system level.

Assertion error status is never returned. If an assertion error occurs, FunnelWeb bombs out ungracefully to the operating system. Assertion errors should never happen. If they do, then there is a bug in FunnelWeb.

To be precise, the status returned by each command is a vector of numbers being the number of each of the different kinds of diagnostic generated by the command. Usually only one kind of diagnostic is generated. However, the `fw` command and a few of the other commands can generate more than one kind of diagnostic. These status vectors are summed internally where they may later be accessed using the `status` command. However, the current diagnostic state evaporates as soon as the next command is encountered.

3.15.3 Command Line Length

The maximum length of a shell command line is guaranteed to be at least 300 characters.

3.15.4 String Substitution

Most command shells provide some form of string substitution so as to provide some degree of parameterization. The FunnelWeb shell provides 36 different string variables named `$0..$9` and `$A..$Z` (case insensitive). Each variable can hold a string containing any sequence of printable characters and can be as long as a command line.

The `define` command allows the user to assign a value to these variables. The `define` command takes two arguments. The first is the digit or letter of the variable to be defined. The second is a double quote delimited string being the string value to be assigned to the variable. If you want to include a double quote character within the string, you don't need to double it.

Examples:

```
define 3 "/root/usr/usrs/users/users5/thisuser/workdir/fwdir/testdir"
define M "/user/local/rubbish/bin/fw"
define Q "You don't need to double" double quotes"
```

Only the identifying character of the variable being assigned is used in the definition. This syntax is a simple way of preventing the variable from being substituted before it has a chance to be defined!

The following points clean up the remaining semantic details:

- There is only one set of variables and they are global to all shellscripts. There are *no local variables.*
- When a shellscript is invoked using the `execute` command, the substitution variables `0` through `9` are affected. See Section 3.15.7.9 for more details.
- If you want to include a dollar sign character in a command use "`$$`".
- FunnelWeb also defines "`$/`" which translates to the character that separates directory and file name fields in file names on the host machine. For example: Sun="`/`", Vax="`]`", Mac="`:`", PC="`\`".
- Substitution is not performed recursively.

3.15.5 How a Command Line is Processed

When FunnelWeb reads in a command line (from the console or a script file), it processes it in the following sequence:

1. The command line is checked for non-printable characters. If there are any, they are flagged with a severe error.
2. All dollar string substitution variables in the command line are replaced by their corresponding string. The command line is processed from left to right. Substitutions are performed non recursively.
3. At this point, if the line is empty, or consists entirely of blanks, it is ignored and the interpreter moves to the next line.
4. A severe error is generated if the line at this stage begins with a blank.
5. If the first character of the line is "`!`", the line is a comment line and is ignored.
6. The run of non-blanks commencing at the start of the line is compared case-insensitively to each of the legal command verbs. If the command is illegal, a severe error is generated, otherwise the command is processed.

3.15.6 Options

The FunnelWeb shell maintains three sets of command line options.

1. The set of options resulting from applying the operating system level command line arguments to the default option settings.
2. A set of shell options that prevail during the shell invocation.
3. The set of option values active during a particular invocation of FunnelWeb proper.

When FunnelWeb is invoked from the operating system with just `+F`, only the first of these three sets comes into existence. If the user invokes the FunnelWeb shell, the shell options come into existence and are initialized with the value of the first set. These shell options are used as the default for all subsequent `fw` commands. However, they can be altered using the script command `set`. If a `fw` command executed in a shell contains additional command line options, these override the shell options for that run, but do not change the shell options. An example follows:

```
$ fw +k +t             ! Original invocation of FunnelWeb from OS.
                       ! Shell options are now default with "+t".
FunnelWeb>fw sloth     ! Equivalent to fw sloth +t.
FunnelWeb>set -l       ! Change the l shell option.
FunnelWeb>fw sloth +q  ! Equivalent to fw sloth +t -l +q.
FunnelWeb>fw sloth     ! Equivalent to fw sloth +t -l.
```

The existence of the shell option set means that the user can set up a set of defaults to be applied to all `fw` commands issued within the shell.

3.15.7 Shell Commands

This section describes each of the FunnelWeb shell commands. The syntax is:

```
shell_command = absent   | codify | compare | define | diff    | diffsummary |
                diffzero | eneo   | execute | exists | fixeols | help        |
                here     | quit   | set     | show   | skipto  | status      |
                tolerate | trace  | write   | writeu
s = {" "}+
```

As a rule, FunnelWeb shell commands return severe status if their arguments are syntactically incorrect or if they are unable to successfully operate on argument files.

3.15.7.1 Absent

The `absent` command performs no action except to return a status. If the file specified in its argument doesn't exist it returns success status, otherwise it returns severe status.

```
Syntax : absent = "absent" s filename
Example: absent result.out
```

This command is useful in regression testing for making sure that FunnelWeb *hasn't* produced a particular output file.

3.15.7.2 Codify

The `codify` command takes two arguments: an input file and an output file. It reads each line of the input file and writes a corresponding line to the output file. The corresponding line consists of a C macro call containing a string containing the input line. The command converts all backslashes in input lines to double backslashes so as to avoid unwanted interpretations by the C compiler. It also converts double quotes in the line to backslashed double quotes.

```
Syntax : codify = "codify" s filename s filename
Example: codify header.tex header.c
```

The following example demonstrates the transformation.

```
Input  Line: \def\par{\leavevmode\endgraf}% A "jolly good hack".
Output Line: WX("\\def\\par{\\leavevmode\\endgraf}% A \"jolly good hack\".");
```

The `codify` command was introduced to assist in the development of FunnelWeb. It is used to convert longish text files into C code to write them out. The C code is then included within the FunnelWeb C program. For example, the set of TeX definitions that appears at the top of every documentation file was `codif`ied and inserted into the FunnelWeb code so that FunnelWeb would not have to look for a file containing the definitions at run time.

3.15.7.3 Compare

The `compare` command takes two filename arguments and performs a binary comparison of the two files. If the files are identical, success status is returned. If they are different, severe status is returned. No information about the manner in which the files differ is conveyed.

```
Syntax : compare = "compare" s filename s filename
Example: compare result.txt answer.txt
```

The `compare` command was created as the main checking mechanism for regression testing. However, its binary output was soon found to be unworkable and the more sophisticated `diff` command was added so that the actual differences between the files could be examined.

3.15.7.4 Define

The `define` command assigns a value to a shell string substitution variable. The `define` command takes two arguments. The first is the digit or letter of the variable to be defined. The second is a double quote delimited string being the string value to be assigned to the variable. If you want to include a double quote character within the string, you don't need to double it.

```
Syntax  : define = "define" s letter s """" text """"
Examples: define 3 "/usr/usrs/thisuser/workdir/fwdir/testdir"
          define M "/user/local/rubbish/bin/fw"
          define Q "You don't need to double" double quotes"
```

The command interpreter expands the command line before it executes the `define` command. This means that you can define string substitution variables in terms of each other with static binding.

The `define` command was introduced to allow the parameterization of the directories involved in regression testing.

See Section 3.15.4 for more details.

3.15.7.5 Diff

The `diff` command reads in two text files and *appends* a report to a log file containing a list of the differences between the two input files. If the log file does not already exist, an empty one is created first.

```
Syntax  : diff = "diff" s filename s filename s filename s ["ABORT"]
Examples: diff result.tex answer.tex diff.log
          diff $Otest23.out $Atest23.out $Ldiff.log ABORT
```

The `diff` command performs a full line-based differences operation. It will identify different sections in a file, even if they are of differing length.

The implementation of the `diff` command is quite complicated. To be sure that it is at least getting its same/different proclamation right, the `diff` command performs a binary comparison as an extra check.

The following points describe the rules for determining the result status.

1. `diff` aborts with a severe error if the log file cannot be opened or created for appending.
2. An ordinary error is generated if either or both of the input files cannot be opened.
3. If, at the end of the run, the two input files have not been proven to be identical, and the `ABORT` keyword is present, `diff` returns severe status.
4. `diff` returns success status if none of the above conditions (or similar conditions) occur, even if the two files are different.

The `diff` command *appends* its differences report rather than merely writing it. This allows a regression test script to perform a series of regression tests and produce a report for the user.

The `diff` command was added to the shell after it had become apparent that the simpler `compare` command was not yielding enough information. Whereas early on, regression testing was treated mainly as a tool to ensure that FunnelWeb was being ported to other machines correctly, it began to place an increasing role during development in identifying the effects of changes made to the code. The `diff` command supports this application of regression testing by pinpointing the differences between nearly-identical text files.

3.15.7.6 Diffsummary

The **diffsummary** command writes a short report to the console giving the number of difference operations that have taken place and how many of the pairs of files compared were identical. Counting starts at the most recent execution of a **diffzero** command, or if there has been none, when FunnelWeb started up.

```
Syntax  : diffsummary = "diffsummary"
Examples: diffsummary
```

The **diffsummary** command was added so as to allow regression testing scripts to display a summary of the results of the test. If the summary indicates that no pair of files differed, then there is no need to look in the **diff** log file.

3.15.7.7 Diffzero

The **diffzero** command zeros the different summary counters used by the **diff** and **diffsummary** commands.

```
Syntax  : diffzero = "diffzero"
Examples: diffzero
```

The **diffzero** command was added so as to allow regression testing shellscripts to zero their differences counters at the start of a run. This allows testers to invoke the same regression testing script twice in one interactive session without receiving an inflated differences summary.

3.15.7.8 Eneo

The **eneo** command takes one filename argument. If the file does not exist, no action is taken. If the file does exist, it is deleted. In both cases success status is returned. However, if the file exists and cannot be deleted, **eneo** returns severe status.

```
Syntax  : eneo = "eneo" s filename
Examples: eneo result.out
```

The **eneo** command was added so as to allow regression testing scripts to ensure that existing output files were not present before proceeding with a test run. If FunnelWeb were to fail to generate an output file, it would be extremely undesirable for the old version to be used.

ENEO stands for **E**stablish the **N**on **E**xistence **O**f. Most operating systems provide a command to delete files. Typically these commands are verbs such as "delete", "remove", and "kill". As a consequence, the designers of delete commands usually consider the command to have failed if it fails to find the file to be deleted. However, in my experience, the most common use for the delete command is to *establish the non-existence of* one or more files. Typically, a script is starting up and needs to clear the air before getting started. If the files are there, they should be deleted; if they are not, then that's OK too.[5]

[5] As far as I know, the **eneo** command is original.

3.15.7.9 Execute

The `execute` command causes a specified text file to be executed as a FunnelWeb shellscript. The first argument is the name of the script file. The remaining arguments are assigned to the substitution variables `$1`, `$2`, ..., `$9`. Substitution variables in the range `$1` to `$9` that do not correspond to an argument are set to the empty string `""`. `$0` is set to the empty string regardless. The execute command can be used recursively, allowing shell scripts to invoke each other. A file extension default of "`.fws`" (FunnelWeb Script) applies to script files.

```
Syntax  : execute = "execute" s filename {argument_string}
Examples: execute megatest.fws /usr/users/ross/fwtest !
          execute sloth
```

The first example above will result in the following substitution variable assignments.

```
$0 = ""
$1 = "/usr/users/ross/fwtest"
$2 = "!"
$3 = ""
...
$9 = ""
```

It should be stressed that there are no local variables in the FunnelWeb command language; the variables above are globally modified.

The `execute` command was added to allow the creation of sub-scripts to test FunnelWeb in particular ways.

3.15.7.10 Exists

The `exists` command performs no action except to return a status. If the file specified in its argument exists it returns success status, otherwise it returns severe status.

```
Syntax : exists = "exists" s filename
Example: exists test6.fw
```

This command is useful in regression testing for ensuring that FunnelWeb has produced a particular output file.

3.15.7.11 Fixeols

The `fixeols` command takes two filename arguments: an input file and an output file. It reads in the input file and writes it to the output file changing all the end of line control character sequences to the local format. It can also take one filename argument, in which case it replaces the target file with its transformation.

```
Syntax  : fixeols = "fixeols" s filename [s filename]
Examples: fixeols imported.hak result.kln
          fixeols sloth.dat
```

The `fixeols` command works by parsing the input file into alternating runs of printable characters (ASCII 20 to ASCII 126) and runs of non-printable characters (all the others). It then parses each run of non-printable characters from left to right into subruns of non-printables not containing the same character twice. It then replaces each subrun with a native EOL.[6] For example, if a native EOL is `X`, and `ABCD` are non-printable characters, and the file to be converted is

[6] Note: A native EOL can be inserted into a text file in a portable manner simply by writing "\n" to the text output stream.

```
thisABisABCDanABABexampleABCCCof the conversion.
```

then `fixeols` would produce

```
thisXisXanXXexampleXXXof the conversion.
```

The `fixeols` command was devised to solve the problem created sometimes when text files are moved from one machine to another (e.g. with the kermit program) using a binary transfer mode rather than a text transfer mode. If such a transfer is made, and the text file line termination conventions differ on the two machines, one can wind up with a set of text files with improperly terminated lines. This can cause problems on a number of fronts, but in particular affects regression testing which relies heavily on exact comparisons between files. The `fixeols` command provides a solution to this problem by providing a portable way to "purify" text files whose end of lines have become incorrect. The regression testing scripts all apply `fixeols` to their input and output files before each test.

3.15.7.12 Fw

The `fw` command allows FunnelWeb proper to be invoked from a shell script. The syntax is almost identical to the syntax with which FunnelWeb is invoked from the operating system.

```
Syntax  : fw = "fw" s ordinary_funnelweb_command_line
Examples: fw sloth +t +d
          fw -l walrus
```

Some important points about this `fw` command are:

- Options are inherited from the default shell options.
- The `F` (input file option) must be turned on.
- The `K`, `H`, and `X` options must be turned off.
- The `J` option must be turned off.
- The options specified in a `fw` command do not affect the default shell options.
- This command performs no action in the VAX VMS version of FunnelWeb.

3.15.7.13 Help

The `help` command provides online help from within the FunnelWeb shell. It provides access to all of the same messages that the `+H` command line option does.

```
Syntax  : help = "help" [s help_message_name]
Examples: help
          help commands
```

If no message name is given, the default message is displayed. It contains a list of the other help messages and their names. The actual messages themselves are not listed here.

3.15.7.14 Here

The `here` command acts as a target for the `skipto` command. When the shell interpreter encounters a `skipto` command, it ignores all the following commands until it encounters a `here` command.

```
Syntax : here = "here"
Example: here
```

The `skipto/here` mechanism was created to allow groups of regression tests to be skipped during debugging without having to comment them out. For more information, see Section 3.15.7.18.

3.15.7.15 Quit

The `quit` command terminates FunnelWeb immediately and returns control to the operating system. This applies regardless of the depth of the script being executed.

```
Syntax : quit = "quit"
Example: quit
```

3.15.7.16 Set

The `set` command modifies the default shell options. For example, `set +t` sets the `+t` option for all subsequent FunnelWeb runs within the shell until another set command sets `-t`.

```
Syntax  : set = "set" s ordinary_funnelweb_command_line
Examples: set sloth +t +d
          set -lwalrus
```

The restrictions on the `set` command are identical to those on the `fw` command except that, in addition, the `+F` option cannot be turned on in the `set` command.

The set command is useful for setting option defaults before a long run of regression tests. It could also be useful to set default options in a FunnelWeb shell kept by a user in a workstation window.

3.15.7.17 Show

The `show` command displays the current default shell options. These options are the options that subsequent `fw` commands will inherit.

```
Syntax : show = "show"
Example: show
```

3.15.7.18 Skipto

The `skipto` command causes the shell to ignore all subsequent commands until a `here` command is encountered.

```
Syntax  : skipto = "skipto"
Examples: skipto
```

The skipto/here mechanism was created to allow groups of regression tests to be skipped during debugging without having to comment them out. It is like a cut price goto. For example, supposing that there were eight tests and that you had debugged the first five. You might want to skip the first five tests so that you can concentrate on the next three. The following code shows how this can be done.

```
skipto
execute test infile1
execute test infile2
execute test infile3
execute test infile4
execute test infile5
here
execute test infile6
execute test infile7
execute test infile8
```

It should be stressed that FunnelWeb performs full command line processing including the dollar substitutions before testing the line to see if it is here. This can lead to non-obvious problems. For example.

```
skipto
! Test the Parser
! ---------------
define X "execute parsertest.fws"
$X infile1
$X infile2
$X infile3
$X infile4
$X infile5
here
```

The above looks correct, but, because the define command isn't executed (and $X is not defined) the subsequent $X lines result in a leading blanks error. The problem can be corrected by defining $X before the skipto command.

3.15.7.19 Status

The status command takes two forms. In its first form in which no arguments are given, it writes out the number of warnings, errors and severe errors that 1) were generated by the previous command and 2) have been generated during the entire shell invocation. In its second form it takes from one to three arguments each of which specifies a diagnostic severity and a number. The status command compares each of these numbers with the number of that diagnostic generated by the previous command and generates a severe error if they differ.

```
Syntax  : status = "status" {s ("w"|"e"|"s") num}0..3
Examples: status
          status w1 e5 s1
          status w4
          status s1 e2
```

The status command was introduced to test the status results of commands during their debugging. It is also useful for checking to see that the right number of diagnostics have been generated at particular points in test scripts.

3.15.7.20 Tolerate

The `tolerate` command instructs the shell not to abort processing of the script if the next command generates one or more warnings, errors, or severe errors. For the purposes of this command, a blank line counts as a command, so be sure to place the `tolerate` command immediately above the command about which you wish to be tolerant.

```
Syntax : tolerate = "tolerate"
Example: tolerate
```

The tolerate command was introduced to allow FunnelWeb (i.e. the `fw` command) to be tested in a script under conditions which would normally cause it to abort the script.

3.15.7.21 Trace

The `trace` command turns on or off command tracing during script execution. By default, tracing is turned off.

```
Syntax  : trace = "trace" [s ("on" | "off")]
Examples: trace on
          trace off
```

The `trace` command was introduced to assist in the debugging of regression test scripts.

3.15.7.22 Write

The `write` command accepts a double-quoted argument and writes it followed by an EOL to the console (standard output). There is no need to double any double quotes occurring within the string.

```
Syntax  : write = "write" s string
Examples: write "Now about to start the next test."
          write "You don't need to " double enclosed double quotes."
```

The `write` command was added so as to allow regression testing scripts to inform the user of their progress.

3.15.7.23 Writeu

The `writeu` command is identical to the `write` command except that it underlines the text on an additional following output line.

```
Syntax  : writeu = "writeu" s string
Examples: writeu "Test 6"
```

3.16 Concluding Remarks

This chapter defines the semantics of the FunnelWeb program. As stated at the start of this chapter, this document takes precedence over the FunnelWeb program. While the definition of FunnelWeb in this chapter is reasonably solid, it is far from watertight, and it is hoped that it can be tightened further in future versions. All constructive criticism will be gratefully received by the author Ross Williams (`ross@spam.adelaide.edu.au`).

Chapter 4

FunnelWeb Installation

This chapter describes how to obtain, compile, and install FunnelWeb. You will need:

- FTP access to the internet *or* a FunnelWeb distribution kit on disk.
- A Sun, VMS VAX, Macintosh, or PC *or* lots of extra time to port FunnelWeb to a new platform.
- About four megabytes of free disk space. You might be able to install it with less, but four megabytes is safe. The distribution kit itself is about two and a half megabytes. If you are short on space, you can throw away everything after installation except the binary executable which will consume about half a megabyte.
- A C compiler.
- An acquaintance with the C programming language and the ability to compile and link C programs on your machine.
- Elementary systems programming knowledege for your machine.
- About an hour.

You will *not* need any sort of system privileges to install FunnelWeb, unless you want the FunnelWeb command `fw` to be automatically available to everyone on your machine as well as yourself.

4.1 Obtaining a Copy of FunnelWeb

The simplest way to obtain a copy of FunnelWeb is by anonymous FTP from:

```
Machine   : sirius.itd.adelaide.edu.au [IP=129.127.40.3].
Directory : ~pub/funnelweb/   (or a directory of similar name).
```

It is not clear at the time of writing whether FunnelWeb will be presented as a "`.tar`" file, or as a directory tree, or both. Just sniff around and use your common sense. Two points deserve attention however:

1. Be sure to use the *text* transfer mode whenever you transfer raw FunnelWeb files. However, you should use *binary* mode for TAR files and other conglomerate representations.
2. If you have to transfer the files individually, don't lump all the FunnelWeb files into a single directory. Refer to the sections that follow for information on the directory tree you should create to receive the FunnelWeb files.

If anonymous FTP is not available to you, contact the author for up-to-date information on other channels of distribution.


```
Name:  Dr Ross N. Williams
Email: ross@spam.adelaide.edu.au
Snail: 16 Lerwick Avenue, Hazelwood Park 5066, Australia.
```


4.2 Establishing The Directory Tree

At this stage, we will assume that you have somehow obtained a set of files that are supposed to be FunnelWeb, and that they are sitting on a disk on the machine on which you wish to compile and install FunnelWeb.

The first thing you have to do is to make sure that the FunnelWeb directory tree has been correctly unpacked. The directory tree should look like this.

```
fwdir       - Root FunnelWeb directory.
   admin    - Administrative files.
   answers  - Answers to test suite.
   hackman  - FunnelWeb Hacker's Manual.
   results  - For test results.
   scripts  - Test scripts.
   sources  - Source code.
   tests    - Test suite.
   userman  - FunnelWeb User's Manual.
```

The following sections describe the contents of each directory in alphabetical order. Check the contents to make sure that you have everything. Do not become fussed if your configuration is not quite as specified as it is very easy for installation guides such as this one to go out of date as minor last minute changes and updates are made to the distribution kit. Check the source from which you obtained the kit, and if it is different too, proceed.

4.2.1 Admin Directory

The `admin` directory contains administrative files to do with licensing and such. It is also a catch-all directory for files that don't belong anywhere else. At the time of writing, it is not clear exactly what will be in the `admin` directory. Why not take a look?

4.2.2 Answers Directory

The `answers` directory contains the "correct answers" to all the regression testing input files. The regression test scripts compare these files to the files generated in the `results` directory.

```
an01.lis ... an04.lis
ex01.lis ... ex16.lis
ex01.out ... ex10.out
ex11.tex ... ex16.tex
generate.lis
hi01.lis ... hi10.lis
hi01.out ... hi05.out
hi06a.out
hi06b.out
hi07a.out
```

```
hi07b.out
hi08.out ... hi10.out
pr01.lis ... pr10.lis
sc01.lis ... sc29.lis
tg01.lis ... tg09.lis
tg01.out ... tg09.out
wv01.lis ... wv06.lis
wv01.tex ... wv06.tex
```

4.2.3 Hackman Directory

The `hackman` directory contains the `.tex` files that make up the *FunnelWeb Hacker's Manual.*

```
h_ch0.tex     - Preface, etc.
h_ch1.tex     - Design.
h_ch2.tex     - Implementation.
h_ch3.tex     - Modification.
h_ch4.tex     - Future.
h_cha.tex     - Appendices.
h_manual.tex  - Main TeX file.
```

See the comment at the top of `h_manual.tex` file for instructions on how to typeset and print the *FunnelWeb Hacker's Manual.*

There is no need to read or print the *FunnelWeb Hacker's Manual* unless you intend to modify FunnelWeb.

4.2.4 Results Directory

The `results` directory exists as a target directory for the output files generated by FunnelWeb during regression testing. This directory is distributed empty and should be empty at the start of regression testing. However, it is permissible for the `results` directory to contain files generated during a previous test run, as the regression testing scripts delete specific unwanted files before each test anyway.

4.2.5 Scripts Directory

The `scripts` directory stores the FunnelWeb command shell scripts that are used to perform regression testing.

```
master.fws    - The master test script. This is the one you run.
test_gen.fws  - Script to generate certain tricky input files.
test_l.fws    - Test FunnelWeb with +L.
test_ld.fws   - Test FunnelWeb with +L +B...
test_lo.fws   - Test FunnelWeb with +L +O.
test_lo2.fws  - Test FunnelWeb with +L +O (two output files).
test_lot.fws  - Test FunnelWeb with +L +O +T.
test_lt.fws   - Test FunnelWeb with +L +T.
```

4.2.6 Sources Directory

The `sources` directory contains *all* of the C source files required to build a FunnelWeb binary executable. In the following list, files given without an extension represent both `.c` and `.h` files.

```
analyse       - The analyser.
as            - Assertions.
clock         - A clock abstraction.
command       - The shell command interpreter.
data          - Shared data structures and global variables.
dump          - Functions to dump internal data structures.
environ.h     - Lightweight machine-dependent, program-independent header.
help          - Module to write out help messages.
help_gnu      - Function to write out the GNU license.
help_gnu.txt  - The GNU license in text form.
help_gnu.ctx  - The GNU license in C code form.
list          - A list abstraction.
lister        - Module to manage the listing file.
machin        - Module to hold machine-dependent, program-dependent stuff.
main.c        - The main() program.
mapper        - Module to read files into memory.
memory        - Memory management.
misc          - Miscellaneous functions.
option        - Command line option processing.
parser        - The parser.
scanner       - The scanner.
section       - A section number abstraction.
style.h       - A machine-independent, program-independent header file.
table         - A table abstraction.
tangle        - The tangler.
texhead       - Module to write out TeX header in documentation files.
texhead.ctx   - The TeX header in C code form.
texhead.tex   - The TeX header in TeX form.
weave         - The weaver.
writfile      - Output abstraction.
```

The "`.txt`", and "`.tex`" files do not participate in the compilation, but are considered part of the source code as they were used to generate the "`.ctx`" files. The "`.ctx`" files are included by `.c` files of the same name. They do not need to be compiled themselves.

4.2.7 Tests Directory

The `tests` directory stores all the input files of the regression test suite. These come in two kinds: FunnelWeb input files with extensions of "`.fw`", and FunnelWeb include files with extensions of "`.fwi`".

```
FunnelWeb Input Files:
   an01.fw ... an04.fw    - Analyser tests.
   ex01.fw ... ex16.fw    - Examples from the tutorial in user manual.
   generate.fw            - Generates a few other tricky input files.
   hi01.fw ... hi10.fw    - Examples from the hints chapter in user manual.
   pr01.fw ... pr10.fw    - Parser tests.
   sc01_note.fw           - A note explaining absence of sc01.fw
   sc02.fw ... sc29.fw    - Scanner tests.
   tg01.fw ... tg09.fw    - Tangler tests.
   wv01.fw ... wv06.fw    - Weaver tests.

FunnelWeb Include Files:
   ex09a.fwi
   sc13a.fwi ... sc13f.fwi
   sc15a.fwi
```

```
tg08a.fwi
```

4.2.8 Userman Directory

The `userman` directory contains the `.tex` files that make up the *FunnelWeb User's Manual*.

```
u_ch0.tex     - Preface, etc.
u_ch1.tex     - Tutorial.
u_ch2.tex     - Hints.
u_ch3.tex     - Definition.
u_ch4.tex     - Installation.
u_ch5.tex     - Administration.
u_cha.tex     - Appendices.
u_manual.tex  - Main TeX file.
```

See the comment at the top of `u_manual.tex` file for instructions on how to typeset and print the *FunnelWeb User's Manual*.

4.3 Compiling FunnelWeb

The FunnelWeb source code is entirely contained within the `sources` directory. However, some simple script files and makefiles can be found in the `admin` directory.

FunnelWeb contains some machine-dependent components, so before compiling FunnelWeb, you need to specify your machine in the source file `environ.h`. To do this, edit the `environ.h` file and set exactly one of the machine name `#define`s to `1`. For example, on the Sun you should set:

```
#define MAC 0
#define SUN 1
#define VMS 0
#define PC  0
```

There should be little difficulty compiling FunnelWeb for any of these platforms. If the machine on which you are compiling FunnelWeb is not one of the ones listed in the `environ.h` file, then choose the closest one you can. Try the `SUN` if you are running a non-Sun Unix. If you run into serious difficulties, you will have to customize `machin.h` and `machin.c` for your machine. See the comments in these files for instructions on how to do this.

Once you have specified a target machine, compile FunnelWeb by pointing your C compiler at all the "`.c`" files in the `sources` directory. The "`.txt`", and "`.tex`" files do not participate in the compilation, but appear in the `sources` directory because they were used to generate the "`.ctx`" files. The "`.ctx`" files are included by `.c` files of the same name and do not need to be compiled separately. Link the results.

The result of all this should be a binary executable called `fw`, or `fw.exe`, or `fw.xxx` where `.xxx` is whatever file extension is appropriate on the target machine. Clean up the `sources` directory by deleting all the listing and object files.

4.4 Testing FunnelWeb

Once you have obtained a binary executable, you should test FunnelWeb before making it available to users. To do this:

1. Set the default directory to be the `scripts` directory.

2. Copy the FunnelWeb executable into the scripts directory (or be able to invoke it from the scripts directory).

3. Edit the script `master.fws`. Locate the section called "Define Symbol For the Root Test Directory" and define the `R` symbol to point to the FunnelWeb root directory `fwdir`. The examples in the comments in the script should make it clear what is required.

4. Invoke FunnelWeb to execute the master test script with the command line `fw +xmaster`

The `master.fws` script should run for a few minutes. If all goes well, you will find a differences report on your screen reporting zero differences. If this happens, then FunnelWeb has been fully tested and is ready to be made available to users. You should delete all the files in the `results` directory and proceed to the next section on installing FunnelWeb for users.

If there were one or more differences, you can either give up and contact the author, or attempt to fix the code yourself. If you decide to fix the code yourself, start with the differences log file and follow your nose. Good luck!

4.5 Installing FunnelWeb

At this stage you should have a `fwdir` directory tree somewhere in your file system. Its contents should be almost identical to the directory contents specified earlier in this chapter except there should now be an additional binary executable file sitting in the `scripts` directory.

To make FunnelWeb available to users, you should:

1. Make the entire directory tree readable to all users.

2. Move the binary executable from the scripts directory to the `admin` directory *or* copy it to somewhere convenient such as a `/bin` directory.

3. Set up a symbol, path, or command of some kind by the name of `fw` that "points" to the binary executable. If possible, set this up so that the `fw` symbol is available to all users. Alternatively, you can inform interested users of how they can add a command to their `login` command file to make the command available to them.

If you are short of disk space or have a system that is stressed in some other way, it may be of assistance to you to know that FunnelWeb has been constructed so that its binary executable is *totally self contained.* The binary executable does not rely on any other files to operate. Nor does it care about its position in the file system. In fact, all that is really required to use FunnelWeb is the binary executable and the *FunnelWeb User's Manual.* Thus, if you are short of disk space, you can move the binary executable to your "`/bin`" directory and delete the entire FunnelWeb tree. However, making the tree available to users is encouraged because:

- It allows users to copy the tree and install it on another machine without bothering you.

- It allows users access to the TEX code for the *FunnelWeb User's Manual* and the *FunnelWeb Hacker's Manual.*

- It allows users access to the regression test suite. This may not seem important, but it could be very convenient for the user as the `ex*` and `hi*` files of the regression test suite contain most of the examples from the *FunnelWeb User's Manual.* By making them available you will save users the trouble of typing them in.

Finally, you should fill in and send off a FunnelWeb registration form. This allows me to get a handle on the size and needs of the user base, and you to be kept informed of new FunnelWeb releases (optional). See Section 5.4 for more information.

4.6 Printing Manuals

FunnelWeb comes with two manuals, a *FunnelWeb User's Manual* and a *FunnelWeb Hacker's Manual*. Instructions for how to typeset and print these manuals appear at the top of the main TEX files for these manuals `u_manual.tex` and `h_manual.tex`.

There is no need to print the *FunnelWeb Hacker's Manual* unless you intend to modify FunnelWeb. However, you should make a few copies of the *FunnelWeb User's Manual* available for users, or at least let them know where the TEX source for the *FunnelWeb User's Manual* is kept so that they can typeset and print it themselves.

4.7 Installation Problems?

If you run into any problems installing FunnelWeb, please write a short report describing the problem and mail it to the author Ross Williams (`ross@spam.adelaide.edu.au`). I may not be able to help you with it immediately, but I certainly want to know that a problem exists so that it can be corrected in future releases of FunnelWeb.

Chapter 5

FunnelWeb Administration

5.1 Introduction

Whether a computer program is useful depends not just on the functionality provided by the program, but also on the totality of the culture and services surrounding it such as license costs, the quality of documentation, presence of a standard and so on. This chapter addresses these issues from a user perspective.

5.2 The User's Commitment To FunnelWeb

One of the problems that might prevent potential users from using FunnelWeb is the level of commitment that it requires. As soon as the user starts creating FunnelWeb `.fw` files, the user becomes dependent on the FunnelWeb program, as the files so created will become unmanageable if the FunnelWeb program were to become unavailable for some reason. There are a number of ways in which computer programs can become unavailable, including operating system upgrades, copyright problems, inability to fix bugs, or just the inability of the program to be ported to a new target platform.

The fact is that FunnelWeb defines an input language and that currently there is only one implementation of the language. Users will only write programs using that language if they feel comfortable about the availability of its implementation.

I have been critically aware of these issues throughout the development of FunnelWeb and have taken every possible step to make FunnelWeb a solid base onto which to build programs. The following points describe the actions I have taken.

> **GNU license:** FunnelWeb's C source code has been released under GNU General Public License Version 2. This means that the source code to FunnelWeb will always be available to anyone who wants it. The FunnelWeb program can never be taken away from you.
>
> **Portability:** FunnelWeb has been designed and written to be extremely portable. First, all FunnelWeb really does is read and write text files. This makes it easy to make portable. Second, FunnelWeb is written in the C programming language[**ANSI**] with portability as a major design goal. Third, FunnelWeb has already been ported to four popular platforms: SunOS, VAX VMS, PC, and Macintosh and it should not prove hard to move it to others. Fourth, FunnelWeb comes with a huge automated regression test suite. This makes it easy to pinpoint problems when it is moved to a new platform. Portability is important because, even if *you* are not using FunnelWeb on some funny platform, you may want to send a computer program written using FunnelWeb to someone who does.

Quality: The FunnelWeb source code is high quality code. Although it has not been formally developed under any particular coding standard, it is well designed and documented. Design decisions have not been taken lightly.

Documentation: FunnelWeb is well documented by this manual and by the *FunnelWeb Hacker's Manual*. This is important because it means that if you want to send someone a program written using FunnelWeb, you can simply point them to this manual rather than having to explain it all yourself. It is also comforting for managers who are controlling source code to know that the format in which the source code is written is well-documented.

Standardization: Users who create source files using FunnelWeb are not only committing to the FunnelWeb program; they are also investing in the FunnelWeb language. If FunnelWeb's language changes radically for some reason, then this investment by users will be undermined. To protect this investment, I intend to maintain an "official" version of FunnelWeb whose language will not change radically, at least not in a non backward compatible manner.

It is my hope that the combination of these factors will alleviate any fears that users may have about committing their source files to FunnelWeb.

5.3 Documentation

The following FunnelWeb documentation is available:

```
"FunnelWeb User's   Manual": Tutorial, Hints, Reference Manual.
"FunnelWeb Hacker's Manual": Notes on Design and Implementation.
```

Everyone involved with FunnelWeb should read the *FunnelWeb User's Manual*. It contains everything you need to know about how to use FunnelWeb. In fact, you are reading it now!

The *FunnelWeb Hacker's Manual* is for those who want to install, modify, fix, fiddle with, and generally hack the FunnelWeb C source code.

Both of these manuals are shipped with the FunnelWeb distribution kit, and should be available on your machine in the form of LaTeX text files. If you cannot find them, you can obtain them from the FunnelWeb FTP archive (see Section 5.11).

5.4 Registration

If you install or use FunnelWeb, please register by filling in and returning the registration form in **Figure 4**. Feel free to expand the form if there is not enough room.

Email the form to `ross@spam.adelaide.edu.au`, or snail mail it to Ross Williams, 16 Lerwick Avenue, Hazelwood Park 5066, Australia. You may wish to make a contribution when you register. See Section 5.5 for more information.

5.5 Support

FunnelWeb is released "as is" under a GNU license, and no formal support is available. You have the right to make changes to FunnelWeb and to use the modified versions created by random programmers. However, this is discouraged (see Section 5.9).

In fact the support that is most needed is your financial support for the FunnelWeb developers! It has taken *months* of full-time *unpaid* work to bring FunnelWeb to you in its current form. I

```
+----------------------FunnelWeb Registration Form----------------------+
| 1. Date:                                                              |
|                                                                       |
| 2. Title and name:                                                    |
|                                                                       |
| 3. Internet email address:                                            |
|                                                                       |
| 4. Snail mail address:                                                |
|                                                                       |
| 5. Work phone number (country,area,number):                           |
|                                                                       |
| 6. Which category of FunnelWeb user best describes you?               |
|       Latent    - Installed FunnelWeb, but don't intend to use it.    |
|       Beginner  - Haven't used FunnelWeb much yet; have an open mind. |
|       Casual    - Use FunnelWeb occasionally.                         |
|       Convert   - Use FunnelWeb to do most programming.               |
|       Fanatic   - FunnelWeb has become a way of life.                 |
|                                                                       |
| 7. What changes or new features would you like to see in FunnelWeb?   |
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
|                                                                       |
| 8. Do you want to be kept informed of new FunnelWeb developments?     |
|                                                                       |
+-----------------------------------------------------------------------+
```

To register as a FunnelWeb user, please fill in this form and email it to `ross@spam.adelaide.edu.au`, or snail mail it to Ross Williams, 16 Lerwick Avenue, Hazelwood Park 5066, Australia. You may wish to make a contribution when you register. See Section 5.5 for more information.

Figure 4: The FunnelWeb registration form.

don't want to inconvenience users who install FunnelWeb, play with it, and then hardly ever use it. If you are in this category, please register, but don't bother contributing. However, if you find that FunnelWeb has become a useful programming tool, a contribution of some positive multiple of US$50 would be appreciated.

To make a contribution, send payment with a completed registration form (see Section 5.4) to:

```
Renaissance Software Pty Ltd
Email: ross@spam.adelaide.edu.au
Snail: 16 Lerwick Avenue, Hazelwood Park 5066, Australia.
```

Payment can be by personal or bank cheque to any bank in the world or by Visa or Mastercard. Please give the card name, number, expiry date, and the amount to be paid in US dollars. All contributions will be appreciated and will encourage further FunnelWeb development. However, no undertaking is made whatsoever about how the money will be used.

5.6 Copyright

The FunnelWeb program is Copyright © 1992 Ross Williams.

However, FunnelWeb has been released by the author and copyright owner Ross Williams (`ross@spam.adelaide.edu.au`) under Version 2 of the GNU General Public License published by the Free Software Foundation. Here are some ways that you can obtain a copy of this license.

- The license appears as an appendix in the *FunnelWeb Hacker's Manual*.
- If you have a working version of FunnelWeb, invoke it with `fw +hlicense +jlicense.txt`.
- Look in the `help` module of the FunnelWeb source code.
- FTP the license from `prep.ai.mit.edu` in `/pub/gnu/COPYING-2`.
- Write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

The license allows you to redistribute FunnelWeb and/or modify it under certain conditions. The license does not cover the *FunnelWeb User's Manual* and *FunnelWeb Hacker's Manual* which are distributed under a simpler license that prohibits changes.

Note: FunnelWeb is distributed WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

5.7 Nowarranty

Users of FunnelWeb should be aware that FunnelWeb comes with no warranty. Here is an extract from the GNU General Public License Version 2, under which FunnelWeb is distributed. For more information see Section 5.6.

```
  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW.  EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.  SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
```

```
REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED
TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.
```

5.8 Distribution

Users of FunnelWeb should be aware that they can distribute the program freely. The following is an extract from the GNU General Public License Version 2, under which FunnelWeb is distributed.

```
  1. You may copy and distribute verbatim copies of the Program's
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.
```

The license also allows you other freedoms. For more information see Section 5.6.

5.9 Modification

FunnelWeb is distributed under a GNU license, and you are free to modify the source code and distribute modified copies (see Section 5.6). However, there are good reasons why you should avoid doing this.

1. If you distribute modified versions of FunnelWeb, you run the risk of creating a version that will diverge from the "official" version of FunnelWeb that I intend to maintain.
2. If you release a version of FunnelWeb with a changed input language, users of your modified version will create source files that will no longer work on other versions of FunnelWeb. The result would be chaos.

For these reasons I request that you do not distribute modified versions of FunnelWeb, particularly versions with a modified language. However, if you must distribute a version with a modified language, *please change its name* (i.e. from "FunnelWeb" to something else). Please also allocate a new file extension to replace "`.fw`" as the extension for source files written in the modified language.

For more information, refer to the *FunnelWeb Hacker's Manual.*

5.10 Versions

FunnelWeb was created in 1986 and was used extensively by Ross Williams (`ross@spam.adelaide.edu.au`) for three years. However, Version 1.0 was written in Ada and was not very portable (it was fairly VAX/VMS specific). David Hulse (`dave@cs.adelaide.edu.au`) took the first step towards a release by translating the Ada code into C. Ross Williams then extensively reworked the C code, making it robust and portable, adding new features, and polishing it to its current form.

```
Vers  Lang  Created  Released   Author          Copyright      Licensing
----  ----  -------  --------   --------------  -------------  ---------
V1.0  Ada   1986     Never      Ross  Williams  Ross Williams
V2.0  C     1989     Never      David Hulse     Public domain  No restriction.
V3.0  C     1992     May-1992   Ross  Williams  Ross Williams  GNU release.
```

This manual was released for:

```
* FunnelWeb       V3.0.
* User's Manual   V1.0.
* Hacker's Manual V1.0.
* TeX Definitions V1.0.
```

5.11 FTP Archive and Author

The FunnelWeb FTP archive is:

```
Machine   : sirius.itd.adelaide.edu.au [IP=129.127.40.3].
Directory : ~pub/funnelweb/   (or a directory of similar name).
```

The author of FunnelWeb and this manual is:

```
Name:  Dr Ross N. Williams
Email: ross@spam.adelaide.edu.au
Snail: 16 Lerwick Avenue, Hazelwood Park 5066, Australia.
```

I intend to maintain an “official” version of FunnelWeb which I will release under GNU license from time to time. I am happy to receive constructive criticism about FunnelWeb and its documentation.

I will always be happy to receive mail about FunnelWeb, but cannot guarantee that I will be able to reply to it immediately.

Appendix A

Glossary

Analyser: A component of the FunnelWeb program that checks the macro table created by the parser for errors. For example, the analyser checks to see if any macro without a `@Z` has not been called.

Argument: A string delimited by blanks appearing on the FunnelWeb command line. Arguments are used to control options.

Directive: A FunnelWeb special sequence or cooperating group of special sequences that do not form part of a macro definition. A directive can take the form of a pragma.

Documentation: Descriptive text.

Documentation file: An output file, produced by the Weave component of FunnelWeb, that contains typesetter commands. When fed into the appropriate typesetter program, the result is a typeset image of the input file.

Free text: The text in an input file that remains if one were to remove macro definitions and directives.

FunnelWeb: This word has a number of different meanings all pertaining to the FunnelWeb system of programming. 1) The entire system of programming as in "Maybe FunnelWeb can help." 2) The computer program that implements the system as in "Run it through FunnelWeb and see what comes out." 3) The language implemented by the FunnelWeb program as in "I wrote the program in FunnelWeb." or "I wrote the program in Ada using FunnelWeb.".

FunnelWeb file: A file whose contents are written in the FunnelWeb language.

FunnelWeb language: The language in which FunnelWeb input files are written.

FunnelWeb proper: Usually, when FunnelWeb is invoked, it processes a single input file and then terminates. However, it also has a command language mode in which it is possible to invoke "FunnelWeb" many times. This leads to confusion between "FunnelWeb" the outer program and "FunnelWeb" the inner program. To avoid this confusion, the inner FunnelWeb is sometimes referred to as "FunnelWeb proper".

FW: An abbreviation for "FunnelWeb" that is used wherever appropriate.

Include file: A file read in by FunnelWeb as the result of an include pragma (`@i filename`).

Input file: Any file read in by FunnelWeb. The phrase "the input file" refers to the root input file (specified using the `+F` option).

Journal file: An output file containing a copy of the output sent to the user's console during an invocation of FunnelWeb. In other systems, this file is sometimes called a "log file".

Listing file: An output file summarizing the result of processing an input file.

Macro: A binding of a name to a string.

Macro definition: A construct appearing in a FunnelWeb file that binds a name to a text string. A FunnelWeb file consists of a series of macro definitions surrounded by documentary text.

Mapper: A component of the FunnelWeb program that reads in the input file and creates a copy of it in memory.

Option: An parameter internal to the FunnelWeb program which can be controlled by command line arguments or pragmas.

Output file: Any file written by FunnelWeb. This includes listing, journal, product, and documentation files. (Warning: During most of FunnelWeb's development the term "output file" was also used to refer to what are now called "product files". This turned out to be extremely confusing and so the term "product file" was invented to distinguish the generic from the specific. However, as this was a late modification, you may find some occurrences of the old use of "output file".).

Parser: A component of the FunnelWeb program that processes the token list generated by the scanner and produces a macro table and a document list. The parser mainly analyses the input file at the syntactic level, but also does some lightweight semantic checking too.

Pragma: Single-line directives that appears in FunnelWeb files. Pragmas control everything from maximum input line length to typesetter dependence. A pragma line starts with "`@p`".

Printed documentation: Sheets of paper resulting from actually typesetting and printing a documentation file.

Product file: An output file, generated by the Tangle component of FunnelWeb, that contains the expansion of the macros in the input file. Note: Other names considered for this were: generated file, expanded file, result file, program file, and tangle file.

Scanner: A component of the FunnelWeb program that scans a copy of the input file in memory and generates a line list and a token list to be fed to the parser. The scanner processes the input at the lexical level.

Script: A file containing FunnelWeb shell commands.

Shell: A command language interpreter built into the FunnelWeb program. The interpreter allows the user to invoke FunnelWeb proper many times during a single invocation of the FunnelWeb program.

Special character: A distinguished character in a FunnelWeb input file that introduces a special sequence. By default the special character is "`@`". However, it can be changed using the "`@=`" special sequence.

Special sequence: A special sequence is a construct introduced by the special character. Special sequences are used to define a structure in a FunnelWeb input file that exists at a higher level to the surrounding text. A FunnelWeb input file may be considered to be a sequence of text and special sequences.

Tangle: This is the name for the component of FunnelWeb that generates one or more product files containing the expansion of macros in the input file.

Typesetting directive: A FunnelWeb directive whose sole effect is to modify the way in which the input file is represented in the documentation file.

Weave: This is the name for the component of FunnelWeb that generates a documentation file containing typesetting commands representing the input file.

Appendix B

References

[ANSI] Australian Standard AS 3955-1991, "Programming Languages — C", (ISBN: 0-7262-6970-0), 12 July 1991. Identical to: International Standard ISO/IEC 9899: 1990 Programming Languages — C.

[ANZE] "Australia, New Zealand Encyclopedia", Entry: "Funnel-web spiders", Vol 7, pp. 564–565, Bay Books, Sydney, (ISBN: 85835–127–7), 1975.

[BSI82] British Standards Institute, "Specification for Computer Programming Language Pascal", Publication BS6192:1982, British Standards Institute, P.O. Box 372, Milton Keynes, MK146LO, 1982.

[Gries81] Gries D., "The Science of Programming", Springer-Verlag, (ISBN: 0-387-90641-X), 1981.

[Humphries91] Humphries B, "Neglected Poems and Other Creatures", Angus and Robertson, Sydney, (ISBN: 0-207-17212-9), 1991.

[Kernighan88] Kernighan B.W., Ritchie D.M., "The C Programming Language", (second edition,"ANSI C"), Prentice Hall, (ISBN: 0-13-110362-8), 1988.

[Knuth83] Knuth D.E., "The WEB System of Structured Documentation", (Web User Manual, Version 2.5, November, 1983), Stanford University, 1983.

[Knuth84] Knuth D.E., "The TEXbook", Addison-Wesley, (ISBN: 0-201-13448-9), 1984.

[Knuth84] Knuth D.E., "Literate Programming", *The Computer Journal*, Vol. 27, No. 2, pp. 97-111, 1984. Note: The author of this manual has not yet obtained this paper.

[Lamport86] Lamport L., "LATEX: A Document Preparation System", Addison-Wesley, (ISBN: 0-201-15790-X), 1986.

[Rosovsky90] Rosovsky H., "The University: An Owner's Manual", W.W.Norton & Company, Inc., (ISBN: 0-393-02782-1), 1990.

[Smith91] Smith L.M.C., "An Annotated Bibliography of Literate Programming", ACM SIGPLAN Notices, Vol. 26, No. 1, January 1991.

[Strunk79] Strunk W., White E.B., "The Elements of Style", Third Edition, MacMillan Publishing Company, New York, (ISBN: 0-02-418200-1), 1979.

[USDOD83] "The Programming Language Ada Reference Manual", American National Standards Institute Inc, ANSI/MIL-STD-1815A-1983, 1983.

Index