FunnelWeb

$Revision: 1.14 $

Based on the FunnelWeb User's Manual by Ross N. Williams

University of Paderborn
D-4790 Paderborn
F.R.G

Table of Contents

FunnelWeb is a literate programming system. It allows the construction of specifications containing both documentation and code fragments. In Eli these specifications have '`.fw`' or '`.fwi`' file types.

A FunnelWeb specification can contain documentation and fragments of code written in other Eli specification languages. Eli will process a FunnelWeb specification as if it was a collection of specifications; just those specifications that it describes. Also, it is possible to generate a documentation file from a FunnelWeb specification. This documentation comprises the literate version of the specifications.

This document is a reference manual for the contents of FunnelWeb specifications emphasizing the use of FunnelWeb in Eli.

The complete FunnelWeb distribution is available via anonymous ftp from "ftp.ross.net" in the directory "clients/ross/funnelweb". The Version 3.0 of the original FunnelWeb Users-Manual can also be found in the Eli-Distribution in directory Eli/pkg/fw.

1 Introduction

This chapter contains an example for a first funnelweb specification. The commands used in this example are explained and Eli is used to generate a documentation and to process the specification contained in this example.

```
@=~
~p maximum_input_line_length = infinity
~p maximum_output_line_length = infinity
~p typesetter = texinfo
~t title titlefont left "A First Funnelweb Specification"

The file ~{helloworld.lido~} contains a minimal Eli-Specification, which
accepts only the empty input file and prints ~{Hello, World~} to the
standard output.

~O~<helloworld.lido~>~{
RULE:   root ::=
COMPUTE
        printf("Hello, World\n");
END;
~}
```

Copy this specification file into the file '`first.fw`'. Now, you can use the following Eli-derivations:

`first.fw:exe > first.exe`
: Generates an executable processor from the specification and copies it as '`first.exe`' onto the harddisk.

`() +cmd=(first.fw:exe) :stdout>`
: Generates an executable processor from the specification, runs it with an empty input file and shows its output.

`first.fw :fwTexinfo :display`
: Generates the processor documentation and shows it in a window.

`first.fw :fwTexinfo :dvi !xdvi`
: Generates the processor documentation and runs it through texinfo to generate a '`.dvi`'-file. Then, it executes `xdvi` to show the documentation.

`first.fw :fwTexinfo :pdf !acroread`
: Generates the processor documentation and runs it through texinfo and pdflatex to generate a '`.pdf`'-file. Then, it executes `acroread` to show the documentation.

In the following, the different components of the above specification are explained.

```
@=~
```

A FunnelWeb-Specification-file is built up from normal characters and FunnelWeb Commands. Every FunnelWeb-command starts with the macro-character which is `@` by default. Some times it would be more convenient to change the macro character to some other, less used character. The above command changes the macro-character to `~`.

```
~p ‘maximum_input_line_length’ = infinity
~p ‘maximum_output_line_length’ = infinity
```

Normally, a funnelweb-specification file consists of lines with a maximal length of 80 and produces output-files consisting of lines, which are also 80 characters long. The above lines switch off these limitations.

```
~p ‘typesetter’ =  ‘texinfo’
```

FunnelWeb can produce documentation in one of a number of different formats. The above line selects `TexInfo` and is needed to use the `:fwTexinfo`-derivation in Eli. See Section 3.1 [Target], page 11, for more information.

```
~t title titlefont left "A First Funnelweb Specification"
```

This is a special insertion that can be used to typeset a document-title for the generated document. Other possible insertins are a table of contents and a page break. See Section 3.5 [Formatting], page 15, for more information.

```
The file ~{helloworld.lido~} contains ... standard output.
```

Any text between Macro-definitions and output-file-definitions contains documentation that is processed when a formatted documentation is requested. You can structure your document (see Section 3.3 [Structure], page 13), change the font for text-parts (see Section 3.4 [Marking Text], page 14) and insert special text parts such as a title or a table of contents (see Section 3.5 [Formatting], page 15).

```
~O~<helloworld.lido~>~{
...
~}
```

The specification-parts that should be processed by Eli are marked as Output-files with this command. Any text between the opening and the closing brace goes into the specified file, ‘`helloworld.lido`’ in this case. Another method of composing output text are macros which can have parameters and are substituted in the output file command.

2 Producing Specifications

2.1 Output Files

outputfile ::= (‘`@N`’ | ‘`@O`’) *filename* [‘`==`’] ‘`@{`’ *expression* ‘`@}`’ .

When processing to obtain the specification, these directives create an output file named *filename* that contains an *expression* consisting of text and calls to macros (defined later). Filenames must be unique within a FunnelWeb-file.

The ‘`==`’-part is optional an can be left out. It is included to pair the same option for macros.

When processing to obtain the documentation, the *expression* is printed using the `tt font`. Above, the filename is set and below a note is included that this code is attached to an output file.

The difference between ‘`@N`’ and ‘`@O`’ lies in the way in which the files are treated by Eli. A file extracted by ‘`@O`’ becomes part of the specification described by the FunnelWeb file, while a file extracted by ‘`@N`’ does not. Most files should be extracted by ‘`@O`’; *non-product files*, extracted by ‘`@N`’, are those that are used in the derivation of product components, but are not themselves components of the product.

As an example of the use of a non-product file, consider the problem of making keywords case-insensitive but retaining case sensitivity in identifiers (see Section “Making Literal Symbols Case Insensitive” in *Lexical Analysis*). Here is a portion of a FunnelWeb file implementing such a processor:

```
@O@<nolit.gla@>==@{
identifier: C_IDENTIFIER
@}

@N@<keyword.gla@>==@{
$[a-z]+
@}

@O@<keyword.specs@>==@{
keyword.gla :kwd
@}
```

Note that the file ‘`keyword.gla`’ can *not* form part of the final product specification. If it did, the specified processor would treat all completely lower case identifiers as comments! Nevertheless, file ‘`keyword.gla`’ is necessary to specify the representation of the keywords in the *grammar* so that they can be extracted and processed separately (see Section “Making Literal Symbols Case Insensitive” in *Lexical Analysis*). Thus file ‘`keyword.gla`’ is extracted by ‘`@N`’, while the other files are extracted by ‘`@O`’.

2.2 Extracting Output Files with Eli

Within Eli, the derivation `:fwGen` can be used to obtain a directory that contains all the output files contained in a FunnelWeb specification. With the sample specification from the Introduction, you can run following Eli-derivations:

`first.fw :fwGen !ls`
: This generates a directory containing all the output files (only '`helloworld.lido`' in this case) and runs the command `ls` with its name.

`first.fw :fwGen/helloworld.lido>`
: This generates a directory containing all the output files from '`first.fw`'. It then selects the file '`helloworld.lido`' and copies its contents to the standard output.

2.3 Macros

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 ::= '`@$`' *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.

By default, a macro must be invoked exactly once by one other macro or by an output file command. 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!

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 `==`.

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.

2.3.1 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*} .

2.3.2 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`' .

2.4 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.

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. If 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.
```

2.5 Controlling Macro Expansion

2.5.1 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_indent ::= *ps* '`indentation`' *s* '`=`' *s* ('`blank`' / '`none`') .

s ::= {' '}+ .

ps ::= ('`@p`' / '`@P`') ' ' .

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.

2.5.2 Output Length

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* .

s ::= {' '}+ .

ps ::= ('`@p`' / '`@P`') ' ' .

number ::= { *decimal_digit* }+ .

numorinf ::= *number* / '`infinity`' .

The default value is 80 characters.

3 Producing Documentation

3.1 Specifying the Typesetter

One of the design goals of FunnelWeb 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.

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.

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' / 'latex' / 'html' / 'texinfo' /
    'latex2html').

s ::= {' '}+ .

ps ::= ('@p' / '@P') ' ' .
```

The six forms of the pragma look like this.

```
@p typesetter = none
@p typesetter = tex
@p typesetter = latex
@p typesetter = html
@p typesetter = texinfo
@p typesetter = latex2html
```

A source file can contain more than one typesetter pragma, but they must all specify the same value. The default is `none`, in which case FunnelWeb assumes that the documentation file is to be typeset with TeX but that the user has not included any explicit TeX markup. If the typesetter setting is not `none`, FunnelWeb assumes that the free text may contain explicit markup for the specified typesetter.

The typesetter setting affects three things:

Handling of free text

If the typesetter setting is not `none`, FunnelWeb writes the free text *directly* to the documentation file without changing it in any way. 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 TeX and replaces them with TeX commands to typeset the sequences so that they will appear as they do in the input. For example, if `$` (the TEX mathematics mode character) appears in the input file, it will be be written to the documentation file as `\$`.

Handling of macro text

Text within macros and literal strings always appears in the documentation exactly as it does in the FunnelWeb source. Weave intercepts any characters or sequences that might have a special meaning to the specified typesetter and replaces them with commands to typeset that material as it appears in the input.

Boilerplate

If the typesetter setting is `none` or `tex`, the documentation file will begin with a collection of TeX macros implementing FunnelWeb constructs and commands to protect TeX markup. A FunnelWeb file using either of these settings need have no explicit markup whatsoever. No such boilerplate is included for any other typesetter settings. Thus any markup required by the specified typesetter must appear explicitly in the FunnelWeb file. (For example, if the typesetter setting is `latex` then the FunnelWeb file must contain an explicit `\documentclass` command.)

3.2 Typesetting Documentation with Eli

Within the Eli-Implementation of FunnelWeb, the usage of a typesetter-directive is a precondition for the generation of a documentation in the same format.

```
@p typesetter = none
@p typesetter = tex
@p typesetter = latex
@p typesetter = latex2html
```

When these or no typesetter-directive is used, the derivation `:fwTex` can be used to obtain a '`.tex`'-type file. A file of this type can be converted into a '`.dvi`'-file by application of the `:dvi`-derivation or into a '`.ps`'-file by application of the `:ps`-derivation. Of course you need a TEX implementation to use these derivations.

The user must supply an appropriate LaTeX preamble, `\begin{document}` and `\end{document}` as part of the document text when using either `latex` or `latex2html`. This means that it is possible to combine the outputs from several `:fwTex` derivations into a single document by using appropriate LaTeX `\input` commands.

The preamble must include the LaTeX command `\usepackage{alltt}` for either `latex` or `latex2html`, and `latex2html` also requires `\usepackage{html}`.

Pragmas `latex` and `latex2html` normally translate the FunnelWeb section directives into LaTeX sectioning commands as follows:

- `@A` `\section`
- `@B` `\subsection`
- `@C` `\subsubsection`
- `@D` `\paragraph`
- `@E` `\subparagraph`

If the `+chapter` parameter is passed to the `:fwTex` derivation, however, the translation is:

- `@A` `\chapter`
- `@B` `\section`
- `@C` `\subsection`
- `@D` `\subsubsection`
- `@E` `\paragraph`

`@p typesetter = texinfo`

When these directive is used, the derivation `:fwTexinfo` can be used to obtain a '`.tnf`'-type file. Since '`.tnf`'-files are valid TEX-files, all the derivations for `.tex`-files are also valid for `.tnf`-files. Additionally, `.tnf`-type files can be converted into info-files which can be shown with the Eli-Info-Browser by application of the `:display`-derivation.

`@p typesetter = html`

When these directive is used, the derivation `:fwHtml` can be used to obtain a '`.html`'-type file. This file can be extracted from Eli and can be loaded into your favorite html-browser.

3.3 Structure

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`' .

name ::= '`@<`' *name_text* '`@>`' .

name_text ::= {*ordinary_char* / *text_special*} .

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` to the lowest level of `E`. 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 `@B`). 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 (ie. 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 (eg. 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.

3.4 Marking Text

3.4.1 Typesetting a literal string

Experience has shown that one of the most common typesetting requirements 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.

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.
```

3.4.2 Emphasising a string

The emphasis directive is very similar to the literal directive except that it causes its argument to be typeset in an emphasised manner (eg. 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.
```

3.5 Formatting

3.5.1 Forcing a Pagebreak

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

ftd_newpage ::= ‘`@t `’ ‘`new_page`’ .

Its 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.

3.5.2 Producing a table of contents

The table of contents pragma is a typesetting pragma with the following syntax.

ftd_toc ::= ‘`@t `’ ‘`table_of_contents`’ .

Its only form looks like this.

```
@t table_of_contents
```

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

3.5.3 Inserting blank vertical space

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 ::= ‘`@t `’ ‘`vskip`’ ‘ ’ *number* ‘ `mm`’.

For example:

```
@t vskip 26 mm
```

3.5.4 Specifying the title of your document

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*} ‘`"`’ .

Its 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"
```

4 Input Processing

4.1 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* which introduces a *special sequence*. Thus, the scanner partitions the input file into ordinary text and special sequences. (The control 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*.)

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 table gives all the possible characters that can follow the special character, and the legality of each sequence. The item headings give the ASCII number of each ASCII character and the special sequence for that character. The descriptions start with 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.

000–008	Unprintable characters and hence illegal.
009	Tab. Converted by Eli (not FunnelWeb) into the appropriate number of spaces.
010–031	Unprintable characters and hence illegal.
032 @	- Illegal (space).
033 @!	C Comment.
034 @"	S Parameter delimiter.
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.
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–@z	Identical to @A–@Z.
123 @{	S Open macro body/Open literal directive.
124 @|	- Illegal.
125 @}	S Close macro body/Close literal directive.
126 @~	- Illegal.
127–255	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.

4.2 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 (ie. 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 (eg. a list of internet electronic mail addresses).

4.3 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 `@#@#@`.

4.4 Inserting Arbitrary Characters into the Text

While FunnelWeb does not tolerate unprintable characters in the input file (except for the end of line character and the tabs that Eli expands into spaces), 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.

4.5 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.

4.6 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 (eg. `@<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
```

4.7 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.

4.8 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.

4.9 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 inserts one (in the copy in memory). In Eli, an include file must have type '`.fwi`'.

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.

4.10 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* .

ps ::= ('`@p`' / '`@P`') ' ' .

number ::= { *decimal_digit* }`+` .

numorinf ::= *number* / '`infinity`' .

s ::= {' '}`+` .

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.

5 Grammar

input_file ::= {*ordinary_text* / *special_sequence*} .

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’ .

quick_name ::= ‘@#’ *non_space_printable* .

pragma ::= *pragma_indent* / *pragma_mill* / *pragma_moll* /
pragma_typesetter .

s ::= {‘ ’}+ .

ps ::= (‘@p’ / ‘@P’) ‘ ’ .

number ::= { *decimal_digit* }+ .

numorinf ::= *number* / ‘infinity’ .

pragma_indent ::= *ps* ‘indentation’ *s* ‘=’ *s* (‘blank’ / ‘none’) .

pragma_mill ::= *ps* ‘maximum_input_line_length’ *s* ‘=’ *s* *numorinf* .

pragma_moll ::= *ps* ‘maximum_output_line_length’ *s* ‘=’ *s* *numorinf* .

pragma_typesetter ::= *ps* ‘typesetter’ *s* ‘=’ *s* (‘none’ / ‘tex’ / ‘texinfo’).

ftd ::= *ftd_newpage* / *ftd_toc* / *ftd_vskip* / *ftd_title* .

ts ::= ‘`@t `’ .

ftd_newpage ::= *ts* ‘`new_page`’ .

ftd_toc ::= *ts* ‘`table_of_contents`’ .

ftd_vskip ::= *ts* ‘`vskip`’ *s number s* ‘`mm`’ .

ftd_title ::= *ts* ‘`title`’ *s font s alignment text* .

font ::= ‘`normalfont`’ / ‘`titlefont`’ / ‘`smalltitlefont`’ .

alignment ::= ‘`left`’ / ‘`centre`’ / ‘`right`’ .

text ::= ‘`"`’ {*printable_char*} ‘`"`’ .

input_file ::= {*text* / *macro* / *directive*} .

free_text ::= *ordinary_text* .

ordinary_text ::= {*ordinary_char* / *eol* / *text_special*}`+` .

text_special ::= ‘`@+`’ / ‘`@@`’ / ‘`@^`’ *char_spec* .

ordinary_char ::= ‘` `’..‘`~`’-*special* .

directive ::= *ftd* / *itd* .

itd ::= *section* / *literal* / *emphasis* .

section ::= ‘`@`’ *levelchar* [*name*] .

levelchar ::= ‘`A`’ / ‘`B`’ / ‘`C`’ / ‘`D`’ / ‘`E`’ /
‘`a`’ / ‘`b`’ / ‘`c`’ / ‘`d`’ / ‘`e`’ .

literal ::= ‘`@{`’ *ordinary_text* ‘`@}`’ .

emphasise ::= ‘`@/`’ *ordinary_text* ‘`@/`’ .

macro ::= (‘`@O`’ / ‘`@$`’) *name* [*formal_parameter_list*] .
[‘`@Z`’] [‘`@M`’] [‘`==`’ / ‘`+=`’] ‘`@{`’ *expression* ‘`@}`’ .

name ::= ‘`@<`’ *name_text* ‘`@>`’ .

name_text ::= {*ordinary_char* / *text_special*} .

formal_parameter_list ::= ‘`@(`’ *formal_parameter* ‘`@)`’ .

formal_parameter ::= ‘`@1`’ / ‘`@2`’ / ‘`@3`’ / ‘`@4`’ / ‘`@5`’ /
‘`@6`’ / ‘`@7`’ / ‘`@8`’ / ‘`@9`’ .

expression ::= {*ordinary_text* / *macro_call* / *formal_parameter*} .

macro_call ::= *name* [*actual_parameter_list*] .

actual_parameter_list ::= ‘`@(`’ *actpar* { ‘`@,`’ *actpar* } ‘`@)`’ .

actpar ::= *expression* /
(*whitespace* ‘`@"`’ *expression* ‘`@"`’ *whitespace*) .

whitespace ::= {‘ ’ / *eol*} .

Index