

## The web<sub>0</sub> system

Dag Langmyhr

11th April 2007



**Department of Informatics  
University of Oslo**

# Contents

<b>1</b>	<b>Using <code>web<sub>0</sub></code></b>	<b>3</b>
1.1	The documentation . . . . .	3
1.2	The code . . . . .	3
1.3	Comparison to <code>web</code> . . . . .	10
<b>2</b>	<b>Implementation</b>	<b>13</b>
2.1	The <code>tangleO</code> program . . . . .	13
2.1.1	Definitions . . . . .	13
2.1.2	Parameter decoding . . . . .	13
2.1.3	Running the pre- and postprocessors . . . . .	15
2.1.4	Auxiliary functions . . . . .	16
2.2	The <code>weaveO</code> program . . . . .	16
2.2.1	Definitions . . . . .	17
2.2.2	Parameter decoding . . . . .	17
2.2.3	Running the pre- and postprocessors . . . . .	20
2.2.4	Auxiliary functions . . . . .	21
2.3	The <code>wOpre</code> filter . . . . .	21
2.3.1	The main program . . . . .	21
2.3.2	Definitions . . . . .	24
2.3.3	Initialization . . . . .	24
2.3.4	Parameter handling . . . . .	24
2.3.5	Extracting tokens . . . . .	25
2.3.6	Meta symbol definitions . . . . .	25
2.3.7	Handling lines of documentation . . . . .	26
2.3.8	File name check . . . . .	26
2.3.9	Utility functions . . . . .	26
2.4	The <code>wOcode</code> filter . . . . .	27
2.4.1	Definitions . . . . .	28
2.4.2	Parameter handling . . . . .	28
2.4.3	Reading the tokens . . . . .	29
2.4.4	Expanding the meta symbols . . . . .	30
2.4.5	Utility functions . . . . .	31
2.5	The <code>wO-f-latex</code> filter . . . . .	32
2.5.1	Initialization . . . . .	32
2.5.2	Parameter handling . . . . .	34
2.5.3	Pass 1 . . . . .	34
2.5.4	Pass 2 . . . . .	35
2.5.5	Pass 3 . . . . .	37
2.5.6	Making an index . . . . .	41
2.5.7	Utility functions . . . . .	43
2.6	Language filters . . . . .	43
2.6.1	The C language filter <code>wO-l-c</code> . . . . .	44
2.6.2	The Java language filter <code>wO-l-java</code> . . . . .	48
2.6.3	The $\LaTeX$ language filter <code>wO-l-latex</code> . . . . .	52
2.6.4	The Perl language filter <code>wO-l-perl</code> . . . . .	55
2.7	Miscellaneous . . . . .	57
2.7.1	The Perl interpreter . . . . .	57
2.7.2	Program version . . . . .	57

---

2.8	Adapting text for processing by $\text{\LaTeX}$ . . . . .	58
2.8.1	Handle “\”, “{”, and “}” . . . . .	58
2.8.2	Handle the other special $\text{\LaTeX}$ characters . . . . .	58
2.8.3	Handle other ISO Latin-1 characters . . . . .	59
2.8.4	Avoing unwanted ligatures . . . . .	59
2.8.5	Handle blanks . . . . .	59
2.9	Expanding TAB characters . . . . .	60
2.10	Printing user messages . . . . .	60
2.10.1	The function <code>&amp;Error</code> . . . . .	60
2.10.2	The function <code>&amp;Message</code> . . . . .	60
2.11	Alphabetical sorting . . . . .	61
<b>3</b>	<b><math>\text{\LaTeX}</math> support</b> . . . . .	<b>63</b>
3.1	The webzero package . . . . .	63
3.1.1	Package identification . . . . .	63
3.1.2	Package options . . . . .	63
3.1.3	Package loading . . . . .	65
3.1.4	Implementation of interface . . . . .	65
3.1.5	Typesetting the index . . . . .	68
3.1.6	Page style . . . . .	69
3.1.7	Utility macros . . . . .	69
3.1.8	End of class . . . . .	71
3.2	The webzero document class . . . . .	71
3.2.1	Class identification . . . . .	71
3.2.2	Class options . . . . .	71
3.2.3	Package and class loading . . . . .	71
3.2.4	Main code . . . . .	72
3.2.5	End of class . . . . .	72
<b>4</b>	<b>Documentation</b> . . . . .	<b>73</b>
4.1	Man page for <code>tangleO</code> . . . . .	73
4.1.1	Identification . . . . .	73
4.1.2	Program description . . . . .	73
4.1.3	The parameters . . . . .	74
4.2	Man page for <code>weaveO</code> . . . . .	74
4.2.1	Identification . . . . .	74
4.2.2	Program description . . . . .	74
4.2.3	The parameters . . . . .	75
4.3	Common man page information . . . . .	75
4.3.1	The name of the author . . . . .	75
4.3.2	Cross reference information . . . . .	75
	<b>References</b> . . . . .	<b>77</b>

## List of Figures

1	The <code>web<sub>0</sub></code> file <code>bubble.wO</code> , page 1 . . . . .	4
2	The <code>web<sub>0</sub></code> file <code>bubble.wO</code> , page 2 . . . . .	5
3	The documentation created from <code>bubble.wO</code> , page 1 . . . . .	6
4	The documentation created from <code>bubble.wO</code> , page 2 . . . . .	7
5	The documentation created from <code>bubble.wO</code> , page 3 . . . . .	8
6	The documentation created from <code>bubble.wO</code> , page 4 . . . . .	9
7	The C code extracted from <code>bubble.wO</code> . . . . .	10
8	The <code>tangleO</code> program . . . . .	14
9	The <code>weaveO</code> program . . . . .	17
10	Another typical <code>web<sub>0</sub></code> source text <code>Hello.wO</code> . . . . .	22
11	Tokens produced by <code>wOpre</code> from <code>Hello.wO</code> . . . . .	23
12	<code>L<sup>A</sup>T<sub>E</sub>X</code> code produced by <code>weaveO</code> from <code>Hello.wO</code> . . . . .	33

# The web<sub>0</sub> system

Dag Langmyhr

THE web<sub>0</sub> system is a tool for *literate programming*, the programming style invented by Donald Knuth[Knu83, Knu84, Knu92]. This style recognizes that programs ought to be written for people rather than computers; consequently, program code and documentation are intermixed. This gives the following advantages:

- The program can be written in a sequence that is easier to explain to human readers, rather than the one required by the programming language.
- Documentation and program are in the same file, making them easier to maintain.
- The programmer can use all kinds of typographical features to enhance the documentation, like mathematical formulæ, section headers, tables, figures, footnotes, and others.

Donald Knuth created the original web system to implement the T<sub>E</sub>X and METAFONT programs in Pascal. Since then, versions for other programming languages have appeared, like CWeb[Lev87] for the C programming language.

In 1989, Norman Ramsay designed noweb[Ram89] which is a language-independent version of web. web<sub>0</sub> is inspired by noweb, but aims to be simpler to understand and adapt. Also, it is written in Perl[WCS96] rather than Awk and Icon.

Compared to Donald Knuth's original web, web<sub>0</sub> is a much simpler tool, but perhaps more general. A comparison between web and web<sub>0</sub> is given in Section 1.3 on page 10.



# 1 Using `web0`

This Section tells the reader how to write documented programs in the `web0` notation. A short article describing an implementation of bubble sorting is given as an example. In Figures 1 and 2 is shown the source file `bubble.w0` containing the combined program and documentation in `web0` notation.

The printed documentation is produced by executing

```
weave0 -l c -e -o bubble.tex bubble.w0
ltx1 bubble.tex
```

and the result is shown in Figures 3–6. The last two pages show the tables that `web0` can generate on request (“`\wzvarindex`” and “`\wzmetaindex`”).

Executing

```
tangle0 -o bubble.c bubble.w0
```

will extract the C code file `bubble.c`. This file is shown in Figure 7 on page 10.<sup>2</sup>

## 1.1 The documentation

Any part of the `web0` source that is not code (see Section 1.2) is treated as documentation. This documentation is written exactly as one would write any other document for the chosen DTP program.

When using  $\text{\LaTeX}$ , however, the user should remember the following:

- The documentation must either use the document class `webzero` (see Section 3.2 on page 71) or the package `webzero` (see Section 3.1 on page 63).

## 1.2 The code

Writing code in the `web0` system consists of defining a lot of *meta symbols*, like

```
<<main program>> =
begin
  <<declarations>>

  <<statements>>
end
@
```

This definition states that the meta symbol  $\langle main\ program \rangle$  is defined to expand to the given text, which may include other meta symbols defined elsewhere. The user may extract any part of code that he or she wants by naming the starting meta symbol.

When writing `web0` code, the following points should be considered:

---

<sup>1</sup>The program `ltx` automatically runs `latex` the required number of times; for more information, see <http://www.ifi.uio.no/it/store/opt/doc/ltx.pdf>.

<sup>2</sup>The code shown in Figure 7 on page 10 is not particularly easy to read, but this code is not intended for human eyes, only for computers.

Figure 1: The web<sub>0</sub> file bubble.w0, page 1

---

```

\documentclass[12pt,a4paper]{webzero}
\usepackage[latin1]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{amssymb,palatino,mathpple}

\title{Bubble sort}
\author{Dag Langmyhr\ Department of Informatics\
  University of Oslo\ [5pt] \texttt{dag@ifi.uio.no}}

\begin{document}
\maketitle

\noindent This short article describes \emph{bubble
  sort}, which quite probably is the easiest sorting
  method to understand and implement.
  Although far from being the most efficient one, it is
  useful as an example when teaching sorting algorithms.

  Let us write a function \texttt{bubble} in C which sorts
  an array \texttt{a} with \texttt{n} elements. In other
  words, the array \texttt{a} should satisfy the following
  condition when \texttt{bubble} exits:
  \[
  \text{forall } i, j \text{ in } \mathbb{N}: 0 \leq i < j < \text{mathtt{n}}
  \text{Rightarrow } \text{mathtt{a}[i] } \leq \text{mathtt{a}[j]
  \]

  <<bubble sort>>=
  void bubble(int a[], int n)
  {
    <<local variables>>

    <<use bubble sort>>
  }
  @
  Bubble sorting is done by making several passes through the
  array, each time letting the larger elements
  “bubble” up. This is repeated until the array is
  completely sorted.

  <<use bubble sort>>=
  do {
    <<perform bubbling>>
  } while (<<not sorted>>);
  @

```

---

Figure 2: The `web0` file `bubble.w0`, page 2

---

Each pass through the array consists of looking at every pair of adjacent elements; \footnote{We could, on the average, double the execution speed of `\texttt{bubble}` by reducing the range of the `\texttt{for}`-loop by `\sim 1` each time. Since a simple implementation is the main issue, however, this improvement was omitted.} if the two are in the wrong sorting order, they are swapped:

```
<<perform bubbling>>=
<<initialize>>
for (i=0; i<n-1; ++i)
  if (a[i]>a[i+1]) { <<swap a[i] and a[i+1]>> }
@
The \texttt{for}-loop needs an index variable
\texttt{i}:

<<local var...>>=
int i;
@
Swapping two array elements is done in the standard way
using an auxiliary variable \texttt{temp}. We also
increment a swap counter named \texttt{n\_swaps}.

<<swap ...>>=
temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
++n_swaps;
@
The variables \texttt{temp} and \texttt{n\_swaps}
must also be declared:

<<local var...>>=
int temp, n_swaps;
@
The variable \texttt{n\_swaps} counts the number of
swaps performed during one “bubbling” pass.
It must be initialized prior to each pass.

<<initialize>>=
n_swaps = 0;
@
If no swaps were made during the “bubbling” pass,
the array is sorted.

<<not sorted>>=
n_swaps > 0
@

\wzvarindex \wzmetaindex
\end{document}
```

---

Figure 3: The documentation created from bubble.w0, page 1

## Bubble sort

Dag Langmyhr  
 Department of Informatics  
 University of Oslo  
 dag@ifi.uio.no  
 April 11, 2007

This short article describes *bubble sort*, which quite probably is the easiest sorting method to understand and implement. Although far from being the most efficient one, it is useful as an example when teaching sorting algorithms.

Let us write a function `bubble` in C which sorts an array `a` with `n` elements. In other words, the array `a` should satisfy the following condition when `bubble` exits:

$$\forall i, j \in \mathbb{N} : 0 \leq i < j < n \Rightarrow a[i] \leq a[j]$$

**#1** `<bubble sort>`  $\equiv$

```

1 void bubble(int a[], int n)
2 {
3     <local variables #4(p.1)>
4
5     <use bubble sort #2(p.1)>
6 }

```

*(This code is not used.)*

Bubble sorting is done by making several passes through the array, each time letting the larger elements “bubble” up. This is repeated until the array is completely sorted.

**#2** `<use bubble sort>`  $\equiv$

```

7 do {
8     <perform bubbling #3(p.1)>
9 } while ( (not sorted #7(p.2)) );

```

*(This code is used in #1 (p.1).)*

Each pass through the array consists of looking at every pair of adjacent elements;<sup>1</sup> if the two are in the wrong sorting order, they are swapped:

**#3** `<perform bubbling>`  $\equiv$

```

10 <initialize #6(p.2)>
11 for (i=0; i<n-1; ++i)
12     if (a[i]>a[i+1]) { <swap a[i] and a[i+1] #5(p.2)> }

```

*(This code is used in #2 (p.1).)*

The `for`-loop needs an index variable `i`:

**#4** `<local variables>`  $\equiv$

```

13 int i;

```

*(This code is extended in #4<sub>s</sub> (p.2). It is used in #1 (p.1).)*

---

<sup>1</sup>We could, on the average, double the execution speed of `bubble` by reducing the range of the `for`-loop by 1 each time. Since a simple implementation is the main issue, however, this improvement was omitted.

File: `bubble.w0` page 1

**Figure 4:** The documentation created from bubble.w0, page 2

Swapping two array elements is done in the standard way using an auxiliary variable `temp`. We also increment a swap counter named `n_swaps`.

```
#5 <swap a[i] and a[i+1]> ≡  
14 temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;  
15 ++n_swaps;  
(This code is used in #3 (p.1))
```

The variables `temp` and `n_swaps` must also be declared:

```
#4a <local variables #4(p.1)> +≡  
16 int temp, n_swaps;
```

The variable `n_swaps` counts the number of swaps performed during one “bubbling” pass. It must be initialized prior to each pass.

```
#6 <initialize> ≡  
17 n_swaps = 0;  
(This code is used in #3 (p.1))
```

If no swaps were made during the “bubbling” pass, the array is sorted.

```
#7 <not sorted> ≡  
18 n_swaps > 0  
(This code is used in #2 (p.1))
```

File: *bubble.w0* page 2

**Figure 5:** The documentation created from bubble.w0, page 3

<b>Variables</b>	
<b>A</b>	
a .....	<u>1</u> , 12, 14
<b>I</b>	
i .....	11, 12, <u>13</u> , 14
<b>N</b>	
n .....	<u>1</u> , 11
n_swaps .....	15, <u>16</u> , 17, 18
<b>T</b>	
temp .....	14, <u>16</u>

VARIABLES page 3

**Figure 6:** The documentation created from bubble.wO, page 4

<b>Meta symbols</b>	
<i>&lt;bubble sort #1&gt;</i> .....	page 1 *
<i>&lt;initialize #6&gt;</i> .....	page 2
<i>&lt;local variables #4&gt;</i> .....	page 1
<i>&lt;not sorted #7&gt;</i> .....	page 2
<i>&lt;perform bubbling #3&gt;</i> .....	page 1
<i>&lt;swap a[i] and a[i+1] #5&gt;</i> .....	page 2
<i>&lt;use bubble sort #2&gt;</i> .....	page 1

(Symbols marked with \* are not used.)

META SYMBOLS page 4

**Figure 7:** The C code extracted from `bubble.w0`


---

```

void bubble(int a[], int n)
{
    int i;
    int temp, n_swaps;

    do {
        n_swaps = 0;
        for (i=0; i<n-1; ++i)
            if (a[i]>a[i+1]) { temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
                ++n_swaps; }
    } while (n_swaps > 0);
}

```

---

- The notation for meta symbols is `<<symbol name>>`. A meta symbol name may contain any character except “<” or “>”.
- A meta symbol definition continues until a line starting with a “@” and containing nothing else (except possibly spaces).
- The notation `<<>>` is shorthand for the last meta symbol defined.
- The notation `<<xxx. . . >>` (ending with three dots) is shorthand for any meta symbol whose name starts with “xxx” and has previously been defined or referenced. Only one meta symbol may match the given prefix.
- The programmer may not use both `<<` and `>>` on the same line in the program code, as this will confuse the scanner (which will regard it as a quaint meta symbol).

### 1.3 Comparison to `web`

As mentioned in the preface, `web0` is based on the ideas of Donald Knuth’s `web` program, but there are quite a few differences:

- `web` supports the Pascal programming language; `web0` is language independent.
- A `web` file can contain only one program; `web0` files may contain several, and these programs may share code.
- A `web` program must be on one file (plus an optional change file); `web0` programs may reside on several files.
- The implementation of `web`’s `tangle` and `weave` consists of 3315 and 4904 lines of “tangled” Pascal code, respectively. The corresponding programs in `web0` (`tangle0` and `weave0` with auxiliary programs) contain only 205 and 468 lines of “tangled” Perl code.
- `web` knows 36 commands; `web0` knows only one.
- In `web`, the programmer can insert `TEX` text into the code parts; this is not possible in `web0`.

- `web0` does not support the change file concept of `web`.
- In `web`, one can insert program code into the documentation part. There are no explicit capabilities for this in `web0`, but the ordinary `LATEX` mechanisms should be sufficient.



## 2 Implementation

The `web0` system consists of two programs `tangle0` and `weave0`, but it is implemented as many small processes. The reasons for this are:

- It is easier to write and maintain smaller programs with a well-defined interface.
- It is easier to modify the system, for instance if documentation in another format than  $\text{\LaTeX}$  is required.

### 2.1 The `tangle0` program

The `tangle0` program reads a set of `web0` files and translates them into executable code. The program itself is just a wrapper for the two programs that do the real work: the preprocessor `wOpre` and the postprocessor `wOcode`, as shown in figure 8 on the following page.

The `tangle0` program is written in Perl:

```
#1 <tangle0> ≡  
1 <perl #105 (p.57)>  
2  
3 <tangle0 definitions #2 (p.13)>  
4 <tangle0 parameter decoding #3 (p.14)>  
5 <tangle0 processing #6 (p.16)>  
6  
7 <tangle0 auxiliary functions #7 (p.16)>  
8 <user message functions #110 (p.60)>  
(This code is not used.)
```

#### 2.1.1 Definitions

All programs should be able to identify themselves with name and version.

```
#2 <tangle0 definitions> ≡  
9 my ($Prog, $Version) = ("tangle0", "<version #106 (p.57)>");  
(This code is extended in #2a (p.13). It is used in #1 (p.13).)
```

Since `tangle0` is to start two auxiliary processes, it must know where to find them. The variable `$Lib_dir` is initialized to the path to the correct directory. In this source listing it is given as `"."` (which means the current directory) but this will be modified by the “make install” command specified in the `Makefile`.

```
#2a <tangle0 definitions #2 (p.13)> +≡  
10 my $Lib_dir = ".";  
(This code is extended in #2b (p.13).)
```

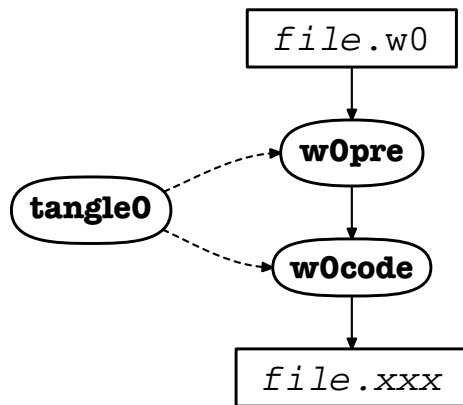
#### 2.1.2 Parameter decoding

This loop looks at all the parameters. They are decoded and put in `@Pre_opt` if they go to `wOpre`, and in `@Code_opt` if they are for `wOcode`.

```
#2b <tangle0 definitions #2 (p.13)> +≡  
11 my (@Code_opt, @Pre_opt);  
(This code is extended in #2c (p.14).)
```

File names are kept in `@Pre_files`.

Figure 8: The tangle0 program



```

#2c <tangle0 definitions #2 (p.13)> +≡
12 my @Pre_files = ();
    (This code is extended in #2d (p.15).)
  
```

Note that all file names are quoted in case they contain strange characters (like spaces). If no input files are given, tangle0 will read from *standard input*. This is handled automatically by Perl.

```

#3 <tangle0 parameter decoding> ≡
13 PARAM:
14 while (@ARGV) { $_ = shift;
15     <tangle0 parameters #4 (p.14)>
16
17     push @Pre_files, "'$_'";
18 }
    (This code is used in #1 (p.13).)
  
```

**2.1.2.1 The `-o` option** is used to specify the name of the output file. This option is passed on to w0code, and — since the file name may contain strange characters, like “&” or “>” — the parameter is quoted.

```

#4 <tangle0 parameters> ≡
19 /^-o$/ and do { $_ = shift or &Usage;
20     push @Code_opt, "-o'$_'"; next PARAM; };
21 /^-o(.+)$/ and do {
22     push @Code_opt, "-o'$1'"; next PARAM; };
    (This code is extended in #4a (p.15). It is used in #3 (p.14).)
  
```

We must not forget a short description of the parameter in the man page.

```

#5 <tangle0 man page parameters> ≡
23 .TP
24 .B -o \fifile\fp
25 Specify the name of the file on which to write the extracted program
26 code. If this option is not used, the output will go to
27 .I standard output.
    (This code is extended in #5a (p.15). It is used in #133 (p.74).)
  
```

**2.1.2.2 The `-v` option** makes tangle0 print its version identification and some information on what it is doing. This parameter is also passed on to both w0pre and w0code so they can do the same.

```
#4a <tangle0 parameters #4 (p.14)> +≡
28 /^-v$/ and do {
29     print STDERR "This is $Prog (version $Version)\n";
30     push @Pre_opt, "-v"; push @Code_opt, "-v";
31     $Verbose = "Yes"; next PARAM; };
    (This code is extended in #4b (p.15).)
```

The default is to be silent.

```
#2a <tangle0 definitions #2 (p.13)> +≡
32 my $Verbose = 0;
```

The `-v` parameter must also be mentioned in the man page.

```
#5a <tangle0 man page parameters #5 (p.14)> +≡
33 .TP
34 .B -v
35 State the program version. Use of this option will also make
36 .I tangle0
37 more verbose so that it will display information on what it does.
    (This code is extended in #5b (p.15).)
```

**2.1.2.3 The `-x` option** names the meta symbol with which to start the program extraction. It is passed on to `wOcode`.

```
#4b <tangle0 parameters #4 (p.14)> +≡
38 /^-x$/ and do { $_ = shift or &Usage;
39     push @Code_opt, "-x'$_'"; next PARAM; };
40 /^-x(.+)/ and do {
41     push @Code_opt, "'$1'"; next PARAM; };
    (This code is extended in #4c (p.15).)
```

And now for the man file description of the `-x` parameter.

```
#5b <tangle0 man page parameters #5 (p.14)> +≡
42 .TP
43 .B -x \fIname\fP
44 Specify the name of the meta symbol with which to start the program
45 extraction. If this option is not used, extraction will start with the
46 first meta symbol defined.
```

**2.1.2.4 Unknown options** All other parameters result in a warning.

```
#4c <tangle0 parameters #4 (p.14)> +≡
47 /^-/ and do { &Message("Unknown option '$_' ignored."); next PARAM; };
```

### 2.1.3 Running the pre- and postprocessors

Running the pre- and postprocessor programs could have been accomplished using

```
system("wOpre ... | wOcode ... ");
```

but that would have made it impossible to detect run-time errors in either program.<sup>3</sup> To be able to detect all errors, `tangle0` must start `wOpre` as an input process and `wOcode` as an output process and pass all data from one

<sup>3</sup>The documentation in “`man sh`” states that

“The exit status of a pipeline is the exit status of the last command in the pipeline.”

to the other. When the `wOpre` process is finished, it is possible to close the two processes and detect an error exit status.

```
#6 <tangle0 processing> ≡
48 &Message("Running $Lib_dir/wOpre @Pre_opt @Pre_files " .
49 " | $Lib_dir/wOcode @Code_opt") if $Verbose;
50 open(PRE, "$Lib_dir/wOpre @Pre_opt @Pre_files |");
51 open(POST, " | $Lib_dir/wOcode @Code_opt");
52 print POST while <PRE>;
53 close PRE; exit $?>>8 if $?>>8;
54 close POST; exit $?>>8;
(This code is used in #1 (p.13).)
```

### 2.1.4 Auxiliary functions

If the user makes a parameter error, he or she should get a short notification on how to do it correctly.

```
#7 <tangle0 auxiliary functions> ≡
55 sub Usage {
56     print STDERR "Usage: $Prog [-o file] [-v] [-x symbol] [file...]\n";
57     exit 1;
58 }
(This code is used in #1 (p.13).)
```

## 2.2 The `weave0` program

The `weave0` program reads a set of `web0` files and produces the documentation in a suitable format. The program is constructed according to the same principle as `tangle0`, as shown in Figure 9 on the next page. `weave0` is a wrapper for up to three programs processing the data:

**wOpre** is the same preprocessor as is used by `tangle0`.

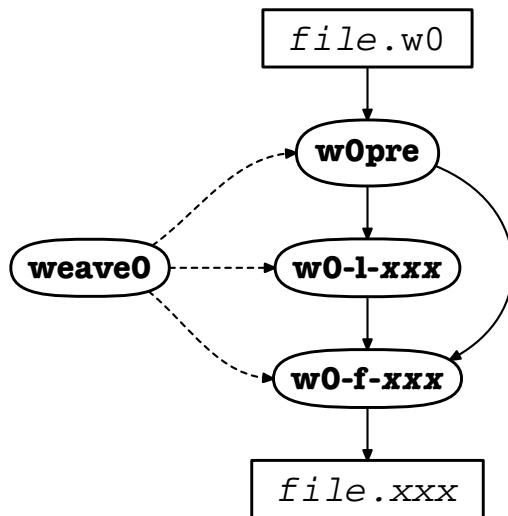
**wO-l-xxx** is an optional language dependant filter, such as `wO-l-c` for C programs and `wO-l-perl` for programs in Perl.

**wO-f-xxx** is the postprocessor producing the actual documentation commands. At present, only one such postprocessor is supplied: `wO-f-latex` that produces  $\text{\LaTeX}$  code.

The `weave0` program is also written in Perl:

```
#8 <weave0> ≡
59 <perl #105 (p.57)>
60
61 <weave0 definitions #9 (p.17)>
62 <weave0 parameter decoding #10 (p.17)>
63 <weave0 processing #16 (p.20)>
64
65 <weave0 auxiliary functions #17 (p.21)>
66 <user message functions #110 (p.60)>
(This code is not used.)
```

Figure 9: The weaveO program



### 2.2.1 Definitions

All programs should be able to identify themselves with name and version.

```
#9 <weaveO definitions> ≡
67 my ($Prog, $Version) = ("weaveO", "<version #106 (p.57)>");
    (This code is extended in #9a (p.17). It is used in #8 (p.16).)
```

Since weaveO is to start two or three auxiliary processes, it must know where to find them. The variable \$Lib\_dir is initialized to the path to the correct directory. In this source listing it is given as "." (which means the current directory) but this will be modified by the “make install” command specified in the Makefile.

```
#9a <weaveO definitions #9 (p.17)> +≡
68 my $Lib_dir = ".";
    (This code is extended in #9b (p.17).)
```

### 2.2.2 Parameter decoding

The following loop will examine all the parameters:

```
#10 <weaveO parameter decoding> ≡
69 <set default parameter values #14 (p.18)>
70 PARAM:
71 while (@ARGV) { $_ = shift;
72     <weaveO parameters #11 (p.18)>
73     push @Pre_files, "'$_'";
74 }
    (This code is used in #8 (p.16).)
```

The parameters are decoded and put in @Pre\_opt if they go to w0pre, in @Lang\_opt if they are sent to the language processor w0-l-xxx, and in @Code\_opt if they are for the postprocessor w0-f-xxx.

```
#9b <weaveO definitions #9 (p.17)> +≡
76 my (@Code_opt, @Lang_opt, @Pre_opt);
    (This code is extended in #9c (p.18).)
```

File names are kept in `@Pre_files`.

```
#9c <weave0 definitions #9 (p.17)> +≡
77 my @Pre_files = ();
    (This code is extended in #9d (p.18).)
```

If no input files are given, `weave0` will read from *standard input*. This is handled automatically by Perl.

**2.2.2.1 The `-e` option** indicates that the user wants to enhance the code by using bold or italic fonts. (This option will only work if a language filter is specified; see Section 2.2.2.3 on the facing page.)

```
#11 <weave0 parameters> ≡
78 /^-e$/ and do { push @Lang_opt, "-e"; next PARAM; };
    (This code is extended in #11a (p.18). It is used in #10 (p.17).)
```

Like all the options, this one must be described in the man page.

```
#12 <weave0 man page parameters> ≡
79 .TP
80 .B -e
81 Enhance the code by using bold and italic fonts.
82 (Works only in conjunction with the
83 .B -l
84 option; see below.)
    (This code is extended in #12a (p.19). It is used in #137 (p.75).)
```

**2.2.2.2 The `-f` option** is used to specify which postprocessor filter to use.

```
#11a <weave0 parameters #11 (p.18)> +≡
85 /^-f$/ and do { $_ = shift;
86     <weave0: note filter #13 (p.18)>; next PARAM; };
87 /^-f(.+)/ and do { $_ = $1;
88     <weave0: note filter #13 (p.18)>; next PARAM; };
    (This code is extended in #11b (p.19).)
```

To check that a correct filter has been specified, it must be checked. The path name of the corresponding filter program will be kept in `$F_prog`.

```
#13 <weave0: note filter> ≡
89 &Usage unless $_;
90 my $pr = "$Lib_dir/wO-f-$_";
91 if (-r $pr && -x $pr) {
92     $F_prog = $pr;
93 } else {
94     &Message("Filter $_ is unknown;", "use of -f option ignored.");
95 }
    (This code is used in #11a (p.18) and #14 (p.18).)
```

The variable `$F_prog` must be declared.

```
#9d <weave0 definitions #9 (p.17)> +≡
96 my $F_prog = "";
    (This code is extended in #9e (p.19).)
```

The default output format is `LATEX`:

```
#14 <set default parameter values> ≡
97 $_ = "latex"; <weave0: note filter #13 (p.18)>;
    (This code is used in #10 (p.17).)
```

The man page contains documentation on this option:

```
#12a <weave0 man page parameters #12 (p.18)> +≡
98 .TP
99 .B -f \fifilter\fP
100 Specify which output filter to use. At present, only
101 .I latex
102 is available, so this is the default.
    (This code is extended in #12b (p.19).)
```

**2.2.2.3 The `-l` option** is used to specify the programming language used, when the user wants to employ the language dependant filter.

```
#11b <weave0 parameters #11 (p.18)> +≡
103 /^-l$/ and do { $_ = shift;
104     <weave0: note language #15 (p.19)>; next PARAM; };
105 /^-l(.+)/ and do { $_ = $1;
106     <weave0: note language #15 (p.19)>; next PARAM; };
    (This code is extended in #11c (p.19).)
```

To ensure that the user has specified an existing language filter, `weave0` must check that the filter program exists. The path name of that filter program will be saved in `$L_prog`.

```
#15 <weave0: note language> ≡
107 &cUsage unless $_;
108 my $pr = "$Lib_dir/wO-l-$_";
109 if (-r $pr && -x $pr) {
110     $L_prog = $pr;
111 } else {
112     &Message("Language $_ is unknown;", "use of -l option ignored.");
113 }
    (This code is used in #11b (p.19).)
```

The variable `$L_prog` must be declared.

```
#9e <weave0 definitions #9 (p.17)> +≡
114 my $L_prog = "";
    (This code is extended in #9f (p.20).)
```

The `-l` option is also described in the man page:

```
#12b <weave0 man page parameters #12 (p.18)> +≡
115 .TP
116 .B -l \fllanguage\fP
117 Specify which language filter to use when processing the data.
118 Currently, there exist filters
119 .IR c " , " java " , "
120 .IR latex " , and " perl.
121 The default is to use no language filter at all.
    (This code is extended in #12c (p.20).)
```

**2.2.2.4 The `-o` option** is used to specify the name of the output file. This option is passed on to the processor.

```
#11c <weave0 parameters #11 (p.18)> +≡
122 /^-o$/ and do { $_ = shift or &cUsage;
123     push @Code_opt, "-o'$_'"; next PARAM; };
124 /^-o(.+)/ and do {
125     push @Code_opt, "-o'$1'"; next PARAM; };
    (This code is extended in #11d (p.20).)
```

The man page description is also necessary:

```
#12c <weave0 man page parameters #12(p.18)> +≡
126 .TP
127 .B -o \fifile\fp
128 Specify the name of the file on which to write the documentation.
129 If this option is not used, the output will go to
130 .I standard output.
    (This code is extended in #12d(p.20).)
```

**2.2.2.5 The `-v` option** makes weave0 print its version identification and some information on what it is doing. The option is also passed on to all sub-processes so they can do the same.

```
#11d <weave0 parameters #11(p.18)> +≡
131 /^-v$/ and do {
132     print STDERR "This is $Prog (version $Version)\n";
133     push @Pre_opt, "-v"; push @Lang_opt, "-v";
134     push @Code_opt, "-v"; $Verbose = "Yes"; next PARAM; };
    (This code is extended in #11e(p.20).)
```

The default is to be silent.

```
#9f <weave0 definitions #9(p.17)> +≡
135 my $Verbose = 0;
```

The `-v` parameter must also be mentioned in the man page.

```
#12d <weave0 man page parameters #12(p.18)> +≡
136 .TP
137 .B -v
138 State the program version. Use of this option will also make
139 .I tangle0
140 more verbose so that it will display information on what it does.
```

**2.2.2.6 Unknown options** All other options result in a warning.

```
#11e <weave0 parameters #11(p.18)> +≡
141 /^-/ and do {
142     &Message("Unknown option '$_' ignored."); next PARAM; };
```

## 2.2.3 Running the pre- and postprocessors

The basic mechanism for running the processes is the same as for the tangle0 program; see Section 2.1.3 on page 15. The language processor wO-l-xxx, however, is run through a pipeline so any status value indicating error will be ignored.<sup>4</sup>

```
#16 <weave0 processing> ≡
143 my $Post_cmd = ($L_prog ? "$L_prog @Lang_opt | " : "") .
144     "$F_prog @Code_opt";
145 &Message("Running $Lib_dir/wOpre @Pre_opt @Pre_files | $Post_cmd")
146     if $Verbose;
147 open(PRE, "$Lib_dir/wOpre @Pre_opt @Pre_files |");
148 open(POST, "| $Post_cmd");
149 print POST while <PRE>;
150 close PRE; exit $?>>8 if $?>>8;
151 close POST; exit $?>>8;
    (This code is used in #8(p.16).)
```

---

<sup>4</sup>Since this program is part of the web<sub>0</sub> package, it should never produce any errors. ☺

### 2.2.4 Auxiliary functions

The `weave0` program uses some utility functions.

**2.2.4.1 The `&Usage` function** This function is called if the user makes a mistake in the parameter list. It gives a short description on how to use the program.

```
#17 <weave0 auxiliary functions> ≡
152 sub Usage {
153     print STDERR "Usage: $Prog [-e] [-f filter] [-l language] ",
154                " [-o file] [-v] [file...]\n";
155     exit 1;
156 }
(This code is used in #8 (p.16).)
```

## 2.3 The `wOpre` filter

The `wOpre` filter is really a scanner. It reads the source text and separates it into tokens that are easy to digest for the other programs in the `web0` package. The following tokens are produced:

**code** shows code in the body of a meta symbol definition. If a code line contains meta symbol references, it will result in several use tokens (for each meta symbol used) and several code tokens (for each part of the rest of the line).

Note that the code token always terminates with a semicolon; this is to make it easy to see any spaces at the end of a line.

**def** indicates that a meta symbol is being defined.

**file** gives the name of the file being read. There will be one such token for each file read.

**nl** is used to separate code lines in a meta symbol definition.

**text** is a line of documentation text. It is also terminated by a semicolon.

**use** marks the use of a meta symbol.

Each line produced by `wOpre` contains exactly one token. For example, the `web0` code shown in Figure 10 on the following page produces the tokens shown in Figure 11 on page 23.

### 2.3.1 The main program

The `wOpre` program is written in Perl.

```
#18 <wOpre> ≡
157 <perl #105 (p.57)>
158
159 <wOpre definitions #19 (p.24)>
160 <wOpre initialization #20 (p.24)>
161 <wOpre parameter handling #21 (p.24)>
162 <wOpre token recognition #23 (p.25)>
163 exit($N_errors ? 1 : 0);
164
```

Figure 10: Another typical web<sub>0</sub> source text Hello.w0

---

```

\documentclass[12pt,norsk]{webzero}
\usepackage[latin1]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{babel}

\title{«Hello, world!»-programmet i Java}
\author{Dag Langmyhr}

\begin{document}
\maketitle

\section{Implementasjon}
Omtrent alle læreboken i programmering starter med
«Hello, world!»-programmet. I Java ser det slik ut:

<<hello world>> =
<<importspesifikasjoner>>
class Hello {
    public static void main(String args[]) {
        <<deklarasjoner>>
        <<setninger>>
    }
}
@

\subsection{Skriv «Hello»}
Det er trivielt å skrive ut teksten «Hello, world!».
<<setninger>> =
System.out.println("Hello, world!");
@

\subsection{Versjonsinformasjon}
Det er nyttig å få informasjon om hvilken versjon av Java man kjører.
<<setn...>> =
System.out.println("Dette er versjon " +
    prop.getProperty("java.version") + ".");
@

Egenskapen \texttt{prop} må lages:
<<dekl...>> =
Properties prop = System.getProperties();
@
Dessuten må klassen \texttt{Properties} importeres.
<<import...>> =
import java.util.*;
@
\end{document}

```

---

**Figure 11:** The tokens produced by wOpre from the web<sub>0</sub> code Hello.wO in Figure 10 on the preceding page

---

```

file:Hello.wO;
text:\documentclass[12pt,norsk]{webzero};
text:\usepackage[latin1]{inputenc};
text:\usepackage[T1]{fontenc};
text:\usepackage{babel};
text:;
text:\title{«Hello, world!»-programmet i Java};
text:\author{Dag Langmyhr};
text:;
text:\begin{document};
text:\maketitle;
text:;
text:\section{Implementasjon};
text:Omtrent alle læreboken i programmering starter med;
text:«Hello, world!»-programmet. I Java ser det slik ut:;
text:;
def:hello world>>
use:importspesifikasjoner>>
nl
code:class Hello {;
nl
code: public static void main(String args[]) {;
nl
code:  ;
use:deklarasjoner>>
nl
code:  ;
use:setninger>>
nl
code: };
nl
code:};
text:;
text:\subsection{Skriv «Hello»};
text:Det er trivielt å skrive ut teksten «Hello, world!».;
def:setninger>>
code:System.out.println("Hello, world! ");;
text:;
text:\subsection{Versjonsinformasjon};
text:Det er nyttig å få informasjon om hvilken versjon av Java man kjører.;
def:setninger>>
code:System.out.println("Dette er versjon " + ;
nl
code: prop.getProperty("java.version") + ".");;
text:;
text:Egenskapen \texttt{prop} må lages:;
def:deklarasjoner>>
code:Properties prop = System.getProperties();;
text:Dessuten må klassen \texttt{Properties} importeres.;
def:importspesifikasjoner>>
code:import java.util.*;;
text:\end{document};

```

---

```

165 <wOpre utility functions #29 (p.26)>
166 <user message functions #110 (p.60)>
(This code is not used.)

```

## 2.3.2 Definitions

**2.3.2.1 Identification** As all the other programs in this package, wOpre can identify itself with its name and version number.

```

#19 <wOpre definitions> ≡
167 my ($Prog, $Version) = ("wOpre", "<version #106 (p.57)>");
(This code is extended in #19a (p.24). It is used in #18 (p.21).)

```

**2.3.2.2 Boolean constants** The values **False** and **True** are used quite often in this program, so the code will be more readable if they are given names.

```

#19a <wOpre definitions #19 (p.24)> +≡
168 my ($False, $True) = (0, 1);
(This code is extended in #19b (p.24).)

```

**2.3.2.3 Meta symbol start pattern** To avoid using <<...>> in the program (see Section 1.2 on page 3), we need to introduce a variable \$Meta\_start to contain the opening brackets.

```

#19b <wOpre definitions #19 (p.24)> +≡
169 my $Meta_start = "<<";
(This code is extended in #19c (p.26).)

```

## 2.3.3 Initialization

Always start in documentation mode.

```

#20 <wOpre initialization> ≡
170 my ($Code_mode, $New_line) = ($False, $False);
(This code is used in #18 (p.21).)

```

## 2.3.4 Parameter handling

The following loop will fetch and decode all the parameters, while the file names in @ARGV will remain.

```

#21 <wOpre parameter handling> ≡
171 PARAM:
172 while (@ARGV && $ARGV[0]=~/^-/){ $_ = shift;
173 <wOpre parameters #22 (p.24)>
174 &Message("Unknown option ‘$_’ ignored.");
175 }
(This code is used in #18 (p.21).)

```

**2.3.4.1 The -v option** will make wOpre state its name and version.

```

#22 <wOpre parameters> ≡
176 /^-v$/ and do {
177 print STDERR "This is $Prog (version $Version)\n";
178 next PARAM; };
(This code is used in #21 (p.24).)

```

### 2.3.5 Extracting tokens

This loop will read each line and scan the `web0` code for tokens.

```
#23 <wOpre token recognition> ≡
179  LINE:
180  while (<>) { chomp;
181      <wOpre check input file #28 (p.26)>
182      <expand TAB characters #109 (p.60)>
183      <wOpre check for end of meta symbol definition #26 (p.25)>
184      <wOpre check for start of meta symbol definition #24 (p.25)>
185      <wOpre handle text line #27 (p.26)>
186      <wOpre check one line of a meta symbol definition #25 (p.25)>
187  }
```

(This code is used in #18 (p.21).)

### 2.3.6 Meta symbol definitions

A new meta symbol is defined when the user writes a line containing only a meta symbol and an “=” sign (and possibly some spaces surrounding the “=”). The variable `$Code_mode` is set to indicate that `wOpre` is currently reading a meta symbol definition.

```
#24 <wOpre check for start of meta symbol definition> ≡
188  /^$Meta_start(?:>>\s*=\s*$/o &&& !$Code_mode and do {
189      $Last_def = &Find_meta_sy($1); $Code_mode = $True;
190      print "def:$Last_def>>\n"; $New_line = $False;
191      next LINE; };
```

(This code is used in #23 (p.25).)

Each line of a meta symbol definition must be checked to see if it contains references to other meta symbols. Each such reference produces a use token; the remainder of the line results in code tokens.

```
#25 <wOpre check one line of a meta symbol definition> ≡
192  print "nl\n" if $New_line;
193  while (s/^(.*)$Meta_start(?:>>\/o) {
194      print "code:$1;\n" if $1;
195      print "use:", &Find_meta_sy($2), ">>\n";
196  }
197  print "code:$_;\n" if $_; $New_line = $True;
```

(This code is used in #23 (p.25).)

(The variable `$New_line` is set whenever a line has been completely processed. We need this variable because the `nl` token *separates* lines in a meta symbol definition.)

The end of a meta symbol definition is recognized by a line with a lone “@” (except for additional spaces<sup>5</sup> after the “@”).

```
#26 <wOpre check for end of meta symbol definition> ≡
198  /^@\s*$/ &&& $Code_mode and do {
199      $Code_mode = $New_line = $False; next LINE; };
```

(This code is used in #23 (p.25).)

---

<sup>5</sup>I decided to allow superfluous spaces since such spaces would be invisible to the user, and he or she might otherwise have problems detecting the cause of any erroneous behavior.

### 2.3.7 Handling lines of documentation

When not defining a meta symbol, the line is just copied as it is into a text token.

```
#27 <wOpre handle text line> ≡
200 unless ($Code_mode) { print "text:$_\n"; next LINE; };
(This code is used in #23 (p.25).)
```

### 2.3.8 File name check

The variable \$Cur\_file keeps track of the source file name.

```
#19c <wOpre definitions #19 (p.24)> +≡
201 my $Cur_file = "";
(This code is extended in #19d (p.27).)
```

A file token is generated whenever we start reading another file.

```
#28 <wOpre check input file> ≡
202 unless ($Cur_file) { $Cur_file = $ARGV; print "file:$Cur_file\n"; }
(This code is extended in #28a (p.26). It is used in #23 (p.25).)
```

When a file has been completely read, we must clear \$Cur\_file to force another file token if another file is read.

```
#28a <wOpre check input file #28 (p.26)> +≡
203 $Cur_file = "" if eof;
```

### 2.3.9 Utility functions

**2.3.9.1 The function &Find\_meta\_sy** This function is used to find the correct name of a meta symbol, in particular when the <<>> or <<xxx. . . >> notation is used. The function has one parameter:

1. the meta symbol name as given by the user (but without the angle brackets).

```
#29 <wOpre utility functions> ≡
204 sub Find_meta_sy {
205     local $_ = shift;
206
207     <find_meta_sy: handle reference to last meta symbol #30 (p.26)>
208     <find_meta_sy: handle abbreviated meta symbol name #31 (p.27)>
209     $Meta{$_} = "Defined"; return $_;
210 }
(This code is extended in #29a (p.27). It is used in #18 (p.21).)
```

<<>> is used to refer to the last meta symbol defined.

```
#30 <find_meta_sy: handle reference to last meta symbol> ≡
211 unless ($_) {
212     &Warning("Illegal use of '$Meta_start>>' notation;",
213         "no meta symbol defined yet.") unless $Last_def;
214     return $Last_def || "??";
215 }
(This code is used in #29 (p.26).)
```

The name of the last meta symbol defined is kept in \$Last\_def.

```
#19d <wOpre definitions #19(p.24)> +≡
216 my $Last_def = "";
    (This code is extended in #19e(p.27).)
```

<<xxx. . .>> is a reference to a meta symbol whose name starts with xxx. This can be found by searching the table %Meta which contains all known meta symbols (as keys).

```
#19e <wOpre definitions #19(p.24)> +≡
217 my %Meta = ();
    (This code is extended in #19f(p.27).)
```

There should be exactly one such meta symbol in %Meta.

```
#31 <find_meta_sy: handle abbreviated meta symbol name> ≡
218 if (/\.{3}$/) {
219     my $abbrev = $';
220     my $ab_len = length $abbrev;
221
222     my @match = grep { substr($_,$ab_len) eq $abbrev } keys %Meta;
223     unless (@match) {
224         &Warning("No match for $Meta_start$_>>.");
225         return "??";
226     }
227     &Warning("Multiple matches for $Meta_start$_>>:",
228             join(", ", map("$Meta_start$_>>",@match))) if @match>1;
229     return $match[0];
230 }
    (This code is used in #29(p.26).)
```

(Note in particular the use of `grep` to find matching meta symbols. The test here cannot use a pattern as the abbreviated meta symbol name may contain characters like “(” or “\*” that will confuse Perl.)

**2.3.9.2 The function `&Warning`** This function gives the user a warning and increases the error count `$N_errors`.

```
#29a <wOpre utility functions #29(p.26)> +≡
231 sub Warning {
232     &Message(@_); ++$N_errors;
233 }
```

The error count must be declared.

```
#19f <wOpre definitions #19(p.24)> +≡
234 my $N_errors = 0;
```

## 2.4 The wOcode filter

The wOcode filter reads tokens produced by wOpre and prints the program code; it forms the last link in the pipeline set up by `tangleO`. Most of this code can be found as parts of the various tokens, but meta symbols must be expanded.

The wOcode filter is written in Perl.

```
#32 <wOcode> ≡
235 <perl #105(p.57)>
236
237 <wOcode definitions #33(p.28)>
```

```

238 <w0code parameter handling #34 (p.28)>
239 <w0code read tokens #37 (p.29)>
240 <w0code expand meta symbols #40 (p.30)>
241 close OUT unless $Output == \*STDOUT;
242 exit ($N_errors ? 1 : 0);
243
244 <w0code utility functions #41 (p.30)>
245 <user message functions #110 (p.60)>
    (This code is not used.)

```

## 2.4.1 Definitions

**2.4.1.1 Identification** This filter should be able to identify itself with its name and version number.

```

#33 <w0code definitions> ≡
246 my ($Prog, $Version) = ("w0code", "<version #106 (p.57)>");
    (This code is extended in #33a (p.28). It is used in #32 (p.27).)

```

**2.4.1.2 Meta symbol start pattern** To avoid using <<...>> in the program (see Section 1.2 on page 3), we need to introduce a variable \$Meta\_start to contain the opening brackets.

```

#33a <w0code definitions #33 (p.28)> +≡
247 my $Meta_start = "<<";
    (This code is extended in #33b (p.29).)

```

## 2.4.2 Parameter handling

This loop will look at all the program's parameters.

```

#34 <w0code parameter handling> ≡
248 PARAM:
249 while (@ARGV &&& $ARGV[0] =~ /^-/ ) { $_ = shift;
250     <w0code parameters #35 (p.28)>
251 }
    (This code is used in #32 (p.27).)

```

**2.4.2.1 The -o option** This option is used to indicate the name of the file on which to write the output. If this option is not used, the output will go to *standard output*.

```

#35 <w0code parameters> ≡
252 /^-o$/ and do { $_ = shift;
253     <w0code: note output file #36 (p.28)>; next PARAM; };
254 /^-o(.+)/ and do { $_ = $1;
255     <w0code: note output file #36 (p.28)>; next PARAM; };
    (This code is extended in #35a (p.29). It is used in #34 (p.28).)

```

When the user specifies an output file, it is opened, and the variable \$Output is set to reference this file.

```

#36 <w0code: note output file> ≡
256 open(OUT, ">$_") or &Error("Could not create $_.");
257 $Output = \*OUT;
    (This code is used in #35 (p.28).)

```

If the user does not specify any output file, the result will be written to *standard output*.

**#33<sub>b</sub>** *<w0code definitions #33 (p.28)>* + ≡  
 258 my \$Output = \\*STDOUT;  
 (This code is extended in #33<sub>c</sub> (p.29).)

**2.4.2.2 The -v option** This option is for debugging. It will report the program's name and version number.

**#35<sub>a</sub>** *<w0code parameters #35 (p.28)>* + ≡  
 259 /^-v\$/ **and do** {  
 260 print STDERR "This is \$Prog (version \$Version)\n";  
 261 **next** PARAM; };  
 (This code is extended in #35<sub>b</sub> (p.29).)

**2.4.2.3 The -x option** This option is used to indicate the meta symbol with which to start the extraction.

**#35<sub>b</sub>** *<w0code parameters #35 (p.28)>* + ≡  
 262 /^-x\$/ **and do** { \$Start = shift; **next** PARAM; };  
 263 /^-x(.+)/ **and do** { \$Start = \$1; **next** PARAM; };  
 (This code is extended in #35<sub>c</sub> (p.29).)

The variable \$Start is used for this symbol.

**#33<sub>c</sub>** *<w0code definitions #33 (p.28)>* + ≡  
 264 my \$Start = " ";  
 (This code is extended in #33<sub>d</sub> (p.29).)

**2.4.2.4 Illegal options** Any other option than those handled above is illegal.

**#35<sub>c</sub>** *<w0code parameters #35 (p.28)>* + ≡  
 265 &Message("Unknown option '\$\_' ignored.");

## 2.4.3 Reading the tokens

Before we can extract any code, all token must be read. The table %Def will contain the code of all meta symbols defined; each value in %Def will be a text string of all the token containing the body of that meta symbol, separated by newlines.

**#33<sub>d</sub>** *<w0code definitions #33 (p.28)>* + ≡  
 266 my %Def = ();  
 (This code is extended in #33<sub>e</sub> (p.30).)

If a meta symbol has several definitions, they will all be concatenated here.

**#37** *<w0code read tokens>* ≡  
 267 DATA:  
 268 **while** (<>) { chomp;  
 269 *<w0code meta symbol definition #38 (p.29)>*  
 270 *<w0code add to meta symbol body #39 (p.30)>*  
 271 }  
 (This code is used in #32 (p.27).)

**2.4.3.1 Meta symbol definition** The def token signals the definition of a meta symbol. Note that this can be an extension of a meta symbol already defined; in that case an nl token is inserted to separate the two.

**#38** *<w0code meta symbol definition>* ≡  
 272 /^def:(.\*)>>\$/ **and do** {  
 273 \$Cur\_def = \$1; \$Start = \$Start || \$Cur\_def;

```

274     $Def{$Cur_def} =
275         exists $Def{$Cur_def} ? "$Def{$Cur_def}nl\n" : " ";
276     next DATA; };

```

(This code is used in #37 (p.29).)

The variable \$Cur\_def always contains the name of the meta symbol whose body is being defined.

**#33e** *<wOcode definitions #33 (p.28)> + ≡*

```

277     my $Cur_def = " ";

```

(This code is extended in #33<sub>f</sub> (p.30).)

**2.4.3.2 Meta symbol body** The code, nl, and use tokens all add to the body of the current meta symbol definition.

**#39** *<wOcode add to meta symbol body> ≡*

```

278     /^(code:|nl|use:)/ and do {
279         $Def{$Cur_def} .= "$_\n"; next DATA; };

```

(This code is used in #37 (p.29).)

All other tokens are ignored.

## 2.4.4 Expanding the meta symbols

Since expanding the meta symbols is a recursive process, wOcode uses the function &Expand for this.

**#40** *<wOcode expand meta symbols> ≡*

```

280     &Error("No meta symbols defined.") unless $Start;
281     &Expand($Start);
282     print $Output "\n";

```

(This code is used in #32 (p.27).)

The function &Expand has one parameter: the meta symbol to expand.

**#41** *<wOcode utility functions> ≡*

```

283     sub Expand {
284         my $sy = shift;
285         <wOcode expand: check that meta symbol is defined #42 (p.30)>
286         <wOcode expand: check for definition cycles #43 (p.31)>
287         <wOcode expand: expand the meta symbol body #45 (p.31)>
288         <wOcode expand: deactivate current meta symbol #44 (p.31)>
289     }

```

(This code is extended in #41<sub>a</sub> (p.31). It is used in #32 (p.27).)

**2.4.4.1 Check definition** If the requested meta symbol has not been defined, there is nothing we can do.

**#42** *<wOcode expand: check that meta symbol is defined> ≡*

```

290     unless (exists $Def{$sy}) {
291         &Warning("Symbol $Meta_start$sy>> has not been defined.")
292         unless exists $Not_def_mess{$sy};
293         $Not_def_mess{$sy} = "reported"; return;
294     }

```

(This code is used in #41 (p.30).)

The table %Not\_def\_mess keeps track of the messages to avoid repeating them.

**#33f** *<wOcode definitions #33 (p.28)> + ≡*

```

295     my %Not_def_mess = ();

```

(This code is extended in #33<sub>g</sub> (p.31).)

**2.4.4.2 Checking for definition cycles** If the user has made an error and written a circular set of definitions, this must be detected by `wOcode` to avoid an endless loop. A check for circular definitions is simple to implement, however.

```
#43 <wOcode expand: check for definition cycles> ≡
296 &Error("Definition cycle found;", "the loop involves:",
297     join(" ", map("$Meta_start$_>>",
298         sort {$Active{$a} <=> $Active{$b}}
299         grep {$Active{$_} >= $Active{$sy}}
300         keys %Active)))
301     if $Active{$sy};
302 $Active{$sy} = 1+keys %Active;
    (This code is used in #41 (p.30).)
```

The table `%Active` contains (as keys) all the meta symbols that are currently being expanded.<sup>6</sup>

```
#33g <wOcode definitions #33 (p.28)> +≡
303 my %Active = ();
    (This code is extended in #33h (p.31).)
```

When the current meta symbol has been completely expanded, it can be removed from the `%Active` list.

```
#44 <wOcode expand: deactivate current meta symbol> ≡
304 delete $Active{$sy};
    (This code is used in #41 (p.30).)
```

**2.4.4.3 The actual expansion** The actual expansion consists of reading the tokens and inserting the text therein. The only exception is the use token which results in a recursive call on `&Expand`.

```
#45 <wOcode expand: expand the meta symbol body> ≡
305 my $line;
306 foreach $line (split(/\n/, $Def{$sy})) {
307     if ($line =~ /^code:(.*)"$/ { print $Output $1; }
308     elsif ($line =~ /^nl$/ { print $Output "\n"; }
309     elsif ($line =~ /^use:(.*)">>$/ { &Expand($1); }
310     else { &Error("Unknown format: $line."); }
311 }
    (This code is used in #41 (p.30).)
```

## 2.4.5 Utility functions

**2.4.5.1 The function `&Warning`** This function gives the user a warning and increases the error count `$N_errors`.

```
#41a <wOcode utility functions #41 (p.30)> +≡
312 sub Warning {
313     &Message(@_); ++$N_errors;
314 }
```

The error count must be declared.

```
#33h <wOcode definitions #33 (p.28)> +≡
315 my $N_errors = 0;
```

---

<sup>6</sup>The `%Active` values are 1, 2, ... which are useful when producing a good error message.

## 2.5 The `w0-f-latex` filter

This program reads tokens produced by `wOpre` (see Section 2.3 on page 21) and produces `LaTeX` code. To be more particular, the following commands and environments are generated:

`\wzdef` is used whenever a new meta symbol is defined; see Section 3.1.4.1 on page 66.

`\wzenddef` terminates the meta symbol definition; see Section 3.1.4.2 on page 66.

`\wzeol` is used as a code line separator; see Section 3.1.4.3 on page 67.

`\wzfile` signals that a new source file is being read; see Section 3.1.4.4 on page 68.

`\wzmeta` is used to typeset a meta symbol; see Section 3.1.4.5 on page 68.

For an example, see Figure 12 on the facing page.

```
#46 <w0-f-latex> ≡
316 <perl #105 (p.57)>
317
318 <alphabetical sorting #111 (p.61)>
319 <w0-f-latex: initialization #47 (p.32)>
320 <w0-f-latex: option handling #48 (p.34)>
321 <w0-f-latex: make LaTeX code #51 (p.35)>
322 close $Output unless $Output == \*STDOUT;
323 exit ($N_errors ? 1 : 0);
324
325 <w0-f-latex: utility functions #61 (p.40)>
326 <latex generation functions #107 (p.58)>
327 <user message functions #110 (p.60)>
(This code is not used.)
```

### 2.5.1 Initialization

**2.5.1.1 Identification** is required for any self-respecting program.

```
#47 <w0-f-latex: initialization> ≡
328 my ($Prog, $Version) = ("w0-f-latex", "<version #106 (p.57)>");
(This code is extended in #47a (p.32). It is used in #46 (p.32).)
```

**2.5.1.2 Boolean constants** are used so often in this program that they should be given names.

```
#47a <w0-f-latex: initialization #47 (p.32)> + ≡
329 my ($False, $True) = (0, 1);
(This code is extended in #47b (p.32).)
```

**2.5.1.3 Start of meta symbol** must be defined in a variable to avoid writing `<<. . . >>` which will confuse `web0`.

```
#47b <w0-f-latex: initialization #47 (p.32)> + ≡
330 my $Meta_start = "<<";
(This code is extended in #47c (p.34).)
```

**Figure 12:** The L<sup>A</sup>T<sub>E</sub>X code produced by weaveO from the web<sub>0</sub> code Hello.wO in Figure 10 on page 22

---

```

\def\wzclassindex{\begin{wzindex}{\wzclassindexname}{2}
\end{wzindex}}
\def\wzfuncindex{\begin{wzindex}{\wzfuncindexname}{2}
\end{wzindex}}
\def\wzvarindex{\begin{wzindex}{\wzvarindexname}{2}
\end{wzindex}}
\def\wzmetaindex{\begin{wzindex}{\wzmetaindexname}{1}
\wzx{\wzmeta{deklarasjoner~~\upshape\#3}}\wzlongpageref{3}\par
\wzx{\wzmeta{hello~world~~\upshape\#1}}\wzlongpageref{1}\rlap{~*}\par
\wzx{\wzmeta{importspesifikasjoner~~\upshape\#4}}\wzlongpageref{4}\par
\wzx{\wzmeta{setninger~~\upshape\#2}}\wzlongpageref{2}\par
\raggedright\wzmetaindexstartext\par
\end{wzindex}}
\ifx \wzfile\undefined
\AtBeginDocument{\wzfile {Hello.wO}}\else \wzfile {Hello.wO}\fi
\documentclass[12pt,norsk]{webzero}
\usepackage[latin1]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{babel}

\title{«Hello, world!»-programmet i Java}
\author{Dag Langmyhr}

\begin{document}
\maketitle

\section{Implementasjon}
Omtrent alle læreboken i programmering starter med
«Hello, world!»-programmet. I Java ser det slik ut:

\wzdef{hello~world}{1}{0}\wzmeta{4}{importspesifikasjoner}\wzeol\relax
class~Hello~{\wzeol\relax
~~public~static~void~main(String~args[]){\wzeol\relax
~~~~\wzmeta{3}{deklarasjoner}\wzeol\relax
~~~~\wzmeta{2}{setninger}\wzeol\relax
~~}\wzeol\relax
}%
\wzenddef{1}{0}{0}\relax

\subsection{Skriv «Hello»}
Det er trivielt å skrive ut teksten «Hello, world!».
\wzdef{setninger}{2}{0}System.out.println("Hello,~world!");%
\wzenddef{2}{0}{1}\wzxref{1}{0}\relax

\subsection{Versjonsinformasjon}
Det er nyttig å få informasjon om hvilken versjon av Java man kjører.
\wzdef{setninger}{2}{1}System.out.println("Dette~er~versjon~"~+~\wzeol\relax
~~~~prop.getProperty("java.version")~+~".");%
\wzenddef{2}{1}{0}\relax

Egenskapen \texttt{prop} må lages:
\wzdef{deklarasjoner}{3}{0}Properties~prop~~System.getProperties();%
\wzenddef{3}{0}{0}\wzxref{1}{0}\relax
Dessuten må klassen \texttt{Properties} importeres.
\wzdef{importspesifikasjoner}{4}{0}import~java.util.*;%
\wzenddef{4}{0}{0}\wzxref{1}{0}\relax
\end{document}

```

---

**2.5.1.4 Default output file name** is *standard out* unless the user tells us otherwise.

```
#47c <w0-f-latex: initialization #47 (p.32)> +≡
331 my $Output = \*STDOUT;
      (This code is extended in #47a (p.35).)
```

## 2.5.2 Parameter handling

This loop will examine all the program options (but leave any file names behind).

```
#48 <w0-f-latex: option handling> ≡
332 PARAM:
333 while (@ARGV &&& $ARGV[0] =~ /^-/) { $_ = shift;
334     <w0-f-latex: examine options #49 (p.34)>
335 }
      (This code is used in #46 (p.32).)
```

**2.5.2.1 The `-o` option** specifies the name of the file on which to put the final `LATEX` code.

```
#49 <w0-f-latex: examine options> ≡
336 /^-o$/ and do { $_ = shift; <w0-f-latex: note output file #50 (p.34)>
337     next PARAM; };
338 /^-o(.$)/ and do { $_ = $1; <w0-f-latex: note output file #50 (p.34)>
339     next PARAM; };
      (This code is extended in #49a (p.34). It is used in #48 (p.34).)
```

Once the output file has been opened, the variable `$Output` must be set so that the output really goes to that file.

```
#50 <w0-f-latex: note output file> ≡
340 open(OUT, ">$_") or &Error("Could not create '$_'.");
341 $Output = \*OUT;
      (This code is used in #49 (p.34).)
```

**2.5.2.2 The `-v` option** is used for debugging. It will make `w0-f-latex` state its name and version number.

```
#49a <w0-f-latex: examine options #49 (p.34)> +≡
342 /^-v$/ and do {
343     print STDERR "This is $Prog (version $Version)\n";
344     next PARAM; };
      (This code is extended in #49b (p.34).)
```

**2.5.2.3 Illegal options** Any option beside those already mentioned is illegal.

```
#49b <w0-f-latex: examine options #49 (p.34)> +≡
345 &Message("Unknown option '$_' ignored.");
```

## 2.5.3 Pass 1

Translating tokens from `wOpre` into `LATEX` code is a three-pass affair:

1. Make a list of all the meta symbols.
2. Collect information on where each meta symbol has been used, as well as the information required to generate the indices.
3. Produce the `LATEX` code.

As mentioned, the first pass determines which meta symbols have been defined in the source files. The following variables are used:

**\$N\_meta** contains the number of distinct meta symbols defined.

```
#47d <w0-f-latex: initialization #47 (p.32)> +≡
346 my $N_meta = 0;
(This code is extended in #47e (p.35).)
```

**%Meta\_id** contains the meta symbols (as keys); the values are the symbols' identification numbers.

```
#47e <w0-f-latex: initialization #47 (p.32)> +≡
347 my %Meta_id = ();
(This code is extended in #47f (p.35).)
```

**@Meta\_n\_ext** contains how many extensions each meta symbol has.

```
#47f <w0-f-latex: initialization #47 (p.32)> +≡
348 my @Meta_n_ext = ();
(This code is extended in #47g (p.36).)
```

As the tokens are read, they must be saved so that they can be read again in the other two passes. The array **@Tokens** is used for this; it is quite a bit faster than using a file.

```
#51 <w0-f-latex: make LaTeX code> ≡
349 my @Tokens = ();
350 while (<<>) { chomp; push(@Tokens, $_);
351   <w0-f-latex: pass1: note meta symbol definition #52 (p.35)>
352 }
(This code is extended in #51a (p.36). It is used in #46 (p.32).)
```

Only the def tokens are of interest in this pass.

```
#52 <w0-f-latex: pass1: note meta symbol definition> ≡
353 /^def:(.*)>>$/ and do { my $symb = $1;
354   if (defined $Meta_id{$symb}) {
355     ++$Meta_n_ext[$Meta_id{$symb}];
356   } else {
357     $Meta_id{$symb} = ++$N_meta; $Meta_n_ext[$N_meta] = 0;
358   }
359 };
(This code is used in #51 (p.35).)
```

### 2.5.4 Pass 2

The purpose of the second pass is to record usage of the meta symbols. The following variables will contain this information:

**@Meta\_use** will tell where each meta symbol has been used. Its index is the meta symbol's identification number, and the value is a text string on the form

$$n_1-e_1 \quad n_2-e_2 \quad \dots$$

where  $n_1-e_1$  is the first occurrence,  $n_2-e_2$  the second, and so forth. Each occurrence is given as two numbers: the identification number  $n_x$  of the meta symbol in whose definition the specific meta symbol occurs, and the extension  $e_x$ .

```
#51a <w0-f-latex: make LaTeX code #51 (p.35)> +≡
360 my @Meta_use = ();
(This code is extended in #51b (p.36).)
```

In addition, the following variables are used during pass 2:

**\$Cur\_line** contains the line number of the current code line.

```
#51b <w0-f-latex: make LaTeX code #51 (p.35)> +≡
361 my $Cur_line = 0;
(This code is extended in #51c (p.36).)
```

**%message\_given** counts how many times an error message for a meta symbol with no definition has been given. Its purpose is to avoid reporting problems with each meta symbol more than once.

```
#51c <w0-f-latex: make LaTeX code #51 (p.35)> +≡
362 my %message_given = ();
(This code is extended in #51d (p.36).)
```

**@meta\_ext\_cnt** counts how many extensions for a given meta symbol have been found so far.

```
#51d <w0-f-latex: make LaTeX code #51 (p.35)> +≡
363 my @meta_ext_cnt = ();
(This code is extended in #51e (p.36).)
```

These variables are deleted at the end of pass 2 to save space.

```
#51e <w0-f-latex: make LaTeX code #51 (p.35)> +≡
364 $Cur_line = 0;
365 foreach (@Tokens) {
366     <w0-f-latex: pass2: handle tokens #53 (p.36)>
367 }
368 %message_given = (); @meta_ext_cnt = ();
(This code is extended in #51f (p.38).)
```

The def tokens indicate which is the current meta symbol and extension.

```
#53 <w0-f-latex: pass2: handle tokens> ≡
369 /^def:(.*)>>$/ and do { ++$Cur_line;
370     $cur_meta = $Meta_id{$1};
371     $cur_ext = $meta_ext_cnt[$cur_meta]++; next; };
(This code is extended in #53a (p.36). It is used in #51e (p.36).)
```

The current meta symbol and extension are kept in \$cur\_meta and \$cur\_ext (in numeric form).

```
#47g <w0-f-latex: initialization #47 (p.32)> +≡
372 my ($cur_meta, $cur_ext) = (0, 0);
(This code is extended in #47h (p.37).)
```

The use tokens show the use of a meta symbol.

```
#53a <w0-f-latex: pass2: handle tokens #53 (p.36)> +≡
373 /^use:(.*)>>$/ and do { my $symb = $1;
374     if (defined $Meta_id{$symb}) {
375         $Meta_use[$Meta_id{$symb}] .= "$cur_meta-$cur_ext ";
376     } elsif ($message_given{$symb}++ == 0) {
377         &Warning("Symbol $Meta_start$symb>> not defined.");
378     }
}
```

```
379     next; };
    (This code is extended in #53b (p.37).)
```

The `nl` token indicates that the line number must be increased.

```
#53b <w0-f-latex: pass2: handle tokens #53 (p.36)> +≡
380 /^nl$/ and do { ++$Cur_line; next; };
    (This code is extended in #53c (p.37).)
```

The `fdef` and `fuse` tokens report that a function has been defined or used on the current line. This information is saved in `%Func_index` as “`nn0`” and “`nn1`”, respectively.<sup>7</sup>

```
#47h <w0-f-latex: initialization #47 (p.32)> +≡
381 my %Func_index = ();
    (This code is extended in #47i (p.37).)
```

```
#53c <w0-f-latex: pass2: handle tokens #53 (p.36)> +≡
382 /^fdef:(.*/ and do {
383     $Func_index{$1} .= $Cur_line . "0 "; next; };
384 /^fuse:(.*/ and do {
385     $Func_index{$1} .= $Cur_line . "1 "; next; };
    (This code is extended in #53d (p.37).)
```

Similarly, the `vdef` and `vuse` tokens report that a variable has been defined or used. This information is saved in `%Var_index` as “`nn0`” and “`nn1`”, respectively.

```
#47i <w0-f-latex: initialization #47 (p.32)> +≡
386 my %Var_index = ();
    (This code is extended in #47j (p.37).)
```

```
#53d <w0-f-latex: pass2: handle tokens #53 (p.36)> +≡
387 /^vdef:(.*/ and do {
388     $Var_index{$1} .= $Cur_line . "0 "; next; };
389 /^vuse:(.*/ and do {
390     $Var_index{$1} .= $Cur_line . "1 "; next; };
    (This code is extended in #53e (p.37).)
```

And, finally, the `cdef` and `cuse` tokens report that a class has been defined or used. This information is saved in `%Class_index` as “`nn0`” and “`nn1`”, respectively.

```
#47j <w0-f-latex: initialization #47 (p.32)> +≡
391 my %Class_index = ();
    (This code is extended in #47k (p.38).)
```

```
#53e <w0-f-latex: pass2: handle tokens #53 (p.36)> +≡
392 /^cdef:(.*/ and do {
393     $Class_index{$1} .= $Cur_line . "0 "; next; };
394 /^cuse:(.*/ and do {
395     $Class_index{$1} .= $Cur_line . "1 "; next; };
    (This code is extended in #53f (p.38).)
```

### 2.5.5 Pass 3

The third and last pass generates the actual  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  code. While generating this code, the program is in two modes:

---

<sup>7</sup>The reason for this notation is that I want to list definition occurrences before the ones that concern use.

- When `$Code_mode` is `$True`, `wO-f-latex` is processing code, i.e., the definition of a meta symbol.
- When `$Code_mode` is `$False`, the program is just passing document code on to the `LATEX` file.

During pass 3, the following global variables are used:

**@Meta\_x** counts how many extensions to a meta symbol have been found so far during this pass.

```
#47k <wO-f-latex: initialization #47 (p.32)> +≡
396 my @Meta_x = ();
(This code is extended in #471(p.39).)
```

Before producing any code, however, the indices must be generated.

```
#51f <wO-f-latex: make LaTeX code #51 (p.35)> +≡
397 <wO-f-latex: generate indices #62 (p.41)>
(This code is extended in #51g(p.38).)
```

Then, the translation of tokens into `LATEX` code can commence. Initially, the program expects documentation.

```
#51g <wO-f-latex: make LaTeX code #51 (p.35)> +≡
398 my ($Code_mode, $Cur_line) = ($False, 0);
399 foreach (@Tokens) {
400     <wO-f-latex: pass3: handle 'code' tokens #54 (p.38)>
401     <wO-f-latex: pass3: handle 'def' tokens #55 (p.38)>
402     <wO-f-latex: pass3: handle 'file' tokens #56 (p.39)>
403     <wO-f-latex: pass3: handle 'nl' tokens #57 (p.39)>
404     <wO-f-latex: pass3: handle 'text' tokens #58 (p.39)>
405     <wO-f-latex: pass3: handle 'use' tokens #59 (p.39)>
406 }
407 <wO-f-latex: end code #60 (p.40)>
```

**2.5.5.1 Handling the ‘code’ tokens** These tokens represent the parts of the program code that need no handling, except that they must be `&Latexify`-ed. The three variants are used when the code is to be typeset in a normal font, a bold font, or italics, respectively.

```
#54 <wO-f-latex: pass3: handle 'code' tokens> ≡
408 /^code:(.*)>$/ and do {
409     print $Output &Latexify($1); next; };
410 /^bcode:(.*)>$/ and do {
411     print $Output "\\textbf{" , &Latexify($1), "}"; next; };
412 /^icode:(.*)>$/ and do {
413     print $Output "\\textit{" , &Latexify($1), "}";
414     next; };
415 /^code:(.*)>$/ and do {
416     print $Output "\\textit{" , &Latexify($1), "}"; next; };
(This code is used in #51g(p.38).)
```

**2.5.5.2 Handling the ‘def’ tokens** These tokens indicate the start of a meta symbol definition (and, thus, “code mode”).

```
#55 <wO-f-latex: pass3: handle 'def' tokens> ≡
417 /^def:(.*)>>$/ and do {
418     my $symb = $1; $def_id = $Meta_id{$symb};
419     if (defined $Meta_x[$def_id]) {
```

```

420     ++$Meta_x[$def_id];
421   } else {
422     $Meta_x[$def_id] = 0;
423   }
424   <w0-f-latex: end code #60 (p.40)>
425   print $Output "\\wzdef{" , &Latexify($symb),
426     "}{$def_id}{$Meta_x[$def_id]}";
427   $Code_mode = $True; ++$Cur_line; next; };
  (This code is used in #51g(p.38).)

```

The name of the meta symbol being defined is kept in variable \$def\_id.

```

#471 <w0-f-latex: initialization #47 (p.32)> +≡
428 my $def_id = "";
  (This code is extended in #47m(p.43).)

```

**2.5.5.3 Handling the ‘file’ tokens** These tokens occur whenever the preprocessor has started reading another file. Note the test on whether \wzfile has been defined. The reason for this test is that when the first file (usually containing the \documentclass directive) is being read, \wzfile is yet undefined, so the call on it must wait until the start of the document.

```

#56 <w0-f-latex: pass3: handle ‘file’ tokens> ≡
429 /^file:(.*)>$/ and do {
430   print $Output "\\ifx \\wzfile\\undefined
431   \\AtBeginDocument{\\wzfile {$1}}\\else \\wzfile {$1}\\fi\n";
432   next; };
  (This code is used in #51g(p.38).)

```

**2.5.5.4 Handling the ‘nl’ tokens** These tokens indicate that a code line is terminated and another one starts.

```

#57 <w0-f-latex: pass3: handle ‘nl’ tokens> ≡
433 /^nl$/ and do {
434   print $Output "\\wzeol\\relax\n"; ++$Cur_line; next; };
  (This code is used in #51g(p.38).)

```

**2.5.5.5 Handling the ‘text’ tokens** These tokens contain documentation text that is copied verbatim to the output.

```

#58 <w0-f-latex: pass3: handle ‘text’ tokens> ≡
435 /^text:(.*)>$/ and do {
436   my $text = $1; <w0-f-latex: end code #60 (p.40)>
437   print $Output "$text\n"; next; };
  (This code is used in #51g(p.38).)

```

**2.5.5.6 Handling the ‘use’ tokens** These tokens are found when a meta symbol is referenced in the code. It is translated into a call on \wzmeta with the symbol and its number as parameters.

```

#59 <w0-f-latex: pass3: handle ‘use’ tokens> ≡
438 /^use:(.*)>>$/ and do {
439   print $Output "\\wzmeta{${Meta_id{$1}}{" , &Latexify($1), "}";
440   next; };
  (This code is used in #51g(p.38).)

```

**2.5.5.7 Terminating a meta symbol definition** (which is detected by reading the first ‘text’ token after a meta symbol definition, or on finding two consecutive meta symbol definitions, or when the last token has been read) also requires some special actions. The macro `\wzenddate` is supplied with the following parameters (see Section 3.1.4.2 on page 66):

1. the meta symbol’s number,
2. its extension number,
3. 1 if there are any further extensions, or 0 if there are none, and
4. usage information (already formatted into  $\LaTeX$  code).

```
#60 <w0-f-latex: end code> ≡
441 print $Output "%\n\n\wzenddate{$def_id}{$Meta_x[$def_id]}" .
442     ($Meta_x[$def_id]<$Meta_n_ext[$def_id] ? $Meta_x[$def_id]+1 : 0) .
443     "}" . ($Meta_x[$def_id]==0 ? &Format_usage($def_id) : 0),
444     "\relax\n" if $Code_mode;
445 $Code_mode = $False;
    (This code is used in #51g (p.38), #55 (p.38), and #58 (p.39).)
```

**2.5.5.8 The `&Format_usage` function** This function is used to format the usage information. This is not as trivial as it might at first seem; for instance, the following rules apply for English:

- When there are two elements, there should be just an “and” between the two.
- When there are three or more elements, the last two should be separated by “, and” and the others just by a comma.

(Other languages have different rules, but most languages can be handled properly by suitable definitions of `\wzsep`, `\wztwosep`, and `\wzlastsep`; see Section 3.1.2 on page 63.)

Also, we want to omit multiple occurrences of identical references.

The function has one parameter:

1. the index of a meta symbol.

It returns the  $\LaTeX$  code as a text string.

```
#61 <w0-f-latex: utility functions> ≡
446 sub Format_usage {
447     local $_ = $Meta_use[shift]; chomp;
448     my @data = ();
449     my ($last_x, $x) = ("", "");
450
451     foreach $x (split) { push(@data, $x) if $x ne $last_x; $last_x = $x; }
452
453     my ($n, $res) = (0, "");
454     foreach (@data) {
455         if (++$n == @data && @data > 2) { $res .= "\\wzlastsep\n"; }
456         elsif ($n == 2 && @data == 2) { $res .= "\\wztwosep\n"; }
457         elsif ($n > 1) { $res .= "\\wzsep\n"; }
458         $res .= "\\wzhref" . (/^\(d+\)-(d+)$/ ? "{$1}{$2}" : "{$_}{0}");
459     }
460     return $res;
461 }
```

(This code is extended in #61<sub>a</sub> (p.41). It is used in #46 (p.32).)

### 2.5.6 Making an index

All L<sup>A</sup>T<sub>E</sub>X code generated by `wO-f-latex` will contain indices for the variables, functions, classes, and meta symbols found in the code. It is up to the user, however, to introduce any of these indices into his or her documentation using the macros `\wzvarindex`, `\wzfuncindex`, `\wzclassindex`, or `\wzmetaindex`, respectively.

```
#62 <wO-f-latex: generate indices> ≡
462 <wO-f-latex: generate the class index #70 (p.43)>
463 <wO-f-latex: generate the function index #63 (p.41)>
464 <wO-f-latex: generate the variable index #69 (p.43)>
465 <wO-f-latex: generate the meta symbol index #71 (p.43)>
(This code is used in #51f (p.38).)
```

**2.5.6.1 Generating the function index** The function index is generated using the `&Generate_index` function.

```
#63 <wO-f-latex: generate the function index> ≡
466 &Generate_index("func", \%Func_index);
(This code is used in #62 (p.41).)
```

This function is used to produce both the function and variable indices. It has two parameters:

1. “func” when generating the function index, or “var” or “class”, and
2. a reference to the index information in a hash table.

```
#61a <wO-f-latex: utility functions #61 (p.40)> + ≡
467 sub Generate_index {
468   my ($cmd, $index) = @_ ;
469   local $_ ;
470   my (@lines, $lx, $initial, $last_initial);
471
472   print $Output "\\def\\wz", $cmd,
473     "index\\{\begin{wzindex}\\wz", $cmd, "indexname}{2}\n";
474   foreach (sort indexwise keys %{$index}) {
475     <wO-f-latex: produce an initial (if required) #68 (p.42)>
476     <wO-f-latex: generate an index entry #64 (p.42)>
477   }
478   print $Output "\\end{wzindex}\\}\n";
479 }
(This code is extended in #61b (p.41).)
```

The sorting is not quite straightforward. If the index entry does not start with a letter,<sup>8</sup> we will ignore the initial character when sorting.

```
#61b <wO-f-latex: utility functions #61 (p.40)> + ≡
480 sub indexwise {
481   my $ax = $a =~ /^\\w/ ? $a : substr($a,1)." $1 ";
482   my $bx = $b =~ /^\\w/ ? $b : substr($b,1)." $1 ";
483   local ($a, $b) = (uc($ax), uc($bx));
484   return &alphabetically;
485 }
(This code is extended in #61c (p.43).)
```

Each index entry starts with a call on `\wzx`.

---

<sup>8</sup>This quaint sorting rule was invented to handle Perl variable and functions properly; these start with a special character like “\$”, “@”, “%”, or “&”.

```
#64 <w0-f-latex: generate an index entry> ≡
486 print $Output "\\wzx{" , &Latexify($_) , "}";
487 <w0-f-latex: format and print index line numbers #65 (p.42)>
488 print $Output "\\par\n";
(This code is used in #61a (p.41).)
```

The index entry consists of a sequence of line numbers. These must be sorted,<sup>9</sup> and duplicates must be removed. The sorted sequence is saved in @lines.

```
#65 <w0-f-latex: format and print index line numbers> ≡
489 @lines = ();
490 foreach $lx (sort { $a <=> $b } split(" ", $index->{$_})) {
491     push @lines, $lx unless @lines && $lines[$#lines] == $lx;
492 }
(This code is extended in #65a (p.42). It is used in #64 (p.42).)
```

Then we can print all the line numbers, but we must first check whether they form a consecutive sub-sequence.

```
#65a <w0-f-latex: format and print index line numbers #65 (p.42)> +≡
493 while (@lines) { $lx = shift @lines;
494     <w0-f-latex: check for range of line numbers #66 (p.42)>
495     <w0-f-latex: print index line number #67 (p.42)>
496     print $Output "\\wzindexsep\n" if @lines;
497 }
```

If there are at least three consecutive line numbers, they are replaced by a sequence.

```
#66 <w0-f-latex: check for range of line numbers> ≡
498 if (@lines>=2 && $lines[0]==$lx+10 && $lines[1]==$lx+20) {
499     <w0-f-latex: print index line number #67 (p.42)>
500     $lx = shift(@lines) while @lines && $lines[0]==$lx+10;
501     print $Output "--";
502     <w0-f-latex: print index line number #67 (p.42)>
503     print $Output "\\wzindexsep\n" if @lines;
504     next;
505 }
(This code is used in #65a (p.42).)
```

When printing a line number, a final 0 must be replaced by a call on \wzul; a final 1 is just removed.

```
#67 <w0-f-latex: print index line number> ≡
506 print $Output $lx%10 ? (" ".int($lx/10)) : ("\\wzul{" .int($lx/10)."}");
(This code is used in #65a (p.42) and #66 (p.42).)
```

Whenever the first letter in the index changes, an initial should be printed to mark this.

```
#68 <w0-f-latex: produce an initial (if required)> ≡
507 $initial = /^[a-z]/i ? uc(substr($_,0,1)) : uc(substr($_,1,1));
508 print $Output "\\wzinitial{" , &Latexify($initial) , "}\n"
509     if $initial && $initial ne $last_initial;
510 $last_initial = $initial;
(This code is used in #61a (p.41).)
```

---

<sup>9</sup>The reason the line numbers must be sorted, is that we want to list defining occurrences before usage on the same line.

**2.5.6.2 Generating the variable index** The `&Generate_index` function can handle the variable index, too.

```
#69 <w0-f-latex: generate the variable index> ≡
511 &Generate_index("var", \%Var_index);
    (This code is used in #62 (p.41).)
```

**2.5.6.3 Generating the class index** The `&Generate_index` function can even handle the class index.

```
#70 <w0-f-latex: generate the class index> ≡
512 &Generate_index("class", \%Class_index);
    (This code is used in #62 (p.41).)
```

**2.5.6.4 Generating the meta symbol index** The meta symbol index can be created by just looking at the contents of `%Meta_id`.

```
#71 <w0-f-latex: generate the meta symbol index> ≡
513 print $Output "\\def\\wzmetaindex{\\begin{wzindex}",
514       "\\wzmetaindexname}{1}\\n";
515 foreach (sort alphabetically keys %Meta_id) {
516     print $Output "\\wzx{\\wzmeta{" , &Latexify($_), "~\\upshape\\#",
517               "$Meta_id{$_}}\\wzlongpageref{$Meta_id{$_}}",
518               ($Meta_use[$Meta_id{$_}] ? " " : "\\rlap{~*}"),
519               "\\par\\n";
520 }
521 print $Output "\\raggedright\\wzmetaindexstartext\\par
522 \\end{wzindex}}\\n";
    (This code is used in #62 (p.41).)
```

## 2.5.7 Utility functions

**2.5.7.1 The function `&Warning`** gives the user a warning and increases the error count `$N_errors`.

```
#61c <w0-f-latex: utility functions #61 (p.40)> + ≡
523 sub Warning {
524     &Message(@_); ++$N_errors;
525 }
```

The error count must be initialized.

```
#47m <w0-f-latex: initialization #47 (p.32)> + ≡
526 my $N_errors = 0;
```

## 2.6 Language filters

This Section contains the various optional language filters that have been written so far. These filters all read tokens from *standard input* and write tokens to *standard output*. They should *never* generate any error messages.

In addition to the tokens received from the preprocessor (see Section 2.3 on page 21) the filters may generate the following new tokens which are used when creating the function and variable indices:

**`cdef`** specifies that a class is being defined.

**`cuse`** shows that the class has been used.

**fdef** specifies that a function has been defined.

**fuse** shows that the function has been used.

**vdef** tells of the declaration of a variable.

**vuse** notes the use of a variable.

(These tokens all contain a name; that name is always surrounded by a “:” and a “;”.)

Also, the following tokens may be generated:

**bcode** is a variant of code when the code should be set in boldface.

**bicode** is another variant to be used when bold italic code is desired.

**icode** is yet another variant to be used when the code is to be set in italics.

### 2.6.1 The C language filter **w0-l-c**

This filter is used to analyze C programs. It is not very advanced, so it is easily confused by obscure C code and things like multi-line comments.

The filter is written in Perl.

```
#72 <w0-l-c> ≡
527 <perl #105 (p.57)>
528
529 <w0-l-c definitions #73 (p.44)>
530 <w0-l-c parameter handling #74 (p.45)>
531 <w0-l-c read C code #75 (p.45)>
532 exit 0;
533
534 <user message functions #110 (p.60)>
    (This code is not used.)
```

#### 2.6.1.1 Definitions

Even the language filters should be able to identify themselves with their name and version number.

```
#73 <w0-l-c definitions> ≡
535 my ($Prog, $Version) = ("w0-l-c", "<version #106 (p.57)>");
    (This code is extended in #73a (p.44). It is used in #72 (p.44).)
```

The syntax for C identifiers is used several times so it is an advantage to name it.

```
#73a <w0-l-c definitions #73 (p.44)> + ≡
536 my $C_id = "[A-Za-z_]\w*";
    (This code is extended in #73b (p.44).)
```

The same goes for an identifier list.

```
#73b <w0-l-c definitions #73 (p.44)> + ≡
537 my $C_id_list = "$C_id([*, ]*$C_id)*";
    (This code is extended in #73c (p.44).)
```

We need a table %Res\_words with all the reserved words in C.

```
#73c <w0-l-c definitions #73 (p.44)> + ≡
538 my %Res_words = ();
```

```

539 my %Special_words = ();
540 for ("auto", "break", "case", "const", "continue", "default", "do",
541      "else", "enum", "extern", "for", "goto", "if", "register",
542      "return", "sizeof", "static", "struct", "switch", "typedef",
543      "union", "volatile", "while")
544   { $Res_words{$_} = $Special_words{$_} = 1; }
  (This code is extended in #73d (p.45).)

```

We also need a table %Type\_words with all the predefined type words in C.

```

#73d <w0-l-c definitions #73 (p.44)> +≡
545 my %Type_words = ();
546 for ("char", "double", "float", "int", "long", "short",
547      "unsigned", "void")
548   { $Type_words{$_} = $Special_words{$_} = 1; }
  (This code is extended in #73e (p.45).)

```

A table %Both\_words contains a union of the two.

**2.6.1.2 Parameter handling** This loop will look at all the parameters; however, only `-e` and `-v` have any effect.

```

#74 <w0-l-c parameter handling> ≡
549 PARAM:
550 while (@ARGV && $ARGV[0] =~ /^-/) { $_ = shift;
551     /^-e$/ and do { $Enhance = "yes"; next PARAM; };
552     /^-v$/ and do {
553         print STDERR "This is $Prog (version $Version)\n";
554         next PARAM; };
555 }
  (This code is used in #72 (p.44).)

```

Variable \$Enhance must be declared.

```

#73e <w0-l-c definitions #73 (p.44)> +≡
556 my $Enhance = 0;

```

**2.6.1.3 Reading the C code** Now it's time to read the C code. All tokens read will be printed, but only the code ones require further handling.

```

#75 <w0-l-c read C code> ≡
557 my $line = "";
558 while (<>) {
559     unless (/^code:(.*)"$/ ) { print; next; };
560     chomp($line = $_);
561     $_ = $line; <w0-l-c check C code for functions and variables #76 (p.45)>
562     if ($Enhance) {
563         $_ = $line; <w0-l-c enhance C code #79 (p.47)>
564     } else {
565         print "code:$line;\n";
566     }
567 }
  (This code is used in #72 (p.44).)

```

**2.6.1.4 Looking for functions and variables** There may be several functions and variables on each line.

```

#76 <w0-l-c check C code for functions and variables> ≡
568 while ($_) {
569     <w0-l-c check for names #77 (p.46)>
570 }
  (This code is used in #75 (p.45).)

```

Before we can look for names, we must remove preprocessor commands and comments:

```
#77 <w0-l-c check for names> ≡
571 redo if s|^#.*$||;
572 redo if s|^\s*/\*.*\*/||;
(This code is extended in #77a (p.46). It is used in #76 (p.45).)
```

We should also remove string and character literals:

```
#77a <w0-l-c check for names #77 (p.46)> +≡
573 redo if s:^\s*"(\\"|["])*"::;
574 redo if s:^\s*'(\\"'|['])*'::;
(This code is extended in #77b (p.46).)
```

Now we can check if we have an alphabetic symbol:

```
#77b <w0-l-c check for names #77 (p.46)> +≡
575 if (s|^\s*($C_id)|o) {
576     my $id = $1;
577     <w0-l-c check alphabetic symbol #78 (p.46)>
578 }
(This code is extended in #77c (p.47).)
```

Reserved words are ignored:

```
#78 <w0-l-c check alphabetic symbol> ≡
579 redo if $Res_words{$id};
(This code is extended in #78a (p.46). It is used in #77b (p.46).)
```

We may have a function call:

```
#78a <w0-l-c check alphabetic symbol #78 (p.46)> +≡
580 if (s|^\s*\(|\)|) {
581     print "fuse:$id;\n" unless $Special_words{$id};
582     redo;
583 }
(This code is extended in #78b (p.46).)
```

Or, we may have a function definition:

```
#78b <w0-l-c check alphabetic symbol #78 (p.46)> +≡
584 if (s|^\s*[ ]*($C_id)\s*\(|\)|) {
585     print "fdef:$1;\n" unless $Special_words{$1};
586     redo;
587 }
(This code is extended in #78c (p.46).)
```

Alternatively, we may have a variable declaration list:

```
#78c <w0-l-c check alphabetic symbol #78 (p.46)> +≡
588 if (s|^\s*[ ]*($C_id_list)|o) {
589     local $_;
590     foreach (split(/[*], [+/, $1)) {
591         print "vdef:$_;\n" unless $Special_words{$_};
592     }
593     redo;
594 }
(This code is extended in #78d (p.46).)
```

If nothing else, our alphabetic symbol is a variable:

```
#78d <w0-l-c check alphabetic symbol #78 (p.46)> +≡
595 print "vuse:$id;\n" unless $Special_words{$id};
```

596 redo;

Non-alphabetic symbols are ignored:

```
#77c <w0-l-c check for names #77 (p.46)> +≡
597 s:^(^s*[^A-Za-z_''/]+:: or s:^(^s*.?::;
```

**2.6.1.5 Enhance the C code** This filter enhances the C code in the following way:

- Preprocessor directives (i.e., lines starting with a “#”) are set in bold italic type.
- Comments are set in italics.
- Reserved words are set in bold type.

First, we look for the preprocessor lines:

```
#79 <w0-l-c enhance C code> ≡
598 /^#/ and do { print "bicode:$_\n"; next; };
(This code is extended in #79a (p.47). It is used in #75 (p.45).)
```

For the other lines, we can look for any special words.

```
#79a <w0-l-c enhance C code #79 (p.47)> +≡
599 while ($_) {
600 <w0-l-c look for special words #80 (p.47)>
601 }
```

However, first we check for comments:

```
#80 <w0-l-c look for special words> ≡
602 if (s:^(^s*/\.*\*/| |) {
603 print "icode:$&\n"; redo;
604 }
(This code is extended in #80a (p.47). It is used in #79a (p.47).)
```

On the other hand, string and character literals are just ordinary code:

```
#80a <w0-l-c look for special words #80 (p.47)> +≡
605 if (s:^(^s*"(\\" | [^"])*::) {
606 print "code:$&\n"; redo;
607 }
608 if (s:^(^s*'(\\" | [^'])*::) {
609 print "code:$&\n"; redo;
610 }
(This code is extended in #80b (p.47).)
```

Now, we can look for reserved words:

```
#80b <w0-l-c look for special words #80 (p.47)> +≡
611 if (s:^(^s*)($C_id)| |o) {
612 if ($Res_words{$2}) {
613 print "code:$1\n" if $1;
614 print "bcode:$2\n";
615 } else {
616 print "code:$&\n";
617 }
618 redo;
619 }
(This code is extended in #80c (p.48).)
```

Anything else is just ordinary code:

```
#80c <w0-l-c look for special words #80 (p.47)> +≡
620 s:^(\\s*[^A-Za-z_''/]+):: or s:^(\\s*\\.?::;
621 print "code:$&e;\n" if $&e;
```

## 2.6.2 The Java language filter w0-l-java

This filter is rather similar to the C one; the major difference is the handling of classes.

The filter is written in Perl.

```
#81 <w0-l-java> ≡
622 <perl #105 (p.57)>
623
624 <w0-l-java definitions #82 (p.48)>
625 <w0-l-java parameter handling #83 (p.49)>
626 <w0-l-java read Java code #84 (p.49)>
627 exit 0;
628
629 <user message functions #110 (p.60)>
(This code is not used.)
```

### 2.6.2.1 Definitions Even the language filters should be able to identify themselves with their name and version number.

```
#82 <w0-l-java definitions> ≡
630 my ($Prog, $Version) = ("w0-l-java", "<version #106 (p.57)>");
(This code is extended in #82a (p.48). It is used in #81 (p.48).)
```

The syntax for Java class names and other identifiers is used several times so it is an advantage to name it.

```
#82a <w0-l-java definitions #82 (p.48)> +≡
631 my $Java_class_id = "[A-Z]([a-z_0-9]\\w*)?";
632 my $Java_other_id = "[a-zA-Z]\\w*";
(This code is extended in #82b (p.48).)
```

The same goes for an identifier list.

```
#82b <w0-l-java definitions #82 (p.48)> +≡
633 my $Java_id_list = "$Java_other_id(\\s*,\\s*$Java_other_id)*";
(This code is extended in #82c (p.48).)
```

We need a table %Res\_words with all the reserved words of Java.

```
#82c <w0-l-java definitions #82 (p.48)> +≡
634 my %Res_words = ();
635 my %Special_words = ();
636 for ("abstract", "assert", "break", "case", "catch", "class", "const",
637      "continue", "default", "do", "else", "extends", "false", "final",
638      "finally", "for", "goto", "if", "implements", "import", "instanceof",
639      "interface", "native", "new", "package", "private", "protected",
640      "public", "return", "static", "strictfp", "super", "switch",
641      "synchronized", "this", "throw", "throws", "transient", "true",
642      "try", "volatile", "while")
643   { $Res_words{$_} = $Special_words{$_} = 1; }
(This code is extended in #82d (p.48).)
```

We also need a list %Type\_words with all the type names:

```
#82d <w0-l-java definitions #82 (p.48)> +≡
644 my %Type_words = ();
```

```

645 for ("boolean", "byte", "char", "double", "float", "int",
646      "long", "short", "void")
647 { $Type_words[$_] = $Special_words[$_] = 1; }

```

(This code is extended in #82<sub>e</sub> (p.49).)

Incidentally, having a table %Special\_words which is the union of the last two, seems a good idea.

**2.6.2.2 Parameter handling** This loop will look at all the parameters; however, only `-e` and `-v` have any effect.

```

#83 <w0-l-java parameter handling> ≡
648 PARAM:
649 while (@ARGV && $ARGV[0] =~ /^-/) { $_ = shift;
650     /^-e$/ and do { $Enhance = "yes"; next PARAM; };
651     /^-v$/ and do {
652         print STDERR "This is $Prog (version $Version)\n";
653         next PARAM; };
654 }

```

(This code is used in #81 (p.48).)

Variable \$Enhance must be declared.

```

#82e <w0-l-java definitions #82 (p.48)> +≡
655 my $Enhance = 0;

```

**2.6.2.3 Reading the Java code** Now it's time to read the Java code. All tokens read will be printed, but only the code ones require further handling.

```

#84 <w0-l-java read Java code> ≡
656 my $line = "";
657 while (<>) {
658     unless (/^code:(.);$/) { print; next; };
659     chomp($line = $1);
660     $_ = $line; <w0-l-java check Java code for names #85 (p.49)>
661     if ($Enhance) {
662         $_ = $line; <w0-l-java enhance Java code #89 (p.51)>
663     } else {
664         print "code:$line;\n";
665     }
666 }

```

(This code is used in #81 (p.48).)

**2.6.2.4 Looking for methods and variables** There may be several methods and variables on each line.

```

#85 <w0-l-java check Java code for names> ≡
667 while ($_) {
668     <w0-l-java check for names #86 (p.49)>
669 }

```

(This code is used in #84 (p.49).)

Before we can look for names, we must remove any comments:

```

#86 <w0-l-java check for names> ≡
670 redo if s|^s*//.*||;
671 redo if s|^s*/\*.*\*/||;

```

(This code is extended in #86<sub>a</sub> (p.50). It is used in #85 (p.49).)

We should also remove string and character literals:

```
#86a <w0-l-java check for names #86 (p.49)> +≡
672 redo if s:^(s*(\|["])*::;
673 redo if s:^(s*(\|['])*::;
(This code is extended in #86b (p.50).)
```

Nor are we interested in package and import specifications:

```
#86b <w0-l-java check for names #86 (p.49)> +≡
674 redo if s|^(s*package\s+[^;]*;|);
675 redo if s|^(s*import\s+[^;]*;|);
(This code is extended in #86c (p.50).)
```

Class declarations are easiest to find:

```
#86c <w0-l-java check for names #86 (p.49)> +≡
676 if (s|^(s*class\s+(\$Java_class_id)\b| |o) {
677     print "cdef:$1;\n";
678     redo;
679 }
(This code is extended in #86d (p.50).)
```

Then we can check if we have an alphabetic symbol:

```
#86d <w0-l-java check for names #86 (p.49)> +≡
680 if (s|^(s*(\$Java_other_id)| |o) {
681     my $id = $1;
682     <w0-l-java check alphabetic symbol #87 (p.50)>
683 }
(This code is extended in #86e (p.51).)
```

Reserved words are ignored (as we have already handled class):

```
#87 <w0-l-java check alphabetic symbol> ≡
684 redo if $Res_words{$id};
(This code is extended in #87a (p.50). It is used in #86d (p.50).)
```

We may have a constructor definition or a method call:

```
#87a <w0-l-java check alphabetic symbol #87 (p.50)> +≡
685 if (s|^(s*\(| |) {
686     if ($id =~ /^\$Java_class_id$/o) {
687         print "cuse:$id;\n";
688     } else {
689         print "fuse:$id;\n" unless $Special_words{$id};
690     }
691     redo;
692 }
(This code is extended in #87b (p.50).)
```

Or, we may have a method definition:

```
#87b <w0-l-java check alphabetic symbol #87 (p.50)> +≡
693 if (s|^(s*(\$Java_other_id)\s*\(| |o) {
694     my $f = $1;
695     print "cuse:$id;\n" if $id =~ /^\$Java_class_id$/o;
696     print "fdef:$f;\n" unless $Special_words{$f};
697     <w0-l-java check formal parameter list #88 (p.51)>
698     redo;
699 }
(This code is extended in #87c (p.51).)
```

Checking the formal parameter list is pretty straightforward:

```
#88 <w0-l-java check formal parameter list> ≡
700 if (s|^\s*[\^]*\)|) {
701     local $_;
702     foreach (split(/,/ , $&e)) {
703         if (/($Java_other_id)\s+($Java_other_id)/o) {
704             my $type = $1;
705             my $id = $2;
706             print "cuse:$type;\n" if $type =~ /^$Java_class_id$/o;
707             print "vdef:$id;\n";
708         }
709     }
710 }
    (This code is used in #87b (p.50).)
```

Alternatively, the first identifier may have a variable declaration list:

```
#87c <w0-l-java check alphabetic symbol #87 (p.50)> +≡
711 if (($Type_words{$id} || $id =~ /^$Java_class_id$/o) &&&
712     s|^\s*($Java_id_list)|)o) {
713     print "cuse:$id;\n" if $id =~ /^$Java_class_id$/o;
714     local $_;
715     foreach (split(/[, ]+/, $1)) {
716         print "vdef:$_;\n" unless $Special_words{$_};
717     }
718     redo;
719 }
    (This code is extended in #87d (p.51).)
```

If nothing else, our alphabetic symbol is a variable or a class:

```
#87d <w0-l-java check alphabetic symbol #87 (p.50)> +≡
720 if ($id =~ /^$Java_class_id$/o) {
721     print "cuse:$id;\n";
722 } else {
723     print "vuse:$id;\n" unless $Special_words{$id};
724 }
725 redo;
```

Non-alphabetic symbols are ignored:

```
#86e <w0-l-java check for names #86 (p.49)> +≡
726 s:\s*[^A-Za-z_ "'/]+:: or s:\s*.\?::;
```

**2.6.2.5 Enhance the Java code** This filter enhances the Java code in the following way:

- Class names are set in bold italic type.
- Comments are set in italics.
- Reserved words are set in bold type.

For each line, we can look for any special words or symbols.

```
#89 <w0-l-java enhance Java code> ≡
727 while ($) {
728     <w0-l-java look for special words or symbols #90 (p.51)>
729 }
    (This code is used in #84 (p.49).)
```

However, first we check for comments:

```
#90 <w0-l-java look for special words or symbols> ≡
730 if (s|^\s*/\.*$|) {
```

```

731     print "icode:$&&\n"; redo;
732 } elsif (s | ^\s*\/*.*\*/ | | ) {
733     print "icode:$&&\n"; redo;
734 }

```

(This code is extended in #90<sub>a</sub> (p.52). It is used in #89 (p.51).)

On the other hand, string and character literals are just ordinary code:

#90<sub>a</sub> *<w0-l-java look for special words or symbols #90 (p.51)> + ≡*

```

735 if (s: ^\s*"(\\" | [^"])*": :) {
736     print "code:$&&\n"; redo;
737 }
738 if (s: ^\s*'(\\" | [^'])*': :) {
739     print "code:$&&\n"; redo;
740 }

```

(This code is extended in #90<sub>b</sub> (p.52).)

Now, we can look for reserved words:

#90<sub>b</sub> *<w0-l-java look for special words or symbols #90 (p.51)> + ≡*

```

741 if (s | ^(\s*)(\$Java_other_id | | o) {
742     my ($space, $id) = ($1, $2);
743
744     if ($id =~ /^$Java_class_id$/o) {
745         print "code:$space;\n" if $space;
746         print "bicode:$id;\n";
747     } elsif ($Res_words{$id}) {
748         print "code:$space;\n" if $space;
749         print "bcode:$id;\n";
750     } else {
751         print "code:$space$id;\n";
752     }
753     redo;
754 }

```

(This code is extended in #90<sub>c</sub> (p.52).)

Anything else is just ordinary code:

#90<sub>c</sub> *<w0-l-java look for special words or symbols #90 (p.51)> + ≡*

```

755 s: ^\s*[^A-Za-z_"/]+:: or s: ^\s*\.?::;
756 print "code:$&&\n" if $&&;

```

### 2.6.3 The `LaTeX` language filter `w0-l-latex`

This filter is used to analyze `LaTeX` code. It will put new commands in boldface and comments in italics; it will also collect information on which commands are used.

The filter is written in Perl.

#91 *<w0-l-latex> ≡*

```

757 <perl #105 (p.57)>
758
759 <w0-l-latex definitions #92 (p.53)>
760 <w0-l-latex parameter handling #93 (p.53)>
761 <w0-l-latex read LaTeX code #94 (p.53)>
762 exit 0;
763
764 <user message functions #110 (p.60)>

```

(This code is not used.)

**2.6.3.1 Definitions** Even the language filters should be able to identify themselves with their name and version number.

```
#92 <w0-l-latex definitions> ≡
765 my ($Prog, $Version) = ("w0-l-latex", "<version #106 (p.57)>");
    (This code is extended in #92a (p.53). It is used in #91 (p.52).)
```

The syntax for L<sup>A</sup>T<sub>E</sub>X identifiers is use more than once so it is an advantage to name it.

```
#92a <w0-l-latex definitions #92 (p.53)> + ≡
766 my $LaTeX_id = "\\ \\ ([A-Za-z@]+|.)";
    (This code is extended in #92b (p.53).)
```

Normally, the text is not enhanced (i.e., not printed in boldface or italics).

```
#92b <w0-l-latex definitions #92 (p.53)> + ≡
767 my ($Bcode, $Icode) = ("code", "code");
    (This code is extended in #92c (p.53).)
```

**2.6.3.2 Parameter handling** This loop will handle all the parameters; however, only `-e` and `-v` have any effect.

```
#93 <w0-l-latex parameter handling> ≡
768 PARAM:
769 while (@ARGV && $ARGV[0] =~ /^-/) { $_ = shift;
770     /^-e$/ and do { $Bcode = "bcode"; $Icode = "icode";
771         next PARAM; };
772     /^-v$/ and do {
773         print STDERR "This is $Prog (version $Version)\n";
774         next PARAM; };
775 }
    (This code is used in #91 (p.52).)
```

Variables `$Bcode` and `$Icode` will record use of the `-e` option.

```
#92c <w0-l-latex definitions #92 (p.53)> + ≡
776 my ($Bcode, $Icode) = (0, 0);
```

**2.6.3.3 Reading the L<sup>A</sup>T<sub>E</sub>X code** Now it's time to read the L<sup>A</sup>T<sub>E</sub>X code. All tokens read will be printed, but only the code requires further handling.

```
#94 <w0-l-latex read LaTeX code> ≡
777 while (<<>) {
778     unless (/^code:(.*)"$/ ) { print; next; };
779     chomp($_ = $1);
780     <w0-l-latex check LaTeX code #95 (p.53)>
781 }
    (This code is used in #91 (p.52).)
```

Is there any L<sup>A</sup>T<sub>E</sub>X code that requires attention?

```
#95 <w0-l-latex check LaTeX code> ≡
782 while (/[%\\]/) {
783     print "code:$';\n" if $';
784     $_ = $&.$';
785     <w0-l-latex check for comments #96 (p.54)>
786     <w0-l-latex check for declarations #97 (p.54)>
787     <w0-l-latex check for use #98 (p.54)>
788 }
789 print "code:$_;\n" if $_;
    (This code is used in #94 (p.53).)
```

**2.6.3.4 Look for comments** A “%” indicates a comment. This extends to the end of the line.

```
#96 <w0-l-latex check for comments> ≡
790 /%^/ and do { print "$Icode:$_\n"; $_ = ""; last; };
(This code is used in #95 (p.53).)
```

**2.6.3.5 Look for declarations** New L<sup>A</sup>T<sub>E</sub>X commands are declared using `\newcommand`, `\renewcommand`, or `\def`.

```
#97 <w0-l-latex check for declarations> ≡
791 s/^(\\(re)?newcommand\s*\*?\s*\{(.+?)\})// and do {
792   print "code:$1\n$Bcode:$3\ncode:$4\n";
793   print "fdef:$3\n";
794   next; };
795 s/^(\\def\s*)($LaTeX_id)//o and do {
796   print "code:$1\n$Bcode:$2\n";
797   print "fdef:$2\n"; next; };
(This code is extended in #97a (p.54). It is used in #95 (p.53).)
```

New environments are declared using `\newenvironment` or `\renewenvironment`.

```
#97a <w0-l-latex check for declarations #97 (p.54)> + ≡
798 s/^(\\(re)?newenvironment\s*\{(.+?)\})// and do {
799   print "code:$1\n$Bcode:$3\ncode:$4\n";
800   print "vdef:$3\n"; next; };

```

**2.6.3.6 Look for use** If the “\” does not start a command definition, it must indicate usage. First we check to see if it starts or terminates an environment.

```
#98 <w0-l-latex check for use> ≡
801 s/^(\\begin\s*\{(.+?)\})// and do {
802   print "code:$&\n";
803   print "vuse:$1\n"; next; };
(This code is extended in #98a (p.54). It is used in #95 (p.53).)
```

The corresponding `\end` is ignored.

```
#98a <w0-l-latex check for use #98 (p.54)> + ≡
804 s/^(\\end\s*\{(.+?)\})// and do {
805   print "code:$&\n"; next; };
(This code is extended in #98b (p.54).)
```

Then we look for use of ordinary L<sup>A</sup>T<sub>E</sub>X commands.

```
#98b <w0-l-latex check for use #98 (p.54)> + ≡
806 s/^\$LaTeX_id//o and do {
807   print "code:$&\n";
808   print "fuse:$&\n"; next; };
(This code is extended in #98c (p.54).)
```

If the “\” does not start a legal L<sup>A</sup>T<sub>E</sub>X command, we will not speculate about why this is so, but just print it as a normal character.

```
#98c <w0-l-latex check for use #98 (p.54)> + ≡
809 print "code:\\\n"; s/^.//; next;
```

### 2.6.4 The Perl language filter **w0-l-perl**

This part of the code contains the optional Perl filter whose job it is to find the variables and functions used in the program. It can also enhance the printing of the code by using a bold or italic font for some of the text.

This filter is not without fault. For instance, in the statement

```
print "Program $Prog [-v] file...\n";
```

the filter will assume that the array `@Prog` is referenced. Getting this correct, however, is too difficult to be worth the bother.

The Perl filter is, of course, written in Perl.

```
#99 <w0-l-perl> ≡
810 <perl #105 (p.57)>
811
812 <w0-l-perl definitions #100 (p.55)>
813 <w0-l-perl parameter handling #101 (p.55)>
814 <w0-l-perl read Perl code #102 (p.56)>
815 exit 0;
816
817 <user message functions #110 (p.60)>
    (This code is not used.)
```

**2.6.4.1 Definitions** Even the language filters should be able to identify themselves with their name and version number.

```
#100 <w0-l-perl definitions> ≡
818 my ($Prog, $Version) = ("w0-l-perl", "<version #106 (p.57)>");
    (This code is extended in #100a (p.55). It is used in #99 (p.55).)
```

The syntax for Perl identifiers is used so often that it should be given a name.

```
#100a <w0-l-perl definitions #100 (p.55)> + ≡
819 my $Perl_id = "[A-Za-z]\\w*";
    (This code is extended in #100b (p.55).)
```

**2.6.4.2 Parameter handling** This loop will handle all the parameters; however, only `-e` and `-v` have any effect.

```
#101 <w0-l-perl parameter handling> ≡
820 PARAM:
821 while (@ARGV &&& $ARGV[0] =~ /^-/) { $_ = shift;
822     /^-e$/ and do { $Enhance = "yes"; next PARAM; };
823     /^-v$/ and do {
824         print STDERR "This is $Prog (version $Version)\n";
825         next PARAM; };
826 }
    (This code is used in #99 (p.55).)
```

The state variables must be declared.

```
#100b <w0-l-perl definitions #100 (p.55)> + ≡
827 my $Enhance = 0;
    (This code is extended in #100c (p.57).)
```

**2.6.4.3 Reading the Perl code** Now it's time to read the Perl code. All tokens read will be printed, but only the code tokens will require further handling.

```
#102 <w0-l-perl read Perl code> ≡
828 my $line;
829 while (<>) {
830     unless (/^code:(.*/);$/) { print; next; };
831     chomp($line = $1);
832     $_ = $line; <w0-l-perl check Perl code for functions and variables #103 (p.56)>
833     if ($Enhance) {
834         $_ = $line; <w0-l-perl enhance Perl code #104 (p.57)>
835     } else {
836         print "code:$line;\n";
837     }
838 }
(This code is used in #99 (p.55).)
```

**2.6.4.4 Check for functions** First, check for function definition and use. Perl functions are defined using the reserved word **sub** and used by being prefixed with a **&**.<sup>10</sup>

```
#103 <w0-l-perl check Perl code for functions and variables> ≡
839 while (s/\bsub\s+(\$Perl_id)//o) { print "fdef:&$1;\n"; }
840 while (s/&(\$Perl_id)//o) { print "fuse:&$1;\n"; }
(This code is extended in #103a (p.56). It is used in #102 (p.56).)
```

**2.6.4.5 Check for variables** Now, we can look for the variables. Since there is no explicit declaration of variables in Perl, only occurrence will be monitored.

First we can check for array variables which can occur in two forms:

```
@var or $var[...]
```

The default variable @\_ is used so often that there is no need to record that.

```
#103a <w0-l-perl check Perl code for functions and variables #103 (p.56)> +≡
841 while (s/\@(\$Perl_id)//o) { print "vuse:@$1;\n" unless $1 eq "_"; }
842 while (s/\$(\$Perl_id)\s*\{/o) { print "vuse:@$1;\n" unless $1 eq "_"; }
(This code is extended in #103b (p.56).)
```

Then we can look for hash variables which also occur in two variant forms:

```
%var or $var{...}
```

```
#103b <w0-l-perl check Perl code for functions and variables #103 (p.56)> +≡
843 while (s/%(\$Perl_id)//o) { print "vuse:%$1;\n"; }
844 while (s/\$(\$Perl_id)\s*\{/o) { print "vuse:%$1;\n"; }
(This code is extended in #103c (p.56).)
```

And finally we can search for ordinary variables. The default variable \$\_ is so commonly used that indexing it will provide little information; consequently, it is ignored.

```
#103c <w0-l-perl check Perl code for functions and variables #103 (p.56)> +≡
845 while (s/\$(\$Perl_id)//o) { print "vuse:\$$1;\n" unless $1 eq "_"; }
```

<sup>10</sup>More lax notation is permitted in Perl, but checking for that becomes too difficult by far.

**2.6.4.6 Enhance the Perl code** This filter enhances the Perl code in the following naïve way:

- All line comments (i.e., lines starting with a “#”) are set in italic type.
- Reserved words are set in bold type. (This is done even if they occur in text strings or comments.)

First, we look for the all comment lines:

```
#104 <w0-l-perl enhance Perl code> ≡
846 /^\s*#/ and do { print "icode:$_\n"; next; };
(This code is extended in #104a (p.57). It is used in #102 (p.56).)
```

For the other lines, we can look for any reserved words.

```
#104a <w0-l-perl enhance Perl code #104 (p.57)> +≡
847 while (/^\b($Res_Perl)\b/o) {
848     print "code:$'\n" if $';
849     print "bcode:$&\n";
850     $_ = $';
851 }
852 print "code:$_\n" if $_;
```

The search pattern \$Res\_Perl is composed from all the reserved words in Perl.<sup>11</sup>

```
#100c <w0-l-perl definitions #100 (p.55)> +≡
853 my @Res_word = ("and", "continue", "die", "do", "dump", "else",
854 "elsif", "eval", "exec", "exit", "for", "foreach", "fork",
855 "if", "kill", "last", "next", "or", "return", "sub",
856 "unless", "until", "wait", "while");
857 my $Res_Perl = join("|", @Res_word);
```

## 2.7 Miscellaneous

### 2.7.1 The Perl interpreter

The meta symbol *<perl>* defines the location of the Perl interpreter.

```
#105 <perl> ≡
858 #! /store/bin/perl
(This code is extended in #105a (p.57). It is used in #1 (p.13), #8 (p.16), #18 (p.21), #32 (p.27),
#46 (p.32), #72 (p.44), #81 (p.48), #91 (p.52), and #99 (p.55).)
```

All Perl programs should use the strictest error checking available.

```
#105a <perl #105 (p.57)> +≡
859 use strict;
```

### 2.7.2 Program version

For compatibility reasons, all subprograms are assigned the same program version.

```
#106 <version> ≡
860 1.09c
(This code is used in #2 (p.13), #9 (p.17), #19 (p.24), #33 (p.28), #47 (p.32), #73 (p.44), #82 (p.48),
#92 (p.53), #100 (p.55), and #116 (p.63).)
```

<sup>11</sup>One might argue what is a reserved word in Perl; is print? or and? or join? I have chosen to include only those words that have a syntactical purpose, or that directly influence the order of execution.

## 2.8 Adapting text for processing by $\text{\LaTeX}$

Most text in the ISO Latin-1 encoding may be processed as it is by  $\text{\LaTeX}$ , but some characters have a particular meaning to  $\text{\LaTeX}$ . Also, we must take care of some other special characters, and we must avoid unwanted ligatures.

The function `&Latexify` translates an ISO Latin-1 text into  $\text{\LaTeX}$  code. It accepts two parameters:

1. the original text and
2. an indication whether the spaces in the text are breakable (by default, they are not, as this parameter is optional).

```
#107 <latex generation functions> ≡
861 sub Latexify {
862     local $_ = shift;
863     my $break_spaces = shift;
864
865     <Latexify: adapt text #108 (p.58)>
866     return $_;
867 }
```

(This code is used in #46 (p.32).)

When using this function, please note that it assumes that the  $\text{\LaTeX}$  package `textcomp` has been loaded.

### 2.8.1 Handle “\”, “{”, and “}”

The three characters “\”, “{”, and “}” are translated as follows:

```
\ → \textbackslash{}
{ → \{
} → \}
```

Since the characters occur in each other’s definition, they are slightly tricky to translate. Using a temporary text — chosen so that it is extremely unlikely to occur by accident in any user text<sup>12</sup> — makes this possible.

```
#108 <Latexify: adapt text> ≡
868 s/\ \ / \ \ \textbackslash\«temporær bakslask»/g;
869 s/{ / \ \ \{/g;
870 s/} / \ \ \}/g;
871 s/\ \ \textbackslash\«temporær bakslask»/ \ \ \textbackslash\{/g;
(This code is extended in #108a (p.58). It is used in #107 (p.58).)
```

### 2.8.2 Handle the other special $\text{\LaTeX}$ characters

The seven other special  $\text{\LaTeX}$  characters all have well-known command equivalents.

```
#108a <Latexify: adapt text #108 (p.58)> + ≡
872 s/\#/ \ \ \#/g;
873 s/\$/ \ \ \$/g;
874 s/\%/ \ \ \%/g;
875 s/\&/ \ \ \&/g;
876 s/\_ / \ \ \_ /g;
```

<sup>12</sup>The text is in Norwegian to be even safer. (The term means “temporary backslash”, in case you wondered.)

```
877 s/\^/\textasciicircum{/g;
878 s/\~/\textasciitilde{/g;
(This code is extended in #108b (p.59).)
```

### 2.8.3 Handle other ISO Latin-1 characters

Some characters from the ISO Latin-1 character set produce math symbols in the inputenc package. We want the text version, both because they look better and because we want to avoid changing into math mode. Fortunately, we can find them in the textcomp package.

Math version	Text version	Math version	Text version
$\neg$	¬	1	1
$\pm$	±	$\mu$	μ
2	2	×	×
3	3	÷	÷

```
#108b <Latexify: adapt text #108 (p.58)> +≡
879 s/\-/\textlnot{/g; s/\pm/\textpm{/g; s/\^2/\texttwosuperior{/g;
880 s/\^3/\textthreesuperior{/g; s/\^1/\textonesuperior{/g;
881 s/\mu/\textmu{/g; s/\times/\texttimes{/g; s/\div/\textdiv{/g;
(This code is extended in #108c (p.59).)
```

### 2.8.4 Avoing unwanted ligatures

Most L<sup>A</sup>T<sub>E</sub>X fonts contain some ligatures that we do not want because they are totally different characters.

Original	Ligature	Original	Ligature
<<	«	>>	»
--	—	---	—
!‘	¡	?‘	¿
“	“	”	”
”	”		

```
#108c <Latexify: adapt text #108 (p.58)> +≡
882 s/<</<\null</g;
883 s/>>/>\null>/g;
884 s/---/-\null-\null-/g;
885 s/--/-\null-/g;
886 s/!‘/!\null‘/g; s/\?‘/\?\null‘/g;
887 s/“/“\null‘/g; s/”/”\null’/g; s/,,/, \null,/g;
(This code is extended in #108d (p.59).)
```

Note that the text ligatures “ff”, “ffi”, “ffl”, “fi”, and “fl” are not translated; they look all right the way they are.

### 2.8.5 Handle blanks

Finally, unless specified by the second parameter, we want to keep all blanks in the input.

```
#108d <Latexify: adapt text #108 (p.58)> +≡
888 s/ /~/g unless $break_spaces;
```

## 2.9 Expanding TAB characters

If the input contains TAB characters, they must usually be expanded to so many space characters that the following character's position is a multiple of 8 (if we start counting from 0).

The following piece of code assumes that the line is kept in the standard `$_` variable. (The code is "stolen" from *Programming Perl*[WCS96, page 66].)

```
#109 <expand TAB characters> ≡
889 while (s/\t+/'x(length($&)*8-length($')%8)/e) {}
      (This code is used in #23 (p.25).)
```

## 2.10 Printing user messages

This section describes two functions that occur in nearly every Perl program:

**&Error** prints an error message and terminates the program.

**&Message** just prints a message.

Both functions start the first message line with the program name; this implies that the variable `$Prog` must be defined.

### 2.10.1 The function &Error

As mentioned, this function will print an error message (using `&Message`) and terminate with status code 1. If something needs to be fixed before exiting, it can be handled by defining a function named `&Tidy_up`.

```
#110 <user message functions> ≡
890 sub Error {
891     &Message(@_);
892     &Tidy_up if defined &Tidy_up;
893     exit 1;
894 }
      (This code is extended in #110a (p.60). It is used in #1 (p.13), #8 (p.16), #18 (p.21), #32 (p.27),
      #46 (p.32), #72 (p.44), #81 (p.48), #91 (p.52), and #99 (p.55).)
```

### 2.10.2 The function &Message

The function `&Message` will print the text strings supplied as parameters. They will automatically be prefixed with the program name (or spaces), and line terminators will be added.

```
#110a <user message functions #110 (p.60)> + ≡
895 sub Message {
896     print STDERR "$Prog: ", shift, "\n";
897     while (@_) {
898         print STDERR " "x(2+length $Prog), shift, "\n";
899     }
900 }
```



```
919 }
    (This code is extended in #111a(p.62). It is used in #46 (p.32).)
```

Using this table, sorting is quite straightforward.

```
#111a <alphabetical sorting #111 (p.61)> + ≡
920 sub alphabetically {
921     <alphabetical sorting: simple first tests #112 (p.62)>
922     <alphabetical sorting: test initial character #113 (p.62)>
923     <alphabetical sorting: test other characters #114 (p.62)>
924 }
```

If the two texts are equal, or one is empty, the result is found quickly.

```
#112 <alphabetical sorting: simple first tests> ≡
925 return 0 if $a eq $b;
926 return -1 unless $a;
927 return 1 unless $b;
    (This code is used in #111a(p.62).)
```

If neither text is empty, we find the result by looking at the rank of the two initial characters.

```
#113 <alphabetical sorting: test initial character> ≡
928 my $ax = $Alpha_rank{substr($a,0,1)};
929 my $bx = $Alpha_rank{substr($b,0,1)};
930 return $ax <=> $bx if $ax != $bx;
    (This code is used in #111a(p.62).)
```

If the two initial characters are equal, we must compare the rest of the two text strings. For this, we can use `&alphabetically` recursively.

```
#114 <alphabetical sorting: test other characters> ≡
931 local ($a, $b) = (substr($a,1), substr($b,1));
932 return &alphabetically;
    (This code is used in #111a(p.62).)
```

## 3 L<sup>A</sup>T<sub>E</sub>X support

This part describes the `webzero` package and the `webzero` document class used when `web0` documents are typeset with L<sup>A</sup>T<sub>E</sub>X.

### 3.1 The `webzero` package

This package contains the necessary definitions for including `web0` documentation into any L<sup>A</sup>T<sub>E</sub>X document.

Names that are part of the user interface have names that start with “wz” and contain no “@”. All internal names start with “wz@” to avoid confusion with declarations in other packages.

```
#115 <webzero.sty> ≡
933 <webzero.sty identification #116 (p.63)>
934 <webzero.sty options #117 (p.63)>
935 <webzero.sty package loading #118 (p.65)>
936 <webzero.sty main code #119 (p.66)>
    (This code is not used.)
```

#### 3.1.1 Package identification

Every L<sup>A</sup>T<sub>E</sub>X package should contain version information.

```
#116 <webzero.sty identification> ≡
937 \NeedsTeXFormat{LaTeX2e}[1994/12/01]
938 \ProvidesPackage{webzero}[2007/04/10 v<version #106 (p.57)>
939   I fi package for web0 documents]
    (This code is used in #115 (p.63).)
```

#### 3.1.2 Package options

The `webzero` package recognizes six options. (Other users can add options for their own language.)

**3.1.2.1 The option `american`** is used when the document is written in American English. This is the default.

Note that some command names contain no “@”; these commands provide headings and may be modified by the user.

```
#117 <webzero.sty options> ≡
940 \DeclareOption{american}{%
941   \def \wzclassindexname{Classes}%
942   \def \wzfuncindexname{Functions}%
943   \def \wzlastsep{, and }%
944   \def \wzmetaindexname{Meta symbols}%
945   \def \wzmetaindexstartext{(Symbols marked with * are not used.)}%
946   \def \wzsep{, }%
947   \def \wztwosep{ and }%
948   \def \wzvarindexname{Variables}%
949   \def \wz@extendedname{extended in}%
950   \def \wz@filename{File}%
951   \def \wz@itisname{It is}%
952   \def \wz@notusedname{not used}%
953   \def \wz@pagename{page}%
954   \def \wz@shortpagename{p.}%
```

```

955 \def \wz@thiscodename{This code is}%
956 \def \wz@usedname{used in}}
  (This code is extended in #117a (p.64). It is used in #115 (p.63).)

```

**3.1.2.2 The option `english`** is used when the document is written in British English. At present, this is equivalent to using option `american`.

```

#117a <webzero.sty options #117 (p.63)> + ≡
957 \DeclareOption{english}{\ExecuteOptions{american}}
  (This code is extended in #117b (p.64).)

```

**3.1.2.3 The option `normalsize`** will produce program code in type size `\small` which normally looks best. The leading may be squeezed, however.

```

#117b <webzero.sty options #117 (p.63)> + ≡
958 \DeclareOption{normalsize}{\def \wz@codesize{\small}%
959 \def \wz@codestretch{0.9}}
  (This code is extended in #117c (p.64).)

```

**3.1.2.4 The option `norsk`** is used when the document is written in Norwegian “Bokmål”.

```

#117c <webzero.sty options #117 (p.63)> + ≡
960 \DeclareOption{norsk}{%
961 \def \wzclassindexname{Klasser}%
962 \def \wzfuncindexname{Funksjoner}%
963 \def \wzlastsep{ og }%
964 \def \wzmetaindexname{Metasymboler}%
965 \def \wzmetaindexstarttext{(Symboler merket med * er ikke brukt.)}%
966 \def \wzsep{, }%
967 \def \wztwosep{ og }%
968 \def \wzvarindexname{Variable}%
969 \def \wz@extendedname{utvidet i}%
970 \def \wz@filename{Fil}%
971 \def \wz@itisname{Den blir}%
972 \def \wz@notusedname{ikke brukt}%
973 \def \wz@pagename{side}%
974 \def \wz@shortpagename{s.}%
975 \def \wz@thiscodename{Denne koden blir}%
976 \def \wz@usedname{brukt i}}
  (This code is extended in #117d (p.64).)

```

**3.1.2.5 The option `nynorsk`** is used when the document is written in Norwegian “Nynorsk”.

```

#117d <webzero.sty options #117 (p.63)> + ≡
977 \DeclareOption{nynorsk}{\ExecuteOptions{norsk}%
978 \def \wzfuncindexname{Funksjonar}%
979 \def \wzmetaindexname{Metasymbol}%
980 \def \wzmetaindexstarttext{(Symboler merkte med * er ikkje nytta.)}%
981 \def \wz@extendedname{utvida i}%
982 \def \wz@itisname{Han vert}%
983 \def \wz@notusedname{ikkje nytta}%
984 \def \wz@thiscodename{Denne koden vert}%
985 \def \wz@usedname{nytta i}}
  (This code is extended in #117e (p.65).)

```

**3.1.2.6 The option `sf`** will use the standard `\sffamily` fonts when printing program code.

```
#117e <webzero.sty options #117 (p.63)> +≡
986 \DeclareOption{sf}{\def \wz@family{\sffamily}}
(This code is extended in #117f (p.65).)
```

**3.1.2.7 The option `small`** will produce program code in type size `\footnotesize` (which is smaller than the standard `\small`). In this case we must compensate by increasing the leading to maintain readability.

```
#117f <webzero.sty options #117 (p.63)> +≡
987 \DeclareOption{small}{\def \wz@codesize{\footnotesize}%
988 \def \wz@codestretch{0.95}}
(This code is extended in #117g (p.65).)
```

**3.1.2.8 The option `tt`** will use the standard `\ttfamily` fonts when printing program code. This is the default.

```
#117g <webzero.sty options #117 (p.63)> +≡
989 \DeclareOption{tt}{\def \wz@family{\ttfamily}}
(This code is extended in #117h (p.65).)
```

**3.1.2.9 Default options** are `american`, `normalsize` and `tt`.

```
#117h <webzero.sty options #117 (p.63)> +≡
990 \ExecuteOptions{american, normalsize, tt}
991 \ProcessOptions \relax
```

### 3.1.3 Package loading

The `calc` and `ifthen` packages are used in the `\wz@alpha` macro; see Section 3.1.7.2 on page 70.

```
#118 <webzero.sty package loading> ≡
992 \RequirePackage{calc,ifthen}
(This code is extended in #118a (p.65). It is used in #115 (p.63).)
```

The `relsize` package is used when typesetting meta symbol numbers; see Section 3.1.7.3 on page 70.

```
#118a <webzero.sty package loading #118 (p.65)> +≡
993 \RequirePackage{relsize}
(This code is extended in #118b (p.65).)
```

As mentioned in Section 2.8 on page 58, the `textcomp` package is required.

```
#118b <webzero.sty package loading #118 (p.65)> +≡
994 \RequirePackage{textcomp}
```

### 3.1.4 Implementation of interface

These command constitute the standard package interface.

**3.1.4.1 Meta symbol definition** is done using the command `\wzdef`. It takes three parameters:

1. the name of the meta symbol,
2. its number, and
3. its extension number.

Starting a new meta symbol definition involves the following:

- Add some vertical space.
- Select a suitable typeface.
- Modify the paragraph parameters. Note that the baseline distance is given with a little stretch and shrink. This is necessary to avoid messages about “Underfull \vbox” when we have pages completely filled with code.<sup>14</sup>
- Print the meta symbol number and its name, followed by a “ $\equiv$ ” or a “ $+ \equiv$ ”. The subsequent line change is preceded by a `\nobreak` to avoid widow lines.

The modifications are done inside a local group (`\beginngroup . . . \endngroup`); this makes it easier to revert to the original parameters afterwards.

```
#119 <webzero.sty main code> ≡
995 \newcommand{\wzdef}[3]{\par
996   \ifthenelse{\parskip>0}{\vspace{\parskip}}{\medskip}
997   \beginngroup
998   \renewcommand{\baselinestretch}{\wz@codestretch}%
999   \wz@codesize\wz@family
1000   \setlength{\parindent}{1em}
1001   \setlength{\parskip}{Opt plus 0.3pt minus 0.1pt}\frenchspacing
1002   \noindent
1003   \llap{\normalfont\bfseries {\wz@num{#2}{#3}}\hspace*{1em}}%
1004   \ifthenelse{#3=0}{\wzmeta{#1}~$\equiv$}%
1005   {\wzmeta{#2}{#1}~$+!\equiv$}\label{wO-#2-#3}%
1006   \wzeol[\nobreak]}
```

(This code is extended in #119<sub>a</sub> (p.67). It is used in #115 (p.63).)

**3.1.4.2 Meta symbol termination** is signaled by use of the `\wzenddef` command. It takes four parameters:

1. the meta symbol’s number,
2. its extension number,
3. value 1 if the meta symbol has an extension or 0 if it has not, and
4. information on its usage.

The termination involves the following actions:

- Print extension and/or usage information.
- Add some vertical space.

---

<sup>14</sup>The standard  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} 2_{\epsilon}$  document classes use a different scheme to avoid such messages: it ensures that the `\textheight` is equal to an integral number of `\baselineskips`. I prefer the stretch and shrink method; in my experience, it is more robust.

```
#119a <webzero.sty main code #119 (p.66)> +≡
1007 \newcommand{\wzenddate}[4]{%
1008 \ifthenelse{#2=0}{<webzero.sty: info on base definition #121 (p.67)>}
1009 {\<webzero.sty: info on extended definition #120 (p.67)>}\par
1010 \endgroup
1011 \ifthenelse{\parskip>0}{\medskip}
1012 <suppress indentation of subsequent paragraph #122 (p.67)>}
(This code is extended in #119b (p.67).)
```

The extensions give only information on further extensions (if any).

```
#120 <webzero.sty: info on extended definition> ≡
1013 \ifthenelse{#3=0}{
1014 {\wz@info{\wz@extendedname\ \wz@numandpage[#3]{#1}}
(This code is used in #119a (p.67).)
```

We provide both extension and usage information the first time a meta symbol is defined.

```
#121 <webzero.sty: info on base definition> ≡
1015 \wz@info{\ifthenelse{#3=0}{
1016 {\wz@extendedname\ \wz@numandpage[#3]{#1}. \wz@itisname\ }%
1017 \ifthenelse{\equal{#4}{}}{\wz@notusedname}
1018 {\wz@usedname\ #4}}
(This code is used in #119a (p.67).)
```

The `\wz@info` command defines the appearance of the information.

```
#119b <webzero.sty main code #119 (p.66)> +≡
1019 \newcommand{\wz@info}[1]{\*[0.2ex]
1020 \textsl{\rmfamily\footnotesize (\wz@thiscodename\ #1.)}}
(This code is extended in #119c (p.67).)
```

We want to suppress any indentation of the paragraph immediately following a meta symbol definition. The code to do this was found in Section A of the L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> source code.

```
#122 <suppress indentation of subsequent paragraph> ≡
1021 \everypar{\setbox0=\lastbox\everypar{}}
(This code is used in #119a (p.67).)
```

**3.1.4.3 Code line termination** is specified using the `\wzeol` command. It has an optional parameter which is inserted just before the new line is started; this parameter is used to suppress a page break before the first code line; see Section 3.1.4.1 on the facing page.

```
#119c <webzero.sty main code #119 (p.66)> +≡
1022 \newcommand{\wzeol}[1][\par #1\leavevmode
1023 \addtocounter{wz@lnum}{1}%
1024 \llap{\normalfont\tiny \thewz@lnum \hspace*{\parindent}}
(This code is extended in #119d (p.67).)
```

The line counter `wz@lnum` must be declared.

```
#119d <webzero.sty main code #119 (p.66)> +≡
1025 \newcounter{wz@lnum}
(This code is extended in #119e (p.68).)
```

**3.1.4.4 File name notification** using the command `\wzfile` occurs whenever a new source file is being read. The file name is saved for later inclusion in the page header.

**#119<sub>e</sub>** *<webzero.sty main code #119 (p.66)> +≡*  
`1026 \newcommand{\wzfile}[1]{\markright{\wz@filename: \textsl{#1}}}`  
*(This code is extended in #119<sub>f</sub> (p.68).)*

**3.1.4.5 Meta symbols** are typeset using the `\wzmeta` command. This command is used in meta symbol definitions, but the user may also employ it if he or she wishes.

The command has an optional parameter. If this is a non-zero number, the meta symbol's number and page where first defined will be included.

**#119<sub>f</sub>** *<webzero.sty main code #119 (p.66)> +≡*  
`1027 \newcommand{\wzmeta}[2][0]{$`  
`1028 \langle \mbox{\it #2\ifthenelse{#1=0}{\sim\smaller[2]\upshape`  
`1029 \wz@numandpage{#1}}\ / } \rangle $}`  
*(This code is extended in #119<sub>g</sub> (p.68).)*

### 3.1.5 Typesetting the index

The `wzindex` environment is used for typesetting the variable, function, and meta symbol indices. It has two parameters:

1. the name of the index (like “Variables”), and
2. the number of columns to use (1 or 2).

**#119<sub>g</sub>** *<webzero.sty main code #119 (p.66)> +≡*  
`1030 \newenvironment{wzindex}[2]%`  
`1031 {\ifnum #2=2 \twocolumn\section*{#1}\else \onecolumn \section*{#1}\fi`  
`1032 \markboth{\MakeUppercase{#1}}{\MakeUppercase{#1}}`  
`1033 \begingroup`  
`1034 \vspace*{4pt}`  
`1035 \setlength{\emergencystretch}{3cm}`  
`1036 \setlength{\parfillskip}{0pt}`  
`1037 \setlength{\parindent}{0pt}%`  
`1038 \setlength{\parskip}{1pt plus 1pt}`  
`1039 \small \sloppy \hbadness = \tolerance }%`  
`1040 {\onecolumn \endgroup }`  
*(This code is extended in #119<sub>h</sub> (p.68).)*

**3.1.5.1 The macro `\wzinitial`** This command is used whenever the index changes the initial letter. It has one parameter: the initial letter.

**#119<sub>h</sub>** *<webzero.sty main code #119 (p.66)> +≡*  
`1041 \newcommand{\wzinitial}[1]{\vspace{16pt plus 4pt}`  
`1042 {\raggedright \textbf{\large #1}\par}\vspace*{2pt plus 1pt minus 0.5pt}}`  
*(This code is extended in #119<sub>i</sub> (p.68).)*

**3.1.5.2 The macro `\wzul`** This macro `\wzul` is used to typeset an underlined line number (which signifies that the element was defined on that line).

**#119<sub>i</sub>** *<webzero.sty main code #119 (p.66)> +≡*  
`1043 \newcommand{\wzul}[1]{\underline{#1}}`  
*(This code is extended in #119<sub>j</sub> (p.69).)*

**3.1.5.3 The macro `\wzindexsep`** This macro defines the separator between successive line numbers. The default definition is a comma followed by a space with a lot of stretch. The comma is placed in an `\rlap` so that it will stick into the margin if it comes at the end of a line.

```
#119j <webzero.sty main code #119 (p.66)> +≡
1044 \newcommand{\wzindexsep}{\rlap{,}\hspace{0.5em plus 1em minus 0.1em}}
(This code is extended in #119k (p.69).)
```

**3.1.5.4 The macro `\wzx`** This macro typesets the name of the variable, function, or meta symbol being indexed presently. It is followed by a `\dotfill` to connect the name with the following line numbers.

```
#119k <webzero.sty main code #119 (p.66)> +≡
1045 \newcommand{\wzx}[1]{\setlength{\hangindent}{2em}%
1046 \hangafter=1 {\wz@family #1}\hspace*{0.5em}\dotfill}
(This code is extended in #119l (p.69).)
```

**3.1.5.5 The macro `\wzxref`** This macro is used when referencing other meta symbols.

```
#119l <webzero.sty main code #119 (p.66)> +≡
1047 \newcommand{\wzxref}[2]{\wz@numandpage[#2]{#1}}
(This code is extended in #119m (p.69).)
```

**3.1.5.6 The macro `\wzlongpageref`** This macro is used when typesetting the meta symbol index. Here we need to reference page numbers, like “page 123”. This definition provides space for page numbers with up to three digits.

```
#119m <webzero.sty main code #119 (p.66)> +≡
1048 \newcommand{\wzlongpageref}[1]{\wz@pagename~%
1049 \makebox[2em][r]{\pageref{wO-#1-O}}}
(This code is extended in #119n (p.69).)
```

### 3.1.6 Page style

A new page style is provided. The page style `webzero` places the file name and the page number in the footer; the header is left empty; an example is shown in Figures 3 and 4 on pages 6 and 7.

```
#119n <webzero.sty main code #119 (p.66)> +≡
1050 \newcommand{\ps@webzero}{%
1051 \renewcommand{\@evenhead}{\let \@oddhead = \@evenhead
1052 \renewcommand{\@evenfoot}{\rightmark\hfill
1053 \wz@pagename\space\thepage}%
1054 \let \@oddfont = \@evenfont
1055 \renewcommand{\sectionmark}[1]{}%
1056 \renewcommand{\subsectionmark}[1]{}%
(This code is extended in #119o (p.69).)
```

### 3.1.7 Utility macros

**3.1.7.1 The name `web0`** is generated by the macro `\webzero`. It uses the same trick as the definition of `\LaTeXe` in the L<sup>A</sup>T<sub>E</sub>X<sub>2<sub>ε</sub></sub> source code to decide when the subscript should be bold.

```
#119o <webzero.sty main code #119 (p.66)> +≡
1057 \DeclareRobustCommand{\webzero}{\mbox{\m@th
1058 \if b\expandafter\@car\f@series\@nil \boldmath \fi
1059 \textsf{web}$_{\mathsf{0}}$}}
(This code is extended in #119p (p.70).)
```

### 3.1.7.2 Extended alphabetical numbering

The extensions are numbered

a, b, ..., z, aa, ab, ..., az, ba, ...

(This is the standard numbering `\alph` extended to arbitrarily large numbers.)

This is implemented in the macro `\wz@alpha`. Note the use of the counter `\wz@val` and the extra grouping `\begingroup ... \endgroup`; this is necessary to preserve the parameter #1 (which is really `\thewz@temp`) across the recursive calls. (The use of `\wz@val` must be written in plain `TEX` as `\setcounter` has an implicit `\global`.)

```
#119p <webzero.sty main code #119 (p.66)> +≡
1060 \newcommand{\wz@alpha}[1]{%
1061   \ifthenelse{#1<27}
1062     {\setcounter{wz@temp}{#1}\alph{wz@temp}}
1063     {\begingroup
1064       \count\wz@val = #1 \relax
1065       \setcounter{wz@temp}{(#1-1)/26}\wz@alpha{\thewz@temp}%
1066       \setcounter{wz@temp}{\the\count\wz@val -
1067         (\the\count\wz@val-1)/26*26}\alph{wz@temp}%
1068       \endgroup }}
```

(This code is extended in #119<sub>q</sub> (p.70).)

The two counters must be declared.

```
#119q <webzero.sty main code #119 (p.66)> +≡
1069 \newcounter{wz@temp} \newcount\wz@val
```

(This code is extended in #119<sub>r</sub> (p.70).)

### 3.1.7.3 Typesetting a meta symbol number

A meta symbol number with its extension is typeset as

**#4<sub>c</sub>**

This is done by the macro `\wz@num` which has two parameters: the meta symbol number and the extension number. (If the extension number is 0, there will be no extension.)

```
#119r <webzero.sty main code #119 (p.66)> +≡
1070 \newcommand{\wz@num}[2]{\##1%
1071   \ifthenelse{#2>0}{\raisebox{-0.4ex}{\smaller[2]}%
1072     \hspace*{-0.01em}\wz@alpha{#2}}{}}
```

(This code is extended in #119<sub>s</sub> (p.70).)

### 3.1.7.4 Typesetting a meta symbol number with page reference

The `LATEX` command `\wz@numandpage` prints the number of a meta symbol (and its extension number if provided by an optional parameter) together with the number of the page on which it was defined:

**#4<sub>c</sub> (p.15)**

```
#119s <webzero.sty main code #119 (p.66)> +≡
1073 \newcommand{\wz@numandpage}[2][0]{\wz@num{#2}{#1}%
1074   \hspace*{0.2em}\wz@shortpagename\pageref{wO-#2-#1}}
```

(This code is extended in #119<sub>t</sub> (p.71).)

### 3.1.8 End of class

It is common in L<sup>A</sup>T<sub>E</sub>X to use a line with `\endinput` as the last line. That way it is easier to detect whether the file has been truncated for some reason.

```
#119t <webzero.sty main code #119 (p.66)> +≡
1075 \endinput
```

## 3.2 The webzero document class

Since most web<sub>0</sub> documents will describe a program and nothing more, a document class `webzero` is provided. This document class is based on the `article` document class and the `webzero` package.

```
#123 <webzero.cls> ≡
1076 <webzero.cls identification #124 (p.71)>
1077 <webzero.cls options #125 (p.71)>
1078 <webzero.cls package and class loading #126 (p.71)>
1079 <webzero.cls main code #127 (p.72)>
1080 <webzero.cls end #128 (p.72)>
(This code is not used.)
```

### 3.2.1 Class identification

```
#124 <webzero.cls identification> ≡
1081 \NeedsTeXFormat{LaTeX2e}[1994/12/01]
1082 \ProvidesClass{webzero}[1999/12/20 v1.0 Ifi class for web0 documents]
(This code is used in #123 (p.71).)
```

### 3.2.2 Class options

This document class will recognize the options known to the `webzero` package (see Section 3.1 on page 63) and send them on. All other options are passed on to the `article` class.

```
#125 <webzero.cls options> ≡
1083 \DeclareOption{american}{\PassOptionsToPackage{american}{webzero}}
1084 \DeclareOption{english}{\PassOptionsToPackage{english}{webzero}}
1085 \DeclareOption{norsk}{\PassOptionsToPackage{norsk}{webzero}}
1086 \DeclareOption{nynorsk}{\PassOptionsToPackage{nynorsk}{webzero}}
1087 \DeclareOption{sf}{\PassOptionsToPackage{sf}{webzero}}
1088 \DeclareOption{tt}{\PassOptionsToPackage{tt}{webzero}}
1089 \DeclareOption*{\PassOptionsToClass{\CurrentOption}{article}}
1090 \ProcessOptions \relax
(This code is used in #123 (p.71).)
```

### 3.2.3 Package and class loading

As mentioned above, the `webzero` document class is based on the `article` document class and the `webzero` package.

```
#126 <webzero.cls package and class loading> ≡
1091 \LoadClass{article}
1092 \RequirePackage{webzero}
(This code is used in #123 (p.71).)
```

### 3.2.4 Main code

As most of the document will be program text, longer lines are useful:

```
#127 ⟨webzero.cls main code⟩ ≡
1093 \addtolength{\textwidth}{3cm}
1094 \addtolength{\evensidemargin}{-1.5cm}
1095 \addtolength{\oddsidemargin}{-1.5cm}
(This code is extended in #127a (p.72). It is used in #123 (p.71).)
```

..., as are taller pages, particularly because there are no page headers in the webzero page style:

```
#127a ⟨webzero.cls main code #127 (p.72)⟩ +≡
1096 \addtolength{\topmargin}{-2.8cm}
1097 \addtolength{\textheight}{4.5cm}
(This code is extended in #127b (p.72).)
```

We want to use the webzero page style.

```
#127b ⟨webzero.cls main code #127 (p.72)⟩ +≡
1098 \pagestyle{webzero}
(This code is extended in #127c (p.72).)
```

Since `\maketitle` issues a call on `\thispagestyle{plain}`, we must also redefine that page style.

```
#127c ⟨webzero.cls main code #127 (p.72)⟩ +≡
1099 \let \ps@plain = \ps@webzero
```

### 3.2.5 End of class

And that's all, folks.

```
#128 ⟨webzero.cls end⟩ ≡
1100 \endinput
(This code is used in #123 (p.71).)
```

## 4 Documentation

Since the `web0` system is likely to be used in a UNIX environment, some users will appreciate manual pages for the `tangle0` and `weave0` programs.

### 4.1 Man page for `tangle0`

The man page consists of the standard head line and the usual parts.

```
#129 <man tangle0> ≡
1101 .TH TANGLE0 1 "<man page date #130 (p.73)>"
1102 <man tangle0 name #131 (p.73)>
1103 <man tangle0 description #132 (p.73)>
1104 <man tangle0 parameters #133 (p.74)>
1105 <man author #138 (p.75)>
1106 <man see also #139 (p.75)>
    (This code is not used.)
```

The date identifies the current version.

```
#130 <man page date> ≡
1107 5 April 2006
    (This code is used in #129 (p.73) and #134 (p.74).)
```

#### 4.1.1 Identification

This specification gives the name of the program and a single-line description of what it does.

```
#131 <man tangle0 name> ≡
1108 .SH NAME
1109 tangle0 - a web0 tool for extracting program code
    (This code is used in #129 (p.73).)
```

#### 4.1.2 Program description

This specification first gives the program name and a list of its parameters:

```
#132 <man tangle0 description> ≡
1110 .SH SYNOPSIS
1111 .B tangle0
1112 .RI [-o " file" ]
1113 [-v]
1114 .RI [-x " name" ]
1115 .RI [ file... ]
    (This code is extended in #132a (p.73). It is used in #129 (p.73).)
```

Then comes a longer description of what the program does.

```
#132a <man tangle0 description #132 (p.73)> + ≡
1116 .SH DESCRIPTION
1117 .I Tangle0
1118 is part of the
1119 .B web0
1120 package. It is used to extract the program code from a
1121 .B web0
1122 source.
```

### 4.1.3 The parameters

This part of the man page lists the parameters and describes their use. The individual parameter is described together with the code implementing it.

```
#133 <man tangle0 parameters> ≡
1123 .SS OPTIONS
1124 .I TangleO
1125 accepts the following options:
1126 <tangle0 man page parameters #5 (p.14)>
    (This code is used in #129 (p.73).)
```

## 4.2 Man page for `weave0`

The man page consists of the standard head line and the usual parts.

```
#134 <man weave0> ≡
1127 .TH WEAVEO 1 "<man page date #130 (p.73)>"
1128 <man weave0 name #135 (p.74)>
1129 <man weave0 description #136 (p.74)>
1130 <man weave0 parameters #137 (p.75)>
1131 <man author #138 (p.75)>
1132 <man see also #139 (p.75)>
    (This code is not used.)
```

### 4.2.1 Identification

This specification gives the name of the program and a single-line description of what it does.

```
#135 <man weave0 name> ≡
1133 .SH NAME
1134 weave0 - a web0 tool for producing program documentation
    (This code is used in #134 (p.74).)
```

### 4.2.2 Program description

This specification first gives the program name and a list of its parameters:

```
#136 <man weave0 description> ≡
1135 .SH SYNOPSIS
1136 .B weave0
1137 .RI "[-e] [-f " filter ]
1138 .RI [-l " language" ]
1139 .RI [-o " file" ]
1140 [-v]
1141 .RI [ file... ]
    (This code is extended in #136a (p.74). It is used in #134 (p.74).)
```

Then comes a longer description of what the program does.

```
#136a <man weave0 description #136 (p.74)> + ≡
1142 .SH DESCRIPTION
1143 .I WeaveO
1144 is part of the
1145 .B web0
1146 package. It is used to produce the program documentation from a
1147 .B web0
1148 source.
```

### 4.2.3 The parameters

This part of the man page lists the parameters and describes their use. The individual parameter is described together with the code implementing it.

```
#137 <man weave0 parameters> ≡  
1149 .SS OPTIONS  
1150 .I Weave0  
1151 accepts the following options:  
1152 <weave0 man page parameters #12 (p.18)>  
(This code is used in #134 (p.74).)
```

## 4.3 Common man page information

Some man page information is the same for the two programs; it is defined here.

### 4.3.1 The name of the author

This information includes the name and address of the program's author.

```
#138 <man author> ≡  
1153 .SH AUTHOR  
1154 Dag Langmyhr, Department of Informatics, University of Oslo.  
(This code is used in #129 (p.73) and #134 (p.74).)
```

### 4.3.2 Cross reference information

Those who read this manual page will quite probably be interested in the complete documentation on the web<sub>0</sub> system.

```
#139 <man see also> ≡  
1155 .SH "SEE ALSO"  
1156 .I The Web0 System  
1157 by Dag Langmyhr; available on  
1158 .I http://www.ifl.uio.no/~drift/doc/programs/web0.pdf.  
(This code is used in #129 (p.73) and #134 (p.74).)
```



## References

- [Knu83] Donald E. Knuth. Literate programming. Technical Report STAN-CS-82-981, Stanford University, Stanford, CA 94305, September 1983.
- [Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
- [Knu92] Donald E. Knuth. «*Literate programming*». Center for the study of language and information, 1992.
- [Lev87] Silvio Levy. WEB adapted to C. *TUGboat*, 8(1):12–13, April 1987.
- [Ram89] Norman Ramsay. Literate programming: weaving a language-independent WEB. *Communications of the ACM*, 32(9):1051–1055, September 1989.
- [WCS96] Larry Wall, Tom Christiansen, and Randal L. Swarz. *Programming Perl*. O'Reilly & associates, second edition, 1996. Known as “The camel book”.

## Functions

### A

&alphabetically ..... 484, 920, 932

### E

&Error ..... 256, 280, 296, 310, 340, 890

&Expand ..... 281, 283, 309

### F

&Find\_meta\_sy ..... 189, 195, 204

&Format\_usage ..... 443, 446

### G

&Generate\_index .... 466, 467, 511, 512

### I

&indexwise ..... 480

### L

&Latexify ..... 409, 411,  
413, 416, 425, 439, 486, 508, 516, 861

### M

&Message ..... 47, 48, 94, 112, 142, 145,  
174, 232, 265, 313, 345, 524, 891, 895

### T

&Tidy\_up ..... 892

### U

&Usage ..... 19, 38, 55, 89, 107, 122, 152

### W

&Warning ..... 212,  
224, 227, 231, 291, 312, 377, 523

## Meta symbols

<i>&lt;alphabetical sorting #111&gt;</i> .....	page	61
<i>&lt;alphabetical sorting: simple first tests #112&gt;</i> .....	page	62
<i>&lt;alphabetical sorting: test initial character #113&gt;</i> .....	page	62
<i>&lt;alphabetical sorting: test other characters #114&gt;</i> .....	page	62
<i>&lt;expand TAB characters #109&gt;</i> .....	page	60
<i>&lt;find_meta_sy: handle abbreviated meta symbol name #31&gt;</i> .....	page	27
<i>&lt;find_meta_sy: handle reference to last meta symbol #30&gt;</i> .....	page	26
<i>&lt;Latexify: adapt text #108&gt;</i> .....	page	58
<i>&lt;latex generation functions #107&gt;</i> .....	page	58
<i>&lt;man author #138&gt;</i> .....	page	75
<i>&lt;man page date #130&gt;</i> .....	page	73
<i>&lt;man see also #139&gt;</i> .....	page	75
<i>&lt;man tangle0 #129&gt;</i> .....	page	73 *
<i>&lt;man tangle0 description #132&gt;</i> .....	page	73
<i>&lt;man tangle0 name #131&gt;</i> .....	page	73
<i>&lt;man tangle0 parameters #133&gt;</i> .....	page	74
<i>&lt;man weave0 #134&gt;</i> .....	page	74 *
<i>&lt;man weave0 description #136&gt;</i> .....	page	74
<i>&lt;man weave0 name #135&gt;</i> .....	page	74
<i>&lt;man weave0 parameters #137&gt;</i> .....	page	75
<i>&lt;perl #105&gt;</i> .....	page	57
<i>&lt;set default parameter values #14&gt;</i> .....	page	18
<i>&lt;suppress indentation of subsequent paragraph #122&gt;</i> .....	page	67
<i>&lt;tangle0 #1&gt;</i> .....	page	13 *
<i>&lt;tangle0 auxiliary functions #7&gt;</i> .....	page	16
<i>&lt;tangle0 definitions #2&gt;</i> .....	page	13
<i>&lt;tangle0 man page parameters #5&gt;</i> .....	page	14
<i>&lt;tangle0 parameter decoding #3&gt;</i> .....	page	14
<i>&lt;tangle0 parameters #4&gt;</i> .....	page	14
<i>&lt;tangle0 processing #6&gt;</i> .....	page	16
<i>&lt;user message functions #110&gt;</i> .....	page	60
<i>&lt;version #106&gt;</i> .....	page	57
<i>&lt;w0-f-latex #46&gt;</i> .....	page	32 *
<i>&lt;w0-f-latex: check for range of line numbers #66&gt;</i> .....	page	42
<i>&lt;w0-f-latex: end code #60&gt;</i> .....	page	40
<i>&lt;w0-f-latex: examine options #49&gt;</i> .....	page	34
<i>&lt;w0-f-latex: format and print index line numbers #65&gt;</i> .....	page	42
<i>&lt;w0-f-latex: generate an index entry #64&gt;</i> .....	page	42
<i>&lt;w0-f-latex: generate indices #62&gt;</i> .....	page	41
<i>&lt;w0-f-latex: generate the class index #70&gt;</i> .....	page	43
<i>&lt;w0-f-latex: generate the function index #63&gt;</i> .....	page	41
<i>&lt;w0-f-latex: generate the meta symbol index #71&gt;</i> .....	page	43
<i>&lt;w0-f-latex: generate the variable index #69&gt;</i> .....	page	43
<i>&lt;w0-f-latex: initialization #47&gt;</i> .....	page	32
<i>&lt;w0-f-latex: make LaTeX code #51&gt;</i> .....	page	35
<i>&lt;w0-f-latex: note output file #50&gt;</i> .....	page	34
<i>&lt;w0-f-latex: option handling #48&gt;</i> .....	page	34
<i>&lt;w0-f-latex: pass1: note meta symbol definition #52&gt;</i> .....	page	35
<i>&lt;w0-f-latex: pass2: handle tokens #53&gt;</i> .....	page	36
<i>&lt;w0-f-latex: pass3: handle 'code' tokens #54&gt;</i> .....	page	38

<i>&lt;w0-f-latex: pass3: handle 'def' tokens #55&gt;</i> .....	page	38
<i>&lt;w0-f-latex: pass3: handle 'file' tokens #56&gt;</i> .....	page	39
<i>&lt;w0-f-latex: pass3: handle 'nl' tokens #57&gt;</i> .....	page	39
<i>&lt;w0-f-latex: pass3: handle 'text' tokens #58&gt;</i> .....	page	39
<i>&lt;w0-f-latex: pass3: handle 'use' tokens #59&gt;</i> .....	page	39
<i>&lt;w0-f-latex: print index line number #67&gt;</i> .....	page	42
<i>&lt;w0-f-latex: produce an initial (if required) #68&gt;</i> .....	page	42
<i>&lt;w0-f-latex: utility functions #61&gt;</i> .....	page	40
<i>&lt;w0-l-c #72&gt;</i> .....	page	44 *
<i>&lt;w0-l-c check alphabetic symbol #78&gt;</i> .....	page	46
<i>&lt;w0-l-c check C code for functions and variables #76&gt;</i> .....	page	45
<i>&lt;w0-l-c check for names #77&gt;</i> .....	page	46
<i>&lt;w0-l-c definitions #73&gt;</i> .....	page	44
<i>&lt;w0-l-c enhance C code #79&gt;</i> .....	page	47
<i>&lt;w0-l-c look for special words #80&gt;</i> .....	page	47
<i>&lt;w0-l-c parameter handling #74&gt;</i> .....	page	45
<i>&lt;w0-l-c read C code #75&gt;</i> .....	page	45
<i>&lt;w0-l-java #81&gt;</i> .....	page	48 *
<i>&lt;w0-l-java check alphabetic symbol #87&gt;</i> .....	page	50
<i>&lt;w0-l-java check for names #86&gt;</i> .....	page	49
<i>&lt;w0-l-java check formal parameter list #88&gt;</i> .....	page	51
<i>&lt;w0-l-java check Java code for names #85&gt;</i> .....	page	49
<i>&lt;w0-l-java definitions #82&gt;</i> .....	page	48
<i>&lt;w0-l-java enhance Java code #89&gt;</i> .....	page	51
<i>&lt;w0-l-java look for special words or symbols #90&gt;</i> .....	page	51
<i>&lt;w0-l-java parameter handling #83&gt;</i> .....	page	49
<i>&lt;w0-l-java read Java code #84&gt;</i> .....	page	49
<i>&lt;w0-l-latex #91&gt;</i> .....	page	52 *
<i>&lt;w0-l-latex check for comments #96&gt;</i> .....	page	54
<i>&lt;w0-l-latex check for declarations #97&gt;</i> .....	page	54
<i>&lt;w0-l-latex check for use #98&gt;</i> .....	page	54
<i>&lt;w0-l-latex check LaTeX code #95&gt;</i> .....	page	53
<i>&lt;w0-l-latex definitions #92&gt;</i> .....	page	53
<i>&lt;w0-l-latex parameter handling #93&gt;</i> .....	page	53
<i>&lt;w0-l-latex read LaTeX code #94&gt;</i> .....	page	53
<i>&lt;w0-l-perl #99&gt;</i> .....	page	55 *
<i>&lt;w0-l-perl check Perl code for functions and variables #103&gt;</i> .....	page	56
<i>&lt;w0-l-perl definitions #100&gt;</i> .....	page	55
<i>&lt;w0-l-perl enhance Perl code #104&gt;</i> .....	page	57
<i>&lt;w0-l-perl parameter handling #101&gt;</i> .....	page	55
<i>&lt;w0-l-perl read Perl code #102&gt;</i> .....	page	56
<i>&lt;w0code #32&gt;</i> .....	page	27 *
<i>&lt;w0code add to meta symbol body #39&gt;</i> .....	page	30
<i>&lt;w0code definitions #33&gt;</i> .....	page	28
<i>&lt;w0code expand meta symbols #40&gt;</i> .....	page	30
<i>&lt;w0code expand: check for definition cycles #43&gt;</i> .....	page	31
<i>&lt;w0code expand: check that meta symbol is defined #42&gt;</i> .....	page	30
<i>&lt;w0code expand: deactivate current meta symbol #44&gt;</i> .....	page	31
<i>&lt;w0code expand: expand the meta symbol body #45&gt;</i> .....	page	31
<i>&lt;w0code meta symbol definition #38&gt;</i> .....	page	29
<i>&lt;w0code parameter handling #34&gt;</i> .....	page	28
<i>&lt;w0code parameters #35&gt;</i> .....	page	28

<code>&lt;w0code read tokens #37&gt;</code> .....	page	29
<code>&lt;w0code utility functions #41&gt;</code> .....	page	30
<code>&lt;w0code: note output file #36&gt;</code> .....	page	28
<code>&lt;w0pre #18&gt;</code> .....	page	21 *
<code>&lt;w0pre check for end of meta symbol definition #26&gt;</code> .....	page	25
<code>&lt;w0pre check for start of meta symbol definition #24&gt;</code> .....	page	25
<code>&lt;w0pre check input file #28&gt;</code> .....	page	26
<code>&lt;w0pre check one line of a meta symbol definition #25&gt;</code> .....	page	25
<code>&lt;w0pre definitions #19&gt;</code> .....	page	24
<code>&lt;w0pre handle text line #27&gt;</code> .....	page	26
<code>&lt;w0pre initialization #20&gt;</code> .....	page	24
<code>&lt;w0pre parameter handling #21&gt;</code> .....	page	24
<code>&lt;w0pre parameters #22&gt;</code> .....	page	24
<code>&lt;w0pre token recognition #23&gt;</code> .....	page	25
<code>&lt;w0pre utility functions #29&gt;</code> .....	page	26
<code>&lt;weave0 #8&gt;</code> .....	page	16 *
<code>&lt;weave0 auxiliary functions #17&gt;</code> .....	page	21
<code>&lt;weave0 definitions #9&gt;</code> .....	page	17
<code>&lt;weave0 man page parameters #12&gt;</code> .....	page	18
<code>&lt;weave0 parameter decoding #10&gt;</code> .....	page	17
<code>&lt;weave0 parameters #11&gt;</code> .....	page	18
<code>&lt;weave0 processing #16&gt;</code> .....	page	20
<code>&lt;weave0: note filter #13&gt;</code> .....	page	18
<code>&lt;weave0: note language #15&gt;</code> .....	page	19
<code>&lt;webzero.cls #123&gt;</code> .....	page	71 *
<code>&lt;webzero.cls end #128&gt;</code> .....	page	72
<code>&lt;webzero.cls identification #124&gt;</code> .....	page	71
<code>&lt;webzero.cls main code #127&gt;</code> .....	page	72
<code>&lt;webzero.cls options #125&gt;</code> .....	page	71
<code>&lt;webzero.cls package and class loading #126&gt;</code> .....	page	71
<code>&lt;webzero.sty #115&gt;</code> .....	page	63 *
<code>&lt;webzero.sty identification #116&gt;</code> .....	page	63
<code>&lt;webzero.sty main code #119&gt;</code> .....	page	66
<code>&lt;webzero.sty options #117&gt;</code> .....	page	63
<code>&lt;webzero.sty package loading #118&gt;</code> .....	page	65
<code>&lt;webzero.sty: info on base definition #121&gt;</code> .....	page	67
<code>&lt;webzero.sty: info on extended definition #120&gt;</code> .....	page	67

(Symbols marked with \* are not used.)