UiO **: Department of Informatics**
University of Oslo

The web_0 system

Program source code and documentation

Dag Langmyhr
4th November 2020

T he web_0 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 TeX 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_0 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_0 is a much simpler tool, but perhaps more general. A comparison between web and web_0 is given in Section 1.3 on page 10.

1 Using web_0

This chapter tells the reader how to write documented programs in the web_0 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 web_0 notation.

The printed documentation is produced by executing

```
weave0 -l c -e -o bubble.tex bubble.w0
latex¹ bubble.tex
```

and the result is shown in Figures 3–6. The last two pages show the tables that web_0 can generate on request ("\wzvarindex" and "\wzmacroindex").

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.[2]

1.1 The documentation

Any part of the web_0 source that is not code (see Section 1.2 on page 10) is treated as documentation. This documentation is written exactly as one would write any other document for the chosen text processor.

When using LaTeX, however, the user should remember the following:

- The documentation must either use the document class webzero (see Section 3.2 on page 66) or the package webzero (see Section 3.1 on page 58).

---

[1]It may be necessary to run latex several times to get all cross-references correct.

[2]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 *web₀* file bubble.w0, page 1

```
\documentclass[12pt,a4paper]{webzero}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc,url}   \urlstyle{sf}
\usepackage{amssymb,mathpazo}

\newcommand{\p}[1]{\textsf{#1}}

\title{Bubble sort}
\author{Dag Langmyhr\\ Department of Informatics\\
  University of Oslo\\[5pt] \url{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 \p{bubble} in C which sorts an array \p{a}
with \p{n} elements. In other words, the array \p{a} should satisfy
the following condition when \p{bubble} exits:
\[
  \forall i, j \in \mathbb{N}:
    0 \leq i < j < \mathtt{n} \Rightarrow \mathtt{a}[i] \leq \mathtt{a}[j]
\]


<<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>>);
@
```

```
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 \p{bubble} by reducing the range of the
  \p{for}-loop by~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 \p{for}-loop needs an index variable \p{i}:

<<local var...>>=
int i;
@

Swapping two array elements is done in the standard way using an
auxiliary variable \p{temp}. We also increment a swap counter named
\p{n\_swaps}.

<<swap ...>>=
temp = a[i];  a[i] = a[i+1];  a[i+1] = temp;
++n_swaps;
@

The variables \p{temp} and \p{n\_swaps} must also be declared:

<<local var...>>=
int temp, n_swaps;
@

The variable \p{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  \wzmacroindex
\end{document}
```

Bubble sort

Dag Langmyhr
Department of Informatics
University of Oslo

dag@ifi.uio.no

November 4, 2020

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:

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

#1 ⟨*bubble sort*⟩ ≡
```
1  void bubble(int a[], int n)
2  {
3      ⟨local variables  #4 (p.2)⟩
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*⟩ ≡
```
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;[1] if the two are in the wrong sorting order, they are swapped:

#3 ⟨*perform bubbling*⟩ ≡
```
10  ⟨initialize  #6 (p.2)⟩
11  for (i=0; i<n-1; ++i)
12      if (a[i]>a[i+1]) {
13          ⟨swap a[i] and a[i+1]  #5 (p.2)⟩
14  }
```
(This code is used in #2 (p.1).)

The for-loop needs an index variable i:

---

[1]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.

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

#4 ⟨*local variables*⟩ ≡
15 `int i;`
(*This code is extended in #4ₐ (p.2). It is used in #1 (p.1).*)

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]*⟩ ≡
16 `temp = a[i];  a[i] = a[i+1];  a[i+1] = temp;`
17 `++n_swaps;`
(*This code is used in #3 (p.1).*)

The variables temp and n_swaps must also be declared:

#4$_a$ ⟨*local variables* #4 (p.2)⟩ + ≡
18 `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*⟩ ≡
19 `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*⟩ ≡
20 `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

Variables

A

I

N

T

Figure 6: The documentation created from `bubble.w0`, page 4

Macro names

(Macro names marked with * are not used internally.)

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

```c
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);
}
```

1.2 The code

Writing code in the web_0 system consists of defining a lot of *macros*, like

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

  <<statements>>
end
@
```

This definition states that the macro ⟨*main program*⟩ is defined to expand to the given text, which may include other macros defined elsewhere. The user may extract any part of code that he or she wants by naming the starting macro name.

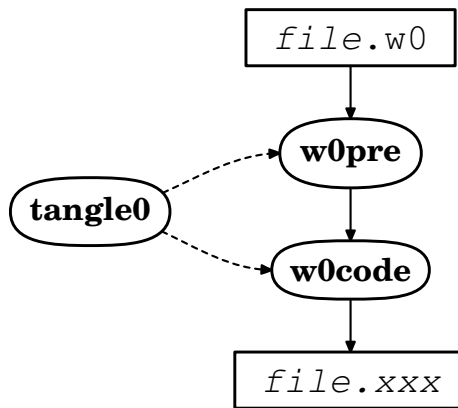When writing web_0 code, the following points should be considered:

- The notation for a macro name is "`<<macro name>>`". A macro name may contain any character except "`<`" or "`>`".

- A macro definition continues until a line starting with a "`@`" and containing nothing else (except possibly spaces).

- The notation "`<<>>`" is shorthand for the name of the last macro defined.

- The notation "`<<xxx...>>`" (ending with three dots) is shorthand for any macro whose name starts with "*xxx*" and has previously been defined or referenced. Only one macro name 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 macro name).[3]

1.3 Comparison to web

As mentioned in the preface, web_0 is based on the ideas of Donald Knuth's web program, but there are quite a few differences:

---

[3]Defining a macro for `<<` may be useful in such circumstances.

Figure 8: The `tangle0` program

```
                    ┌──────────────┐
                    │   file.w0    │
                    └──────────────┘
                            │
                            ▼
                        ╭────────╮
              ┌ ─ ─ ─ ─►│ w0pre  │
    ╭──────────╮         ╰────────╯
    │ tangle0  │             │
    ╰──────────╯             ▼
              └ ─ ─ ─ ─►╭────────╮
                        │ w0code │
                        ╰────────╯
                            │
                            ▼
                    ┌──────────────┐
                    │   file.xxx   │
                    └──────────────┘
```

- web supports the Pascal programming language; web_0 is language independent.

- A web file can contain only one program; web_0 files may contain several, and these programs may share code.

- A web program must be on one file (plus an optional change file); web_0 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 web_0 (`tangle0` and `weave0` with auxiliary programs) contain only 51 and 94 lines of "tangled" Perl code.

- web knows 36 commands; web_0 knows only one.

- In web, the programmer can insert TEX text into the code parts; this is not possible in web_0.

- web_0 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 web_0, but the ordinary LATEX mechanisms should be sufficient.

2 Implementation

The web_0 system consists of two programs `tangle0` and `weave0`, but it is implemented as several 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 LATEX is required.

2.1 The `tangle0` program

The `tangle0` program reads a set of web_0 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 `w0pre` and the postprocessor `w0code`, as shown in figure 8.

  The `tangle0` program is written in Perl:

#1 ⟨*tangle0*⟩ ≡
  1 *#!* ⟨*perl* #105 (p.54)⟩

  2

  3 ⟨*tangle0 definitions* #2 (p.12)⟩
  4 ⟨*tangle0 parameter decoding* #3 (p.12)⟩
  5 ⟨*tangle0 processing* #6 (p.14)⟩

  6

  7 ⟨*tangle0 auxiliary functions* #7 (p.14)⟩
  8 ⟨*user message functions* #112 (p.57)⟩
  (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, $Version) = ("tangle0", "⟨*version* #107 (p.54)⟩");
  (This code is extended in #2$_a$ (p.12). It is used in #1 (p.12).)

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) which makes it easy to test the code, but in the production version this will be modified by the "`make install`" command specified in the `Makefile`.

#2$_a$ ⟨*tangle0 definitions* #2 (p.12)⟩ +≡
  10 my $Lib_dir = ".";
  (This code is extended in #2$_b$ (p.12).)

2.1.2 Parameter decoding

This loop looks at all the parameters. They are decoded and put in `@Pre_opt` if they go to w0pre, and in `@Code_opt` if they are for w0code.

#2$_b$ ⟨*tangle0 definitions* #2 (p.12)⟩ +≡
  11 my (@Code_opt, @Pre_opt);
  (This code is extended in #2$_c$ (p.12).)

File names are kept in `@Pre_files`.

#2$_c$ ⟨*tangle0 definitions* #2 (p.12)⟩ +≡
  12 my @Pre_files = ();
  (This code is extended in #2$_d$ (p.13).)

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.13)⟩

  16

  17   push @Pre_files, "'$_'";
  18 }
  (This code is used in #1 (p.12).)

2.1.2.1 The -o option   is used to specify the name of the output file. This option is passed on to w0code.

#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.13). It is used in #3 (p.12).)

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.13). It is used in #135 (p.68).)

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.13)⟩ +≡
```
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 #4ᵦ (p.13).)

The default is to be silent.

#2_d   ⟨*tangle0 definitions* #2 (p.12)⟩ +≡
```
32   my $Verbose = 0;
```

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

#5ₐ   ⟨*tangle0 man page parameters* #5 (p.13)⟩ +≡
```
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 #5ᵦ (p.13).)

2.1.2.3 The -x option   names the name of the macro with which to start the program extraction. It is passed on to w0code.

#4ᵦ   ⟨*tangle0 parameters* #4 (p.13)⟩ +≡
```
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 #4_c (p.14).)

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

#5ᵦ   ⟨*tangle0 man page parameters* #5 (p.13)⟩ +≡
```
42   .TP
43   .B -x \fIname\fP
44   Specify the name of the macro with which to start the program
45   extraction. If this option is not used, extraction will start with the
46   first macro defined.
```

2.1.2.4 Unknown options   All other parameters result in a warning.

#4c ⟨*tangle0 parameters* #4 (p.13)⟩ + ≡
```
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("perl w0pre ...  | perl w0code ...");
```

but that would have made it impossible to detect run-time errors in either program.[4] To be able to detect all errors, tangle0 must start w0pre as an input process and w0code as an output process and pass all data from one to the other. When the w0pre process is finished, it is possible to close the two processes and detect an error exit status.

#6 ⟨*tangle0 processing*⟩ ≡
```
48  my $Pre_cmd = "$Lib_dir/w0pre @Pre_opt @Pre_files";
49  my $Post_cmd = "$Lib_dir/w0code @Code_opt";
50
51  &Message("Running $Pre_cmd | $Post_cmd") if $Verbose;
52  open(PRE, "$Pre_cmd |");
53  open(POST, "| $Post_cmd");
54  print POST while <PRE>;
55  close PRE;  exit $?>>8 if $?>>8;
56  close POST;  exit $?>>8;
```
(This code is used in #1 (p.12).)

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*⟩ ≡
```
57  sub Usage {
58      print STDERR "Usage: $Prog [-o file] [-v] [-x name] [file...]\n";
59      exit 1;
60  }
```
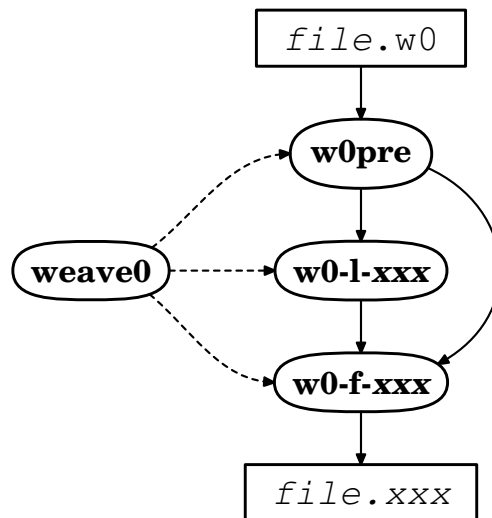*(This code is used in #1 (p.12).)*

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:

w0pre is the same preprocessor as is used by tangle0.

w0-l-*xxx* is an optional language dependant filter, such as w0-l-c for C programs and w0-l-perl for programs in Perl.

w0-f-*xxx* is the postprocessor producing the actual documentation commands. At present, only one such postprocessor is supplied: w0-f-latex that produces LaTeX code.

Figure 9: The weave0 program

The weave0 program is also written in Perl:

#8 ⟨*weave0*⟩ ≡
61   *#!* ⟨*perl* #105 (p.54)⟩
62
63   ⟨*weave0 definitions* #9 (p.15)⟩
64   ⟨*weave0 parameter decoding* #10 (p.15)⟩
65   ⟨*weave0 processing* #16 (p.18)⟩
66
67   ⟨*weave0 auxiliary functions* #17 (p.19)⟩
68   ⟨*user message functions* #112 (p.57)⟩
(This code is not used.)

2.2.1 Definitions

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

#9 ⟨*weave0 definitions*⟩ ≡
69   my ($Prog, $Version) = ("weave0", "⟨*version* #107 (p.54)⟩");
(This code is extended in #9$_a$ (p.15). It is used in #8 (p.15).)

Since weave0 is to start two or three auxiliary processes, it must know where to find them. For more information, see Section 2.1.1 on page 12

#9$_a$ ⟨*weave0 definitions* #9 (p.15)⟩ +≡
70   my $Lib_dir = ".";
(This code is extended in #9$_b$ (p.16).)

2.2.2 Parameter decoding

The following loop will examine all the parameters:

#10 ⟨*weave0 parameter decoding*⟩ ≡
71   ⟨*set default parameter values* #14 (p.17)⟩
72   PARAM:
73   **while** (@ARGV) { $_ = shift;
74   ⟨*weave0 parameters* #11 (p.16)⟩

---

[4]The documentation in "man sh" states that

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

```
75
76        push @Pre_files, "'$_'";
77   }
```
(This code is used in #8 (p.15).)

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*.

#9$_b$ ⟨*weave0 definitions* #9 (p.15)⟩ +≡
```
78   my (@Code_opt, @Lang_opt, @Pre_opt);
```
(This code is extended in #9$_c$ (p.16).)

File names are kept in @Pre_files.

#9$_c$ ⟨*weave0 definitions* #9 (p.15)⟩ +≡
```
79   my @Pre_files = ();
```
(This code is extended in #9$_d$ (p.17).)

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 next page.)

#11 ⟨*weave0 parameters*⟩ ≡
```
80   /^-e$/ and do { push @Lang_opt, "-e";  next PARAM; };
```
(This code is extended in #11$_a$ (p.16). It is used in #10 (p.15).)

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

#12 ⟨*weave0 man page parameters*⟩ ≡
```
81   .TP
82   .B -e
83   Enhance the code by using bold and italic fonts.
84   (Works only in conjunction with the
85   .B -l
86   option; see below.)
```
(This code is extended in #12$_a$ (p.17). It is used in #139 (p.69).)

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

#11$_a$ ⟨*weave0 parameters* #11 (p.16)⟩ +≡
```
87   /^-f$/ and do { $_ = shift;
88       ⟨weave0: note filter #13 (p.16)⟩;  next PARAM; };
89   /^-f(.+)/ and do { $_ = $1;
90       ⟨weave0: note filter #13 (p.16)⟩;  next PARAM; };
```
(This code is extended in #11$_b$ (p.17).)

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*⟩ ≡
```
91   &Usage unless $_;
92   my $pr = "$Lib_dir/w0-f-$_";
93   if (-r $pr) {
94       $F_prog = $pr;
95   } else {
96       &Message("Filter $_ is unknown;", "use of -f option ignored.");
97   }
```
(This code is used in #11$_a$ (p.16) and #14 (p.17).)

The variable $F_prog must be declared.

#9$_d$ ⟨*weave0 definitions* #9 (p.15)⟩ +=

```
 98  my $F_prog = "";
```

(This code is extended in #9$_e$ (p.17).)

The default output format is LATEX:

#14 ⟨*set default parameter values*⟩ ≡

```
 99  $_ = "latex";    ⟨weave0: note filter  #13 (p.16)⟩;
```

(This code is used in #10 (p.15).)

The man page contains documentation on this option:

#12$_a$ ⟨*weave0 man page parameters* #12 (p.16)⟩ +=

```
100  .TP
101  .B -f \fIfilter\fP
102  Specify which output filter to use. At present, only
103  .I latex
104  is available, so this is the default.
```

(This code is extended in #12$_b$ (p.17).)

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.

#11$_b$ ⟨*weave0 parameters* #11 (p.16)⟩ +=

```
105  /^-l$/ and do { $_ = shift;
106       ⟨weave0: note language  #15 (p.17)⟩;  next PARAM; };
107  /^-l(.+)/ and do { $_ = $1;
108       ⟨weave0: note language  #15 (p.17)⟩;  next PARAM; };
```

(This code is extended in #11$_c$ (p.18).)

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*⟩ ≡

```
109  &Usage unless $_;
110  my $pr = "$Lib_dir/w0-l-$_";
111  if (-r $pr) {
112      $L_prog = $pr;
113  } else {
114      &Message("Language $_ is unknown;", "use of -l option ignored.");
115  }
```

(This code is used in #11$_b$ (p.17).)

The variable $L_prog must be declared.

#9$_e$ ⟨*weave0 definitions* #9 (p.15)⟩ +=

```
116  my $L_prog = "";
```

(This code is extended in #9$_f$ (p.18).)

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

#12$_b$ ⟨*weave0 man page parameters* #12 (p.16)⟩ +=

```
117  .TP
118  .B -l \fIlanguage\fP
119  Specify which language filter to use when processing the data.
120  Currently, there exist filters
121  .IR c ", " java ", "
122  .IR latex ", and " perl.
123  The default is to use no language filter at all.
```

(This code is extended in #12$_c$ (p.18).)

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.

#11_c ⟨*weave0 parameters* #11 (p.16)⟩ +=

```
124   /^-o$/ and do { $_ = shift or &Usage;
125       push @Code_opt, "-o'$_'";  next PARAM; };
126   /^-o(.+)/ and do {
127       push @Code_opt, "-o'$1'";  next PARAM; };
```
(This code is extended in #11_d (p.18).)

The man page description is also necessay:

#12_c ⟨*weave0 man page parameters* #12 (p.16)⟩ +=

```
128   .TP
129   .B -o \fIfile\fP
130   Specify the name of the file on which to write the documentation.
131   If this option is not used, the output will go to
132   .I standard output.
```
(This code is extended in #12_d (p.18).)

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.

#11_d ⟨*weave0 parameters* #11 (p.16)⟩ +=

```
133   /^-v$/ and do {
134       print STDERR "This is $Prog (version $Version)\n";
135       push @Pre_opt, "-v";  push @Lang_opt, "-v";
136       push @Code_opt, "-v";  $Verbose = "Yes";  next PARAM; };
```
(This code is extended in #11_e (p.18).)

The default is to be silent.

#9_f ⟨*weave0 definitions* #9 (p.15)⟩ +=

```
137   my $Verbose = 0;
```

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

#12_d ⟨*weave0 man page parameters* #12 (p.16)⟩ +=

```
138   .TP
139   .B -v
140   State the program version. Use of this option will also make
141   .I weave0
142   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.

#11_e ⟨*weave0 parameters* #11 (p.16)⟩ ≡

```
143   /^-/ and do {
144       &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 14. The language processor w0-l-*xxx*, however, is run through a pipeline so any status value indicating error will be ignored.[5]

#16 ⟨*weave0 processing*⟩ ≡

```
145   my $Pre_cmd = "$Lib_dir/w0pre @Pre_opt @Pre_files";
146   my $Post_cmd = "";
147   $Post_cmd .= "$L_prog @Lang_opt | "
```

---

[5]Since this program is part of the *web₀* package, it should never produce any errors. ☺

```
148      if $L_prog;
149  $Post_cmd .= "$F_prog @Code_opt";

150
151  &Message("Running $Pre_cmd | $Post_cmd") if $Verbose;
152  open(PRE, "$Pre_cmd |");
153  open(POST, "| $Post_cmd");
154  print POST while <PRE>;
155  close PRE;   exit $?>>8 if $?>>8;
156  close POST;  exit $?>>8;
```
(This code is used in #8 (p.15).)

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*⟩ ≡
```
157  sub Usage {
158      print STDERR "Usage: $Prog [-e] [-f filter] [-l language]",
159                   " [-o file] [-v] [file...]\n";
160      exit 1;
161  }
```
(This code is used in #8 (p.15).)

2.3 The w0pre filter

The w0pre 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 web_0 package. The following tokens are produced:

code shows code in the body of a macro definition. If a code line contains macro name references, it will result in several use tokens (one for each macro name used) and several code tokens (one 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 macro 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 macro definition.

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

use marks the use of a macro.

Each line produced by w0pre contains exactly one token. For example, the web_0 code shown in Figure 10 on the next page produces the tokens shown in Figure 11 on page 21.

2.3.1 The main program

The w0pre program is written in Perl.

#18 ⟨*w0pre*⟩ ≡
```
162  #! ⟨perl  #105 (p.54)⟩
163
```

Figure 10: Another typical web_0 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 w0pre from the web_0 code Hello.w0 in Figure 10 on the facing page

```
file:Hello.w0;
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};
```

21

```
164   ⟨w0pre definitions  #19 (p.22)⟩
165   ⟨w0pre initialization  #20 (p.22)⟩
166   ⟨w0pre parameter handling  #21 (p.22)⟩
167   ⟨w0pre token recognition  #23 (p.23)⟩
168   exit($N_errors ? 1 : 0);
169
170   ⟨w0pre utility functions  #29 (p.24)⟩
171   ⟨user message functions  #112 (p.57)⟩
```
(This code is not used.)

2.3.2   Definitions

2.3.2.1   Identification   As all the other programs in this package, w0pre can identify itself with its name and version number.

#19   ⟨*w0pre definitions*⟩ ≡
```
172   my ($Prog, $Version) = ("w0pre", "⟨version  #107 (p.54)⟩");
```
(This code is extended in #19ₐ (p.22). It is used in #18 (p.19).)

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.

#19$_a$ ⟨*w0pre definitions* #19 (p.22)⟩ + ≡
```
173   my ($False, $True) = (0, 1);
```
(This code is extended in #19$_b$ (p.22).)

2.3.2.3   Macro name start pattern   To avoid using <<...>> in the program (see Section 1.2 on page 10), we need to introduce a variable $Macro_start to contain the opening brackets.

#19$_b$ ⟨*w0pre definitions* #19 (p.22)⟩ + ≡
```
174   my $Macro_start = "<<";
```
(This code is extended in #19$_c$ (p.24).)

2.3.3   Initialization

Always start in documentation mode.

#20   ⟨*w0pre initialization*⟩ ≡
```
175   my ($Code_mode, $New_line) = ($False, $False);
```
(This code is used in #18 (p.19).)

2.3.4   Parameter handling

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

#21   ⟨*w0pre parameter handling*⟩ ≡
```
176   PARAM:
177   while (@ARGV && $ARGV[0]=~/^-/) { $_ = shift;
178       ⟨w0pre parameters  #22 (p.23)⟩
179       &Message("Unknown option '$_' ignored.");
180   }
```
(This code is used in #18 (p.19).)

2.3.4.1 The −v option will make w0pre state its name and version.

#22 ⟨*w0pre parameters*⟩ ≡
```
181    /^-v$/ and do {
182        print STDERR "This is $Prog (version $Version)\n";
183        next PARAM; };
```
(This code is used in #21 (p.22).)

2.3.5 Extracting tokens

This loop will read each line and scan the web_0 code for tokens.

#23 ⟨*w0pre token recognition*⟩ ≡
```
184    LINE:
185    while (<>) { chomp;
186        ⟨w0pre check input file  #28 (p.24)⟩
187        ⟨expand TAB characters  #111 (p.56)⟩
188        ⟨w0pre check for end of macro definition  #26 (p.23)⟩
189        ⟨w0pre check for start of macro definition  #24 (p.23)⟩
190        ⟨w0pre handle text line  #27 (p.24)⟩
191        ⟨w0pre check one line of a macro definition  #25 (p.23)⟩
192    }
```
(This code is used in #18 (p.19).)

2.3.6 Macro definitions

A new macro is defined when the user writes a line containing only a macro name and an "=" sign (and possibly some spaces surrounding the "="). The variable $Code_mode is set to indicate that w0pre is currently reading a macro definition.

#24 ⟨*w0pre check for start of macro definition*⟩ ≡
```
193    /^$Macro_start(.*?)>>\s*=\s*$/o && !$Code_mode and do {
194        $Last_def = &Find_macro_sy($1);  $Code_mode = $True;
195        print "def:$Last_def>>\n";  $New_line = $False;
196        next LINE; };
```
(This code is used in #23 (p.23).)

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

#25 ⟨*w0pre check one line of a macro definition*⟩ ≡
```
197    print "nl\n" if $New_line;
198    while (s/^(.*?)$Macro_start(.*?)>>//o) {
199        print "code:$1;\n" if $1;
200        print "use:", &Find_macro_sy($2), ">>\n";
201    }
202    print "code:$_;\n" if $_;  $New_line = $True;
```
(This code is used in #23 (p.23).)

(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 macro definition.)

The end of a macro definition is recognized by a line with a lone "@" (except for additional spaces[6] after the "@").

#26 ⟨*w0pre check for end of macro definition*⟩ ≡
```
203    /^@\s*$/ && $Code_mode and do {
204        $Code_mode = $New_line = $False;  next LINE; };
```
(This code is used in #23 (p.23).)

---

[6]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 macro, the line is just copied as it is into a `text` token.

#27 ⟨*w0pre handle text line*⟩ ≡
```
205   unless ($Code_mode) { print "text:$_;\n";  next LINE; };
```
(This code is used in #23 (p.23).)

2.3.8 File name check

The variable `$Cur_file` keeps track of the source file name.

#19_c ⟨*w0pre definitions* #19 (p.22)⟩ + ≡
```
206   my $Cur_file = "";
```
(This code is extended in #19_d (p.24).)

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

#28 ⟨*w0pre check input file*⟩ ≡
```
207   unless ($Cur_file) { $Cur_file = $ARGV;  print "file:$Cur_file;\n"; }
```
(This code is extended in #28_a (p.24). It is used in #23 (p.23).)

When a file has been completely read, we must clear `$Cur_file` to force another `file` token if another file is read.

#28_a ⟨*w0pre check input file* #28 (p.24)⟩ + ≡
```
208   $Cur_file = "" if eof;
```

2.3.9 Utility functions

2.3.9.1 The function &Find_macro_sy This function is used to find the correct name of a macro, in particular when the <<>> or <<*XXX*...>> notation is used. The function has one parameter:

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

#29 ⟨*w0pre utility functions*⟩ ≡
```
209   sub Find_macro_sy {
210       local $_ = shift;
211
212       ⟨find_macro_sy: handle reference to last macro  #30 (p.24)⟩
213       ⟨find_macro_sy: handle abbreviated macro name  #31 (p.25)⟩
214       $Macro{$_} = "Defined";  return $_;
215   }
```
(This code is extended in #29_a (p.25). It is used in #18 (p.19).)

<<>> is used to refer to the last macro defined.

#30 ⟨*find_macro_sy: handle reference to last macro*⟩ ≡
```
216   unless ($_) {
217       &Warning("Illegal use of '$Macro_start>>' notation;",
218           "no macro defined yet.") unless $Last_def;
219       return $Last_def || "??";
220   }
```
(This code is used in #29 (p.24).)

The name of the last macro defined is kept in `$Last_def`.

#19_d ⟨*w0pre definitions* #19 (p.22)⟩ + ≡
```
221   my $Last_def = "";
```
(This code is extended in #19_e (p.25).)

<<*XXX*...>> is a reference to a macro whose name starts with *XXX*. This can be found by searching the table %Macro which contains all known macro names (as keys).

#19_e ⟨*w0pre definitions* #19(p.22)⟩ + ≡
    ₂₂₂ `my %Macro = ();`
    (This code is extended in #19_f(p.25).)

There should be exactly one such macro name in `%Macro`.

#31 ⟨*find_macro_sy: handle abbreviated macro name*⟩ ≡
    ₂₂₃ `if (/\.{3}$/) {`
    ₂₂₄   `my $abbrev = $';`
    ₂₂₅   `my $ab_len = length $abbrev;`
    ₂₂₆
    ₂₂₇   `my @match = grep { substr($_,0,$ab_len) eq $abbrev } keys %Macro;`
    ₂₂₈   `unless (@match) {`
    ₂₂₉   `&Warning("No match for $Macro_start$_>>.");`
    ₂₃₀   `return "??";`
    ₂₃₁   `}`
    ₂₃₂   `&Warning("Multiple matches for $Macro_start$_>>:",`
    ₂₃₃   `join(", ", map("$Macro_start$_>>",@match))) if @match>1;`
    ₂₃₄   `return $match[0];`
    ₂₃₅ `}`
    (This code is used in #29(p.24).)

(Note in particular the use of `grep` to find matching macro names. The test here cannot use a pattern as the abbreviated macro 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`.

#29_a ⟨*w0pre utility functions* #29(p.24)⟩ + ≡
    ₂₃₆ `sub Warning {`
    ₂₃₇   `&Message(@_);  ++$N_errors;`
    ₂₃₈ `}`

The error count must be declared.

#19_f ⟨*w0pre definitions* #19(p.22)⟩ + ≡
    ₂₃₉ `my $N_errors = 0;`

2.4 The w0code filter

The `w0code` filter reads tokens produced by `w0pre` and prints the program code; it forms the last link in the pipeline set up by `tangle0`. Most of this code can be found as parts of the various tokens, but macro names must be expanded.
    The `w0code` filter is written in Perl.

#32 ⟨*w0code*⟩ ≡
    ₂₄₀ `#! `⟨*perl* #105(p.54)⟩
    ₂₄₁
    ₂₄₂ ⟨*w0code definitions* #33(p.26)⟩
    ₂₄₃ ⟨*w0code parameter handling* #34(p.26)⟩
    ₂₄₄ ⟨*w0code read tokens* #37(p.27)⟩
    ₂₄₅ ⟨*w0code expand macros* #40(p.28)⟩
    ₂₄₆ `close OUT unless $Output == \*STDOUT;`
    ₂₄₇ `exit ($N_errors ? 1 : 0);`
    ₂₄₈
    ₂₄₉ ⟨*w0code utility functions* #41(p.28)⟩
    ₂₅₀ ⟨*user message functions* #112(p.57)⟩
    (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*⟩ ≡
251   my ($Prog, $Version) = ("w0code", "⟨*version* #107(p.54)⟩");
(This code is extended in #33ₐ (p.26). It is used in #32 (p.25).)

2.4.1.2 Macro name start pattern To avoid using <<...>> in the program (see Section 1.2 on page 10), we need to introduce a variable $Macro_start to contain the opening brackets.

#33ₐ ⟨*w0code definitions* #33(p.26)⟩ + ≡
252   my $Macro_start = "<<";
(This code is extended in #33_b (p.26).)

2.4.2 Parameter handling

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

#34 ⟨*w0code parameter handling*⟩ ≡
253   PARAM:
254   **while** (@ARGV && $ARGV[0] =~ /^-/) { $_ = shift;
255   ⟨*w0code parameters* #35(p.26)⟩
256   }
(This code is used in #32 (p.25).)

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*⟩ ≡
257   /^-o$/ **and do** { $_ = shift;
258   ⟨*w0code: note output file* #36(p.26)⟩; **next** PARAM; };
259   /^-o(.+)/ **and do** { $_ = $1;
260   ⟨*w0code: note output file* #36(p.26)⟩; **next** PARAM; };
(This code is extended in #35ₐ (p.26). It is used in #34 (p.26).)

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*⟩ ≡
261   open(OUT, ">$_") **or** &Error("Could not create $_.");
262   $Output = *OUT;
(This code is used in #35 (p.26).)

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

#33_b ⟨*w0code definitions* #33(p.26)⟩ + ≡
263   my $Output = *STDOUT;
(This code is extended in #33_c (p.27).)

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

#35ₐ ⟨*w0code parameters* #35(p.26)⟩ + ≡
264   /^-v$/ **and do** {
265   print STDERR "This is $Prog (version $Version)\n";
266   **next** PARAM; };
(This code is extended in #35_b (p.27).)

2.4.2.3 The -x option This option is used to indicate the macro name with which to start the extraction.

#35ᵦ ⟨*w0code parameters* #35 (p.26)⟩ +≡
```
267  /^-x$/ and do { $Start = shift;  next PARAM; };
268  /^-x(.+)/ and do { $Start = $1;  next PARAM; };
```
(This code is extended in #35c (p.27).)

The variable `$Start` is used for this name.

#33ᵪ ⟨*w0code definitions* #33 (p.26)⟩ +≡
```
269  my $Start = "";
```
(This code is extended in #33d (p.27).)

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

#35ᵪ ⟨*w0code parameters* #35 (p.26)⟩ +≡
```
270  &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 macros defined; each value in `%Def` will be a text string of all the token containing the body of that macro, separated by newlines.

#33ᵈ ⟨*w0code definitions* #33 (p.26)⟩ +≡
```
271  my %Def = ();
```
(This code is extended in #33e (p.27).)

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

#37 ⟨*w0code read tokens*⟩ ≡
```
272  DATA:
273  while (<>) { chomp;
274      ⟨w0code macro definition  #38 (p.27)⟩
275      ⟨w0code add to macro body  #39 (p.28)⟩
276  }
```
(This code is used in #32 (p.25).)

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

#38 ⟨*w0code macro definition*⟩ ≡
```
277  /^def:(.*)>>$/ and do {
278      $Cur_def = $1;  $Start = $Start || $Cur_def;
279      $Def{$Cur_def} =
280          exists $Def{$Cur_def} ? "$Def{$Cur_def}nl\n" : "";
281      next DATA; };
```
(This code is used in #37 (p.27).)

The variable `$Cur_def` always contains the name of the macro whose body is being defined.

#33ₑ ⟨*w0code definitions* #33 (p.26)⟩ +≡
```
282  my $Cur_def = "";
```
(This code is extended in #33f (p.28).)

27

2.4.3.2 Macro body   The code, nl, and use tokens all add to the body of the current macro definition.

#39   ⟨*w0code add to macro body*⟩ ≡
```
283  /^(code:|nl|use:)/ and do {
284      $Def{$Cur_def} .= "$_\n";  next DATA; };
```
(This code is used in #37 (p.27).)

All other tokens are ignored.

2.4.4 Expanding the macros

Since expanding the macro is a recursive process, w0code uses the function &Expand for this.

#40   ⟨*w0code expand macros*⟩ ≡
```
285  &Error("No macros defined.") unless $Start;
286  &Expand($Start);
287  print $Output "\n";
```
(This code is used in #32 (p.25).)

The function &Expand has one parameter: the name of the macro to expand.

#41   ⟨*w0code utility functions*⟩ ≡
```
288  sub Expand {
289      my $sy = shift;
290      ⟨w0code expand: check that macro is defined  #42 (p.28)⟩
291      ⟨w0code expand: check for definition cycles  #43 (p.28)⟩
292      ⟨w0code expand: expand the macro body  #45 (p.29)⟩
293      ⟨w0code expand: deactivate current macro  #44 (p.29)⟩
294  }
```
(This code is extended in #41$_a$ (p.29). It is used in #32 (p.25).)

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

#42   ⟨*w0code expand: check that macro is defined*⟩ ≡
```
295  unless (exists $Def{$sy}) {
296      &Warning("Macro name $Macro_start$sy>> has not been defined.")
297          unless exists $Not_def_mess{$sy};
298      $Not_def_mess{$sy} = "reported";  return;
299  }
```
(This code is used in #41 (p.28).)

The table %Not_def_mess keeps track of the messages to avoid repeating them.

#33$_f$ ⟨*w0code definitions #33 (p.26)*⟩ +≡
```
300  my %Not_def_mess = ();
```
(This code is extended in #33$_g$ (p.29).)

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 w0code to avoid an endless loop. A check for circular definitions is simple to implement, however.

#43   ⟨*w0code expand: check for definition cycles*⟩ ≡
```
301  &Error("Definition cycle found;", "the loop involves:",
302      join(" ", map("$Macro_start$_>>",
303               sort {$Active{$a} <=> $Active{$b}}
304                   grep {$Active{$_} >= $Active{$sy}}
305                       keys %Active)))
306          if $Active{$sy};
307  $Active{$sy} = 1+keys %Active;
```
(This code is used in #41 (p.28).)

The table `%Active` contains (as keys) the names of all the macros that are currently being expanded.[7]

#33ᵍ ⟨*w0code definitions* #33 (p.26)⟩ +=

```
308   my %Active = ();
```
(This code is extended in #33ₕ (p.29).)

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

#44 ⟨*w0code expand: deactivate current macro*⟩ ≡

```
309   delete $Active{$sy};
```
(This code is used in #41 (p.28).)

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 ⟨*w0code expand: expand the macro body*⟩ ≡

```
310   my $line;
311   foreach $line (split(/\n/,$Def{$sy})) {
312       if ($line =~ /^code:(.*);$/) { print $Output $1; }
313       elsif ($line =~ /^nl$/) { print $Output "\n"; }
314       elsif ($line =~ /^use:(.*)>>$/) { &Expand($1); }
315       else { &Error("Unknown format: $line."); }
316   }
```
(This code is used in #41 (p.28).)

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 ⟨*w0code utility functions* #41 (p.28)⟩ +=

```
317   sub Warning {
318       &Message(@_);  ++$N_errors;
319   }
```

The error count must be declared.

#33ₕ ⟨*w0code definitions* #33 (p.26)⟩ +=

```
320   my $N_errors = 0;
```

2.5 The w0-f-latex filter

This program reads tokens produced by w0pre (see Section 2.3 on page 19) and produces LaTeX code. To be more particular, the following commands and environments are generated:

\wzdef is used whenever a new macro is defined; see Section 3.1.4.1 on page 61.

\wzenddef terminates the macro definition; see Section 3.1.4.2 on page 61.

\wzeol is used as a code line separator; see Section 3.1.4.3 on page 62.

\wzfile signals that a new source file is being read; see Section 3.1.4.4 on page 63.

\wzmacro is used to typeset a macro name; see Section 3.1.4.5 on page 63.

Figure 12: The LaTeX code produced by weave0 from the *web₀* code Hello.w0 in Figure 10 on page 20

```
\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\wzmacroindex{\begin{wzindex}{\wzmacroindexname}{1}
\wzx{\wzmacro{deklarasjoner~~\upshape\#3}}\wzlongpageref{3}\par
\wzx{\wzmacro{hello~world~~\upshape\#1}}\wzlongpageref{1}\rlap{~*}\par
\wzx{\wzmacro{importspesifikasjoner~~\upshape\#4}}\wzlongpageref{4}\par
\wzx{\wzmacro{setninger~~\upshape\#2}}\wzlongpageref{2}\par
\smallskip\raggedright\wzmacroindexstartext\par
\end{wzindex}}
\ifx \wzfile\undefined
\AtBeginDocument{\wzfile {Hello.w0}}\else \wzfile {Hello.w0}\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}\wzmacro[4]{importspesifikasjoner}\wzeol\relax
class~Hello~\{\wzeol\relax
~~public~static~void~main(String~args[])~\{\wzeol\relax
~~~~\wzmacro[3]{deklarasjoner}\wzeol\relax
~~~~\wzmacro[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}{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}
```

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

#46 ⟨*w0-f-latex*⟩ ≡
321 *#!* ⟨*perl* #105 (p.54)⟩
322
323 ⟨*alphabetical sorting* #113 (p.57)⟩
324 ⟨*w0-f-latex: initialization* #47 (p.31)⟩
325 ⟨*w0-f-latex: option handling* #48 (p.31)⟩
326 ⟨*w0-f-latex: make LaTeX code* #51 (p.33)⟩
327 `close $Output` **`unless`** `$Output == \*STDOUT;`
328 **`exit`** `($N_errors ? 1 : 0);`
329
330 ⟨*w0-f-latex: utility functions* #61 (p.38)⟩
331 ⟨*latex generation functions* #108 (p.54)⟩
332 ⟨*user message functions* #112 (p.57)⟩
(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*⟩ ≡
333 `my ($Prog, $Version) = ("w0-f-latex", "`⟨*version* #107 (p.54)⟩`");`
(This code is extended in #47$_a$ (p.31). It is used in #46 (p.31).)

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

#47$_a$ ⟨*w0-f-latex: initialization* #47 (p.31)⟩ + ≡
334 `my ($False, $True) = (0, 1);`
(This code is extended in #47$_b$ (p.31).)

2.5.1.3 Start of macro name   must be defined in a variable to avoid writing <<...>> which will confuse *web$_0$*.

#47$_b$ ⟨*w0-f-latex: initialization* #47 (p.31)⟩ + ≡
335 `my $Macro_start = "<<";`
(This code is extended in #47$_c$ (p.31).)

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

#47$_c$ ⟨*w0-f-latex: initialization* #47 (p.31)⟩ + ≡
336 `my $Output = \*STDOUT;`
(This code is extended in #47$_d$ (p.32).)

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*⟩ ≡
337 `PARAM:`
338 **`while`** `(@ARGV && $ARGV[0] =~ /^-/) { $_ = shift;`
339   ⟨*w0-f-latex: examine options* #49 (p.32)⟩
340 `}`
(This code is used in #46 (p.31).)

---

[7]The %Active values are 1, 2, . . . which are useful when producing a good error message.

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*⟩ ≡
   341 /^-o$/ **and do** { $_ = shift; ⟨*w0-f-latex: note output file* #50 (p.32)⟩
   342   **next** PARAM; };
   343 /^-o(.+)$/ **and do** { $_ = $1; ⟨*w0-f-latex: note output file* #50 (p.32)⟩
   344   **next** PARAM; };
   (This code is extended in #49ₐ (p.32). It is used in #48 (p.31).)

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*⟩ ≡
   345 open(OUT, ">$_") **or** &Error("Could not create '$_'.");
   346 $Output = *OUT;
   (This code is used in #49 (p.32).)

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.32)⟩ +≡
   347 /^-v$/ **and do** {
   348   print STDERR "This is $Prog (version $Version)\n";
   349   **next** PARAM; };
   (This code is extended in #49ᵦ (p.32).)

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

#49ᵦ ⟨*w0-f-latex: examine options* #49 (p.32)⟩ +≡
   350 &Message("Unknown option '$_' ignored.");

2.5.3 Pass 1

Translating tokens from w0pre into LᴬTEX code is a three-pass affair:

1. Make a list of all the macro names.

2. Collect information on where each macro 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 macros have been defined in the source files. The following variables are used:

$N_macro contains the number of distinct macros defined.

#47d ⟨*w0-f-latex: initialization* #47 (p.31)⟩ +≡
   351 my $N_macro = 0;
   (This code is extended in #47ₑ (p.32).)

%Macro_id contains the macro names (as keys); the values are the macros' identification numbers.

#47ₑ ⟨*w0-f-latex: initialization* #47 (p.31)⟩ +≡
   352 my %Macro_id = ();
   (This code is extended in #47f (p.33).)

`@Macro_n_ext` contains how many extensions each macro has.

```
353  my @Macro_n_ext = ();
```
(This code is extended in #47$_g$ (p.34).)

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.

```
354  my @Tokens = ();
355  while (<>) { chomp;  push(@Tokens, $_);
356    ⟨w0-f-latex: pass1: note macro definition  #52 (p.33)⟩
357  }
```
(This code is extended in #51$_a$ (p.33). It is used in #46 (p.31).)

Only the def tokens are of interest in this pass.

```
358  /^def:(.*)>>$/ and do {
359    my $symb = $1;
360    if (defined $Macro_id{$symb}) {
361      ++$Macro_n_ext[$Macro_id{$symb}];
362    } else {
363      $Macro_id{$symb} = ++$N_macro;  $Macro_n_ext[$N_macro] = 0;
364    }
365  };
```
(This code is used in #51 (p.33).)

2.5.4   Pass 2

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

`@Macro_use` will tell where each macro has been used. Its index is the macro's identification number, and the value is a text string on the form

$$n_1\text{-}e_1 \quad n_2\text{-}e_2 \quad \ldots$$

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

```
366  my @Macro_use = ();
```
(This code is extended in #51$_b$ (p.33).)

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

`$Cur_line` contains the line number of the current code line.

```
367  my $Cur_line = 0;
```
(This code is extended in #51$_c$ (p.34).)

`%message_given` counts how many times an error message for a macro name with no definition has been given. Its purpose is to avoid reporting problems with each macro name more than once.

#51_c ⟨*w0-f-latex: make LaTeX code* #51(p.33)⟩ +≡
368 my %message_given = ();
(This code is extended in #51_d (p.34).)

@macro_ext_cnt counts how many extensions for a given macro have been found so far.

#51_d ⟨*w0-f-latex: make LaTeX code* #51(p.33)⟩ +≡
369 my @macro_ext_cnt = ();
(This code is extended in #51_e (p.34).)

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

#51_e ⟨*w0-f-latex: make LaTeX code* #51(p.33)⟩ +≡
370 $Cur_line = 0;
371 **foreach** (@Tokens) {
372   ⟨*w0-f-latex: pass2: handle tokens* #53(p.34)⟩
373 }
374 %message_given = ();   @macro_ext_cnt = ();
(This code is extended in #51_f(p.35).)

The def tokens indicate which is the current macro name and extension.

#53 ⟨*w0-f-latex: pass2: handle tokens*⟩ ≡
375 /^def:(.*)>>$/ **and do** { ++$Cur_line;
376      $cur_macro = $Macro_id{$1};
377      $cur_ext = $macro_ext_cnt[$cur_macro]++; **next**; };
(This code is extended in #53_a (p.34). It is used in #51_e(p.34).)

The current macro name and extension are kept in $cur_macro and $cur_ext (in numeric form).

#47_g ⟨*w0-f-latex: initialization* #47(p.31)⟩ +≡
378 my ($cur_macro, $cur_ext) = (0, 0);
(This code is extended in #47_h(p.34).)

The use tokens show the use of a macro name.

#53_a ⟨*w0-f-latex: pass2: handle tokens* #53(p.34)⟩ +≡
379 /^use:(.*)>>$/ **and do** { my $symb = $1;
380   **if** (defined $Macro_id{$symb}) {
381        $Macro_use[$Macro_id{$symb}] .= "$cur_macro-$cur_ext ";
382   } **elsif** ($message_given{$symb}++ == 0) {
383   &Warning("Macro name $Macro_start$symb>> not defined.");
384   }
385   **next**; };
(This code is extended in #53_b (p.34).)

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

#53_b ⟨*w0-f-latex: pass2: handle tokens* #53(p.34)⟩ +≡
386 /^nl$/ **and do** { ++$Cur_line; **next**; };
(This code is extended in #53_c (p.34).)

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 "*nn*0" and "*nn*1", respectively.[8]

#47_h ⟨*w0-f-latex: initialization* #47(p.31)⟩ +≡
387 my %Func_index = ();
(This code is extended in #47_i (p.35).)

#53_c ⟨*w0-f-latex: pass2: handle tokens* #53(p.34)⟩ +≡
388 /^fdef:(.*);/ **and do** {
389      $Func_index{$1} .= $Cur_line . "0 "; **next**; };
390 /^fuse:(.*);/ **and do** {

---

[8]The reason for this notation is that I want to list definition occurrences before the ones that concern use.

```
391        $Func_index{$1} .= $Cur_line . "1 ";   next; };
```
(This code is extended in #53$_d$ (p.35).)

Similarly, the `vdef` and `vuse` tokens report that a variable has been defined or used. This information is saved in `%Var_index` as "*nn*0" and "*nn*1", respectively.

#47$_i$ ⟨*w0-f-latex: initialization* #47(p.31)⟩ + ≡
```
392  my %Var_index = ();
```
(This code is extended in #47$_j$ (p.35).)

#53$_d$ ⟨*w0-f-latex: pass2: handle tokens* #53(p.34)⟩ + ≡
```
393  /^vdef:(.*);/ and do {
394      $Var_index{$1} .= $Cur_line . "0 ";   next; };
395  /^vuse:(.*);/ and do {
396      $Var_index{$1} .= $Cur_line . "1 ";   next; };
```
(This code is extended in #53$_e$ (p.35).)

And, finally, the `cdef` and `cuse` tokens report that a class has been defined or used. This information is saved in `%Class_index` as "*nn*0" and "*nn*1", respectively.

#47$_j$ ⟨*w0-f-latex: initialization* #47(p.31)⟩ + ≡
```
397  my %Class_index = ();
```
(This code is extended in #47$_k$ (p.35).)

#53$_e$ ⟨*w0-f-latex: pass2: handle tokens* #53(p.34)⟩ + ≡
```
398  /^cdef:(.*);/ and do {
399      $Class_index{$1} .= $Cur_line . "0 ";   next; };
400  /^cuse:(.*);/ and do {
401      $Class_index{$1} .= $Cur_line . "1 ";   next; };
```

2.5.5   Pass 3

The third and last pass generates the actual LaTeX code. While generating this code, the program is in two modes:

- When `$Code_mode` is `$True`, `w0-f-latex` is processing code, i.e., the definition of a macro.

- 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:

@Macro_x counts how many extensions to a macro have been found so far during this pass.

#47$_k$ ⟨*w0-f-latex: initialization* #47(p.31)⟩ + ≡
```
402  my @Macro_x = ();
```
(This code is extended in #47$_l$ (p.36).)

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

#51$_f$ ⟨*w0-f-latex: make LaTeX code* #51(p.33)⟩ + ≡
```
403  ⟨w0-f-latex: generate indices  #62(p.38)⟩
```
(This code is extended in #51$_g$ (p.35).)

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

#51$_g$ ⟨*w0-f-latex: make LaTeX code* #51(p.33)⟩ + ≡
```
404  my ($Code_mode, $Cur_line) = ($False, 0);
405  foreach (@Tokens) {
406      ⟨w0-f-latex: pass3: handle 'code' tokens  #54(p.36)⟩
407      ⟨w0-f-latex: pass3: handle 'def' tokens  #55(p.36)⟩
```

⟨*w0-f-latex: pass3: handle 'file' tokens* #56 (p.36)⟩
409 ⟨*w0-f-latex: pass3: handle 'nl' tokens* #57 (p.37)⟩
410 ⟨*w0-f-latex: pass3: handle 'text' tokens* #58 (p.37)⟩
411 ⟨*w0-f-latex: pass3: handle 'use' tokens* #59 (p.37)⟩
412 }
413 ⟨*w0-f-latex: end code* #60 (p.37)⟩

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 itailics, respectively.

#54 ⟨*w0-f-latex: pass3: handle 'code' tokens*⟩ ≡
414 `/^code:(.*);$/` **and do** {
415   `print $Output &Latexify($1);` **next**; };
416 `/^bcode:(.*);$/` **and do** {
417   `print $Output "\\textbf{", &Latexify($1), "}";` **next**; };
418 `/^bicode:(.*);$/` **and do** {
419   `print $Output "\\textit{\\textbf{", &Latexify($1), "}}";`
420   **next**; };
421 `/^icode:(.*);$/` **and do** {
422   `print $Output "\\textit{", &Latexify($1), "}";` **next**; };
(This code is used in #51$_g$ (p.35).)

2.5.5.2 Handling the 'def' tokens These tokens indicate the start of a macro definition (and, thus, "code mode").

#55 ⟨*w0-f-latex: pass3: handle 'def' tokens*⟩ ≡
423 `/^def:(.*)>>$/` **and do** {
424   **my** `$symb = $1;  $def_id = $Macro_id{$symb};`
425   **if** `(defined $Macro_x[$def_id]) {`
426   `++$Macro_x[$def_id];`
427   } **else** {
428   `$Macro_x[$def_id] = 0;`
429   }
430   ⟨*w0-f-latex: end code* #60 (p.37)⟩
431   `print $Output "\\wzdef{", &Latexify($symb),`
432   `"}{$def_id}{$Macro_x[$def_id]}";`
433   `$Code_mode = $True;  ++$Cur_line;` **next**; };
(This code is used in #51$_g$ (p.35).)

The name of the macro being defined is kept in variable `$def_id`.

#47$_l$ ⟨*w0-f-latex: initialization* #47 (p.31)⟩ +≡
434 **my** `$def_id = "";`
(This code is extended in #47$_m$ (p.40).)

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*⟩ ≡
435 `/^file:(.*);$/` **and do** {
436   ⟨*w0-f-latex: end code* #60 (p.37)⟩
437   `print $Output "\\ifx \\wzfile\\undefined`
438 `\\AtBeginDocument{\\wzfile {$1}}\\else \\wzfile {$1}\\fi\n";`
439   **next**; };
(This code is used in #51$_g$ (p.35).)

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*⟩ ≡
```
440  /^nl$/ and do {
441      print $Output "\\wzeol\\relax\n";  ++$Cur_line;  next; };
```
(This code is used in #51$_g$ (p.35).)

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*⟩ ≡
```
442  /^text:(.*);$/ and do {
443      my $text = $1;   ⟨w0-f-latex: end code #60 (p.37)⟩
444      print $Output "$text\n";  next; };
```
(This code is used in #51$_g$ (p.35).)

2.5.5.6 Handling the 'use' tokens These tokens are found when a macro is referenced in the code. It is translated into a call on \wzmacro with the name and its number as parameters.

#59 ⟨*w0-f-latex: pass3: handle 'use' tokens*⟩ ≡
```
445  /^use:(.*)>>$/ and do {
446      print $Output "\\wzmacro[$Macro_id{$1}]{", &Latexify($1), "}";
447      next; };
```
(This code is used in #51$_g$ (p.35).)

2.5.5.7 Terminating a macro definition (which is detected by reading the first 'text' token after a macro definition, or on finding two consecutive macro definitions, or when the last token has been read from a file) also requires some special actions. The macro \wzenddef is supplied with the following parameters (see Section 3.1.4.2 on page 61):

1. the macro name'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*⟩ ≡
```
448  print $Output "%\n\\wzenddef{$def_id}{$Macro_x[$def_id]}{" .
449      ($Macro_x[$def_id]<$Macro_n_ext[$def_id] ? $Macro_x[$def_id]+1 : 0) .
450      "}{" . ($Macro_x[$def_id]==0 ? &Format_usage($def_id) : 0),
451      "}\\relax\n" if $Code_mode;
452  $Code_mode = $False;
```
(This code is used in #51$_g$ (p.35), #55 (p.36), #56 (p.36), and #58 (p.37).)

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 58.)

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

The function has one parameter:

1. the index of a macro name.

It returns the LaTeX code as a text string.

#61 ⟨*w0-f-latex: utility functions*⟩ ≡
```
453    sub Format_usage {
454        local $_ = $Macro_use[shift];   chomp;
455        my @data = ();
456        my ($last_x, $x) = ("", "");
457
458        foreach $x (split) { push(@data, $x) if $x ne $last_x;   $last_x = $x; }
459
460        my ($n, $res) = (0, "");
461        foreach (@data) {
462            if (++$n == @data && @data > 2) { $res .= "\\wzlastsep\n"; }
463            elsif ($n == 2 && @data == 2) { $res .= "\\wztwosep\n"; }
464            elsif ($n > 1) {$res .= "\\wzsep\n"; }
465            $res .= "\\wzxref" . (/^(\d+)-(\d+)$/ ? "{$1}{$2}" : "{$_}{0}");
466        }
467        return $res;
468    }
```
(This code is extended in #61$_a$ (p.38). It is used in #46 (p.31).)

2.5.6   Making an index

All LaTeX code generated by w0-f-latex will contain indices for the variables, functions, classes, and macro names found in the code. It is up to the user, however, to introduce any of these indices into his or her documentation using the macroes \wzvarindex, \wzfuncindex, \wzclassindex, or \wzmacroindex, respectively.

#62 ⟨*w0-f-latex: generate indices*⟩ ≡
```
469    ⟨w0-f-latex: generate the class index  #70 (p.40)⟩
470    ⟨w0-f-latex: generate the function index  #63 (p.38)⟩
471    ⟨w0-f-latex: generate the variable index  #69 (p.40)⟩
472    ⟨w0-f-latex: generate the macro index  #71 (p.40)⟩
```
(This code is used in #51$_f$ (p.35).)

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

#63 ⟨*w0-f-latex: generate the function index*⟩ ≡
```
473    &Generate_index("func", \%Func_index);
```
(This code is used in #62 (p.38).)

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.

#61$_a$ ⟨*w0-f-latex: utility functions #61 (p.38)*⟩ +≡
```
474    sub Generate_index {
475        my ($cmd, $index) = @_;
476        local $_;
```

```
477        my (@lines, $lx, $initial, $last_initial);
478
479        print $Output "\\def\\wz", $cmd,
480            "index\{\\begin{wzindex}{\\wz", $cmd, "indexname}{2}\n";
481        foreach (sort indexwise keys %{$index}) {
482            ⟨w0-f-latex: produce an initial (if required)  #68 (p.40)⟩
483            ⟨w0-f-latex: generate an index entry  #64 (p.39)⟩
484        }
485        print $Output "\\end{wzindex}\}\n";
486    }
```
(This code is extended in #61ᵦ (p.39).)

The sorting is not quite straightforward. If the index entry does not start with a letter,[9] we will ignore the initial character when sorting.

#61ᵦ ⟨*w0-f-latex: utility functions* #61 (p.38)⟩ +≡
```
487    sub indexwise {
488        my $ax = $a=~/^\w/ ? $a : substr($a,1)." $1";
489        my $bx = $b=~/^\w/ ? $b : substr($b,1)." $1";
490        local ($a, $b) = (uc($ax), uc($bx));
491        return &alphabetically;
492    }
```
(This code is extended in #61_c (p.40).)

Each index entry starts wih a call on \wzx.

#64 ⟨*w0-f-latex: generate an index entry*⟩ ≡
```
493    print $Output "\\wzx{", &Latexify($_), "}";
494    ⟨w0-f-latex: format and print index line numbers  #65 (p.39)⟩
495    print $Output "\\par\n";
```
(This code is used in #61ₐ (p.38).)

The index entry consists of a sequence of line numbers. These must be sorted,[10] and duplicates must be removed. The sorted sequence is saved in @lines.

#65 ⟨*w0-f-latex: format and print index line numbers*⟩ ≡
```
496    @lines = ();
497    foreach $lx (sort { $a <=> $b } split(" ",$index->{$_})) {
498        push @lines, $lx unless @lines && $lines[$#lines] == $lx;
499    }
```
(This code is extended in #65ₐ (p.39). It is used in #64 (p.39).)

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.39)⟩ +≡
```
500    while (@lines) { $lx = shift @lines;
501        ⟨w0-f-latex: check for range of line numbers  #66 (p.39)⟩
502        ⟨w0-f-latex: print index line number  #67 (p.40)⟩
503        print $Output "\\wzindexsep\n" if @lines;
504    }
```

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*⟩ ≡
```
505    if (@lines>=2 && $lines[0]==$lx+10 && $lines[1]==$lx+20) {
506        ⟨w0-f-latex: print index line number  #67 (p.40)⟩
507        $lx = shift(@lines) while @lines && $lines[0]==$lx+10;
508        print $Output "--";
509        ⟨w0-f-latex: print index line number  #67 (p.40)⟩
510        print $Output "\\wzindexsep\n" if @lines;
511        next;
```

---

[9]This quaint sorting rule was invented to handle Perl variable and functions properly; these start with a special character like "$", "@", "%", or "&".

[10]The reason the line numbers must be sorted, is that we want to list defining occurrences before usage on the same line.

```
512   }
```
(This code is used in #65ₐ (p.39).)

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*⟩ ≡
```
513   print $Output $lx%10 ? ("".int($lx/10)) : ("\\wzul{".int($lx/10)."}");
```
(This code is used in #65ₐ (p.39) and #66 (p.39).)

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)*⟩ ≡
```
514   $initial = /^[a-z]/i ? uc(substr($_,0,1)) : uc(substr($_,1,1));
515   print $Output "\\wzinitial{", &Latexify($initial), "}\n"
516       if $initial && $initial ne $last_initial;
517   $last_initial = $initial;
```
(This code is used in #61ₐ (p.38).)

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*⟩ ≡
```
518   &Generate_index("var", \%Var_index);
```
(This code is used in #62 (p.38).)

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*⟩ ≡
```
519   &Generate_index("class", \%Class_index);
```
(This code is used in #62 (p.38).)

2.5.6.4 Generating the macro index
The macro index can be created by just looking at the contents of `%Macro_id`.

#71 ⟨*w0-f-latex: generate the macro index*⟩ ≡
```
520   print $Output "\\def\\wzmacroindex{\\begin{wzindex}",
521       "{\\wzmacroindexname}{1}\n";
522   foreach (sort alphabetically keys %Macro_id) {
523       print $Output "\\wzx{\\wzmacro{", &Latexify($_), "~~\\upshape\\#",
524           "$Macro_id{$_}}}\\wzlongpageref{$Macro_id{$_}}",
525           ($Macro_use[$Macro_id{$_}] ? "" : "\\rlap{~*}"),
526           "\\par\n";
527   }
528   print $Output "\\smallskip\\raggedright\\wzmacroindexstartext\\par
529   \\end{wzindex}}\n";
```
(This code is used in #62 (p.38).)

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.38)⟩ + ≡
```
530   sub Warning {
531       &Message(@_);  ++$N_errors;
532   }
```

The error count must be initialized.

#47m ⟨*w0-f-latex: initialization* #47 (p.31)⟩ + ≡
```
533   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 19) 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*⟩ ≡
534 *#!* ⟨*perl* #105 (p.54)⟩
535
536 ⟨*w0-l-c definitions* #73 (p.41)⟩
537 ⟨*w0-l-c parameter handling* #74 (p.42)⟩
538 ⟨*w0-l-c read C code* #75 (p.42)⟩
539 **exit** 0;
540
541 ⟨*user message functions* #112 (p.57)⟩
(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*⟩ ≡
542 my ($Prog, $Version) = ("w0-l-c", "⟨*version* #107 (p.54)⟩");
(This code is extended in #73ₐ (p.41). It is used in #72 (p.41).)

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

#73ₐ ⟨*w0-l-c definitions* #73 (p.41)⟩ +≡
543 my $C_id = "[A-Za-z_]\\w*";
(This code is extended in #73_b (p.42).)

The same goes for an identifier list.

#73_b ⟨*w0-l-c definitions* #73 (p.41)⟩ +≡

```
544  my $C_id_list = "$C_id(([*, ]*$C_id)*)";
```
(This code is extended in #73_c (p.42).)

We need a table `%Res_words` with all the reserved words in C.

#73_c ⟨*w0-l-c definitions* #73 (p.41)⟩ +≡

```
545  my %Res_words = ();
546  my %Special_words = ();
547  for ("auto", "break", "case", "const", "continue", "default", "do",
548      "else", "enum", "extern", "for", "goto", "if", "register",
549      "return", "sizeof", "static", "struct", "switch", "typedef",
550      "union", "volatile", "while")
551     { $Res_words{$_} = $Special_words{$_} = 1; }
```
(This code is extended in #73_d (p.42).)

We also need a table `%Type_words` with all the predefined type words in C.

#73_d ⟨*w0-l-c definitions* #73 (p.41)⟩ +≡

```
552  my %Type_words = ();
553  for ("char", "double", "float", "int", "long", "short",
554      "unsigned", "void")
555     { $Type_words{$_} = $Special_words{$_} = 1; }
```
(This code is extended in #73_e (p.42).)

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*⟩ ≡

```
556  PARAM:
557  while (@ARGV && $ARGV[0] =~ /^-/) { $_ = shift;
558      /^-e$/ and do { $Enhance = "yes";  next PARAM; };
559      /^-v$/ and do {
560          print STDERR "This is $Prog (version $Version)\n";
561          next PARAM; };
562  }
```
(This code is used in #72 (p.41).)

Variable `$Enhance` must be declared.

#73_e ⟨*w0-l-c definitions* #73 (p.41)⟩ +≡

```
563  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*⟩ ≡

```
564  my $line = "";
565  while (<>) {
566      unless (/^code:(.*);$/) { print; next; };
567      chomp($line = $1);
568      $_ = $line;  ⟨w0-l-c check C code for functions and variables #76 (p.43)⟩
569      if ($Enhance) {
570          $_ = $line;  ⟨w0-l-c enhance C code #79 (p.44)⟩
571      } else {
572          print "code:$line;\n";
573      }
574  }
```
(This code is used in #72 (p.41).)

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*⟩ ≡
```
575   while ($_) {
576       ⟨w0-l-c check for names  #77 (p.43)⟩
577   }
```
(This code is used in #75 (p.42).)

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

#77 ⟨*w0-l-c check for names*⟩ ≡
```
578   redo if s|^#.*$||;
579   redo if s|^\s*/\*.*\*/||;
```
(This code is extended in #77ₐ (p.43). It is used in #76 (p.43).)

We should also remove string and character literals:

#77ₐ ⟨*w0-l-c check for names* #77 (p.43)⟩ +≡
```
580   redo if s:^\s*"(\\"|[^"])*"::;
581   redo if s:^\s*'(\\'|[^'])'::;
```
(This code is extended in #77_b (p.43).)

Now we can check if we have an alphabetic name:

#77_b ⟨*w0-l-c check for names* #77 (p.43)⟩ +≡
```
582   if (s|^\s*($C_id)||o) {
583       my $id = $1;
584       ⟨w0-l-c check alphabetic name  #78 (p.43)⟩
585   }
```
(This code is extended in #77_c (p.44).)

Reserved words are ignored:

#78 ⟨*w0-l-c check alphabetic name*⟩ ≡
```
586   redo if $Res_words{$id};
```
(This code is extended in #78ₐ (p.43). It is used in #77_b (p.43).)

We may have a function call:

#78ₐ ⟨*w0-l-c check alphabetic name* #78 (p.43)⟩ +≡
```
587   if (s|^\s*\(||) {
588       print "fuse:$id;\n" unless $Special_words{$id};
589       redo;
590   }
```
(This code is extended in #78_b (p.43).)

Or, we may have a function definition:

#78_b ⟨*w0-l-c check alphabetic name* #78 (p.43)⟩ +≡
```
591   if (s|^\s*[* ]*($C_id)\s*\(||o) {
592       print "fdef:$1;\n" unless $Special_words{$1};
593       redo;
594   }
```
(This code is extended in #78_c (p.43).)

Alternatively, we may have a variable declaration list:

#78_c ⟨*w0-l-c check alphabetic name* #78 (p.43)⟩ +≡
```
595   if (s|^\s*[* ]*($C_id_list)||o) {
596       local $_;
597       foreach (split(/[*, ]+/, $1)) {
598           print "vdef:$_;\n" unless $Special_words{$_};
599       }
600       redo;
601   }
```
(This code is extended in #78_d (p.44).)

If nothing else, our alphabetic name is a variable:

#78_d ⟨*w0-l-c check alphabetic name* #78 (p.43)⟩ + ≡
```
602   print "vuse:$id;\n" unless $Special_words{$id};
603   redo;
```

Non-alphabetic names are ignored:

#77_c ⟨*w0-l-c check for names* #77 (p.43)⟩ + ≡
```
604   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*⟩ ≡
```
605   /^#/ and do { print "bicode:$_;\n"; next; };
```
(This code is extended in #79_a (p.44). It is used in #75 (p.42).)

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

#79_a ⟨*w0-l-c enhance C code* #79 (p.44)⟩ + ≡
```
606   while ($_) {
607       ⟨w0-l-c look for special words  #80 (p.44)⟩
608   }
```

However, first we check for comments:

#80 ⟨*w0-l-c look for special words*⟩ ≡
```
609   if (s|^\s*/\*.*\*/||) {
610       print "icode:$&;\n";  redo;
611   }
```
(This code is extended in #80_a (p.44). It is used in #79_a (p.44).)

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

#80_a ⟨*w0-l-c look for special words* #80 (p.44)⟩ + ≡
```
612   if (s:^\s*"(\\"|[^"])*"::) {
613       print "code:$&;\n";  redo;
614   }
615   if (s:^\s*'(\\'|[^'])'::) {
616       print "code:$&;\n";  redo;
617   }
```
(This code is extended in #80_b (p.44).)

Now, we can look for reserved words:

#80_b ⟨*w0-l-c look for special words* #80 (p.44)⟩ + ≡
```
618   if (s|^(\s*)($C_id)||o) {
619       if ($Res_words{$2}) {
620           print "code:$1;\n" if $1;
621           print "bcode:$2;\n";
622       } else {
623           print "code:$&;\n";
624       }
625       redo;
626   }
```
(This code is extended in #80_c (p.44).)

Anything else is just ordinary code:

#80_c ⟨*w0-l-c look for special words* #80 (p.44)⟩ + ≡
```
627   s:^\s*[^A-Za-z_"'/]+:: or s:^\s*.?::;
628   print "code:$&;\n" if $&;
```

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*⟩ ≡
```
629   #! ⟨perl #105 (p.54)⟩
630
631   ⟨w0-l-java definitions #82 (p.45)⟩
632   ⟨w0-l-java parameter handling #83 (p.46)⟩
633   ⟨w0-l-java read Java code #84 (p.46)⟩
634   exit 0;
635
636   ⟨user message functions #112 (p.57)⟩
```
(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*⟩ ≡
```
637   my ($Prog, $Version) = ("w0-l-java", "⟨version #107 (p.54)⟩");
```
(This code is extended in #82$_a$ (p.45). It is used in #81 (p.45).)

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

#82$_a$ ⟨*w0-l-java definitions* #82 (p.45)⟩ + ≡
```
638   my $Java_class_id = "[A-Z]([a-z_0-9]\\w*)?";
639   my $Java_other_id = "[a-zA-Z]\\w*";
```
(This code is extended in #82$_b$ (p.45).)

The same goes for an identifier list.

#82$_b$ ⟨*w0-l-java definitions* #82 (p.45)⟩ + ≡
```
640   my $Java_id_list = "$Java_other_id(\\s*,\\s*$Java_other_id)*";
```
(This code is extended in #82$_c$ (p.45).)

We need a table `%Res_words` with all the reserved words of Java.

#82$_c$ ⟨*w0-l-java definitions* #82 (p.45)⟩ + ≡
```
641   my %Res_words = ();
642   my %Special_words = ();
643   for ("abstract", "assert", "break", "case", "catch", "class", "const",
644       "continue", "default", "do", "else", "extends", "false", "final",
645       "finally", "for", "goto", "if", "implements", "import", "instanceof",
646       "interface", "native", "new", "package", "private", "protected",
647       "public", "return", "static", "strictfp", "super", "switch",
648       "synchronized", "this", "throw", "throws", "transient", "true",
649       "try", "volatile", "while")
650     { $Res_words{$_} = $Special_words{$_} = 1; }
```
(This code is extended in #82$_d$ (p.45).)

We also need a list `%Type_words` with all the type names:

#82$_d$ ⟨*w0-l-java definitions* #82 (p.45)⟩ + ≡
```
651   my %Type_words = ();
652   for ("boolean", "byte", "char", "double", "float", "int",
653       "long", "short", "void")
654     { $Type_words{$_} = $Special_words{$_} = 1; }
```
(This code is extended in #82$_e$ (p.46).)

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*⟩ ≡
```
655   PARAM:
656   while (@ARGV && $ARGV[0] =~ /^-/) { $_ = shift;
657       /^-e$/ and do { $Enhance = "yes";  next PARAM; };
658       /^-v$/ and do {
659           print STDERR "This is $Prog (version $Version)\n";
660           next PARAM; };
661   }
```
(This code is used in #81 (p.45).)

Variable $Enhance must be declared.

#82ₑ ⟨*w0-l-java definitions* #82 (p.45)⟩ +≡
```
662   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*⟩ ≡
```
663   my $line = "";
664   while (<>) {
665       unless (/^code:(.*);$/) { print;  next; };
666       chomp($line = $1);
667       $_ = $line;  ⟨w0-l-java check Java code for names  #85 (p.46)⟩
668       if ($Enhance) {
669           $_ = $line;  ⟨w0-l-java enhance Java code  #89 (p.48)⟩
670       } else {
671           print "code:$line;\n";
672       }
673   }
```
(This code is used in #81 (p.45).)

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*⟩ ≡
```
674   while ($_) {
675       ⟨w0-l-java check for names  #86 (p.46)⟩
676   }
```
(This code is used in #84 (p.46).)

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

#86 ⟨*w0-l-java check for names*⟩ ≡
```
677   redo if s|^\s*//.*||;
678   redo if s|^\s*/\*.*\*/||;
```
(This code is extended in #86ₐ (p.46). It is used in #85 (p.46).)

We should also remove string and character literals:

#86ₐ ⟨*w0-l-java check for names* #86 (p.46)⟩ +≡
```
679   redo if s:^\s*"(\\"|[^"])*"::;
680   redo if s:^\s*'(\\'|[^'])'::;
```
(This code is extended in #86ᵦ (p.46).)

Nor are we interested in package and import specifications:

#86ᵦ ⟨*w0-l-java check for names* #86 (p.46)⟩ +≡
```
681   redo if s|^\s*package\s+[^;]*;||;
682   redo if s|^\s*import\s+[^;]*;||;
```
(This code is extended in #86c (p.47).)

Class declarations are easiest to find:

#86_c ⟨*w0-l-java check for names* #86 (p.46)⟩ +≡

```
683   if (s|^\s*class\s+($Java_class_id)\b||o) {
684       print "cdef:$1;\n";
685       redo;
686   }
```
(This code is extended in #86_d (p.47).)

Then we can check if we have an alphabetic name:

#86_d ⟨*w0-l-java check for names* #86 (p.46)⟩ +≡

```
687   if (s|^\s*($Java_other_id)||o) {
688       my $id = $1;
689       ⟨w0-l-java check alphabetic name  #87 (p.47)⟩
690   }
```
(This code is extended in #86_e (p.48).)

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

#87 ⟨*w0-l-java check alphabetic name*⟩ ≡

```
691   redo if $Res_words{$id};
```
(This code is extended in #87_a (p.47). It is used in #86_d (p.47).)

We may have a constructor definition or a method call:

#87_a ⟨*w0-l-java check alphabetic name* #87 (p.47)⟩ +≡

```
692   if (s|^\s*\(||) {
693       if ($id =~ /^$Java_class_id$/o) {
694           print "cuse:$id;\n";
695       } else {
696           print "fuse:$id;\n" unless $Special_words{$id};
697       }
698       redo;
699   }
```
(This code is extended in #87_b (p.47).)

Or, we may have a method definition:

#87_b ⟨*w0-l-java check alphabetic name* #87 (p.47)⟩ +≡

```
700   if (s|^\s*($Java_other_id)\s*\(||o) {
701       my $f = $1;
702       print "cuse:$id;\n" if $id =~ /^$Java_class_id$/o;
703       print "fdef:$f;\n" unless $Special_words{$f};
704       ⟨w0-l-java check formal parameter list  #88 (p.47)⟩
705       redo;
706   }
```
(This code is extended in #87_c (p.47).)

Checking the formal parameter list is pretty straightforward:

#88 ⟨*w0-l-java check formal parameter list*⟩ ≡

```
707   if (s|^\s*[^)]*\)||) {
708       local $_;
709       foreach (split(/,/, $&)) {
710           if (/($Java_other_id)\s+($Java_other_id)/o) {
711               my $type = $1;
712               my $id = $2;
713               print "cuse:$type;\n" if $type =~ /^$Java_class_id$/o;
714               print "vdef:$id;\n";
715           }
716       }
717   }
```
(This code is used in #87_b (p.47).)

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

#87_c ⟨*w0-l-java check alphabetic name* #87 (p.47)⟩ +≡

```
718   if (($Type_words{$id}||$id=~/^$Java_class_id$/o) &&
```

```
719          s|^\s*($Java_id_list)||o) {
720      my $id_list = $1;
721      print "cuse:$id;\n" if $id =~ /^$Java_class_id$/o;
722      local $_;
723      foreach (split(/[, ]+/, $id_list)) {
724          print "vdef:$_;\n" unless $Special_words{$_};
725      }
726      redo;
727  }
```
(This code is extended in #87d (p.48).)

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

#87d ⟨*w0-l-java check alphabetic name* #87(p.47)⟩ +≡
```
728  if ($id =~ /^$Java_class_id$/o) {
729      print "cuse:$id;\n";
730  } else {
731      print "vuse:$id;\n" unless $Special_words{$id};
732  }
733  redo;
```

Non-alphabetic names are ignored:

#86e ⟨*w0-l-java check for names* #86(p.46)⟩ +≡
```
734  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*⟩ ≡
```
735  while ($_) {
736      ⟨w0-l-java look for special words or symbols #90(p.48)⟩
737  }
```
(This code is used in #84 (p.46).)

However, first we check for comments:

#90 ⟨*w0-l-java look for special words or symbols*⟩ ≡
```
738  if (s|^\s*//.*$||) {
739      print "icode:$&;\n";  redo;
740  } elsif (s|^\s*/\*.*\*/||) {
741      print "icode:$&;\n";  redo;
742  }
```
(This code is extended in #90a (p.48). It is used in #89 (p.48).)

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

#90a ⟨*w0-l-java look for special words or symbols* #90(p.48)⟩ +≡
```
743  if (s:^\s*"(\\"|[^"])*"::) {
744      print "code:$&;\n";  redo;
745  }
746  if (s:^\s*'(\\'|[^'])'::) {
747      print "code:$&;\n";  redo;
748  }
```
(This code is extended in #90b (p.49).)

Now, we can look for reserved words:

#90ᵦ ⟨*w0-l-java look for special words or symbols* #90 (p.48)⟩ +≡

```
749    if (s|^(\s*)($Java_other_id)||o) {
750        my ($space, $id) = ($1, $2);
751
752        if ($id =~ /^$Java_class_id$/o) {
753            print "code:$space;\n" if $space;
754            print "bicode:$id;\n";
755        } elsif ($Res_words{$id}) {
756            print "code:$space;\n" if $space;
757            print "bcode:$id;\n";
758        } else {
759            print "code:$space$id;\n";
760        }
761        redo;
762    }
```
(This code is extended in #90ᵧ (p.49).)

Anything else is just ordinary code:

#90ᵧ ⟨*w0-l-java look for special words or symbols* #90 (p.48)⟩ +≡

```
763    s:^\s*[^A-Za-z_"'/]+:: or s:^\s*.?:::;
764    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*⟩ ≡

```
765    #! ⟨perl #105 (p.54)⟩
766
767    ⟨w0-l-latex definitions #92 (p.49)⟩
768    ⟨w0-l-latex parameter handling #93 (p.50)⟩
769    ⟨w0-l-latex read LaTeX code #94 (p.50)⟩
770    exit 0;
771
772    ⟨user message functions #112 (p.57)⟩
```
(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*⟩ ≡

```
773    my ($Prog, $Version) = ("w0-l-latex", "⟨version #107 (p.54)⟩");
```
(This code is extended in #92ₐ (p.49). It is used in #91 (p.49).)

The syntax for LaTeX identifiers is use more than once so it is an advantage to name it.

#92ₐ ⟨*w0-l-latex definitions* #92 (p.49)⟩ +≡

```
774    my $LaTeX_id = "\\\\([A-Za-z@]+|.)";
```
(This code is extended in #92ᵦ (p.49).)

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

#92ᵦ ⟨*w0-l-latex definitions* #92 (p.49)⟩ +≡

```
775    my ($Bcode, $Icode) = ("code", "code");
```
(This code is extended in #92ᵧ (p.50).)

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*⟩ ≡

```
776  PARAM:
777  while (@ARGV && $ARGV[0] =~ /^-/) { $_ = shift;
778      /^-e$/ and do { $Bcode = "bcode";  $Icode = "icode";
779          next PARAM; };
780      /^-v$/ and do {
781          print STDERR "This is $Prog (version $Version)\n";
782          next PARAM; };
783  }
```

(This code is used in #91 (p.49).)

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

#92$_c$ ⟨*w0-l-latex definitions* #92 (p.49)⟩ +≡

```
784  my ($Bcode, $Icode) = (0, 0);
```

2.6.3.3 Reading the LaTeX code Now it's time to read the LaTeX code. All tokens read will be printed, but only the `code` requires further handling.

#94 ⟨*w0-l-latex read LaTeX code*⟩ ≡

```
785  while (<>) {
786      unless (/^code:(.*);$/) { print;  next; };
787      chomp($_ = $1);
788      ⟨w0-l-latex check LaTeX code  #95 (p.50)⟩
789  }
```

(This code is used in #91 (p.49).)

Is there any LaTeX code that requires attention?

#95 ⟨*w0-l-latex check LaTeX code*⟩ ≡

```
790  while (/[%\\]/) {
791      print "code:$`;\n" if $`;
792      $_ = $&.$';
793      ⟨w0-l-latex check for comments  #96 (p.50)⟩
794      ⟨w0-l-latex check for declarations  #97 (p.50)⟩
795      ⟨w0-l-latex check for use  #98 (p.51)⟩
796  }
797  print "code:$_;\n" if $_;
```

(This code is used in #94 (p.50).)

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*⟩ ≡

```
798  /^%/ and do { print "$Icode:$_;\n";  $_ = "";  last; };
```

(This code is used in #95 (p.50).)

2.6.3.5 Look for declarations New LaTeX commands are declared using \newcommand, \renewcommand, or \def.

#97 ⟨*w0-l-latex check for declarations*⟩ ≡

```
799  s/^(\\(re)?newcommand\s*\*?\s*\{)(.+?)(\})// and do {
800      print "code:$1;\n$Bcode:$3;\ncode:$4;\n";
801      print "fdef:$3;\n";
802      next; };
803  s/^(\\def\s*)($LaTeX_id)//o and do {
804      print "code:$1;\n$Bcode:$2;\n";
805      print "fdef:$2;\n";  next; };
```

(This code is extended in #97$_a$ (p.51). It is used in #95 (p.50).)

New environments are declared using \newenvironment or \renewenvironment.

#97ₐ ⟨*w0-l-latex check for declarations* #97 (p.50)⟩ +≡
```
806  s/^(\\(re)?newenvironment\s*\{)(.+?)(\})// and do {
807      print "code:$1;\n$Bcode:$3;\ncode:$4;\n";
808      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*⟩ ≡
```
809  s/^\\begin\s*\{(.+?)\}// and do {
810      print "code:$&;\n";
811      print "vuse:$1;\n";  next; };
```
(*This code is extended in #98ₐ (p.51). It is used in #95 (p.50).*)

The corresponding \end is ignored.

#98ₐ ⟨*w0-l-latex check for use* #98 (p.51)⟩ +≡
```
812  s/^\\end\s*\{(.+?)\}// and do {
813      print "code:$&;\n";   next; };
```
(*This code is extended in #98ᵦ (p.51).*)

Then we look for use of ordinary LaTeX commands.

#98ᵦ ⟨*w0-l-latex check for use* #98 (p.51)⟩ +≡
```
814  s/^$LaTeX_id//o and do {
815      print "code:$&;\n";
816      print "fuse:$&;\n";  next; };
```
(*This code is extended in #98ᵧ (p.51).*)

If the "\" does not start a legal LaTeX command, we will not speculate about why this is so, but just print it as a normal character.

#98ᵧ ⟨*w0-l-latex check for use* #98 (p.51)⟩ +≡
```
817  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 varibales 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*⟩ ≡
```
818  #! ⟨perl #105 (p.54)⟩
819
820  ⟨w0-l-perl definitions #100 (p.52)⟩
821  ⟨w0-l-perl parameter handling #101 (p.52)⟩
822  ⟨w0-l-perl read Perl code #102 (p.52)⟩
823  exit 0;
824
825  ⟨user message functions #112 (p.57)⟩
```
(*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*⟩ ≡
826 `my ($Prog, $Version) = ("w0-l-perl", "`⟨*version* #107 (p.54)⟩`");`
(This code is extended in #100ₐ (p.52). It is used in #99 (p.51).)

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

#100ₐ   ⟨*w0-l-perl definitions* #100 (p.52)⟩ +≡
827 `my $Perl_id = "[A-Za-z]\\w*";`
(This code is extended in #100_b (p.52).)

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*⟩ ≡
828 `PARAM:`
829 `while (@ARGV && $ARGV[0] =~ /^-/) { $_ = shift;`
830 `    /^-e$/ and do { $Enhance = "yes";  next PARAM; };`
831 `    /^-v$/ and do {`
832 `        print STDERR "This is $Prog (version $Version)\n";`
833 `        next PARAM; };`
834 `}`
(This code is used in #99 (p.51).)

The state variables must be declared.

#100_b   ⟨*w0-l-perl definitions* #100 (p.52)⟩ +≡
835 `my $Enhance = 0;`
(This code is extended in #100_c (p.53).)

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*⟩ ≡
836 `my $line;`
837 `while (<>) {`
838 `    unless (/^code:(.*);$/) { print;  next; };`
839 `    chomp($line = $1);`
840 `    $_ = $line;  `⟨*w0-l-perl check Perl code for functions and variables* #103 (p.52)⟩
841 `    if ($Enhance) {`
842 `        $_ = $line;  `⟨*w0-l-perl enhance Perl code* #104 (p.53)⟩
843 `    } else {`
844 `        print "code:$line;\n";`
845 `    }`
846 `}`
(This code is used in #99 (p.51).)

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 **&**.[11]

#103   ⟨*w0-l-perl check Perl code for functions and variables*⟩ ≡
847 `while (s/\bsub\s+($Perl_id)//o) { print "fdef:&$1;\n"; }`
848 `while (s/&($Perl_id)//o) { print "fuse:&$1;\n"; }`
(This code is extended in #103ₐ (p.53). It is used in #102 (p.52).)

---

[11]More lax notation is permitted in Perl, but checking for that becomes too difficult by far.

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.52)⟩ +≡
```
849  while (s/\@($Perl_id)//o) { print "vuse:\@$1;\n" unless $1 eq "_"; }
850  while (s/\$($Perl_id)\s*\[//o) { print "vuse:\@$1;\n" unless $1 eq "_"; }
```
(This code is extended in #103ᵦ (p.53).)

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

    %var   or $var{...}

#103ᵦ ⟨*w0-l-perl check Perl code for functions and variables* #103 (p.52)⟩ +≡
```
851  while (s/%($Perl_id)//o) { print "vuse:%$1;\n"; }
852  while (s/\$($Perl_id)\s*\{//o) { print "vuse:%$1;\n"; }
```
(This code is extended in #103ᵤ (p.53).)

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.

#103ᵤ ⟨*w0-l-perl check Perl code for functions and variables* #103 (p.52)⟩ +≡
```
853  while (s/\$($Perl_id)//o) { print "vuse:\$$1;\n" unless $1 eq "_"; }
```

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*⟩ ≡
```
854  /^\s*#/ and do { print "icode:$_;\n"; next; };
```
(This code is extended in #104ₐ (p.53). It is used in #102 (p.52).)

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

#104ₐ ⟨*w0-l-perl enhance Perl code* #104 (p.53)⟩ +≡
```
855  while (/\b($Res_Perl)\b/o) {
856      print "code:$`;\n" if $`;
857      print "bcode:$&;\n";
858      $_ = $';
859  }
860  print "code:$_;\n" if $_;
```

The search pattern $Res_Perl is composed from all the reserved words in Perl.[12]

#100ᵤ ⟨*w0-l-perl definitions* #100 (p.52)⟩ +≡
```
861  my @Res_word = ("and", "continue", "die", "do", "dump", "else",
862      "elsif", "eval", "exec", "exit", "for", "foreach", "fork",
863      "if", "kill", "last", "next", "or", "return", "sub",
864      "unless", "until", "wait", "while");
865  my $Res_Perl = join("|", @Res_word);
```

---

[12]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.

2.7 Miscellaneous

2.7.1 The Perl interpreter

The macro named ⟨*perl*⟩ defines the location of the Perl interpreter. This definition may be modified if necessary.

#105 ⟨*perl*⟩ ≡
866 /usr/bin/perl ⟨*perl utf-8 specification* #106 (p.54)⟩
(This code is extended in #105ₐ (p.54). It is used in #1 (p.12), #8 (p.15), #18 (p.19), #32 (p.25), #46 (p.31), #72 (p.41), #81 (p.45), #91 (p.49), and #99 (p.51).)

The option –CSD specifies use of UTF-8 character encoding as default.

#106 ⟨*perl utf-8 specification*⟩ ≡
867 –CSD
(This code is used in #105 (p.54).)

Using the strict option reduces the number of errors.

#105ₐ ⟨*perl* #105 (p.54)⟩ + ≡
868 use strict;

2.7.2 Program version

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

#107 ⟨*version*⟩ ≡
869 2.0.4
(This code is used in #2 (p.12), #9 (p.15), #19 (p.22), #33 (p.26), #47 (p.31), #73 (p.41), #82 (p.45), #92 (p.49), #100 (p.52), #118 (p.58), and #126 (p.66).)

2.8 Adapting text for processing by LATEX

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

The function &Latexify translates a text into LATEX code. It accepts up to three parameters:

1. the original text,
2. an indication whether the spaces in the text are breakable (by default, they are not, as this parameter is optional) and
3. an indication that the text may contain embedded LATEX commands (which they normally won't, as this parameter is optional).

#108 ⟨*latex generation functions*⟩ ≡
870 **sub** Latexify {
871   local $_ = shift;
872   my $break_spaces = shift;
873   my $allow_latex = shift;
874
875   ⟨*Latexify: handle embedded LaTeX* #109 (p.55)⟩
876   ⟨*Latexify: adapt text* #110 (p.55)⟩
877   **return** $_;
878 }
(This code is used in #46 (p.31).)

When using this function, please note that it assumes that the LATEX package textcomp has been loaded.

2.8.1 Handling embedded LaTeX code

The user may be allowed to embed LaTeX commands in the text by placing it in a pair of vertical bars, as in

    Pu|\v{z}|ar

#109 ⟨*Latexify: handle embedded LaTeX*⟩ ≡
```
879   if ($allow_latex && /\|(.*?)\|/) {
880       return &Latexify($`,$break_spaces,$allow_latex) . $1 .
881           &Latexify($',$break_spaces,$allow_latex);
882   }
```
(This code is used in #108 (p.54).)

2.8.2 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[13] — makes this possible.

#110 ⟨*Latexify: adapt text*⟩ ≡
```
883   s/\\/\\textbackslash%temporaer bakslask%/g;
884   s/\{/\\{/g;
885   s/\}/\\}/g;
886   s/\\textbackslash%temporaer bakslask%/\\textbackslash{}/g;
```
(This code is extended in #110$_a$ (p.55). It is used in #108 (p.54).)

2.8.3 Handle the other special LaTeX characters

The seven other special LaTeX characters all have well-known command equivalents.

#110$_a$ ⟨*Latexify: adapt text* #110 (p.55)⟩ +≡
```
887   s/\#/\\\#/g;
888   s/\$/\\\$/g;
889   s/\%/\\\%/g;
890   s/\&/\\\&/g;
891   s/\_/\\\_/g;
892   s/\^/\\textasciicircum{}/g;
893   s/\~/\\textasciitilde{}/g;
```
(This code is extended in #110$_b$ (p.56).)

2.8.4 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
¬	¬	^{1}	₁
±	±	μ	µ
^{2}	₂	×	×
^{3}	₃	÷	÷

---

[13]The text is in Norwegian to be even safer.

#110_b ⟨*Latexify: adapt text* #110 (p.55)⟩ + ≡

```
894  s/¬/\\textlnot{}/g;  s/±/\\textpm{}/g;  s/²/\\texttwosuperior{}/g;
895  s/³/\\textthreesuperior{}/g;  s/¹/\\textonesuperior{}/g;
896  s/µ/\\textmu{}/g;  s/×/\\texttimes{}/g;  s/÷/\\textdiv{}/g;
```
(This code is extended in #110_c (p.56).)

2.8.5 Avoing unwanted ligatures

Most LaTeX fonts contain some ligatures that we do not want because they are totally different characters.

Original	Ligature	Original	Ligature
<<	«	>>	»
--	–	---	—
!`	¡	?`	¿
``	"	''	"
,,	„		

#110_c ⟨*Latexify: adapt text* #110 (p.55)⟩ + ≡

```
897  s/<</<\\null</g;
898  s/>>/>\\null>/g;
899  s/---/-\\null-\\null-/g;
900  s/--/-\\null-/g;
901  s/!`/!\\null`/g;  s/\?`/\?\\null`/g;
902  s/``/`\\null`/g;  s/''/'\\null'/g;  s/,,/,\\null,/g;
```
(This code is extended in #110_d (p.56).)

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

2.8.6 Handle blanks

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

#110_d ⟨*Latexify: adapt text* #110 (p.55)⟩ + ≡

```
903  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].)

#111 ⟨*expand TAB characters*⟩ ≡

```
904  while (s/\t+/' 'x(length($&)*8-length($`)%8)/e) {};
```
(This code is used in #23 (p.23).)

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`.

#112 ⟨*user message functions*⟩ ≡
```
905   sub Error {
906       &Message(@_);
907       &Tidy_up if defined &Tidy_up;
908       exit 1;
909   }
```
(This code is extended in #112ₐ (p.57). It is used in #1 (p.12), #8 (p.15), #18 (p.19), #32 (p.25), #46 (p.31), #72 (p.41), #81 (p.45), #91 (p.49), and #99 (p.51).)

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.

#112ₐ ⟨*user message functions* #112 (p.57)⟩ + ≡
```
910   sub Message {
911       print STDERR "$Prog: ", shift, "\n";
912       while (@_) {
913           print STDERR " "x(2+length $Prog), shift, "\n";
914       }
915   }
```

2.11 Alphabetical sorting

Alphabetical is quite simple in Perl, but we need a small modification to handle Perl variable names like $a or @Names.

#113 ⟨*alphabetical sorting*⟩ ≡
```
916   sub alphabetically {
917       ⟨alphabetical sorting: simple first tests  #114 (p.57)⟩
918       ⟨alphabetical sorting: test initial character  #115 (p.57)⟩
919       ⟨alphabetical sorting: test other characters  #116 (p.58)⟩
920   }
```
(This code is used in #46 (p.31).)

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

#114 ⟨*alphabetical sorting: simple first tests*⟩ ≡
```
921   return 0 if $a eq $b;
922   return -1 unless $a;
923   return 1 unless $b;
```
(This code is used in #113 (p.57).)

If neither text is empty, we examine the initial characters for the special Perl notation. If they are identical, we can ignore them.

#115 ⟨*alphabetical sorting: test initial character*⟩ ≡
```
924   if ($a=~/^[$@%&]/ && $b=~/^[$@%&]/) {
925       return substr($a,0,1) cmp substr($b,0,1)
926           if substr($a,0,1) ne substr($b,0,1);
927       local($a, $b) = (substr($a,1), substr($b,1));
928       &alphabetically;
929   }
```
(This code is used in #113 (p.57).)

Then we may sort according to the internal representation, which works fine for English.

#116 ⟨*alphabetical sorting: test other characters*⟩ ≡
930   **return** lc($a) cmp lc($b);
  (This code is used in #113 (p.57).)

3   LaTeX support

This part describes the webzero package and the webzero document class used when *web₀* documents are typeset with LaTeX.

3.1   The webzero package

This package contains the necessary definitions for including *web₀* documentation into any LaTeX document.

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

#117 ⟨*webzero.sty*⟩ ≡
931   ⟨*webzero.sty identification* #118 (p.58)⟩
932   ⟨*webzero.sty options* #119 (p.58)⟩
933   ⟨*webzero.sty package loading* #120 (p.60)⟩
934   ⟨*webzero.sty main code* #121 (p.61)⟩
  (This code is not used.)

3.1.1   Package identification

Every LaTeX package should contain version information.

#118 ⟨*webzero.sty identification*⟩ ≡
935   \NeedsTeXFormat{LaTeX2e}[1994/12/01]
936   \ProvidesPackage{webzero}[2020/10/04 v⟨*version* #107 (p.54)⟩
937   Ifi package **for** web0 documents]
  (This code is used in #117 (p.58).)

3.1.2   Package options

The webzero package recognizes these options:

3.1.2.1   The options `american` and `USenglish` are used when the document is written in American English. This is the default.

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

#119 ⟨*webzero.sty options*⟩ ≡
938   \DeclareOption{american}{%
939   \def \wzclassindexname{Classes}%
940   \def \wzfuncindexname{Functions}%
941   \def \wzlastsep{, **and** }%
942   \def \wzmacroindexname{Macro names}%
943   \def \wzmacroindexstartext{(Macro names marked with ∗ are not
944   used internally.)}%
945   \def \wzsep{, }%
946   \def \wztwosep{ **and** }%
947   \def \wzvarindexname{Variables}%
948   \def \wz@extendedname{extended in}%
949   \def \wz@filename{File}%
950   \def \wz@itisname{It is}%
951   \def \wz@notusedname{not used}%
952   \def \wz@pagename{page}%

```
953    \def \wz@shortpagename{p.}%
954    \def \wz@thiscodename{This code is}%
955    \def \wz@usedname{used in}}
956 \DeclareOption{USenglish}{\ExecuteOptions{american}}
```
(This code is extended in #119$_a$ (p.59). It is used in #117 (p.58).)

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

#119$_a$ ⟨*webzero.sty options* #119 (p.58)⟩ + ≡
```
957    \DeclareOption{english}{\ExecuteOptions{american}}
958    \DeclareOption{UKenglish}{\ExecuteOptions{english}}
```
(This code is extended in #119$_b$ (p.59).)

3.1.2.3 The option `normalsize` will produce program code in normal type size.[14] The leading is reduced, however.

#119$_b$ ⟨*webzero.sty options* #119 (p.58)⟩ + ≡
```
959    \DeclareOption{normalsize}{\def \wz@codesize{\small}%
960      \def \wz@codestretch{0.9}}
```
(This code is extended in #119$_c$ (p.59).)

3.1.2.4 The option `norsk` is used when the document is written in Norwegian "Bokmål".

#119$_c$ ⟨*webzero.sty options* #119 (p.58)⟩ + ≡
```
961    \DeclareOption{norsk}{%
962      \def \wzclassindexname{Klasser}%
963      \def \wzfuncindexname{Funksjoner}%
964      \def \wzlastsep{ og }%
965      \def \wzmacroindexname{Makronavn}%
966      \def \wzmacroindexstartext{(Makronavn merket med * er ikke
967                              brukt internt.)}%
968      \def \wzsep{, }%
969      \def \wztwosep{ og }%
970      \def \wzvarindexname{Variable}%
971      \def \wz@extendedname{utvidet i}%
972      \def \wz@filename{Fil}%
973      \def \wz@itisname{Den blir}%
974      \def \wz@notusedname{ikke brukt}%
975      \def \wz@pagename{side}%
976      \def \wz@shortpagename{s.}%
977      \def \wz@thiscodename{Denne koden blir}%
978      \def \wz@usedname{brukt i}}
```
(This code is extended in #119$_d$ (p.59).)

3.1.2.5 The option `nynorsk` is used when the document is written in Norwegian "Nynorsk".

#119$_d$ ⟨*webzero.sty options* #119 (p.58)⟩ + ≡
```
979    \DeclareOption{nynorsk}{\ExecuteOptions{norsk}%
980      \def \wzfuncindexname{Funksjonar}%
981      \def \wzmacroindexname{Makronamn}%
982      \def \wzmacroindexstartext{(Makronamn merkte med * er ikkje
983                              nytta internt.)}%
984      \def \wz@extendedname{utvida i}%
985      \def \wz@itisname{Han vert}%
986      \def \wz@notusedname{ikkje nytta}%
```

---

[14]Actually, the code for `normalsize` is set in `\small` which looks better with normal size text.

```
987    \def \wz@thiscodename{Denne koden vert}%
988    \def \wz@usedname{nytta i}}
```
(This code is extended in #119$_e$ (p.60).)

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

#119$_e$ ⟨*webzero.sty options* #119 (p.58)⟩ + ≡
```
989    \DeclareOption{sf}{\def \wz@family{\sffamily}}
```
(This code is extended in #119$_f$ (p.60).)

3.1.2.7 The option small   will produce program code in a type size smaller than `normalsize`.[15] The leading must be adjusted accordingly.

#119$_f$ ⟨*webzero.sty options* #119 (p.58)⟩ + ≡
```
990    \DeclareOption{small}{\def \wz@codesize{\footnotesize}%
991    \def \wz@codestretch{0.95}}
```
(This code is extended in #119$_g$ (p.60).)

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

#119$_g$ ⟨*webzero.sty options* #119 (p.58)⟩ + ≡
```
992    \DeclareOption{tt}{\def \wz@family{\ttfamily}}
```
(This code is extended in #119$_h$ (p.60).)

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

#119$_h$ ⟨*webzero.sty options* #119 (p.58)⟩ + ≡
```
993    \ExecuteOptions{american,normalsize,tt}
994    \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 65.

#120 ⟨*webzero.sty package loading*⟩ ≡
```
995    \RequirePackage{calc,ifthen}
```
(This code is extended in #120$_a$ (p.60). It is used in #117 (p.58).)

The `relsize` package is used when typesetting macro numbers; see Section 3.1.7.3 on page 65.

#120$_a$ ⟨*webzero.sty package loading* #120 (p.60)⟩ + ≡
```
996    \RequirePackage{relsize}
```
(This code is extended in #120$_b$ (p.60).)

As mentioned in Section 2.8 on page 54, the `textcomp` package is required.

#120$_b$ ⟨*webzero.sty package loading* #120 (p.60)⟩ + ≡
```
997    \RequirePackage{textcomp}
```


3.1.4 Implementation of interface

These command constitute the standard package interface.

---

[15]The actual font size for the code will be `\footnotesize`; for an explanation, see footnote no 14.

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

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

Starting a new macro 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.[16]

- Print the macro number and its name, followed by a "≡" or a "+ ≡". The subsequent line change is preceded by a `\nobreak` to avoid widow lines.

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

#121 ⟨*webzero.sty main code*⟩ ≡
```
998    \newcommand{\wzdef}[3]{\par
999      \ifthenelse{\parskip>0}{\vspace{\parskip}}{\medskip}
1000     \begingroup
1001     \renewcommand{\baselinestretch}{\wz@codestretch}%
1002     \wz@codesize\wz@family
1003     \setlength{\parindent}{1em}
1004     \setlength{\parskip}{0pt plus 0.3pt minus 0.1pt}\frenchspacing
1005     \noindent
1006     \llap{\normalfont\bfseries {\wz@num{#2}{#3}}\hspace*{1em}}%
1007     \ifthenelse{#3=0}{\wzmacro{#1}~$\equiv$}%
1008       {\wzmacro[#2]{#1}~$+\!\equiv$}\label{w0-#2-#3}%
1009     \wzeol[\nobreak]}
```
(This code is extended in #121$_a$ (p.62). It is used in #117 (p.58).)

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

1. the macro's number,
2. its extension number,
3. value `1` if the macro 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.

---

[16]The standard LaTeX 2_ε document classes use a different scheme to avoid such messages: it ensures that the `\textheight` is equal to an integral number of `\baselineskips`. This produces better results for regular texts like novels, but I prefer the stretch and shrink method for documents with greatly varying font sizes.

#121ₐ ⟨*webzero.sty main code* #121(p.61)⟩ + ≡
1010   \newcommand{\wzenddef}[4]{%
1011   \ifthenelse{#2=0}{⟨*webzero.sty: info on base definition* #123(p.62)⟩}
1012   {⟨*webzero.sty: info on extended definition* #122(p.62)⟩}\par
1013   \endgroup
1014   \ifthenelse{\parskip>0}{}{\medskip}
1015   ⟨*suppress indentation of subsequent paragraph* #124(p.62)⟩}
(This code is extended in #121ᵇ(p.62).)

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

#122 ⟨*webzero.sty: info on extended definition*⟩ ≡
1016   \ifthenelse{#3=0}{}
1017   {\wz@info{\wz@extendedname\ \wz@numandpage[#3]{#1}}}
(This code is used in #121ₐ(p.62).)

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

#123 ⟨*webzero.sty: info on base definition*⟩ ≡
1018   \wz@info{\ifthenelse{#3=0}{}
1019   {\wz@extendedname\ \wz@numandpage[#3]{#1}. \wz@itisname\ }%
1020   \ifthenelse{\equal{#4}{}}{\wz@notusedname}
1021   {\wz@usedname\ #4}{}}
(This code is used in #121ₐ(p.62).)

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

#121ᵇ ⟨*webzero.sty main code* #121(p.61)⟩ + ≡
1022   \newcommand{\wz@info}[1]{\hspace*{-\wzext}*[0.2ex]
1023   \textsl{\rmfamily\footnotesize (\wz@thiscodename\ #1.)}}
(This code is extended in #121ᶜ(p.62).)

We want to suppress any indentation of the paragraph immediately following a macro definition. The code to do this was found in Section A of the LATEX 2ε source code.

#124 ⟨*suppress indentation of subsequent paragraph*⟩ ≡
1024   \everypar{{\setbox0=\lastbox}\everypar{}}
(This code is used in #121ₐ(p.62).)

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 preceding page.

#121ᶜ ⟨*webzero.sty main code* #121(p.61)⟩ + ≡
1025   \newcommand{\wzeol}[1][]{\hspace*{-\wzext}\par #1\leavevmode
1026   \addtocounter{wz@lnum}{1}%
1027   \llap{\normalfont\tiny \thewz@lnum \hspace*{\parindent}}}
(This code is extended in #121ᵈ(p.62).)

The line counter wz@lnum must be declared.

#121ᵈ ⟨*webzero.sty main code* #121(p.61)⟩ + ≡
1028   \newcounter{wz@lnum}
(This code is extended in #121ᵉ(p.62).)

The length \wzext specifies how far the code lines may extend into the right-hand margin. The default is 0 pt.

#121ᵉ ⟨*webzero.sty main code* #121(p.61)⟩ + ≡
1029   \newlength{\wzext}
(This code is extended in #121ᶠ(p.63).)

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.

#121$_f$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡
1030   \newcommand{\wzfile}[1]{\markright{\wz@filename: \textsl{#1}}}
(This code is extended in #121$_g$(p.63).)

3.1.4.5 Macro names are typeset using the \wzmacro command. This command is used in macro 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 macro's number and page where first defined will be included.

#121$_g$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡
1031   \newcommand{\wzmacro}[2][0]{$
1032   \langle \mbox{\it #2\ifthenelse{#1=0}{}{~~\smaller[2]\upshape
1033   \wz@numandpage{#1}}\/} \rangle $}
(This code is extended in #121$_h$(p.63).)

Previously, this command was called \wzmeta, and the old name is kept for campatablity reasons.

#121$_h$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡
1034   \let \wzmeta = \wzmacro
(This code is extended in #121$_i$(p.63).)

3.1.5 Typesetting the index

The wzindex environment is used for typesetting the variable, function, and macro name indices. It has two parameters:

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

#121$_i$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡
1035   \newenvironment{wzindex}[2]%
1036   {\ifnum #2=2 \twocolumn[\section*{#1}]**\else** \onecolumn \section*{#1}\fi
1037   \markboth{\MakeUppercase{#1}}{\MakeUppercase{#1}}
1038   \begingroup
1039   \vspace*{4pt}
1040   \setlength{\emergencystretch}{3cm}
1041   \setlength{\parfillskip}{0pt}
1042   \setlength{\parindent}{0pt}%
1043   \setlength{\parskip}{1pt plus 1pt}
1044   \small \sloppy \hbadness = \tolerance }%
1045   {\onecolumn \endgroup }
(This code is extended in #121$_j$(p.63).)

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.

#121$_j$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡
1046   \newcommand{\wzinitial}[1]{\vspace{16pt plus 4pt}
1047   {\raggedright \textbf{\large #1}\par}\vspace*{2pt plus 1pt minus 0.5pt}}
(This code is extended in #121$_k$(p.63).)

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).

#121$_k$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡
1048   \newcommand{\wzul}[1]{\underline{#1}}
(This code is extended in #121$_l$(p.64).)

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.

#121$_l$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡
```
1049    \newcommand{\wzindexsep}{\rlap{,}\hspace{0.5em plus 1em minus 0.1em}}
```
(This code is extended in #121$_m$ (p.64).)

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

#121$_m$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡
```
1050    \newcommand{\wzx}[1]{\setlength{\hangindent}{2em}%
1051      \hangafter=1 {\wz@family #1}\hspace*{0.5em}\dotfill}
```
(This code is extended in #121$_n$ (p.64).)

3.1.5.5 The macro `\wzxref` This macro is used when referencing other macros.

#121$_n$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡
```
1052    \newcommand{\wzxref}[2]{\wz@numandpage[#2]{#1}}
```
(This code is extended in #121$_o$ (p.64).)

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

#121$_o$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡
```
1053    \newcommand{\wzlongpageref}[1]{ \wz@pagename~%
1054      \makebox[2em][r]{\pageref{w0-#1-0}}}
```
(This code is extended in #121$_p$ (p.64).)

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.

#121$_p$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡
```
1055    \newcommand{\ps@webzero}{%
1056      \renewcommand{\@evenhead}{}\let \@oddhead = \@evenhead
1057      \renewcommand{\@evenfoot}{\rightmark\hfill
1058        \wz@pagename\space\thepage}%
1059      \let \@oddfoot = \@evenfoot
1060      \renewcommand{\sectionmark}[1]{}%
1061      \renewcommand{\subsectionmark}[1]{}}
```
(This code is extended in #121$_q$ (p.65).)

3.1.7 Utility macroes

3.1.7.1 The name _web$_0$_ is generated by the macro `\webzero`. It will try to use the `\webzerologo` command to produce the proper logo, but the hyperref package is used, we also provide a much plainer version of the logo for the bookmarks.

The `\webzerologo` command uses the same trick as the definition of `\LaTeXe` in the LaTeX 2_ε source code to decide when the subscript should be bold.

#121$_q$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡

```
1062    \DeclareRobustCommand{\webzerologo}{\mbox{\m@th
1063      \if b\expandafter\@car\f@series\@nil \boldmath \fi
1064      $\mathsf{web}_{\mathsf{O}}$}}
1065    \newcommand{\webzero}{\@ifundefined{texorpdfstring}%
1066      {\webzerologo}%
1067      {\texorpdfstring{\webzerologo}{web0}}}
```

(This code is extended in #121$_r$(p.65).)

3.1.7.2 Extended alphabetical numbering The extensions are numbered

$$a, b, \ldots, z, aa, ab, \ldots, az, ba, \ldots$$

(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.)

#121$_r$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡

```
1068    \newcommand{\wz@alpha}[1]{%
1069      \ifthenelse{#1<27}
1070      {\setcounter{wz@temp}{#1}\alph{wz@temp}}
1071      {\begingroup
1072        \count\wz@val = #1\relax
1073        \setcounter{wz@temp}{(#1-1)/26}\wz@alpha{\thewz@temp}%
1074        \setcounter{wz@temp}{\the\count\wz@val -
1075          (\the\count\wz@val-1)/26*26}\alph{wz@temp}%
1076      \endgroup }}
```

(This code is extended in #121$_s$(p.65).)

The two counters must be declared.

#121$_s$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡

```
1077    \newcounter{wz@temp}  \newcount\wz@val
```

(This code is extended in #121$_t$(p.65).)

3.1.7.3 Typesetting a macro number A macro number with its extension is typeset as

$$\#4_c$$

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

#121$_t$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡

```
1078    \newcommand{\wz@num}[2]{\##1%
1079      \ifthenelse{#2>0}{\raisebox{-0.4ex}{\smaller[2]%
1080        \hspace*{-0.01em}\wz@alpha{#2}}}{}}
```

(This code is extended in #121$_u$(p.65).)

3.1.7.4 Typesetting a macro number with page reference The LaTeX command \wz@numandpage prints the number of a macro (and its extension number if provided by an optional parameter) together with the number of the page on which it was defined:

$$\#4_c \; \text{(p.14)}$$

#121$_u$ ⟨*webzero.sty main code* #121(p.61)⟩ +≡

```
1081    \newcommand{\wz@numandpage}[2][0]{\wz@num{#2}{#1}%
1082      \hspace*{0.2em}(\wz@shortpagename\pageref{w0-#2-#1})}
```

(This code is extended in #121$_v$(p.66).)

3.1.8 End of class

It is common in LaTeX 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.

#121ᵥ ⟨*webzero.sty main code* #121 (p.61)⟩ $+\equiv$
1083 \endinput

3.2 The webzero document class

Since most *web₀* 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.

#125 ⟨*webzero.cls*⟩ $\equiv$
1084 ⟨*webzero.cls identification* #126 (p.66)⟩
1085 ⟨*webzero.cls options* #127 (p.66)⟩
1086 ⟨*webzero.cls package and class loading* #128 (p.66)⟩
1087 ⟨*webzero.cls main code* #129 (p.67)⟩
1088 ⟨*webzero.cls end* #130 (p.67)⟩
(This code is not used.)

3.2.1 Class identification

#126 ⟨*webzero.cls identification*⟩ $\equiv$
1089 \NeedsTeXFormat{LaTeX2e}[1994/12/01]
1090 \ProvidesClass{webzero}[2019/10/16 v⟨*version* #107 (p.54)⟩ Ifi class **for** web0 documents]
(This code is used in #125 (p.66).)

3.2.2 Class options

This document class will recognize the options known to the webzero package (see Section 3.1 on page 58) and send them on. All other options are passed on to the article class.

#127 ⟨*webzero.cls options*⟩ $\equiv$
1091 \DeclareOption{american}{\PassOptionsToPackage{american}{webzero}}
1092 \DeclareOption{english}{\PassOptionsToPackage{english}{webzero}}
1093 \DeclareOption{norsk}{\PassOptionsToPackage{norsk}{webzero}}
1094 \DeclareOption{nynorsk}{\PassOptionsToPackage{nynorsk}{webzero}}
1095 \DeclareOption{sf}{\PassOptionsToPackage{sf}{webzero}}
1096 \DeclareOption{tt}{\PassOptionsToPackage{tt}{webzero}}
1097 \DeclareOption{UKenglish}{\PassOptionsToPackage{UKenglish}{webzero}}
1098 \DeclareOption{USenglish}{\PassOptionsToPackage{USamerican}{webzero}}
1099 \DeclareOption*{\PassOptionsToClass{\CurrentOption}{article}}
1100 \ProcessOptions \relax
(This code is used in #125 (p.66).)

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.

#128 ⟨*webzero.cls package and class loading*⟩ $\equiv$
1101 \LoadClass{article}
1102 \RequirePackage{webzero}
(This code is used in #125 (p.66).)

3.2.4 Main code

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

#129 ⟨*webzero.cls main code*⟩ ≡
1103 `\addtolength{\textwidth}{3cm}`
1104 `\addtolength{\evensidemargin}{-1.5cm}`
1105 `\addtolength{\oddsidemargin }{-1.5cm}`
 (This code is extended in #129$_a$ (p. 67). It is used in #125 (p. 66).)

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

#129$_a$ ⟨*webzero.cls main code* #129 (p. 67)⟩ + ≡
1106 `\addtolength{\topmargin}{-2.8cm}`
1107 `\addtolength{\textheight}{4.5cm}`
 (This code is extended in #129$_b$ (p. 67).)

We want to use the `webzero` page style.

#129$_b$ ⟨*webzero.cls main code* #129 (p. 67)⟩ + ≡
1108 `\pagestyle{webzero}`
 (This code is extended in #129$_c$ (p. 67).)

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

#129$_c$ ⟨*webzero.cls main code* #129 (p. 67)⟩ + ≡
1109 `\let \ps@plain = \ps@webzero`

3.2.5 End of class

And that's all, folks.

#130 ⟨*webzero.cls end*⟩ ≡
1110 `\endinput`
 (This code is used in #125 (p. 66).)

4 Documentation

Since the *web$_0$* 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 headline and the usual parts.

#131 ⟨*man tangle0*⟩ ≡
1111 `.TH TANGLE0 1 "`⟨*man page date* #132 (p. 67)⟩`"`
1112 ⟨*man tangle0 name* #133 (p. 68)⟩
1113 ⟨*man tangle0 description* #134 (p. 68)⟩
1114 ⟨*man tangle0 parameters* #135 (p. 68)⟩
1115 ⟨*man author* #140 (p. 69)⟩
1116 ⟨*man see also* #141 (p. 70)⟩
 (This code is not used.)

The date identifies the current version.

#132 ⟨*man page date*⟩ ≡
1117 `17 October 2013`
 (This code is used in #131 (p. 67) and #136 (p. 68).)

4.1.1 Identification

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

#133   ⟨*man tangle0 name*⟩ ≡
```
1118   .SH NAME
1119   tangle0 - a web0 tool for extracting program code
```
(This code is used in #131 (p.67).)

4.1.2 Program description

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

#134   ⟨*man tangle0 description*⟩ ≡
```
1120   .SH SYNOPSIS
1121   .B tangle0
1122   .RI [-o " file" ]
1123   [-v]
1124   .RI [-x " name" ]
1125   .RI [ file... ]
```
(This code is extended in #134ₐ (p.68). It is used in #131 (p.67).)

Then comes a longer desciption of what the program does.

#134ₐ   ⟨*man tangle0 description* #134 (p.68)⟩ + ≡
```
1126   .SH DESCRIPTION
1127   .I Tangle0
1128   is part of the
1129   .B web0
1130   package. It is used to extract the program code from a
1131   .B web0
1132   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.

#135   ⟨*man tangle0 parameters*⟩ ≡
```
1133   .SS OPTIONS
1134   .I Tangle0
1135   accepts the following options:
1136   ⟨tangle0 man page parameters  #5 (p.13)⟩
```
(This code is used in #131 (p.67).)

4.2 Man page for weave0

The man page consists of the standard headline and the usual parts.

#136   ⟨*man weave0*⟩ ≡
```
1137   .TH WEAVE0 1 "⟨man page date  #132 (p.67)⟩"
1138   ⟨man weave0 name  #137 (p.69)⟩
1139   ⟨man weave0 description  #138 (p.69)⟩
1140   ⟨man weave0 parameters  #139 (p.69)⟩
1141   ⟨man author  #140 (p.69)⟩
1142   ⟨man see also  #141 (p.70)⟩
```
(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.

#137 ⟨*man weave0 name*⟩ ≡
    1143 `.SH NAME`
    1144 `weave0 – a web0 tool `**`for`**` producing program documentation`
    (This code is used in #136 (p.68).)

4.2.2 Program description

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

#138 ⟨*man weave0 description*⟩ ≡
    1145 `.SH SYNOPSIS`
    1146 `.B weave0`
    1147 `.RI "[-e] [-f " filter ]`
    1148 `.RI [-l " language" ]`
    1149 `.RI [-o " file" ]`
    1150 `[-v]`
    1151 `.RI [ file... ]`
    (This code is extended in #138$_a$ (p.69). It is used in #136 (p.68).)

Then comes a longer desciption of what the program does.

#138$_a$ ⟨*man weave0 description* #138 (p.69)⟩ + ≡
    1152 `.SH DESCRIPTION`
    1153 `.I Weave0`
    1154 `is part of the`
    1155 `.B web0`
    1156 `package. It is used to produce the program documentation from a`
    1157 `.B web0`
    1158 `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.

#139 ⟨*man weave0 parameters*⟩ ≡
    1159 `.SS OPTIONS`
    1160 `.I Weave0`
    1161 `accepts the following options:`
    1162 ⟨*weave0 man page parameters* #12 (p.16)⟩
    (This code is used in #136 (p.68).)

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.

#140 ⟨*man author*⟩ ≡
    1163 `.SH AUTHOR`
    1164 `Dag Langmyhr, Department of Informatics, University of Oslo.`
    (This code is used in #131 (p.67) and #136 (p.68).)

4.3.2 Cross reference information

Those who read this manual page will quite probably be interested in the complete documentation on the web_0 system.

#141 ⟨*man see also*⟩ ≡
```
1165  .SH "SEE ALSO"
1166  .I The Web0 System
1167  by Dag Langmyhr; available on
1168  .I http://dag.at.ifi.uio.no/littprog/web0.pdf.
```
(This code is used in #131 (p.67) and #136 (p.68).)

References

[Knu83]   Donald E. Knuth. *Literate Programming*. Tech. rep. STAN-CS-82-981. Stanford, CA 94305: Stanford University, Sept. 1983.

[Knu84]   Donald E. Knuth. 'Literate Programming'. In: *The Computer Journal* 27.2 (1984), pp. 97–111.

[Knu92]   Donald E. Knuth. *«Literate programming»*. Center for the study of language and information, 1992.

[Lev87]   Silvio Levy. 'WEB adapted to C'. In: *TUGboat* 8.1 (Apr. 1987), pp. 12–13.

[Ram89]   Norman Ramsay. 'Literate programming: weaving a language-independent WEB'. In: *Communications of the ACM* 32.9 (Sept. 1989), pp. 1051–1055.

[WCS96]   Larry Wall, Tom Christiansen and Randal L. Scwarz. *Programming Perl*. 2nd ed. Known as "The camel book". O'Reilly & associates, 1996.

Functions

Macro names

74

(Macro names marked with * are not used internally.)