Literate Programming in C

The CWEB System of Structured Software Documentation

Manual for CWEBx3.0

Marc A. A. van Leeuwen

1 Overview

This document describes CWEBx3.0, a particular implementation of CWEB, a system that supports the concept of "literate programming" for programs written in the language C (more particularly for this version, in ANSI/ISO C). As this manual aims to supply all information possibly relevant to a wide variety of users, it is necessarily rather extensive. However, CWEB is not a complicated system, and just a few simple commands suffice for practical programming purposes; these are discussed in the section 4 (the remainder serves mainly to allow fine-tuning of the presentation of the printed documents describing literate programs).

As the somewhat contrived name of the system indicates, CWEBx is not the only version of CWEB; indeed it is based on an older version of CWEB by Silvio Levy, which is an adaptation to C of the WEB system (for Pascal) written by Donald E. Knuth, the founder of literate programming. That CWEB system has independently evolved into a system that is currently distributed, under joint responsibility of Levy and Knuth, as CWEB 3.4. Those with experience using Levy/Knuth CWEB will already be acquainted with most aspects of CWEBx, and may turn to section 9 for a summary of the differences between the two systems; however, CWEBx also provides a compatibility mode (selectable by specifying '+c' on the command line) in which it should be able to process C programs written using Levy/Knuth CWEB without any modification.

The structure of this manual is as follows. In section 2 an exposition of the ideas underlying the concept of literate programming is given, and a description of how systems of the WEB family provide concrete tools to support this programming methodology. This section is directed particularly to those who are new to literate programming; it explains the purpose of CWEB and the logical connection between the various elements that literate programming adds with respect to traditional programming. As an illustration we then give a small example program using CWEB in section 3. The main commands of CWEB, which define the structure of the source text and tell the CWEB tools what to do with the various pieces of that text, are discussed in section 4. In section 5 we discuss how the behaviour of CWEB tools can be further affected by means of command line options and file name arguments supplied when the programs are invoked. In section 6 we discuss some facilities for distributing the source text over several input files, including a "change file" that allows applying local patches without affecting the original source files; this is not directly related to literate programming, but can be quite useful in larger projects. In section 7 the remaining minor CWEB commands are explained, and section 8 discusses some features of the TEX format employed by CWEB, which can be used to affect the appearance of CWEB documents. Section 9 is devoted to a comparison with Levy/Knuth CWEB; finally section 10 summarises all CWEB codes recognised in the source text.

2 About literate programming

Literate programming is a concept that was developed, implemented, and propagated by D. E. Knuth in the early 1980's, as a natural sequel to the concept of "structured programming" that had caused a revolution in the world of software development about a decade earlier. (At this moment, another decade further, one may conclude that literate programming has not caused a similar revolution, since many programmers practicing illiterate programming do not feel at all as guilty about this as they would if they were to be found practicing unstructured programming.) So let us first consider the idea of structured programming more closely.

Structured programming Without attempting a definition of the term, it seems fair to say that structured programming involves designing a program by hierarchical decomposition of the task at hand, and constructing a program that has a similar decomposition into parts, where each part "solves" the corresponding subtask. The subdivision of the program manifests itself in its division into subroutines (procedures, functions), and at the more detailed level in the syntactic composition of those routines (control structures, blocks); this explains why much emphasis is placed on the use of subroutines, and why languages with a linear program model and (conditional) jumps as their main control structure (like assembly language) do not form a natural vehicle for structured programming. The question as to which criteria should be used in subdividing problems into smaller ones is difficult to answer in general, but a good rule of thumb is that at the level of abstraction at which a task is defined it should be possible to give a reasonably simple informal description of its subtasks, in which a new level of detail about the method used to perform the task is given. (It would be nice if the subtasks also had simple formal specifications, but in general this is too much to ask, since informal descriptions do not only abstract from implementation details, they may also conceal numerous obvious specification details).

It can be said that the idea of structured programming, undone of any dogmatism that has been associated with it, has been rather universally accepted, and has proven to be an effective methodology in software development. Other methodologies have been put forward, such as data encapsulation, but this has been in addition to rather than as an alternative for structured programming. Literate programming however is related in a different way to structured programming, as it concerns not the contents of the program itself, but rather the way it is presented. Literate programming presupposes structured programming, but is independent of other programming paradigms; any program that has been designed in a structured way can in principle be rendered as a literate program, without requiring a change of the program text itself.

Limitations of traditional structured programming Although the composition of a structured program should reflect the design decisions that led to its construction, the traditional way of presenting such programs lacks appropriate facilities for communicating this information effectively to readers of the program, seriously limiting the readability, especially to people other than the programmer of the code. Yet readability is of vital importance, because it is only by careful reading that we can verify that the design of a program is sound and well-implemented, and to understand where and how changes can be made when such a need arises. Of course the code may be documented by adding an arbitrary amount of comments, but there are various reasons why this has a limited effectiveness, so that in practice the level of documentation is nearly always (much) lower than would be desirable from the point of view of maintainability.

The syntactic decomposition of a piece of a structured program into a hierarchy of control structures, compound statements, etc., is crucial to understanding the way it functions, yet the human eye is much less capable of performing this task then a parser is, even when proper indentation is applied. The difficulty rapidly increases with the size of the program fragment, and can become a serious factor when this size exceeds a dozen or two lines (depending on the complexity of the structure). And even when we can recognise the structure, the meaning of the individual parts cannot be immediately perceived, but must be derived from close inspection of the code, or from the comments. However, as comments are localised in the code, and it is hard to attach a comment clearly to a construct of some considerable extent. Also, adding too many comments may actually decrease the readability of the program, by making the structure recognition problem worse. Even indentation, useful as it may be, can be more of a nuisance than of any help when it becomes so deep that it forces code and comments to be squeezed together into (or fall off) the right margin. Finally the fact that program sources are usually plain text files, represented in a single rather crude font, does not improve human readability. The limitations of the character set hamper formulation of comments, where special symbols, formulae, tables or illustrations might convey the information much

more effectively; in the program itself the eye has to do without visual clues marking the distinction between various types of program elements (identifiers, keywords, constants, literal strings, operators, comments, etc.). As a consequence of all this, few people will find much pleasure in reading source listings, even if the program is well designed and documented, and possibly even contains some interesting and subtle algorithms.

Many of these problems would not be too grave if all subroutines were severely limited in size and complexity. However, although it might be possible to live up to such a restriction for certain kinds of programs (in particular if the task involves mostly simple actions of administrative nature rather than any really complicated algorithms), it would be a very impractical requirement in general, certainly for procedural programming languages. First of all, having to break up a subroutine using auxiliary subroutines solely because of the size of the code, violates the basic principle that such a decomposition should be the result of design decisions. Furthermore, there are several technical reasons why such a decomposition could be either impossible, or might involve a large amount of additional code that has little to do with the actual task being performed, and that might deteriorate performance unacceptably. Examples of such reasons are for instance the need to perform a large multi-way branching, to have local variables that are visible throughout the execution of a complex algorithm, or to have the possibility to jump out of nested structures on certain (error) conditions (while the language might not allow jumps out of a subroutine into the routine calling it). Finally, introducing many small subroutines, for reasons that cannot be described easily outside the immediate context in which they will be used, creates a serious problem of giving them sensible names and remembering the tasks they perform.

Concluding we may say that structured programming in its traditional form does not encourage or even allow the level of documentation that would be desirable for maintenance and intelligibility by people other than the author of the program.

Requirements for literate programming The basic idea of literate programming is to take a fundamentally different starting point for the presentation of programs to human readers, without any direct effect on the program as seen by the computer. Rather than to present the program in the form in which it will be compiled (or executed), and to intercalate comments to help humans understand what is going on (and which the compiler will kindly ignore), the presentation focuses on explaining to humans the design and construction of the program, while pieces of actual program code are inserted to make the description precise and to tell the computer what it should do. The program description should describe parts of the algorithm as they occur in the design process, rather than in the completed program text. For reasons of maintainability it is essential however that the program description defines the actual program text; if this were defined in a separate source document, then inconsistencies would be almost impossible to prevent. If programs are written in a way that concentrates on explaining their design to human readers, then they can be considered as works of (technical) literature; it is for this reason that Knuth has named this style of software construction and description "literate programming". More background information about this concept and its history concept can be found in Knuths book "Literate Programming", CSLI Lecture notes #27, Leland Stanford Junior University, 1992 (ISBN 0-937073-81-4).

From the discussion above it will be clear that traditional programming languages are not directly suitable for literate programming. We shall now try to formulate requirements for a system that supports literate programming. Doing so we shall especially keep in mind programs that evolve after their original design, possibly altering certain parts of that design, and possibly being realised by different persons, since it is in such cases that the benefits of literate programming are particularly crucial.

The documentation parts of the program description should allow for the same freedom of expression that one would have in an ordinary technical paper. This means that the document describing the program should consist of formatted text, rather than being a plain text file. This does not exclude the possibility that the source is written as a plain text file, but then it should undergo some form of processing to produce the actual program description. The document should moreover contain fragments of a program written in some traditional (structured) programming language, in such a way that they can be mechanically extracted and arranged into a complete program; in the formatted document on the other hand layout and choice of fonts for these program fragments should be so as to maximise readability.

Parts of the program that belong together logically should appear near to each other in the description, so that they are visible from the part of the documentation that discusses their function. This means that

it should be possible to rearrange program text with respect to the order in which it will be presented to the computer, for otherwise the parts that deal with the actions at the outer level of a subroutine will be pushed apart by the pieces specifying the details of inner levels. The most obvious and natural way to do this is to suppress the program text for those inner levels, leaving an outline of the outer level, while the inner levels may be specified and documented elsewhere; this is a bit like introducing subroutines for the inner levels, but without the semantic implications that that would have. There should be no restrictions on the order in which the program fragments resulting from this decomposition are presented, so that this order can be chosen so as to obtain an optimal exposition; this may even involve bringing together fragments whose location in the actual program is quite unrelated, but which have some logical connection.

Obviously there should be a clear indication of where pieces of program have been suppressed, and which other program fragments give the detailed specifications of those pieces. From the programming language point of view the most obvious method of identification would be to use identifiers, resulting in a simple system of parameterless macros, with as only unusual aspect that uses of the macro are allowed to precede the definition, and indeed do so more often than not. Actually, literate programming uses a method that differs from this only trivially from a formal standpoint, but has a great advantage in practical terms: identification is by means of a more or less elaborate phrase or sentence, marked in a special way to indicate that it is a reference to a program fragment. This description both stands for the fragment that is being specified elsewhere, and also serves as a comment describing the function of that fragment at a level of detail that is appropriate for understanding the part of the program containing it. In this way several purposes are served at once: a clear identification between use and definition is established, the code at the place of use is readable because irrelevant detail is suppressed, with a relevant description of what is being done replacing it, and at the place of definition a reminder is given of the task that the piece of code presented is to perform. The documenting power of such a simple device is remarkable. In some cases the result is so clear that there is hardly any need to supply further documentation; also it can sometimes be useful to use this method to replace small pieces of somewhat cryptic code by a description that is actually longer than the code itself. It is hard to give a sharp limit on the length of the description for a program fragment, but if substantially more than a sentence is needed, say a full paragraph, then the fragment probably does not represent a well chosen abstraction, which might be an indication that the design of the program has some room for improvement. On the other hand, it is good practice to explicitly mention any unusual control flow that might be caused by executing the abstracted fragment, like jumping to a label outside the fragment, since such information is vital for a proper understanding of the program at the place where the fragment is used.

WEB systems for literate programming Until now we have discussed literate programming as a general concept, independent of any particular implementation; this was done to stress the generality of the idea. We shall now indicate how these ideas are realised by "WEB systems", a family of systems that were modelled after Knuth's original `WEB`, and of which `CWEB` is a member. In these systems the program source is written as a plain text file, and a pair of programs is provided, which transform this source into other text files suitable for processing by a compiler respectively by a typesetting program. Other kinds of literate programming tools are conceivable (e.g., ones that would provide the programmer with a direct graphical representation of the typeset document while editing, possibly with hypertext facilities), but this approach has the advantage of being fairly simple and portable across many platforms. The approach is not the simplest possible however, as a substantial part of the work done by the tools deals with transforming the program fragments from their plain text form into typeset text with proper fonts and layout (i.e., with pretty-printing); this part of the task also depends essentially on the programming language being used. By reverting to verbatim representation of program fragments one could make simple tools that support literate programming in a language independent way—and indeed such tools exist—but then a price is paid in terms of readability. We should also note that WEB systems support writing documents whose purpose is to simultaneously specify and document a program; if one is primarily writing a theoretic paper, in which only occasionally pieces of program text are mentioned, then one might prefer a slightly different kind of system that does not impose as much global structure on the document as WEB systems do.

Not surprisingly, WEB systems satisfy all the requirements for literate programming formulated above, and they do so in a fairly straightforward manner. A typeset WEB document consists of a sequence of consecutively numbered *sections*, whose size is typically less than half a page. Each section may contain

a program fragment, called a *module*, preceded by a commentary in ordinary text, although in some cases either one of these parts might be absent. (The programming language used for the program fragments depends on the particular WEB system used, as does the typesetting system that eventually produces the printed document; for the CWEB system described here these are respectively C and TEX.) In most cases a module is headed by a text in angle brackets called its *module name*, which gives a description of the task it performs. This name is followed by '≡' and the program code that constitutes the module itself; this is called a defining occurrence of the module name. (We make a distinction between the words 'section' and 'module', using the former for a numbered portion of the WEB document, and the latter for a named portion of the program described by it.) A module name can also be used in the body of some other module, either before or after its definition in the document, to represent the corresponding piece of program text. WEB facilitates locating the definition of a module from the place where it is used by automatically incorporating the number of the defining section in the module name. The actual program text is then constructed by recursively replacing module names by the text specified in their definition (this should of course follow the grammatical structure of the program, lest the most basic principles of literate programming be violated). A program fragment occurring at the outermost level is distinguished by the fact that it is not headed by a module name.

Most module names will have just one defining occurrence, and will also be referenced just once; in both cases there may however be exceptions, where a module name has more than one occurrence of the indicated kind. If a module is multiply referenced, this simply means that the corresponding part of the program text is repeated identically in more than one place. If a module name has more than one defining occurrence, then the text of the corresponding module is obtained by concatenating the program fragments of all its definitions in the order in which they occur in the document. In a similar fashion all fragments without module name are combined into an "unnamed module". These are the only occasions where the order in which the sections are given can have any effect on the final program; apart from this the literate programmer has complete freedom of ordering the sections in a way that facilitates understanding the program as much as possible.

We need not discuss all aspects of WEB systems here, but a few points that contribute to literate programming by improving readability should be mentioned. The proper formatting of all program fragments is automatically taken care of by the WEB system, providing a uniform style of presentation. The system also provides a large amount of cross-reference information; this greatly facilitates reading the program and searching for specific pieces of code. Not only is the number of the (first) section defining a module incorporated in its name, but in that defining section indications are also given of the section(s) in which the module is used, and possibly of any further defining occurrences of the same module name. At the end of the document an alphabetically sorted index is added, listing for each identifier all the sections in which it is defined or used; the programmer may also add additional entries to the index by indicating in the program source that certain sections should be referenced for particular terms. A list of all module names is also given, which may help locating the part of the program dealing with some issue. So in many ways the WEB system tries to aid human understanding of the program, but of course the literacy of the programmer will always remain the decisive factor in this respect.

3 What a CWEB program looks like

Enough now of general considerations, let us turn our attention to the CWEB system this manual is really about. The best way to learn about it is probably to read a CWEB document. Therefore we include a small but complete CWEB program below; the program is about as small as possible without rendering a decomposition into modules pointless. The example is not intended as a showpiece of programming literacy, but it demonstrates various aspects of the system. The index at the end of the program is included, but not the list of the four module names in this program or the table of contents. One will notice that some symbols appear to be different from their official representation in C, for instance the assignment operator '`=`' appears as '⇐', the equality operator '`==`' as '=', the logical "and" and "or" operators '`&&`' and '`||`' as '∧' and '∨' respectively, the variable '`f1`' as 'f_1', and the null pointer '`NULL`' as '⊙'; thus the possibilities of the typesetting system are used to improve the appearance of the program.

1. Comparing text files. This is an entirely trivial program, that tests whether two text files are equal, and if not so, points out the first point of difference.

```
#include <stdio.h>
#include <stdlib.h>
  typedef char bool;
```

2. The outline of the program is simple. We read characters from both input files into c_1 and c_2 until the comparison is complete. Line and column counts are maintained in *line* and *col*.

```
#define left_margin 1    /* leftmost column number; change to 0 if you prefer */
  ⟨ Functions 5 ⟩
  int main (int n, char **arg)
  { FILE *f1, *f2;    /* the two input files */
    int c1, c2, col ⇐ left_margin;
    long line ⇐ 1;
    ⟨ Open the files f1 and f2, taking their names from the command line or from the terminal; in case of
        an error for which no recovery is possible, call exit(1) 6 ⟩
    ⟨ Search for first difference, leaving c1 ≠ c2 if and only if a difference was found 3 ⟩
    ⟨ Report the outcome of the comparison 4 ⟩
    return 0;    /* successful completion */
  }
```

3. The heart of the program is this simple loop. When we reach the end of one of the files, the files match if and only if the other file has also reached its end. For this reason the test $c_1 = c_2$, which requires characters to be read from both files, must precede the test for file end; when only one file ends, it is the former test which breaks the loop.

```
⟨ Search for first difference, leaving c1 ≠ c2 if and only if a difference was found 3 ⟩ ≡
  while ((c1 ⇐ getc(f1)) = (c2 ⇐ getc(f2)) ∧ c1 ≠ EOF)
    if (c1 = '\n') { ++line; col ⇐ left_margin; } else ++col;
```

This code is used in section 2.

4. When the first difference occurs at the end of one of the files, or at the end of a line, we give a message indicating this fact.

```
⟨ Report the outcome of the comparison 4 ⟩ ≡
  if (c1 = c2) printf("Files␣match.\n");
  else
  { printf("Files␣differ.\n");
    if (c1 = EOF ∨ c2 = EOF)
      { the_file(c1 = EOF); printf("is␣contained␣in␣the␣other␣as␣initial␣segment.\n"); }
    else if (c1 = '\n' ∨ c2 = '\n')
      { the_file(c1 = '\n'); printf("has␣a␣shorter␣line␣number␣%ld␣than␣the␣other.\n", line); }
    else printf("First␣difference␣at␣line␣%ld,␣column␣%d.\n", line, col);
  }
```

This code is used in section 2.

5. The function *the_file* starts a sentence about the first or second file, depending on its boolean argument.

```
⟨ Functions 5 ⟩ ≡
  void the_file (bool is_first) { printf("The␣%s␣file␣", is_first ? "first" : "second"); }
```

See also section 7.

This code is used in section 2.

6. There can be be zero, one or two command line arguments. If there are none, the user is prompted to supply them, and if there are two these are taken as the file names, prompting the user only in case a file could not be opened. In case just one argument is present, the first file is assumed to be the standard input, which does not have to be opened; in this case however we will not read a file name from terminal in case the second file cannot be opened.

```
#define read_mode "r"
⟨ Open the files f1 and f2, taking their names from the command line or from the terminal; in case of an
      error for which no recovery is possible, call exit(1) 6 ⟩ ≡
  −−n; ++arg;      /* ignore "argument" 0, which is the program name */
  if (n = 0)
    { open_file(&f1, "First␣file␣to␣compare", ⊙); open_file(&f2, "Second␣file␣to␣compare", ⊙); }
  else if (n = 1)
  { f1 ⇐ stdin;
    if ((f2 ⇐ fopen(*arg, read_mode)) = ⊙) { printf("Could␣not␣open␣file␣%s.\n", *arg); exit(1); }
  }
  else if (n = 2)
  { open_file(&f1, "Give␣another␣first␣file", *arg++);
    open_file(&f2, "Give␣another␣second␣file", *arg);
  }
  else { printf("No␣more␣than␣two␣command␣line␣arguments␣are␣allowed.\n"); exit(1); }
```

This code is used in section 2.

7. The function *open_file* will try to open the file *name* for reading, and if this fails it will prompt for another file name until it has success. If called with *name* = ⊙, the function starts with prompting right away.

```
⟨ Functions 5 ⟩ +≡
  void open_file (FILE **f, char *prompt, char *name)
  { char buf[80];
    if (name = ⊙ ∨ (*f ⇐ fopen(name, read_mode)) = ⊙)
      do { printf("%s:␣", prompt); fflush(stdout); scanf("%79s", buf); }
      while ((*f ⇐ fopen(buf, read_mode)) = ⊙);
  }
```

8. Index.

Some remarks about the example program Reading the program should not cause great problems to anyone familiar with the C language, once one gets used to the representation of the symbols. We mention a number of points that will have become clear in the course of the example.

The commentary text at the beginning of the sections is set in ordinary paragraphs, which contrasts sufficiently with the appearance of the program text that the dividing line between the two can be easily perceived, even though it is only marked by a bit of white space. In case the section defines (part of) a named module, the module name heading the program fragment is set flush left, and is followed by '≡', or in case this is not the first defining occurrence of that name, by '+≡' (therefore, the occurrence of the module name '⟨ Functions 5 ⟩' in §2 is not a defining one, whereas the occurrences of that name in §5 and §7 are). The style in which the module names and comments contained in the program fragments are set is similar to that of ordinary text; indeed if they are too long to fit on the line, they will be broken across lines (with proper care taken to respect the indentation level). In CWEB embedded comments are always attached to the right of a program element (usually a statement or declaration); in the example we can see there is relatively little need for embedded comments, because of the other means provided for documentation. An embedded comment that is split across lines will not look very good, and should only occur in cases of emergency; in most cases it is better to use the commentary part at the beginning of the section for any elaborate explanation. On the other hand, long module names (occupying up to about four lines) are not uncommon when the task performed by the module calls for an extensive description.

As one can see in the example, it is common to refer to small pieces of C code (in most cases just variables or simple expressions) from within the commentaries, module names and comments. The CWEB system makes it easy to include such pieces, by providing a variant of the formatting routines used for the actual program fragments (differing from them by the omission of any 2-dimensional layout features such as indentation). In many cases the pieces of C code are so simple that they could easily be typeset directly (using TEX's math mode), producing the same formatted output without using the facilities of CWEB. But even then it is preferable to use CWEB instead, because it will then guarantee that all identifiers mentioned in such a way in the documentation part of a section or in a comment, will be included in the index at the end of the program. Although in many cases a reference would have been generated anyway by the program fragment in the same section (as happens in all cases for our example program), this mechanism ensures that even remarks about the use of variables and functions made in sections that contain no program fragment at all can be traced from the index. Incidentally, identifiers that are used only in a module name are not indexed, which is why there is no reference to §2 in the index entry for *exit*.

When an index entry is recorded, whether from within a program fragment or a piece of C code embedded in text, the occurrence may be flagged as 'defining', depending on the context; this happens for instance in the case of parameters in an ANSI/ISO style function heading, of variable declarations and of labels. If at least one occurrence of an identifier in some section is a defining one, then the corresponding section number in the index entry for that identifier will be underlined. Single-letter identifiers, the special identifier NULL (appearing as '⊙'), and keywords of the language are considered so ubiquitous that no index references for them are generated, except those that are underlined; e.g., in the example there is no reference to §6 for the variable *n*. For keywords this means that they will not appear in the index at all (unless the programmer explicitly marks certain occurrences as defining); note however that identifiers defined in a **typedef** declaration (like **bool** in the example) will be indexed, even though they are set in boldface just like keywords are.

Further attributes of CWEB ***programs*** An aspect of CWEB programs that does not stand out very clearly in our miniature example is that it allows sets of related sections to be grouped together into "chapters". Each chapter is identified by its title, which appears in boldface after the number of its first section; in our example sections 1 and 8 start new chapters. The division into chapters has a few more effects on the document, which were suppressed in our example, since they would interfere with the overall structure of this manual: each chapter starts on a fresh page, its title appears in the running head of all its pages, and all chapter titles are collected in the table of contents. (Style changes such as employed in this manual are easy to obtain, since the style is not determined by the CWEB system, but rather by a separate format consisting of TEX macros; a few small changes to standard format can change the overall appearance of the document, and it would be equally easy to change for instance the page size or the symbols used to represent operators.)

There is one important point left to explain about the example, which is the special position of the lines starting with **#define** and **#include**. Although they look like ordinary preprocessor lines, which could have been included in the program fragments, they are in fact separate items that are given between the documentation part and the program part of a section (this can be seen best in §6), forming a third type of constituent of sections (although in most sections they will be absent). Their place in CWEB is less distinctive then that of their analogues in WEB systems for languages that have no preprocessor (like the original WEB for Pascal, which provides a separate macro facility itself): indeed the directives are just passed on to the C preprocessor. Yet there is some advantage in specifying them as special items to CWEB, and in most cases using these facilities is preferable to embedding the directives in the C program fragments.

One reason is that one usually wants the effects of preprocessor directives to be visible throughout the C file that is generated, while this would not always be the case if they were specified inside the program fragments; for instance if the definition of *read_mode* in §6 had been included in the program fragment, it could not have been validly used in §7, because that section will precede §6 in the C file produced. This difficulty could be overcome by collecting all macro definitions in a module that is used at the start of the program and defined in many sections throughout the CWEB document. In fact this is just about how CWEB treats the separately specified preprocessor directives: they are collected in order of appearance, and placed at the very beginning of the C file. (Some other place of insertion for the preprocessor directives can be specified by means of a pseudo-module named '⟨ Preprocessor directives ⟩', but this is quite rare.) Since a section can define only one module, the CWEB facility for preprocessor directives may help avoid having to split up sections merely because they contain such a directive. Furthermore, an important reason to specify **#include** directives to CWEB, is that this allows it to inspect those header files for any typedef declarations, so that programs can be formatted properly; without this programs using typedef identifiers defined in header files would seriously confuse the syntax analysis that CWEB performs, resulting in very poor quality formatting of program fragments.

Preprocessor directives other than those mentioned above can only be incorporated in a program by including them in an ordinary program module, but there is relatively little need for such directives. In situations where one would use conditional compilation in ordinary C, one can usually use the "change file" mechanism provided by CWEB instead (this will be discussed below), especially if it involves system dependent modifications; this has the advantage that such modifications do not affect the main source files, and only those modifications that are actually applied will be visible in the CWEB document. In the rare cases that one does include a preprocessor directive in a program fragment, the fact that it is not being specified as a separate item to CWEB is usually easy to recognise in the CWEB document, because the module name being defined or some program text precedes it; however even if this should not be the case then such embedded directives can still be distinguished by a slight difference in horizontal and vertical spacing.

Output to multiple files There is one important construction one may encounter in CWEB documents, that we have not mentioned yet. There may be module names that consist of a file name in typewriter type, like '⟨ `common.h` 14 ⟩'; usually such module names are nowhere referenced, but only have one or more defining occurrences. CWEB documents containing such a module will produce a file of that name in addition to the C program that is normally produced. The module bearing the name of the file will form the root module of the C code written to that file, in the same way as the unnamed module forms the root module for the ordinary output. This feature is particularly useful for the production of header files that can be included by other compilation units (and even by the program produced as main output). It allows one for instance to state function prototype declarations that go to the header file and the matching function definitions in the C program in the immediate vicinity of one another within the CWEB document. The module with the file name can refer to submodules, and so on to any depth, just like the modules contributing to the main output. This possibility should be used with some restraint however, lest readers have difficulty finding out to which file the program fragment defined by some module will be sent. The preprocessor lines that are handled by CWEB will normally only become part of the main program output, not of any additional output files; this provides one valid reason for sometimes bypassing the facilities of CWEB, and incorporating **#define** and **#include** directives directly into program modules.

4 How to create a CWEB program

In the previous section we have explained how one should read CWEB documents; in this section we shall discuss how they can be written. The "CWEB document" we have been discussing is the printed text that is eventually produced from the source file written by the programmer, but that file does not look quite like the printed version; on the other hand the difference in appearance is not so great that there is any difficulty finding the place in the source file corresponding to some part of the printed text.

The general setup The programmer creates a plain text file using the format explained below, which contains both program fragments and commentary, and has file name extension '`.w`'; e.g., the file from which the example above was produced is `compare.w` (it is included in the CWEBx distribution). The CWEB system consists of two utility programs '`CTANGLE`' and '`CWEAVE`' that can be applied to this source file. In order to create an executable program, one issues the command '`ctangle compare`', which will read the file `compare.w` and write a file `compare.c` containing the corresponding C program. This file can then be processed in the ordinary way by any C compiler to produce an executable program. To produce a printed document on the other hand, one issues the command '`cweave compare`', which will again read the file `compare.w`, and this time write a file `compare.tex`. This file serves as input for the typesetting program TEX: by giving the command '`tex compare`' it will be processed, and the result is a file `compare.dvi`. This file can be either previewed or converted to hardcopy output by the system dependent programs for this purpose that accompany TEX. Despite the somewhat elaborate processing trajectories, it will become apparent that the programmer has good control over the final result produced in both cases.

A word of explanation about the names of CWEB and its constituent programs. The initial C's stand for the programming language, of course; the rest of the names are the same as those chosen by Knuth for the original WEB system (which existed long before the World Wide Web). The CWEB language allows one to separately describe small parts of a C program and their interconnections, both formal (via module references) and informal (by some semantic relationship); with some fantasy this evokes the image of a web of connected pieces. These parts are linearised quite differently in their presentation for human readability than in the "official" form in which they are presented to the C compiler, and it is the program CTANGLE that does the somewhat complicated reordering to obtain the latter from the former. This process is traditionally called "tangling" the code, although one could also call it untangling if one prefers formal to human order. The CWEAVE program intertwines the TEX and C parts of the source text and "weaves" them together like warp and weft, resulting in a beautifully formatted document. Despite these pretty metaphors, you will be forgiven if you sometimes get these names mixed up.

This general organisation of CWEB has some immediate consequences. First of all, one needs to have an operational TEX system and (not surprisingly) a C compiler in order to use CWEB; the CWEB programs form only a comparatively small part of the utilities needed. Second, the CWEB language must be such that both valid C code and TEX input can be derived mechanically from it, which are rather different formats. Nevertheless the CWEB language is quite simple: this is because for almost all of the CWEB source text the required format is either that of TEX or that of C. The main function of the specific CWEB commands is to structure the source file and determine which parts of the input will be processed further in what way. Finally, a somewhat unfortunate consequence of CWEB's setup is that errors may be detected by any one of CTANGLE, the C compiler, CWEAVE and TEX. The knowledge about C and TEX built into the CWEB programs is far from sufficient to ensure that they will always produce error-free output code, although of course they do their best not to introduce any errors themselves. A bright point in the case of C errors, is that the **#line** directives produced by CTANGLE enable the compiler to refer directly to lines in the CWEB source file in its errors messages, rather than to the intermediate C file (but TEX does not have a similar facility).

It follows from these facts that the CWEB programmer must be acquainted both with C and with TEX; however, the depth of the knowledge required it not the same in both cases. Obviously, one cannot write a computer program without a good understanding of the programming language used, but a very superficial knowledge of TEX will suffice: in most cases no TEXpertise beyond the basic facts in chapters 2–6 of *The TEXbook* is required (but please don't skip chapter 2, as only too many people have done). The reason for this is that one rarely needs to instruct TEX to do sophisticated formatting. It is true that the proper typesetting of computer programs is a subtle matter, but it is precisely this part that is taken care of by CWEAVE (even for references to C constructs in the commentary), and the programmer can just concentrate

on writing syntactically correct C code. On the other hand the full power of TEX is available if one wishes to use it, for instance to illuminate the program with things like complicated tables, or math formulae of a different nature than those occurring in a computer program.

Since the CWEB commands deal only with the structure of the source file, not with its contents, they can be very brief: they consist of '@' followed by one other character, and are commonly referred to as *control codes*. For instance, '@ ' (i.e., '@' followed by white space) indicates the start of a new section, and '@c' marks the start of the C part of a section that contributes to the unnamed module. Control codes may placed at any position within the source lines, although it is customary to place the ones defining the coarse structure of the source file at the beginning of a line for better visibility. In some cases a control code marks the beginning of a piece of text that will be interpreted by CWEB in a special way, as for instance '@<' which starts a module name; the end of these *control texts* is always marked by the special code '@>'. The character '@' was selected because it is quite uncommon both in C and in TEX source code, but in those cases where one does need to pass on the character itself (e.g., in C strings and comments) it should be written as '@@'.

We now discuss the various control codes, grouped by their function. Here we shall treat only the most important control codes, which are used regularly in ordinary programs. Treatment of a number of additional control codes, that either serve for fine tuning in special cases, or are intended to allow emergency fixes in unforeseen cases, is deferred to a later section, in order not to confuse novice CWEB users. For the codes that are discussed, we do however provide full details of their use; most of these can be skipped on first reading. A summary of all CWEB control codes can be found an the end of this manual.

Sectioning codes: '@*', '@ ', '@~' The most important control codes are those that specify the division of the CWEB program into sections. There are three codes that indicate the start of a new section, and are therefore called sectioning codes. Each of them has a slightly different effect, and each section must start with one of them (i.e., a section is never implicitly started). The three sectioning codes are '@*', '@ ', and '@~', of which the second one is the most commonly used. No section numbers should be given in the source file: these will be automatically computed and inserted by CWEB. A tab or newline following '@' is considered equivalent to a space, and for any of these three control codes, (further) white space separating it from the TEX text that follows is ignored, as long as there is no completely blank line (which TEX would interpret as the end of the paragraph that started with the section number).

A section starting with '@*' will start a new chapter of the CWEB document; it should be followed by the title of the chapter, which is terminated by the two-character sequence '. ' (again the space might be any white space character). The title is not recognised by CWEB itself, but rather by TEX, as a delimited macro argument†. This means that if one wants to have an occurrence of the sequence '. ' in the title itself, this can be achieved by enclosing the title (but not the '. ' terminating it) in braces. If one wants to put other things than plain text in a chapter title, one should be aware that it is converted to upper case in the running heads of pages and also written to the table of contents file; only items that behave properly under these operations should be used in a chapter title. Apart from issuing a title that will appear in several places, a section starting a chapter will force a page break before it, and it will cause the section number to be printed on the terminal during the execution of CTANGLE and CWEAVE, as a progress report.

As a feature for advanced users of CWEB, some extra information may be supplied with the '@*' control code: if it is immediately followed by '*' or by a decimal number, than this is not included in the chapter title, but rather interpreted as an indication of the "level" of the chapter. Here '@**' indicates the start of a grouping of sections even coarser than a chapter, and the grouping started by '@*n' becomes finer as n increases, with '@*0' corresponding to unadorned '@*'. The effect of this level depends on the definition of the TEX macros that format the chapter title and the lines in the table of contents, '\N' respectively '\contentsline', to which the level is passed as first argument (for '@**' the level is -1); it could effect for instance the font used for the chapter title or the amount of indentation of that title in the table of contents. In the default definitions of these macros the level is largely ignored, except that '@*n' will not force a page break for $n \geq s$, where s is the value of the '\secpagedepth' register, which is set initially to 2.

† Therefore, if no correctly specified title follows '@*', then CWEAVE will find nothing wrong, but TEX will complain about a "Runaway argument" of a macro that the programmer did not explicitly write (namely '\N'); this is one of the scarier error message that novice users can come across, so please be warned.

In contrast to '@*', a section starting with '@~' instead of '@ ' will tie itself to the previous section, in the sense that a page break between these sections will be avoided. More precisely this is what happens: normally CWEAVE will instruct TEX to break pages only between sections (except when one is too large too fit on a single page) and put as many sections on each page as possible subject to this restriction; however, a section starting with '@~' will be considered to be continuation of the previous section for the purpose of page breaking. A situation where one would use '@~' is the following: suppose we define a function, and also want to state its prototype, which will belong to a different module, since it has to appear earlier in the program or even on a separate (header) file. A natural place to give the prototype in the CWEB document is directly before the function, so that it can easily be seen that the prototype matches the actual definition. Now without special measures there is a substantial chance that a page break will occur between these two sections, since the short section with the prototype might fit on an already partially filled page, whereas the larger section with the definition might not. By starting the latter section with '@~', it can be achieved that in such cases the former section is moved together with the latter to the new page.

Like any other section the very first section starts with s sectioning code (usually '@*'), and any text that might precede it is not part of any section; this text is said to be "in limbo". This material is ignored by CTANGLE, and copied literally into the TEX file by CWEAVE (except for the replacement of '@@' by '@'), following the first line which always reads '\input cwebxmac' (in order to load the standard format). The purpose of the text in limbo is to allow issuing TEX commands that apply to the whole document (such as macro definitions, possibly modifications or additions to the standard format), or producing a title page or an introduction preceding the sections of the CWEB document. No control codes are allowed in the limbo text (well, almost; there are two exceptions, that will be mentioned below). The last section ends simply at the end of the CWEB source file; there is no way to add material after it (or elsewhere outside the sections). However, CWEAVE will append some material at the end itself (unless it is invoked with a '-x' flag): an index of identifier uses, a list of module names, and a table of contents. Because the index is seamlessly attached to the last section, it is customary to give that section the title "Index" and not to include any program fragment in it.

Subsectioning codes: '@d', '@h', '@f', '@c', '@< ... @>=' Each section, as delimited by the sectioning codes, contains a TEX part (although it may be empty), and in addition at most one C part, which always comes at the end of the section, and zero or more intermediate parts, of which there are three kinds: those that specify **#define** and **#include** directives, and format definitions (see below). Intermediate parts can be given in an arbitrary order, as long as they come after the TEX part and before the C part, if present. The beginning of any part other than the TEX part (which starts directly after the sectioning code) is marked by an appropriate control code, which is called a subsectioning code; these codes are optional in the sense that they need only be given if the corresponding part is present. The end of the TEX part is determined by the first subsectioning code, or in absence of any of them by the next sectioning code.

The C part, if present, begins at the first occurrence of '@<' or '@c'; the former starts a defining occurrence of a module name, and the latter is used when the C part belongs to the unnamed module. The code '@c' may also be written as '@C' (in fact all alphabetic codes are equivalent to their upper case counterparts). Once the C part of a section is started, any further module names are interpreted as modules references rather than as defining occurrences. A module name, whether defining or not, consists of TEX code between the '@<' and the next occurrence of '@>'. As a measure against accidental misinterpretation of module names, due for instance to a forgotten '@ ' or '@c', the closing '@>' of a defining occurrence must be followed (optionally with some white space in between, but no newline) by one of '=', '==', '+=' and '+==', while for a non-defining occurrence this must not be the case. The possibilities '+=' and '+==' are included for those who like their source code for continuations of modules to resemble the printed output, but the distinction is ignored by CWEB: it will simply print '≡' after the first defining occurrence of a module name and '+≡' after any further defining occurrences.

The subsectioning codes that mark the beginning of intermediate parts are '@d', '@h', and '@f'. Of these the first two specify preprocessor directives for respectively a macro definition and the inclusion of a header file, and the last specifies a so-called format definition. The codes '@d' and '@h' will be replaced by **#define** respectively by **#include** in both the program and the printed document. We already mentioned how the effect of using '@d' or '@h' differs from that of using **#define** or **#include** directly in the C part of the

section: the directive will be moved to the beginning of the C file, and in case of '@h', the header file will be scanned for typedef definitions. Here we mention a few more points that are relevant when writing the source file.

Macro definitions following '@d' are not line-oriented like those in C: everything up to the next subsectioning or sectioning code is considered to belong to the macro, and newlines need not be escaped, as CTANGLE will take care of escaping any newlines while writing to the C file. There are some mild restrictions on the replacement text of a '@d' macro definition: parentheses and braces should be balanced (this is a deliberate requirement, made in order to allow detection of programming errors that would otherwise be very hard to track; the same requirement also holds for each complete C part of a section), and no module names should be referenced. It is not possible to use other preprocessor directives in macro definitions either, but that is because this is already impossible in C. After a '@h' command, at least one newline should occur before the next sectioning or subsectioning code.

Apart from this, '@d' and '@h' are followed by whatever would follow **#define** respectively **#include**, with the same deviant lexical rules as in C. So whether a macro introduced by '@d' is defined with or without arguments depends on whether the first character after the identifier following '@d' is a left parenthesis or not, where spaces *are* significant. The file name after '@h' may be enclosed either in double quotes or in angle brackets; the latter indicates that the header file is located in some system include file area. After the file name a comment may be placed.

The header file specified after '@h' itself should of course contain ordinary C code rather than CWEB input; after all, it will be read directly by the C compiler. As as mentioned before, the file will be scanned by CWEAVE as well, searching for any typedef definitions; moreover, if it contains any lines starting with **#include**, then those files will be scanned recursively as well. In the case of system header files (specified with angle brackets), CWEAVE will refrain from scanning the file unless the file is found on an explicitly specified search path (see below); in fact it is better not to scan any of the ANSI/ISO standard header files, since CWEAVE already knows about all typedef definitions that can occur in such header files. It is not uncommon that a header file specified after '@h' (using quotes) is itself an auxiliary output file produced from a CWEB source file, possibly even from the very source file containing the '@h' command. There is no circularity or other problematic aspect of such a situation, but one should remember to run CTANGLE to produce the header file, before the run of CWEAVE that needs it.

The way CWEAVE searches for the header file depends on how the name following '@h' is specified: if it is enclosed in quotes then CWEAVE will look first in the current directory. There may have been specified one or more alternative places to look for header files, in the form of strings that can be prefixed to the file name (given on the command line or compiled into CWEAVE, or both). If so, these will be tried in order, regardless of the delimiters used for the file name, until a match is found; CWEAVE will only insist on actually finding a header file if the file name was enclosed in quotes.

There is one aspect of scanning header files that might cause a problem in some cases: when scanning a header file, CWEAVE is unaware of other preprocessor directives that may disable certain nested **#include** directives; CWEAVE will therefore obey such **#include** directives unconditionally. Such a problem is not very likely, but it could be serious if the nested header file cannot be found (and is enclosed in quotes), or if there are circular references between header files. Various solutions could be found for such a problem, depending on the precise situation, varying from creating dummy files or avoiding conditional compilation by the use of change files to (as a last resort) avoiding the scan of the header file altogether, by using **#include** in a program fragment rather than '@h'; in the latter case relevant information could be extracted from the header file manually, and converted into format definitions ('@f') described below.

When preprocessor directives are incorporated in the C part of a section, the ordinary rules of C apply: they should be spelled out in full, as '#define' or '#include', and occur at the beginning of a line; the directive ends at the next non-escaped newline. Although in C it is permissible to extend a preprocessor directive into the following line by placing a multi-line comment that contains the newline, this should not be done in CWEB, since the comment will be removed by CTANGLE but the newline will remain. If one needs a very long comment after a preprocessor directive, one should start it on the line following the directive; in the formatted document such a comment will be placed on the same line as the directive. The same holds for comments placed after a '@h' command.

Format definitions, indicated by the code '`@f`', are entirely specific to CWEB, and have no effect on the C program that is defined. They are not needed very often, but when they are, a proper use of them is essential for obtaining acceptably formatted output. To understand why they are sometimes needed, one has to consider the way CWEAVE formats program fragments. The input is broken up into tokens (like identifiers, constants, operator symbols), and a syntactic category is attached to each; the resulting sequence of categories is then analysed according to a grammar, and formatted correspondingly. Certain identifier tokens are recognised as reserved words and get a corresponding the syntactic category, others are recognised as typedef identifiers and get the same syntactic category as for instance **size_t**, and the remaining ones are treated as ordinary identifiers. This scheme usually works fine, but occasionally there can be problems, caused by the fact that CWEAVE is not aware of all the information that is available to the compiler. The main reasons for this are macros (which may cause the code seen by the compiler to be quite different from that seen by CWEAVE), typedef declarations that are hidden from CWEAVE's sight, and module names that stand for a construct of a different syntactic category than *statement* (which is what CWEAVE expects them to be by default). In all these cases CWEB provides mechanisms for the user to put CWEAVE on the right track, and format definitions are one such mechanism (others will be discussed below).

Format definitions allow the programmer to explicitly state the syntactic category that CWEAVE should attach to a given identifier. They have the form '`@f x y`', which will become '**format** *x y*' in the typeset output; here *x* and *y* can be arbitrary identifiers or keywords. This definition has the effect of associating to *x* the same syntactic category that is associated to *y*. Such a change of category is required when an identifier is defined as a macro to stand for a keyword: whenever you say '`@d ident keyword`', say '`@f ident keyword`' as well. For instance, the author of this manual thinks the keyword **static** is not very informative when applied to functions, and therefore often creates an alias for it by saying '**#define** *local* **static**'; this directive is then followed by '**format** *local* **static**'. We see that the first identifier after '`@d`' or '`@f`' is always typeset in italics; this is so despite the fact that in the example, as a consequence of the format definition, this identifier will be typeset as **local** in all other places. Another reason to change a category could be that an identifier is in fact a typedef identifier, but CWEAVE cannot deduce this fact (presumably the declaration occurs in some header file that is not scanned by CWEAVE); in such cases one can use a standard defined type like **FILE** or **size_t** as the second argument to '`@f`'. Finally, it is possible that some C implementation uses additional, non-standard keywords (or macros that behave as a keyword); such an identifier should be formatted like a standard keyword that has a similar syntactic function as it (which hopefully exists). In fact the identifier **va_dcl**, which is used in a convention for functions with variable argument lists that is not part of ANSI/ISO C, is nevertheless built into CWEAVE, because there is no keyword that has the required syntax category (namely *declaration*), so that it would otherwise not be possible to introduce it; one can on the other hand easily undo the reservation by saying '**format** *va_dcl x*'.

Format definitions can also be used for a reason that does not have to do with syntax analysis. There are two classes of identifiers that are parsed like ordinary identifiers, but are nevertheless treated specially; these classes consist initially of the identifiers *TeX* respectively *NULL*. The main distinction of these classes is that their identifiers are typeset differently, namely as TEX macros; the mentioned identifiers will therefore be written to the TEX file as '`\TeX`' respectively '`\NULL`', which causes them to be typeset as 'TEX' respectively '⊙'. This mechanism gives the user the ability to change the appearance of identifiers in any desired way, simply by defining the macro appropriately. The class of *TeX* is intended for identifiers that are still alphabetic in appearance (possibly with letters being accented or shifted), while the class of *NULL* is intended for identifiers that are represented by mathematical symbols. Hence the TEX macro will be processed in horizontal mode with italic font selected in the first case, and in math mode in the second case. Simply saying '`@f alpha NULL`' suffices to make *alpha* print as α; the format definition is typeset as '**format** *alpha* ⊙ (α)' to make the correspondence of the identifier and typeset symbol evident.

Unlike C identifiers, TEX macros cannot contain underscores and digits. On writing of the macros to the TEX file, underscores are replaced by '`x`', so that they will become part of the macro. Digits however are not changed, so identifiers containing digits should not be put into the class of *TeX* or *NULL* by a format definition, unless special care is taken: the macro will only consist of the part up to the first digit. No index entries for identifiers of the class of *NULL* are recorded (the same holds for keywords); on the other hand index entries for typedef identifiers are recorded, despite the fact that they are formatted as keywords.

Text within* C *program fragments: comments and module names Within the program part of a section, the input should basically follow the rules of the C syntax, but amidst the C tokens there may also occur module names and comments. In both cases the C code is temporarily interrupted by a piece of ordinary text that is processed directly by TEX, just like the TEX part of a section. In the case of module names this text is delimited by '`@<`' and '`@>`', in the case of comments by '`/*`' and '`*/`'. So comments are actually valid C comments, but the converse is not true: the contents of a comment is processed by TEX, so not all C comments can be used without modification; a point to keep in mind if one is converting ordinary C code to `CWEB`. Like C comments, the comments of `CWEB` cannot contain the two-character sequence '`*/`' (regardless of the TEX context, because comments are recognised before TEX even gets to see them). The sequence '`/*`' is forbidden as well, which allows `CTANGLE` to warn the programmer about unclosed comments, that might otherwise lead to particularly elusive errors. In the TEX texts of comments and module names no control codes are allowed (except in embedded pieces of C code, described below), but '`@@`' can be used to represent the character '`@`' (this is true in all contexts); a module name is terminated by the first occurrence of the code '`@>`'. During the processing of these TEX texts, line ends are replaced by spaces, which implies that TEX comments (starting with '`%`') cannot be used. (In the TEX part of a section on the other hand, such comments can safely be used: they are completely ignored by `CWEAVE`, and not even copied to the TEX file.)

The text for module names serves a dual purpose: apart from determining the text representing the module in the printed output, it also serves to identify defining occurrences of a module name with references to it. For the latter purpose it is irrelevant how the contents of a module name will be further processed; there should basically be a character-by-character match. This rule is however alleviated in two ways to make matching easier. First, any amount of consecutive white space is replaced by a single space, and white space at either end of a module name is discarded. Second, an abbreviation mechanism for module names may be used. A module name may be specified by a prefix of the full name, followed by '`...`'. A few conditions must be satisfied to allow this mechanism to work. All specifications of one same module name must be extensions of the one among them of minimal length, which must not be a prefix of any other (full) module name. All specifications of the name that do not end with '`...`' must be equal; there must be at least one such specification, which defines the full module name of which all other specifications give a prefix. Loosely speaking, the minimal specification is used for identification purposes, and the maximal specification is used for typesetting all occurrences. With the help of these rules, and a text editor, there should be little reason to choose module names any shorter than what is needed to express the function of a module clearly. There is a limit on the length of a module name, but it is so generous that this could hardly be a problem: 1000 characters after replacement of consecutive white space characters by single spaces.

The parser of `CWEAVE` normally assumes that references to modules stand for (compound) statements, which is likely to be a valid assumption in the majority of the cases (or at least one that does not upset parsing, for instance when the module is actually a statement sequence). Occasionally however, one of two other syntactic categories applies instead, namely *declaration* or *expression* (the remaining categories are extremely unlikely). When this is the case, the programmer should make it clear to `CWEAVE`, lest the parser might choke on the input and produce badly formatted output. This can be done by placing the control code '`@;`' once (for a *declaration*) respectively twice (for an *expression*) directly after the module name (in the latter case this also conveniently provides a separation from any '`=`' or '`+=`' that might follow).

At the end of the `CWEB` document, after the index, a list will be placed of all module names used. This list sorted lexicographically, with sorting based on the source strings for the full module names, collated (unlike the identifier index) in the order of the internal (ASCII) character codes. For this reason it is a good convention to ensure that all module names are already distinguished by a prefix consisting of alphabetic characters and spaces only, of which the first word is capitalised; then the order of the list will be natural and independent of any internal details that the reader is not aware of.

C ***code within text: '|...|' fragments*** In order to mention a piece of C code within TEX text, it can simply be enclosed in vertical bar characters ('`|`'); then `CWEAVE` will format it in a way similar to to C code of modules. This feature may be used in any kind of TEX text except in limbo, i.e., in the ordinary TEX part of a section, in comments and in module names. The piece of C code itself should not contain any comment.

The "lightweight" construction with vertical bars resembles the math shift characters ('`$`') for TEX's math mode, and indeed in simple cases like '`|a[i+3]|`' the output would be identical if the '`|`' characters were

replaced by '$'. The two modes should not be confused however: the "C mode" is implemented by CWEAVE, which translates the C constructs before TEX ever gets to see them; it often uses math mode itself, and as a consequence it should never be used when TEX is already in math mode. The syntax used by CWEAVE is of a stricter kind than that of TEX's math mode, but it can still be used for some expressions that are not quite proper C; in particular there is no objection to writing things like $begin \leq p < end$, which humans understand better than compilers. On the other hand an incomplete formula like '$\leq n$' (which can be used in sentences, with the missing operand expressed in words) is better written as '$\leq n$' than as '|<=n|': the latter is not understood by CWEAVE's parser, and therefore the '<=' and the 'n' are translated separately with an ordinary space in between; the result looks reasonable, but TEX may very well decide to break the line at the space.

There is a lexical price to pay for using delimiters that are not control codes: it is impossible to use character '|' in any piece of TEX text where '|...|' constructions are allowed (even if one tries for instance to set up a verbatim context, because CWEAVE acts before TEX does). This should not cause great problems however, since '|' is not a character in ordinary text fonts, and for '|' and '\|' in math mode, plain TEX already has the substitutes '\vert' and '\Vert'; for exceptional text fonts (like typewriter type) that do have '|', the standard format for CWEB provides '\v' as a substitute (by means of \chardef) for '|'. Inside '|...|' one has a similar problem of not being able to write the bitwise-or operator '|' in the usual way. For this purpose CWEB provides the control code '@v' to represent that operator (which you may also use in an actual program fragment, although there is no need to do so there). Note that the composite operators '|=' and '||' can be used without problem; consequently no '|...|' should be immediately followed by '=' or by another '|...|'.

Although C comments are forbidden inside '|...|', it is possible to mention a module in TEX text by enclosing the module name in vertical bars; this TEX text can either be the TEX part of a section or a comment, but not another module name. Mentioning a module in this way does not imply any inclusion of the module body, so it is not considered to be a use of the module; in the cross-references it is referred to as a "citation" of the module. For the module name itself the same rules apply as for other occurrences of module names; in particular the abbreviation mechanism can be used, and CWEAVE will automatically insert the relevant section number in the module name. Citing a module may form an exception to the rule that an occurrence of a module name when the C part of a section has not already started must be a defining one. Since CTANGLE normally ignores the vertical bars of '|...|' constructions together with the surrounding TEX text, it needs a simple rule to decide whether a module is being cited or defined. It does this by inspecting the next token (where a newline counts as a token, but codes like '@;' that are ignored by CTANGLE are skipped): if this is '=' (or '+=' etc.), then it assumes that the module is being defined, and if it is '|' that the module is being cited; in other cases it signals an error (this could for instance happen if a '@c' code is missing). Therefore it is not really necessary that the module name is the only item in the '|...|' construction, as long as it is the final item; this extra freedom is not likely to be of much practical use, however.

Modules producing additional output files: '@(... @>' As was mentioned before, there are special module names that will cause the program produced by that module to be written to a separate output file. Such a module name is specified by enclosing the file name in '@(' and '@>'; in fact it is sufficient to use '@(' instead of '@<' in just one occurrence of the module name. The file name will be set in typewriter type by CWEAVE, so that the difference with an ordinary module name is easily perceived. Although hardly relevant for this case, the compression of white space and the abbreviation mechanism for module names also applies to these special module names. The file name can contain any special characters, including '|' and '@'; the latter must as always be doubled.

Control codes that help parsing in special situations: '@;', '@[', '@]' In the discussion of the **format** command we already mentioned the way CWEAVE parses and formats program fragments, and the fact that some programming constructions can confuse the parser, leading to badly formatted output. Like '@f', the control codes in this subsection provide ways to avoid such problems, but they do so on a local basis in the code itself, rather than by global definitions. They are mainly used in connection with macros with replacement texts and/or arguments that are not expressions. Since macro invocations look like identifiers or function calls, and macro arguments appear to be function arguments, a piece of code containing a macro

invocation whose replacement text and arguments are not all expressions may seem syntactically incorrect when not expanded. An example of such a scenario is a macro whose replacement text is a compound statement; an invocation of such a macro needs no semicolon following it, and sometimes placing a semicolon would actually cause an error (e.g., if the invocation is used as the first branch of an **if-else** statement, since the semicolon would be taken to be an empty statement *after* the conditional statement, and the **else** would be unmatched). Since the parser of CWEAVE does not expand macros, it will fail to recognise a macro invocation without a following semicolon as a statement, and like many parsers it is not good at recovering from such a failure. Although no error message is usually issued, formatting can be severely disrupted; indeed, correct formatting will only be inserted locally for constructions that do not contain the "error", so one unrecognised construction can easily destroy the layout of the entire program fragment it occurs in.

CWEAVE provides some simple mechanisms for guiding the parser through such unusual code, and by applying them in several ways nearly all problems that arise in practice can be solved. One of these is the control code '`@;`', which produces no C code (nor any printed output), but which can be used in places where the CWEAVE parser would require a semicolon for a successful parse; another is the combination '`@[`', ..., '`@]`', used as a pair of parentheses, which will cause whatever is enclosed to get the syntactic category 'expression', regardless of its actual category.

The most obvious use of '`@;`' is in the case already mentioned of a macro invocation that expands to a (compound) statement: placing '`@;`' after such a macro invocation will cause it to be recognised as a statement by CWEAVE, keeping its parser happy while not affecting the actual C program. There are other situations as well where one does not want to place a semicolon, yet wishes CWEAVE to act as if it were there. If a macro stands for statement that happens to end in a semicolon, then it is a good idea to suppress the final semicolon in the definition: in that case all invocations can supply the semicolon, and one does not have to remember writing '`@;`' instead of '`;`' at the invocations of this macro. For instance, the macro replacement text could be '**do** ⟨ statement ⟩ **while** (⟨ condition ⟩)', or '**if** (⟨ condition ⟩) ⟨ statement ⟩ **else** ⟨ expression ⟩', or even '**if** (⟨ condition ⟩) ⟨ statement ⟩ **else**', where the final **else** was placed with the purpose of picking up the following semicolon as an empty statement; in all these cases the macro invocation together with the following semicolon is a complete statement that can be used without special precaution, even as the first branch of an **if-else** statement. However, in these cases the macro definition itself needs a bit of extra care: a '`@;`' should be placed at the end to represent the semicolon that will follow in invocations, so that CWEAVE can properly format the replacement text of the macro. Finally, it there can be purely aesthetic reasons for wanting to suppress a semicolon at the end of a '`|...|`' construction, for instance when referring to a declaration as '**char** $*p$', which strictly speaking requires a final semicolon to become a declaration; to let CWEAVE format this properly, one should write '`|char *p @;|`'. Constructions like '**return** *home*' and '**goto** *sleep*', which are fairly common to mention in module names, would also fall into this category, but in this particular case no '`@;`' is necessary, since CWEAVE parses these as expressions, even though strictly speaking they are not.

Since '`@;`' is invisible in the output, yet can be sensed by the parser, it can conveniently be used to pass information to the parser, and there are a few instances of such use where it does not stand for a semicolon. We already mentioned placing one or two copies of '`@;`' after a module name to indicate the syntactic category. Another use is to place it before a typedef identifier to cause it to be treated as an ordinary identifier; this is useful if the identifier is locally redeclared, or used as field selector in a **struct** or **union** specifier. When the identifier is used as a tag immediately after **struct** or **union**, or as a selector after '.' or '→', it is not necessary to place '`@;`' before it.

Unlike '`@;`', the control codes '`@[`' and '`@]`' themselves do not participate in parsing. The material between them is parsed normally, which may or may not succeed in recognising a single construct; then the pieces recognised are concatenated (without separation), and the result is given the category *expression* for the purpose of parsing further items outside. The most obvious use of this mechanism is to encapsulate any arguments in a macro invocation that are not expressions (e.g., some storage allocation macros have a type as argument), so that the invocation can be parsed as a function call. There need not be anything in between '`@[`' and '`@]`', so '`@[ @]`' can be used as an "invisible expression" in the same way as '`@;`' can be used an invisible semicolon. An example where this is useful, is a module standing for an initialiser list, that is moreover defined in multiple sections (see for instance the module '⟨ Rules 157 ⟩' in the source document for

CWEAVE): it is natural to end each program fragment defining a part of such a module with a comma, but this will not be parsed properly unless an expression follows, which can be achieved by adding '@[@]'. Finally, if for some tricky piece of code none of the mentioned methods suffice to get it parsed properly by CWEAVE, one may use '@[' and '@]' (followed by '@;' if necessary) to minimise the damage: by placing '@[' and '@]' around an appropriate part of the program containing of the problem area, we can ignore the fact that the parser failed to recognise it, and force it to continue as if it ad recognised an expression; thus we can contain the problem, and prevent the effects from spreading any further.

5 Invocation of CTANGLE and CWEAVE

The simplest form of calling CTANGLE and CWEAVE is to supply one command line argument, which is the name of the CWEB source file without the '.w' suffix. It is possible however to modify the behaviour of the programs by selecting certain optional settings, and small patches to the master source file can be achieved by supplying a "change file". The general syntax for invoking CTANGLE is

ctangle [(+ | -)⟨ options ⟩] ⟨ CWEB file ⟩[.w] [(⟨ change file ⟩[.ch] | + | -) [⟨ output file ⟩[.c]]]

where square brackets indicate optionality, vertical bars separate alternatives, and parentheses are used for grouping. Here ⟨ options ⟩ is a string of one or more characters designating options, as described below; there may be more than one such string of options, and they may be given between or after the files names instead of before them, with no difference in meaning. For CWEAVE the situation is entirely similar, except that the default extension for the output file is '.tex' instead of '.c'.

Command line options A command parameter that starts with '+' or '-' and has at least one more character, serves to control optional settings of the program being invoked. The characters after the initial character '+' or '-' denote individual options that are turned on respectively off; option characters are case-insensitive. The character 'i' forms an exception, since it is used to supply a string argument rather than to set a switch; the string is the remainder of the option string (following the 'i'), and '+i' and '-i' are equivalent. All option characters will be accepted, but only the ones listed below have any effect on the operation of the program. We list the switches in the direction that alters the default setting.

switch	*program*	*effect*
-b	both	do not write a banner line to the terminal
-p	both	do not show a progress report on the terminal
-h	both	omit confirmation of successful completion
-l	CTANGLE	omit **#line** directives, make C file look nice
-x	CWEAVE	do not attach index and other information at the end of the document
+d	CWEAVE	report failure to completely parse pieces of C code
+t	CWEAVE	write three files, with separate ones for index and list of module names
+e	CWEAVE	even out number of pages before table of contents
+i	CWEAVE	add alternative search path for header files (takes argument)
+f	CWEAVE	force a line break after each statement
+a	CWEAVE	force all statements to be on a line by themselves
+u	CWEAVE	"unaligned brace style": do not align '{' and '}' vertically
+w	CWEAVE	"wide brace style": force line breaks before and after '{'
+m	CWEAVE	"merged declarations style": do not force line breaks between local declarations
+c	both	run in compatibility mode with Levy/Knuth CWEB
+s	both	show memory usage statistics at completion
++	both	handle C++ language instead of C

The options '+d' and '+s' only operate if CWEAVE or CTANGLE was compiled with the preprocessor symbol DEBUG respectively STAT defined (with most C compilers this can be accomplished by including a command line parameter -DDEBUG respectively -DSTAT when compiling the CWEB system).

The options '`b`', '`h`', and '`p`' can be used to control the amount of output that CWEB writes to the user terminal; the combination '`-bph`' will eliminate terminal output altogether when no errors are encountered.

The option '`-l`' of CTANGLE is intended either for use with broken compilers or debuggers that cannot handle **#line** directives properly, or for cases where the C file is of more importance than just as an intermediate file, for instance when the program is transferred to people who do not wish to practice literate programming. Apart from omitting **#line** directives and comments that indicate the section number from which code originates, an attempt is made to make the C file more readable to humans: the spacing and (almost all) comments of the source file are preserved in the C output, and when modules are substituted into others, indentation levels are accumulated, so as to produce indentation that looks natural. Doubtlessly the result is not perfect (and lines may get quite long), but it is definitely more readable than the output normally produced. Since layout and comments of the source file need to be preserved by CTANGLE, this option consumes significantly more memory than its contrary.

The option '`+d`' causes CWEAVE to issue a warning when it could not properly parse some piece of C code; this could happen either because a code fragment is incomplete in the sense that it does not represent a single complete syntactic entity (as in the '`|<=n|`' example above, or when a module body ends with a label without a following statement), or because the code is actually unsyntactic, or because CWEAVE has been fooled by an unusual construction. In all cases however the result can be (very) badly formatted output, and a correction should be made; users who care about the quality of the typeset output are advised to always set this option (or at least when the document is being finalised). Setting the '`+d`' switch is equivalent to placing a control code '`@1`' at the beginning of the first section; the nature of the warning messages and possible remedies will be discussed later in this manual.

The two output files that the option '`+t`' will cause CWEAVE to create in addition to its main output file, are called ⟨ name ⟩`.idx` and ⟨ name ⟩`.scn`, where ⟨ name ⟩ is the name of the main output file without its extension. These files will be read by '`\input`' commands in the main output file, so that the typeset document will not be any different; on large projects however it can be helpful to have this information on separate files, for instance for making a global index. The option '`+e`' is intended for use with two-sided printers: it ensures that the table of contents comes out on a fresh sheet of paper, so that it can conveniently be moved to the front.

The option '`+i`' (or equivalently '`-i`') can be used to specify a directory for CWEAVE to search for header files in '`@h`' commands. Although directory structures are system-dependent, CWEB assumes that a file can be looked up in a specified directory by prefixing a string indicating that directory to the file name (this works for many systems); the desired prefix string should then be supplied as the remainder of the option string after the '`i`' character. E.g., on the UNIX system the author uses, CWEAVE can be told about the location of the 'Xlib' header files by supplying an argument '`+i/usr/local/X11R5/include/`' (one could replace '`+i`' by '`-I`' to make it look more like the similar option passed to the C compiler); the important thing to note is the final pathname separator '`/`'. Up to 8 additional prefixes can be specified by giving several such arguments; they will be tried in order from left to right. It is also possible to fix one such prefix at compile time, by defining the preprocessor symbol '`CWEBHEADERS`' to be the desired prefix string when compiling the compilation unit `common.c` of CWEB; this will behave as if it were the first prefix specified by a '`+i`' argument.

The last five options mentioned will alter layout style of program fragments. The option '`+f`' will result in a more vertical style than the default, and '`+a`' will do so even more; the difference between them is that '`+f`' will not force a simple statement to start on a new line if it follows a label or the condition of an **if** or **while** statement, whereas '`+a`' will start a new line in such cases. The option '`+u`' selects a style in which corresponding opening and closing braces are unaligned because a line break is inserted after '{' instead of before it. The option '`+w`' on the other hand selects a brace style that has more vertical symmetry than the default one, since opening braces will appear on a line by themselves, like closing braces; the price is that listings will consume more paper. The option '`+a`' overrides '`+f`', and similarly '`+w`' overrides '`+u`'. Finally, the option '`+m`' is for people (like the author) who are extremely keen on saving paper: it avoids forced line breaks between the declarations in a compound statement, just like they are not placed by default between the statements; the separation between declarations and statements within a compound statement is still indicated by a line break, that even has some extra vertical space, because this separation is significant in

the C syntax (unlike the C++ syntax).

In compatibility mode, specified by '+c', both CTANGLE and CWEAVE modify their behaviour in such a way that they try to ensure that they can handle any file that can be correctly processed by Levy/Knuth CWEB, and that the output is an equivalent C program, respectively a valid TEX file (this is the hard part) that produces a comparable printed document. In the current version this claim can only be made for programs written in C; a wholehearted attempt to do the same for C++ programs would cost a substantial amount of extra work. There are so many differences in the details of formatting between CWEBx and Levy/Knuth CWEB that one cannot expect formatted output that is identical to what would be produced under Levy/Knuth CWEB, but to get the best approximation, one should in addition to '+c' specify the options '+uft'.

The option '+s' is included because the CWEB utilities use statically allocated memory areas, which may therefore run out; using this option one can see how close one is to the limits of CWEB. The most important limited resources that it provides information about are are: (a) The name tables in which CTANGLE and CWEAVE store all distinct identifiers and index entires, respectively module names (the entries 'identifiers', 'module names', and 'bytes'); (b) CTANGLE's main memory, in which the complete C program file processed during a single run has to be stored, albeit in a compactified form ('replacement texts' and 'tokens'); (c) CWEAVE's cross-reference memory, in which all the data for the index and list of module names are stored ('cross-references'); (d) its parsing buffers, which must be able to hold any one program fragment or piece of C code ('scraps', 'texts', and 'tokens'). There should be no immediate need to increase the size of these memory areas, since even for the main program of CWEAVE, the largest of CWEB's own compilation units, the use of any of these resources is less than a third of the amount available. There is one resource of which a larger fraction is used, namely 'trie nodes', but its usage depends only on the set of grammar rules used, which is independent of the particular CWEB source file. When for some source file CWEB is approaching its limits, one can of course try to recompile CWEB with larger arrays, but alternatively one may restructure the source file: when one of (a), (b), or (c) runs out, one might consider breaking up the file into several separately processed pieces; when (d) runs out, a remedy could be splitting up some huge module body into smaller ones, by introducing submodules or multiple definitions of the module.

Switching to the C++ language has only a minor influence on the operation of CWEB: one-line comments starting with '//' will be recognised, the main output file produced by CTANGLE will have default extension '.C' instead of '.c', and CWEAVE will recognise a few more reserved words and use a slightly different syntax. Since there is no general agreement about the proper extension for C++ files, and alternative default extension for C++ mode (instead of "C") may be built in by setting the preprocessor symbol CPPEXT to the desired string (that should not contain the leading period) when compiling common.c. Currently the CWEAVE grammar will handle only a basic subset of the C++ language, which does not include templates or exception handling.

File name arguments Any command line arguments that do not have the form of an option are taken to indicate file names; their number can vary from 1 to 3. The first one specifies the main source file, the second (if present) indicates the change file, and the third optionally defines a non-standard name for the main output file. The contents and function of the change file is discussed in the next section; here we we just indicate how the actual file names used are derived from the given file name arguments. As far as CWEB is concerned a file name is composed of a base name and an extension. Loosely speaking, the extension of the main file defaults to 'w', that of the change file to 'ch', and that of the output files to 'c' or 'tex' for CTANGLE respectively CWEAVE (but see also the discussion of the '++' option above); the base names of the change file and the main output file default to that of the main file. If in place of a change file name an argument '-' is specified, no change file is used; also if only one file name argument was given, or if the change file name was specified as '+', then the default change file name is tried, but if no such file exists, processing proceeds without a change file. (Specifying the change file as '+' is only useful if a third file name argument is given.) Therefore, assuming regular naming conventions, there is no need to specify more than the main file name without extension, whether or not a change file is being used.

The precise rules are as follows. On file systems where an extension is not a standard property of file names, like that of UNIX, it is assumed the a period is a valid character in file names; a full file name is then formed by concatenation of the base name, a period and the extension (note that this implies that on such systems CWEB cannot access files whose name contains no period at all). Conversely, a string designating a full file name is broken up into a base name and an extension at the last occurrence of a period; if no

period is present, then the string is taken to specify a base name only, and is said to have no extension. If the first file name argument has an extension, it specifies both base name and extension of the main source file, otherwise it specifies the base name, and the extension is taken to be '`w`' (if no such file is found, the extension '`web`' is also tried, but this feature is obsolete). The base name of the main source file is also the default base name of the change file and the main output file; their default extensions are as described above. If a second and possibly third file name argument is present and is not '`+`' or '`-`', it overrides the base name, and also the extension if it has one, of the change file respectively of the main output file. No change file will be used either if the second file name argument is '`-`', or if no change file is found when the second file name argument is '`+`' or absent.

6 Subsidiary input files and change files

As we have described it so far, the CWEB tools read a single source file, from which a main output file and possibly some auxiliary output files are produced. Since C programs can be built from several compilation units, it is not uncommon that several CWEB source files contribute independently to the same program, and there might be non-CWEB source files as well. However, even what is conceptually a single CWEB source, described by a single printed document, may in fact be composed from several input files. Two mechanisms are provided for combining information from several files, with different purposes. First, subsidiary files may be read in from the main source file in a way similar to the way **#include** files are handled by a C compiler. In the case of CWEB however, the main purpose is usually not to share information among several sources, but merely to allow breaking up large source files into more easily manageable parts. Second there is the change file mechanism already mentioned above, which serves to install system dependent patches to a master source, allowing that master to remain free of system dependencies.

When a line of the form '`@i` ⟨ file name ⟩' appears in a CWEB source file, CWEB will read in the indicated file at that point, and continue reading at the next line when it reaches the end of the subsidiary file. The ⟨ file name ⟩ may either be delimited by white space, or be enclosed in double-quote characters (but not in angle brackets). Source files may be nested in this way up to 10 levels deep. Nothing in the printed CWEB document will indicate the switch from one source file to another, nor will there be any effect on the C file(s) written by CTANGLE, except that **#line** directives will of course always point to the proper point of origin for each piece of code written to such files.

Like for header files, there is a way to indicate that if a file included by '`@i`' is not found in the current directory, an alternative place can be tried; unlike header files however there is relatively little need to use this facility, unless one has files that are useful to include identically in more than one project. At most one alternative place to search can be given, and it is specified by a prefix to be applied to the file name, in the same way as for header files. This prefix may either be compiled into the CWEB programs by setting the preprocessor symbol CWEBINPUTS equal to that string when compiling `common.c` (analogously to CWEBHEADERS), or it can be specified at run time by setting the environment variable CWEBINPUTS; when both methods are used, the latter takes precedence.

The change file, if present, contains a sequence of "changes", each of which specifies the replacement of one or more lines from the main input stream by another set of lines. Each change has the form '`@x` ⟨ original lines ⟩ `@y` ⟨ replacement lines ⟩ `@z`', where each of the codes '`@x`', '`@y`', and '`@z`' occupies a line by itself. The ⟨ original lines ⟩ is a non-empty set of lines that should match exactly with some sequence of lines in the main input stream (except for the fact that trailing white space on any line is ignored). Furthermore, different changes should affect non-overlapping sets of lines, and their order in the change file should be the same as that of the parts of the main input stream that they replace. For each change in succession, a sequence of lines matching ⟨ original lines ⟩ is searched for, and replaced by the corresponding ⟨ replacement lines ⟩; like for '`@i`' file insertions, the resulting stream of lines will be processed in the usual way as if it constituted a single CWEB source file. The "main input stream" referred to here is the result of (recursively) inserting any auxiliary files indicated by '`@i`' lines into the main CWEB source file. It therefore makes no sense to specify '`@i`' in the ⟨ original lines ⟩, nor is '`@i`' allowed in the ⟨ replacement lines ⟩: it should simply not occur anywhere in the change file. On the other hand it is legitimate for the ⟨ original lines ⟩ to match a sequence of lines coming from more than one physical source file.

The fact that input is temporarily switched to the change file is not entirely transparent to the CWEB document, as it was in the case if '@i' files: CWEAVE will mark all sections that were modified under control of the change file, by attaching an asterisk to their section number, and to all references to that number. (If some changes should add or remove entire sections in the middle of the CWEB source, which is allowed although not encouraged, then the section numbering will be altered, but sections for which this is the only change will not be flagged with an asterisk.) If one is only interested in sections that are modified, then it is even possible to restrict printing to only those sections, by including the TEX command '\changesonly' in the text in limbo, preferably by means of the change file.

In order to facilitate efficient implementation of the change file mechanism, an additional constraint is placed on the changes: once an exact match of a line in the main input stream with the first line of a change is found, the remaining lines of the change (up to the '@y') should also match. Any empty lines immediately following '@x' are not used for matching (and are in fact completely ignored) so the first matching line is never an empty one; it is preferable to choose changes such that their first line matches a unique line of the main input. It is a good idea to start changes in the TEX part of sections (after all, if the program changes, so should its explanation); in this case uniqueness of the match of the first change line can always be ensured (even when the TEX part is empty) by placing a TEX comment in the main input, that serves merely as a target for replacement by the change file. All text in the change file that is not part of a change is ignored, except that there should be no lines starting with '@i', '@y', or '@z'; this text can be used for instance to explain the purpose of the change to the person installing the program on a new system, rather than to the ordinary reader of the program.

As we have said earlier, the change file mechanism provides an alternative to system dependent conditional compilation, and it is usually a much more elegant way to incorporate system dependencies. The main reason for this is that one does not have to anticipate all possible systems that a program could be ported to, nor is the main source polluted by such considerations: it suffices to provide a separate change file each time the program is moved to a system with different system dependent requirements. Users of a particular system need to know about the change file for that system only, and the responsibility for maintaining main source and the change file might lie with different persons; additional effort is only required when the main source changes in such a way that a change file fails to match.

One should not get carried away by the benefits of change files though: they provide only a rather crude mechanism (due to the inflexible matching rules), and if there are many changes, they will become difficult to maintain when the master file evolves. Portability is still best obtained by limiting system dependent features as much as possible, and if inevitable, confining them to some well defined part of the program. If one should wish to create variants of a program that involve significant changes, then writing extensive change files is probably not the best way to go. This method could lead to a form of "rigor mortis" for the original version of the program, caused by fear that any alterations could upset one of the change files, even trivial changes that only involve the commentary, or even just the layout of the source file. A better approach would be to collect routines of general utility as much as possible into separate compilation units used by all variants, and to complement these with completely independent compilation units to define the specific behaviour of each of the variants. It is certainly pointless to use a change file for such things as bug fixes or further development of a program; the whole idea is that such modifications can be made in the master file while the change files for various systems need little or no adjustment.

The codes '@i', '@x', '@y', and '@z' of this section have the appearance of control codes, but they are not really part of the CWEB language, and obey different rules than control codes. For instance, they are line oriented (and rightly so, since their goal is to select which lines will be actually processed by CWEB): they should appear at the beginning of a line, and any further text on the line (in case of '@i', after the file name) is ignored. Also they act quite independently of CWEB's current mode of operation: rules such as the one forbidding control codes in limbo do not apply to these codes.

7 Control codes for advanced or emergency use

In this section we discuss control codes that are not essential for everyday use of CWEB, but are provided to enable either refinements in the presentation of the CWEB document, or special manoeuvres to deal with certain unusual situations or requirements. Most of them serve to allow the programmer some form direct control over the contents of either the CWEB document, the C file, or the source file, bypassing the automatic processing by which these are normally related to each other; there are also a few that serve as debugging aid, eliciting explicit information from the CWEAVE parser about its actions.

Control codes for cross-referencing: *'@!', '@^', '@.', '@?', '@:', '@#'* Some control codes are provided that allow the programmer to influence indexing and to perform explicit cross-referencing. The codes in this subsection are the only ones that are allowed to occur in the TEX part of sections, outside '|...|'; with the exception of '@#', they can also be used in C text. Control codes such as these, that are intended only to affect the printed document, are ignored completely by CTANGLE. Incidentally, cross-referencing in CWEB always means referring to section numbers rather than to page numbers: CWEAVE cannot know about page numbers since these are determined only at the TEX processing stage. It would be possible to have TEX produce a table mapping section numbers to page numbers; in fact the table of contents provides a coarse approximation to such a map.

Whenever CWEAVE can determine from the context that an occurrence of an identifier is a defining one, it will make the corresponding section reference in the index underlined. If some case is missed by CWEAVE's normal rules, or if one wants to make a reference to a reserved word (which is only made if it is underlined), then one can place the code '@!' in front of the identifier to create an underlined reference. Cases where this may be required include arguments of functions with an old-style (pre-ANSI) heading for which no declaration is given before the function body (i.e., the default type **int** applies), and enumeration constants that appear out of context of the **enum** keyword (e.g., because the enumeration list is given as a separate module). In general, the occasions where one needs '@!' are quite rare.

A group of three codes serves to include additional entries in the index, amidst those generated automatically by CWEAVE for identifiers. It may be useful for instance to maintain references to concepts like 'system dependencies', or to all error messages that can be generated. The three codes are '@^', '@.', and '@?'; they differ only in the way the index entry will be typeset. In each case the index entry is specified as a control text terminated by '@>'; control code and control text will be removed by CWEAVE, but the control text will appear in the index, followed by the section number(s) where the control code occurred. For '@^', '@.', and '@?', the index entry will be set respectively in roman type, in typewriter type, and as argument to the control sequence '\9' (which is undefined in the standard format, but which the programmer may define in limbo). The first possibility is most suited for general concepts, the second for strings that occur in the program, and the third for any further special purpose one may think of. These control codes can be put either in the TEX part of a section or within C code; the effect will be the same, but this allows the programmer to put the control code in such a place that it is most likely to remain in the right place in case the section should be reorganised and possibly subdivided. Like for references to identifiers, one can make an index reference underlined by prefixing the corresponding control code with '@!'.

Unlike the control text forming a module name, the control texts discussed here (as well as those that have not been introduced yet) should be contained in a single line of input; also, no spaces are contracted or removed. The control texts are passed unchanged to TEX (with only '@@' being undoubled as usual), so that they can use TEX commands for special effects. Inside '@....@>' one can get the special characters occurring in '#$%^&{}~_\' by prepending a backslash, '\v' gives a vertical bar '|', and '\ ' gives a visible space '␣'.

The control texts are also used as a sort key to determine the place in the index where the entry appears. Different occurrences of these control codes are combined in the index only if there is an exact match of both control code and control text, and no merging takes place with identifiers whose name happens to be equal to the control text (however, their relative order in the index is unpredictable). In sorting, a collating sequence is used that differs from the standard ASCII order: alphanumeric characters appear at the end of the sequence, with upper and lower case being considered equivalent, and the space character appears at the beginning of the sequence. In case there are entries that cannot be correctly positioned by ordinary means, the following trick has been suggested by Knuth: define '\def\9#1{}' and represent the tricky entries as '@?⟨ sort key ⟩}{⟨ TEX code ⟩@>', where ⟨ sort key ⟩ contains sufficiently many characters to

uniquely determine the position of the entry in the index, and ⟨ TEX code ⟩ produces the index entry itself; this works because CWEAVE will write the index entry '\9{⟨ sort key ⟩}{⟨ TEX code ⟩}', which "expands" to '{⟨ TEX code ⟩}'.

Besides references from the index, CWEAVE provides cross-references, in the form of the section numbers that link the (first) defining occurrence of a module name with the places where it is used and cited. There is also a mechanism for the user to explicitly state similar cross-references in the TEX part of a section, so that it is possible make a reference to another section (where some related matters are treated), that will remain correct if sections are renumbered. The mechanism is simple: in the section referred to, one places the control code '@:', followed by a control text serving as a label, and at the place of reference one uses '@#', followed by the identical control text (both control texts are terminated by '@>'). The rules for placing '@:' are the same as for '@^' and its relatives, except that '@!' has no effect here; the control text will not appear in the index, and there is no conflict when the same string is used as an identifier or index entry.

For '@#' and its control text, CWEAVE basically substitutes the section number of the matching '@:' code, but because there might be multiple occurrences of '@:' with the same control text, the precise replacement rule is a bit more complicated. The replacing text is precisely what would follow "See also section" in a cross-reference for a module name: one or more section numbers in increasing order, separated by commas and "and" as appropriate, and preceded by a space and, in case there is more than one section number, by an 's' before that space. This is set up so that a reference of the form 'section@#label@>' will generate a proper reference, whether or not there are multiple definitions of the label. One can also use '\Sec@#label@>' since in the standard format '\Sec' expands to '§' and '\Secs' to '§§' (in this case the space produced by '@#' is ignored after the TEX control sequence); by defining other TEX macros one could do anything one likes with the text provided by '@#'. Although '@#' cannot be used directly in comments and module names, it is possible to capture its text in a macro definition (within a TEX part) and use that macro instead.

Control codes for layout in programs: '@,', '@|', '@/', '@)', '@\', '@+', '@;' As we mentioned before, CWEAVE formats the program fragments and pieces of C code by inserting formatting controls in the the output based on a syntactic analysis of the C tokens of the program fragments; in particular the layout of the code in the source file is completely ignored. Although this automatic formatting usually works well provided that CWEAVE succeeds in parsing the program fragment (possibly with help of some codes already discussed), there may still be occasions where one is not quite satisfied by the result. If one wishes certain constructions to be systematically treated in a different way, then a more pleasing style might be available by calling CWEAVE with certain options set; if not, then there is always the possibility of changing the grammar or layout rules of CWEAVE (that program was written in a way that tries to make this as easy as possible, but it still requires some careful study of the relevant chapters of the CWEAVE source document). However in some cases one simply wants to override the general rules in specific cases by adding or removing a few formatting controls. There are a number of control codes which can be used to do that. These codes are ignored by CTANGLE; since most of them deal with line breaks, their importance for '|...|' fragments is minimal.

The control code '@,' will insert a thinspace (a small amount of horizontal white space) where it is placed. Within an statement '@|' may be used to indicate a place where a line break may be optionally taken (with no associated penalty), when the statement is too long to fit on a single line. Note however that optional breaks are already allowed at most operator symbols, with a penalty that increases with the operator priority and the number of enclosing parentheses, so CWEB will almost always succeed in finding very a reasonable break point in long expressions. A line break can be forced by '@/'; this can be used for instance between statements (if line breaks are not already forced there), in order to group related statements on one line rather than simply as much as possible. The code '@)' will also force a line break, and in addition create a bit of vertical white space to give an even more visible separation. (CWEAVE will never issue more than one line break on the same place, so there is no problem if a line break was already present on that spot.) The code '@\' is another variation: it forces a line break and backs up the next line by one indentation unit. It is useful before a module name that represents one or more cases in a **switch** statement: this will make the name line up with the case labels.

Finally, '@+' cancels any (forced) line break that might be inserted by CWEAVE at the point where it is placed, and replaces it by a space with optional line break (the kind of space that is usually inserted between

statements). Its main use is to force small conditional or loop statements onto a single line when CWEAVE would otherwise use a multiple-line layout. Because the line can still be broken at the inserted space, such one-liners do not make it impossible to retypeset the program in a narrower column. A warning is in place however if, as a result of applying '@+', a substantial stretch of C code is void of forced breaks, and that code contains constructions that affect the indentation level. TEXnically speaking, the indentation at optional breaks is governed by the hanging indentation parameter of TEX, whose value is constant throughout a paragraph, which in this case is everything between two forced breaks; under the mentioned circumstances the amount of indentation at optional breaks can be unexpected and inappropriate.

For convenience, an alternative method is provided to fit compound statements on a single line, and similarly for **struct** and **union** specifiers. Instead of writing '@+' on every place where CWEAVE would otherwise force a line break (which incidentally depends on the chosen layout style), it suffices to place '@;' immediately after the opening brace. This will activate a different set of layout rules than is normally used, which will not insert forced breaks between the declarations and statements of the compound statement, respectively between the fields of the **struct** or **union** specifier. In the case of a compound statement, any forced breaks caused by conditional or loop statements appearing directly inside the compound statement are also avoided (but nested statements are not affected, so they should be handled separately if present, possibly using another '@;'). Compound statements starting with '{@;' will be treated as if they were simple statements in further parsing, which may affect formatting; for instance, if the statement is the branch of a conditional it will be placed on the same line as the **if** or **else** controlling it. If this is too much of a good thing, a forced break may be explicitly inserted at the beginning and/or end of the compound statement; in fact the sequence '@/{@;' is a fairly common one.

There is another use of '@+', which does not cause any breaks to be cancelled, but where on the contrary the purpose is insert white space. It applies when a long string constant is needed, for which the string-break feature is used: a sequence of strings separated by white space only will be concatenated by the compiler into a single string. Although CTANGLE will correctly insert a space between any two consecutive strings, CWEAVE (guided by syntax rather than by lexical structure) will simply juxtapose them; by inserting '@+' between the strings, one guarantees that in the printed document there will either be a horizontal separation or (if the constituent strings themselves are already long) a line break. Incidentally, if the problem of breaking a string is in the source file rather than in the printed output, one can use the traditional solution of an escaped newline within the string; CWEAVE will treat this as if the parts of the string were on the same source line. If one should create a string in this way that does no fit on a single line of output, a break will be introduced automatically at a some point, which will be typeset as if a string-break was used. In very long strings however it is better to write string-breaks explicitly; for strings broken only by escaped newlines, the same length limit holds as for module names (1000 characters).

Codes for special items in* C *code: '@p', '@v', '@t', '@&', '@=', '@' ' Contrary to TEX text, pieces of C code are broken up into tokens by both CTANGLE and CWEAVE, stored internally and output at some later time after having undergone some processing. This makes it potentially difficult to put something into C code that CWEB is not prepared to handle. Since C is a much more regular language that TEX, occasions where one would need to do such a thing should be quite rare, yet some escape mechanisms have been provided, which we treat in this subsection.

The code '@p' can be used to explicitly specify the place where the preprocessor directives generated by '@d' and '@h' commands will be placed in the C file. Multiple use of '@p' is allowed; as soon as it is used at least once, the default placement at the beginning of the C file is cancelled. This code provides the only way to write the directives generated by '@d' and '@h' to an auxiliary output file. In the formatted output this code is represented by the pseudo-module '⟨ Preprocessor directives ⟩', which (like preprocessor directives embedded in a program fragment) is set on a separate line and does not otherwise affect the formatting of the surrounding code.

Two other codes are intended mainly for use within '|...|'. As mentioned earlier, '@v' represents the bitwise-or operator. The code '@t' is followed by a control text, which can be used to insert any TEX symbols into a C expression; the result gets category *expression* but (if used in a program fragment) does not produce any actual C code. It is for instance possible to obtain '$phi < \pi/2$' by writing '| phi < @tπ@> / 2 |', or if one prefers, to get '$phi < \frac{\pi}{2}$' by writing '| phi < @t$\pi\over2$@> |'. The control text is put

into an `\hbox` that will appear at the specified point in the formula. One might imagine using '`@t`' as a means to sneak in TEX commands that will modify the formatting produced by `CWEAVE`, but this is strongly discouraged unless one thoroughly understands that formatting and the way it is obtained.

The codes '`@&`' and '`@=`' are intended as a means to alter or bypass the processing of C tokens by `CTANGLE`; they should only be used in very exceptional situations. The code '`@&`' forces `CTANGLE` to output the symbols to the left and right of it directly adjacent to each other. Normally `CTANGLE` inserts space between two symbols if it thinks this is necessary for lexical reasons, regardless of whether a space was present in the input. Items with a lexical structure unknown to `CTANGLE` might confuse it, so that it would output a spurious space; this space could then be eliminated by '`@&`'. For instance, an earlier version of `CTANGLE` would not recognise '`1000000UL`' as a constant, and consequently it output a space before the '`U`', so that the C compiler could not recognise it either; this problem could then be remedied by inserting '`@&`'. No similar cases are known for the current version of `CTANGLE`.

The code '`@=`' can be used to place some text in the C file that `CTANGLE` will not produce by ordinary means: the control text following '`@=`', up to the next '`@>`' is copied verbatim to the C file (with '`@@`' undoubled as usual). If some special compiler activity, or some action by another tool, is triggered by the occurrence of some special form of comment in the C code, then such a comment can be placed using '`@=`' (normally comments are removed by `CTANGLE`). Also, should `CTANGLE` unjustly decide that two symbols need no space in between them, then a space can be forced by writing '`@= @>`'[†]. The control text will be set in typewriter type and framed in a box by `CWEAVE`, so that it stands out clearly; it is syntactically neutral (like a comment).

Finally, the code '`@'`' can be used to introduce a single-character constant, in the same way as the character '`'`' does in C. The difference between the two ways of specifying this value is that `CTANGLE` will replace `@'`*c*`'` by the (decimal) numeric ASCII value of the character *c*, whereas `'`*c*`'` is passed on to the C compiler, which will evaluate it to the same value. The feature is therefore of little use in the current version of `CWEB`, which assumes the ASCII character set, but is provided as an aid in writing programs that will be easier to port to non-ASCII versions of `CWEB`. In such systems `CTANGLE` should still use the ASCII code to compute `@'`*c*`'`, while `'`*c*`'` represents the internal code for *c*. The idea is that one can then (as is done in the program TEX) map all characters on input to their ASCII equivalents, perform all internal manipulations independently of the externally used character set, and convert back to that code on output.

Control codes behind the scenes: '@s', '@q', '@l' The control codes of this subsection have in common that their use is never essential, but can be convenient in some situations, and is largely or completely invisible in the `CWEB` document. They are also the only control codes allowed in limbo.

The code '`@s`' has the same effect as '`@f`', but produces no output in the `CWEB` document. It can be used as a subsectioning code, just like '`@f`', but no comment should follow the two identifiers it applies to in this case (since there is nothing to attach the comments to); alternatively '`@s`' can be used in limbo. In either case the format definition is noted but nothing is written to the TEX file. One might prefer to use '`@s`' in situations where showing a **format** definition is considered to be more distracting than informative. Also, if a header file *h* is included by **#include** rather than by '`@h`' or is located in a place where `CWEAVE` cannot find it, and it contains typedef declarations, then *h* could be accompanied by a file containing a line '`@s ident FILE`' for each typedef identifier defined in *h*, which can be read in by means of '`@i`' by any `CWEB` file that includes *h*. This method of passing information between files is more error-prone than having `CWEAVE` scan the header file however, so the latter method is to be preferred whenever possible.

The code '`@q`' is followed by a control text, and is completely ignored both by `CTANGLE` and `CWEAVE`; it can be used either in TEX text (even in limbo) or in C code. It can be used to make comments relevant only when the source file itself is being read, particularly within C code, where TEX comments cannot be used for this purpose. For instance, it can be used to put a descriptive or identifying comment at the beginning

[†] One case where this would be necessary is the famous example '`#123E + 1`': the C standard states that unless a space is put between the '`E`' and the '`+`', the preprocessor should treat this as a single number (a kind of mixture of a hexadecimal and a floating point constant), which turns out not to be valid, causing an error. `CTANGLE` however never places a space between an identifier and an operator (even if one was present in the input), so the way to get this expression properly through the compiler is to write '`0x123E @= @> + 1`' (since this bug is now documented, it has become a feature).

of a file included using '@i'. This code can also be used to accommodate any other tools than CTANGLE and CWEAVE that might inspect the source file, e.g., if a text editor tries to match braces and the like, it is unlikely to correctly handle the complicated lexical structure of CWEB files in all cases, and an occasional brace contained in a '@q' control text may help to keep it happy. Such occurrences of '@q' are best removed however when source files are made public.

The code '@l' is used to allow certain 8-bit characters (i.e., characters with values in the range 128–255) to be used in identifiers. Doing so is only useful if measures are taken to ensure that TEX can handle these characters properly. TEX version 3.0 and newer can handle 8-bit characters in the input, but the standard fonts do not have any characters in positions 128–255, so one has to either load other fonts that do have characters in those positions, or define such characters to be active characters that somehow produce an appropriate glyph in the current font. For identifiers the relevant font is text italic (selected by '\it'), but if these characters are available for identifiers, one will probably also want to use them in TEX text (including module names and comments), so other fonts should be provided for as well. CWEAVE does not take any special measures for 8-bit characters, and just passes them on to TEX (when they occur in C code outside comments and module names, they are assumed to be part of an identifier). However, since such characters cannot be used in actual C identifiers, CTANGLE must replace them by characters that are valid in C identifiers (letters, digits, and underscores). The code '@l' can be used to specify which translation CTANGLE is to use for a given 8-bit character. The code should only be used in limbo, and have the form '@l ⟨ char number ⟩ ⟨ translation ⟩', where ⟨ char number ⟩ specifies the character by a pair of hexadecimal digits in the range 80–FF (without leading '0x'), and ⟨ translation ⟩ is a string of up to 9 characters that are valid in C identifiers, terminated by a space. While copying limbo material, CWEAVE replaces '@l' by '\ATL'; its default definition will make '@l fc ue ' print a paragraph saying '**letter ü tangles as "ue"**', assuming that '{\it\char "FC}' indeed produces '*ü*'; by stating '\noatl' the definition can be changed so that nothing appears at all.

Control codes for tracing CWEAVE***: '@0', '@1', '@2', '@3'*** As will have become clear by now, the most sensitive part of CWEB is CWEAVE's parsing mechanism, and occasionally something may go wrong with it, so that an awful result is produced. We have already discussed the means available for corrective action, but sometimes it can be a problem to find out just what is causing the trouble. Sometimes the reason is an actual syntax error, which is best located by applying CTANGLE and a C compiler to the CWEB source, but as noted before, CWEAVE may have a problem that a C compiler does not experience. For this reason, its parser can produce diagnostic messages on the terminal, showing details about its actions and any anomalies found. The amount of diagnostics produced is controlled by a level that may take values from 0 to 3, and can be selected by one of the control codes '@0', '@1', '@2', and '@3'. These codes can be placed in the TEX part of sections or within C code, and they determine the level until the next such code or the end of the file; the initial level is 0, or 1 if CWEAVE was called with the option '+d'. Since a complete C fragment is read in before parsing starts, the level is constant throughout each fragment, and determined by the value at the end of the fragment.

The diagnostic output uses abbreviations for syntactic categories, e.g., 'unop' and 'binop' stand for unary respectively binary operators, and 'op' stands for operators like '∗' that can be used either way; 'exp' stands for an expression, 'decl' for one or more declarations, 'stmt' for one or more statements. Simple symbols like braces and commas stand for themselves, as do many keywords; 'for' stands for '**for**' or '**while**', and 'int' for a type, storage class specifier or typedef identifier. A complete list of category abbreviations can be found in the source code for the function *print_cat* in CWEAVE. (To fully understand the parser's diagnostic messages one has to be familiar with the parsing algorithm and the grammar rules, but for error detection a detailed understanding is usually not required.)

At level 0 the parser will not produce any diagnostic output. At level 1 it will report any C fragment that could not be recognised as a single syntactic entity, which is a good indicator of grammatical problems and possibly of ugly output. It can be argued that level 1 is the natural level to use (which is why the '+d' option is provided), since not getting any diagnostics when there are syntax problems only gives a false impression that things are in order; after all, nobody would want to use a compiler that would spare the programmer its diagnostics for syntax errors, but instead would produce unreliable code. In a syntactically correct CWEB program it is almost always possible to apply '@f', '@;', '@[' and '@]' in such a way that no diagnostic output

is produced at level 1; indeed this is has been done for all sources of CWEBx itself.

The diagnostic messages produced at level 1 print the successive categories of the sequence of recognised items, which could not be combined into any larger entity. Interpreting such a message takes a bit of practice, as one has to guess which part of the program fragment corresponds to each category printed; however, a look at the (badly) formatted output can often be helpful. The boundaries between the entities corresponding to the printed categories can often be recognised by the fact that some form of layout is obviously missing, and a space appears instead; for instance, if a closing brace of a compound statement is not preceded by a line break, then something inside that statement must have prevented it from being recognised by the parser, and its opening and closing brace will occur among the printed categories.

At levels 2 and 3 the parser will print the result of every single step it takes; this extremely verbose mode can be used to trace the exact steps by which the parser obtains its result. Detailed knowledge of the set of grammar rules is assumed, and these levels are mostly useful to those who wish to study or modify the rules. The set of rules can be found as a chapter of the CWEB source document for CWEAVE, or can be obtained separately by running '`cweave -x rules`' (ignoring the warning about an unused module) and '`tex rules`'. After each reduction step the number of the rule used is printed, followed by a list of categories after reduction, with the one that was formed by the reduction step enclosed in inverted angle brackets. Tracing at level 3 is even more esoteric than at level 2: all categories printed will have an additional character at both ends, indicating whether TEX should be in math mode ('`+`') or in horizontal mode ('`-`') at that end of the item, or that it doesn't matter ('`?`'); this may help to explain the positioning of math shifts ('`$`') in the TEX output, which is controlled indirectly by the grammar rules.

To reduce the amount of output, all categories that have not yet been considered by the parser are replaced by an ellipsis. The sequence of categories before the reduction can be found by looking up the reduction rule with the given number. Here is some sample output for a simple piece of C code at level 2.

```
Tracing after l.3:
@2 |if (n>0) printf("n-1 = %d.\n",n-1);|
  2: if ( >exp< ) ...
 10: if >exp< exp ...
110: >if_head< exp ( ...
  6: if_head exp ( >exp< op ...
  3: if_head exp ( >exp< ) ...
 10: if_head exp >exp< ;.
  7: if_head >exp< ;.
 80: if_head >stmt<.
117: >stmt<.
```

We see that the first three steps reduce '**if** $(n < 0)$' to an `if_head`; then '`"n-1␣=␣%d.\n"`, n' is combined to an expression, after which '-1' is incorporated as well; then the statement calling *printf* is reduced in three steps, and finally it is combined with the '`if_head`' to form another statement.

Even this small example shows that CWEAVE parses the code in a different way than a C compiler would. This is partly due to its strict bottom-up strategy, which is largely unaware of context: parentheses (rule 10) and a comma (rule 6) are incorporated by the expression syntax, even when they actually figure in a conditional statement or function call. Furthermore, some distinctions are irrelevant for determining the proper layout: the "comma operator" is unjustly given precedence over the minus operator, but the printed output will be no different for it.

8 Some features of the standard format

The TEX file produced by CWEAVE will begin with loading the standard format from the file cwebxmac.tex, whose definitions control the typesetting process: CWEAVE communicates with TEX mostly by using macros defined there. Most of them have very short names, in order to limit the size of the TEX file; one should be aware that most of the single-letter control sequences and numerous two-letter ones are in use by CWEB, and are not available for other uses (when in doubt, consult cwebxmac.tex). All of the macros defined in plain TEX for accenting letters have retained their meaning however, except '\.' (for the dot accent), which is replaced by '\:'. Some of the macros of the standard format can be of interest to the literate programmer, either because they can be used directly in TEX text (indeed, some are not used by CWEAVE, and are only intended for this purpose), or because they can be redefined in limbo to alter the formatting of the program.

In formatted C text, many operators are represented by macros that produce the appropriate symbols; by changing the definition of these macros, one can alter their appearance. Here is a table of the relevant cases.

operator	=	==	!=	<=	>=	&&	||	!	&	|	^	~	<<	>>	++	--	%	->	##
macro	\K	\E	\I	\Z	\G	\W	\V	\R	\AND	\OR	\XOR	\CM	\LL	\GG	\PP	\MM	\MOD	\MG	\SS
symbol	⇐	=	≠	≤	≥	∧	∨	¬	&	|	⊕	∼	≪	≫	++	−−	%	→	##

When such a macro is redefined, it is best to consult the original definition first, since it often issues a penalty, and it is best to retain this. Formatting of ordinary identifiers and keywords is performed by '\\' and '\&', which have one argument, that is typeset in italic respectively boldface type; similarly '\.' is used for items in typewriter type, such as strings and all-caps identifiers. In the argument of '\.' special characters can be used if escaped, as discussed for '@.'. For '&' in ordinary text '\AM' can be used (rather than '\&'). For names in all caps, like 'ASCII', or 'UNIX', the macro '\caps' is provided, which makes them slightly less obtrusive by selecting a smaller font; for 'C' and 'C++' the macros '\Cee' and '\Cpp' are provided. Typesetting of comments, C++ one-line comments, and numeric constants is controlled by the macros '\C', '\SHC', and '\T', respectively; these can be redefined if a different style is desired.

The dimensions of the pages can be controlled by setting the parameters '\pagewidth', '\pageheight' (the height of the text area), '\fullpageheight' (the height including running head), and '\pageshift' (extra displacement of odd numbered pages with respect to even numbered ones), and then invoking the macro '\setpage'. A magnification can be applied to the entire document by saying '\magnify{n}',where n is the magnification in thousandths of the ordinary scale; this should precede any changes of the page dimensions, but if no changes are made, the page dimensions will be set to their standard values, unmagnified. The unit of indentation can be set by '\indentation{⟨ size ⟩}'.

The title of the program is taken from the macro '\title', whose default value is the basename of the program source file, converted to upper case. It is used in running heads and in the table of contents. Another part of the running heads is set to the chapter title by sections starting with '@*', but by defining the macro '\gtitle' in limbo the corresponding text for the running heads on any pages before the first such section can be set (the default is 'CWEB output'). By invoking '\titletrue' the running head can be suppressed for one page; this is useful if the text in limbo produces a title page. The date of processing (by TEX) can be included in the document before the first section by putting '\datethis' in limbo; it can be placed on the table of contents by saying '\datecontentspage'. At the end of the document one normally has an index, a list of module names and the table of contents, in that order, but TEX can be made to stop short of any one of these by invoking respectively '\noinx', '\nomods', or '\nocon'; as already mentioned, stating '\changesonly' will limit the printed output to the sections affected by the change file. The appearance of the table of contents can be controlled by redefining '\topofcontents' and '\botofcontents': these macros determine the material that comes above the table and below it, including its title and the glue needed to fill up the page height. The page number of the table of contents is assigned from '\contentspagenumber' (the default is 0), but it will not appear in print because the running head is suppressed on that page.

It may be noted that CWEB documents contain some fixed phrases in the English language, such as the cross-references at the end of sections. These are not produced directly by CWEAVE however: one could adapt CWEB to a different language by redefining the macros '\A', '\As', '\Q', '\Qs', '\U', '\Us', '\ET', '\ETs', '\ch', '\postATL', '\ATP', '\today', '\now', and parts of '\fin' and '\con'.

9 Comparison with Levy/Knuth CWEB

As was mentioned in the introduction, CWEBx is derived from an earlier CWEB system (itself derived from Knuth's WEB), that was written and distributed by Sylvio Levy and Donald E. Knuth, and that CWEB system has independently evolved into a version currently distributed as CWEB 3.3. Both CWEBx and Levy/Knuth CWEB have undergone changes with respect to their common ancestor, although the spirit of the system has not fundamentally changed in either case. Since we considered it undesirable to have a great divergence between systems that both intend to be "a WEB system for C", we made a conscious effort to reduce the differences between CWEBx and Levy/Knuth CWEB by including the extensions of the latter into CWEBx as well. There was one deliberate exception: we made no attempt to extend the grammar used by CWEAVE to handle the full C++ language[†]. On the other hand, hoping to fully bridge the gap between Levy/Knuth CWEB and CWEBx for C programs, a compatibility mode was added to CWEBx in which it tries to mimic the behaviour of Levy/Knuth CWEB in all aspects that are relevant to the programmer (even in cases where that behaviour is undocumented), at the price of losing some possibilities that CWEBx normally has.

The description of the differences between Levy/Knuth CWEB and CWEBx can be divided into two parts: the differences between Levy/Knuth CWEB and the compatibility mode of CWEBx, and the differences between CWEBx with and without compatibility mode. The former differences are minimal, but hard to enumerate precisely, as they are mainly a matter of difference in implementation. The latter differences are much more significant, but they can easily be listed, since precise details of those differences can be found by looking up all index references of the identifier *compatibility_mode* in the sources for the programs of CWEBx, and (for the differences that only involve processing by TEX) the contents of the file cwebcmac.tex that modifies the cwebxmac format to emulate the environment provided by the cwebmac format of Levy/Knuth CWEB.

As was stated, the differences between Levy/Knuth CWEB and the compatibility mode of CWEBx should not be relevant to the programmer, but if one uses Levy/Knuth CWEB in a way that relies on knowledge of intimate details of its implementation (which are not described in the manual but can be learned from studying the sources), then it is certainly possible to find such differences; this applies particularly to using the TEX code produced by CWEAVE in unusual ways. Unfortunately there is no clear specification of which aspects of CWEB are well defined so that the user can safely rely on them, and which aspects are implementation details. We have taken a pragmatic attitude by reproducing all aspects that are described in the manual, and moreover many undocumented aspects, enough to process the sources of Levy/Knuth CWEB itself and of the Stanford GraphBase without problems. To give an impression of the kind of differences that remain, we shall list some of the known ones.

The output files written by CWEBx are not equal to those written by Levy/Knuth CWEB, so processing them otherwise than directly by a C compiler respectively by TEX may reveal some deviations. For instance, entries for the index and the list of module names are written using the control sequence '\I' by Levy/Knuth CWEB, but since '\I' is also used for representing the operator '≠', this causes problems for module names in which that operator is used; therefore, CWEBx uses '\@' instead. In CWEBx unbalanced braces or parentheses in program fragments or macro replacement texts are reported and corrected by CTANGLE, as an aid in catching programming errors early; in Levy/Knuth CWEB this is not done (but programs with such unbalanced symbols will still bring CWEAVE into serious problems). In compatibility mode the definition of '\PB' ensures that '|...|' can always be used from within math mode; in Levy/Knuth CWEB this is true only in simple cases. Comments in the TEX parts of sections (following a non-escaped '%' character) are ignored and removed by CWEBx, whereas in Levy/Knuth CWEB they are processed normally and copied to the output, which may cause spurious index entries, and in exceptional cases may cause part of the comment to appear in print. The grammars used by CWEAVE in the two systems are quite unrelated; for CWEBx, the only guideline in constructing the grammar has been the ANSI/ISO C syntax. When processed by CWEBx with the proper options selected, CWEB documents will look similar to the result produced by Levy/Knuth CWEB, but not identical. Unlike Levy/Knuth CWEB, CWEBx places optional breaks at operators, reflecting their priority and

[†] C++ has a significantly more complicated syntax than that of C, which is already far from simple, and it has some forms of context dependence that make it doubtful whether to CWEAVE could ever reliably handle C++ in full generality (and even then, C++ is a moving target). Most effort was spent on getting the grammar for C correct; support for C++ was restricted to some extensions of C that could be incorporated easily.

nesting inside parentheses. In Levy/Knuth CWEB, if '@␣' is immediately followed by a subsectioning code, then the output from the subsectioning code (e.g., #define for '@d') will be placed on the same line as the section number, but if anything, even an extra space, comes in between, or if the '@' in the sectioning code was followed by a newline rather than by a space, then that output is moved to the beginning of a fresh line; in CWEBx output from a subsectioning code never appears on the same line as the section number.

CWEBx has a number of control codes and a number of command line options that Levy/Knuth CWEB does not have; moreover there are some control codes that Levy/Knuth CWEB does have, but under different names (that are used for other purposes in CWEBx). In compatibility mode such control codes have the same interpretation as in Levy/Knuth CWEB, but if there is no such interpretation while there is one in CWEBx, the latter is taken;. This means that in compatibility mode one can use the control codes '@v', '@\', and '@~', which are ignored in Levy/Knuth CWEB, while '@?' and '@)' can be used as aliases for '@:' and '@#', respectively; furthermore the command line options controlled by the characters 'l', 'd', 't', 'e', 'a', 'u', 'w', 'm', and '+' are also extras with respect to Levy/Knuth CWEB. Finally the control code '@;' retains all its CWEBx uses in compatibility mode, except that of modifying the category of module names (which is different anyway), whereas in Levy/Knuth CWEB it can only be used as an invisible semicolon.

The most direct difference between CWEBx with and without compatibility mode is that in compatibility mode the control codes '@h', '@:', '@#', and '@p' are translated into respectively '@p', '@?', '@)', and '@c', which implies that the meaning of '@h', '@:', and '@#' as described in this manual are not available in compatibility mode. There is also an important syntactic adjustment: in compatibility mode module names are always treated as expressions (which means they must almost always followed by '@;' to make the combination behave as a statement, or by ';', which will however become an empty statement in the C program). Then there are a few points where compatibility mode lifts certain restrictions (thereby reducing the diagnostic capabilities). All 8-bit characters will be accepted by CTANGLE, whether or not an explicit translation was specified using '@l'; the default translation used corresponds to '@l *NN* X*NN*', where *NN* is the 2-digit hexadecimal code, in upper case, for the character. Module names used within '|...|' do not have to be the final item. Rather than performing '@i' inclusions before the change file is matched, the order is more or less reversed (but if some '@i' line is not replaced by the change file, then the included file will again be scanned for changes); consequently '@i' is allowed (and meaningful) in the change file.

The remaining alterations affected by compatibility mode are fairly minor. Trailing digits in identifiers will not be set as subscripts. The TEX control sequence corresponding to identifiers that are given the category of *TEX* by means of a format definition will be processed in math mode rather than horizontal mode. The code '@t' together with the following control text will not be treated as an expression in parsing, but as an inert item that sticks to the token to the right of it (unlike comments that are attached to the token to their left); this allows '@/@t\4@>' to be used in place of '@\'. Compound assignment operators like '+=' are treated as two separate tokens; this implies among other things that the operator '|=' must be entered as '@v=' when used inside '|...|'. In index entries produced by '@^', '@.', or '@:', the underscore character will be automatically escaped by a backslash, unlike other special characters (this makes it harder to enter formulas with subscripts into the index). The output of '|...|' is made an argument to the control sequence '\PB', whose default definition puts its argument into an '\hbox'; this makes it safe to use '|...|' inside math mode when using compatibility mode. Finally there are a few other small changes to the format used: C++ one-line comments will be formatted as if they were ordinary C comments, and the formatting of lines in the table of contents will be affected in font and spacing by the "depth" specified for that chapter title.

10 Summary of CWEB codes

For reference, we give a table with all the codes used in CWEB with their main characteristics. The letters in the column *where* indicate in which parts of the source text may immediately precede the code: 'L' indicates text in limbo, 'T' the TEX part of a section, 'M' indicates an intermediate part of a section (from '@d', '@h', '@f', or '@s'), 'C' the C part of a section, and 'c' pieces of C code within '|...|' (the letter 'M' is only used when the code terminates an intermediate part; inside the parts after '@d', '@h', and '@f', the letter 'C' applies). The column *frequency* indicates how commonly the code is used, with *regular* > *incidental* > *rare* > *emergency*; for the codes with *frequency* ≤ *rare*, no sensible use could be found within any source file for the CWEB system itself. The codes '@0'–'@3' that are only of temporary use are omitted from the table.

code	*meaning*	*where*	*frequency*	*remarks*
Sectioning codes				
`@*`	Start of chapter	LTMC	*regular*	`@**`, `@*`n, or `@*` Title.
`@␣`	Start of section	LTMC	*regular*	
`@~`	Start of section tied to previous one	LTMC	*regular*	
Subsectioning codes				
`@c`	Start of unnamed program fragment	TM	*regular*	also `@C`
`@<`	Start of module name	TMCc	*regular*	`@<` Module name `@>`
`@d`	'`#define`'; start macro definition	TM	*regular*	also `@D`
`@h`	'`#include`'; specify included header file	TM	*regular*	also `@H`
`@f`	Format definition; change syntactic category	TM	*incidental*	also `@F`
`@(`	Start module name defining output file	TMCc	*incidental*	`@(` file name `@>`
Parsing control codes				
`@;`	Invisible semicolon, or magic wand for syntax	Cc	*incidental*	
`@[`	Start of item forced to expression	Cc	*incidental*	
`@]`	End of item forced to expression	Cc	*incidental*	
Cross-referencing codes				
`@!`	Make index reference underlined	TCc	*rare*	
`@^`	Index entry in roman type	TCc	*regular*	`@^`index entry`@>`
`@.`	Index entry in typewriter type	TCc	*regular*	`@.` index entry`@>`
`@?`	Index entry formatted by '`\9`'	TCc	*rare*	`@?`index entry`@>`
`@:`	Define label for explicit cross-reference	TCc	*incidental*	`@:`label`@>`
`@#`	Explicit cross-reference to defined label	T	*incidental*	`@#`label`@>`
Layout control codes				
`@,`	Thin space	Cc	*rare*	
`@|`	Optional line break	Cc	*rare*	
`@/`	Forced line break	Cc	*incidental*	
`@)`	Forced line break with vertical white space	Cc	*incidental*	
`@\`	Forced line break, next line backed up	Cc	*incidental*	
`@+`	Cancel any line break, replace by space	Cc	*incidental*	
C *control codes*				
`@p`	Insert output from '`@d`' and '`@h`'	Cc	*rare*	also `@P`
`@v`	Bitwise or operator '|'	Cc	*incidental*	also `@V`
`@t`	TEX code within expression	Cc	*incidental*	`@t`TEX code`@>`; also `@T`
`@&`	Glue together adjacent tokens	Cc	*emergency*	
`@=`	Insert verbatim C code	Cc	*emergency*	`@=`verbatim C code`@>`
`@'`	ASCII constant converted to number	Cc	*rare*	`@'`c`'`
Silent control codes				
`@s`	Non-printing version of '`@f`'	LTM	*rare*	
`@q`	Ignored control text	LTCc	*rare*	`@q`any text`@>`; also `@Q`
`@l`	Specify translation of 8-bit character	L	*rare*	`@l` xx string ; also `@L`
Miscellaneous codes (these are not control codes)				
`@@`	Representation of '`@`'	LTCc	*incidental*	legal in control text too
`@i`	Insert subsidiary source file	any	*incidental*	also `@I`
`@x`	Start of change; old lines follow	any	*incidental*	also `@X`
`@y`	Middle of change; replacement lines follow	any	*incidental*	also `@Y`
`@z`	End of change	any	*incidental*	also `@Z`

CWEBx Manual