The WEB System of Structured Software Design and Documentation for C, C++, Fortran, Ratfor, and TEX User Manual

JOHN A. KROMMES
Princeton University
krommes@princeton.edu

FWEB Version 1.30
June 15, 1993

1. INTRODUCTION

This manual describes how to write programs in Knuth's WEB system, as adapted and extended by J. A. Krommes to handle the programming languages of FORTRAN (including both FORTRAN–77 and FORTRAN–90), RATFOR, C, C++, and TEX. We shall call this new version FWEB when necessary, but generally we'll just refer to the WEB system. This adaptation is a substantial modification of S. Levy's C version CWEB of WEB, which in turn was a complete rewrite in C of Knuth's original Pascal source. (TEX is written in the original Pascal WEB.) The principal design contributions to this version of WEB are (1) the concept of a current *language*, so that one can process code written in multiple languages in the same WEB run; (2) new production rules for FORTRAN, RATFOR, and TEX (and some modifications of Levy's rules for C); (3) a C-like built-in macro preprocessor; and (4) the ability to directly translate RATFOR into FORTRAN. In addition, many miscellaneous details have been changed and a variety of convenience features have been added. For example, a general style-file mechanism allows the user to customize various actions of the system.

FWEB understands C, C++, Fortran–77, Fortran–90, Ratfor, and TEX.

This manual covers quite a breadth of material, from the most introductory to very advanced. To help identify the level of difficulty, the sections are marked with 0, 1, or 2 dangerous bends (here shown as ♣) a la the TEXbook.

1.1 Previous authors, and the structure of this manual

The WEB system and this manual have evolved through contributions of several authors, principally Knuth and Levy, and the flavor of the system is best captured with those authors' original words. Levy attempted to highlight Knuth's original text by indenting it. Quoting from Levy's manual, "The bulk of this document consists of quotes from Knuth's memo 'The WEB System of Structured Documentation'; these quotes are clearly distinguished by their indentation, and apart from such substitutions as 'code' for 'Pascal', all other changes to them are explicitly indicated. This also serves to indicate which commands and features are common to all versions of WEB and which are characteristic of this version (of course if you're new to WEB you don't have to worry about this)." In practice, new users of WEB seem to find the indented style somewhat confusing; furthermore, the length of this manual has grown to a point where more than 50% of the text is new. Therefore, in an attempt to accomodate everyone, this user manual can be TEXed in two ways, depending on the setting of the TEX macro `\ifbilevel`, defined near the beginning of the macro package `fmanmacs.tex` for this manual. By default, that switch is false and most indentation is suppressed. (A few of the most significant quotes are indented in both formats.) To obtain the indented style, say `\bileveltrue`. In the bilevel mode, we shall preface Levy's remarks by "*[Levy]:*", and we shall feel free to interject additional remarks by enclosing them in square brackets or to skip extraneous material by using an ellipsis. The brackets and ellipses are also suppressed when the `bilevel` switch is false.

1.2 The origins of FWEB

(The contents of this subsection are totally irrelevant to the narrow goal of learning to use FWEB*. But they might be of some "cultural" interest.)*

FWEB was born of necessity. One of the focuses of my research in theoretical plasma physics has been the construction and solution of so-called "statistical closure approximations," the most famous example of which is Kraichnan's 1959 direct-interaction approximation (DIA). At some point around 1985 I decided that further progress would benefit greatly from numerical solutions of the DIA and related closures applied specifically to plasma turbulence. (Kraichnan, Herring, and others had long before done the analogous calculations for neutral-fluid turbulence.) So I began writing my DIA code—in standard FORTRAN. The result was a disaster.

I needed extensive memory allocation facilities, but FORTRAN–77 doesn't provide them. I simulated them using tricky array manipulations, but the code was difficult to debug and to understand. I wanted complicated data structures, but again FORTRAN–77 wasn't up to the task. Ditto for simple string manipulations, communication with the command line, and transparent, easy-to-write I/O statements. In other words, I didn't want FORTRAN–77 at all; I wanted C! So I threw away a year's worth of work and started over.

The second time around, I at least had a better sense of the scale of the project—it was large. Also, upon looking back at my original attempt, I realized that I couldn't understand the code at all, even though I had been fairly liberal with FORTRAN-style comments. Looking ahead, I also realized that graduate students would likely become involved. Since I have strong feelings about wanting my students to actually do physics instead of just computing, it became imperative to write the code using the most powerful documentation scheme available so they could quickly understand how it worked and could modify it as easily as possible. Knuth's article on "Literate programming" [1] had already provided a "religious experience" for me when I read it several years previously, tempered only by the fact that I hadn't been in a computing phase at that time. But faced with the demands of the DIA project, Knuth's excellent points all came back and I realized that WEB was the natural choice. Luckily, Levy had just announced his beautiful CWEB, so everything seemed to be in place.

Then, another setback. One of the important characteristics of plasma, as opposed to incompressible fluid, turbulence is the presence of linearly propagating waves—described most naturally with complex numbers. Unfortunately, C doesn't have an intrinsic **complex** type. C++ can easily be taught complex arithmetic, but good C++ compilers weren't widely available, especially on the Cray supercomputers on which I intended to work. One can, of course, easily **typedef** a C structure that duplicates FORTRAN's layout of a complex number, but short of function calls one can't in C perform arithmetic on complex structures in the natural way one does in FORTRAN. Readability was important. So was vectorization, which also wasn't available in the C compilers available at that time and in any case would probably be broken by function calls. So reluctantly I decided that the innermost, computational routines of the code should be written in FORTRAN.

Of course, no way was CWEB going to handle FORTRAN code gracefully. But my colleague Henry Greenside had been arguing strongly for the use of RATFOR as a superior tool, given that one was stuck with some sort of FORTRAN. I agreed, and also hoped that RATFOR's C-like syntax would be close enough to C's that CWEB would be adequate. And so I plunged ahead, writing in a mixture of C and RATFOR.

CWEB worked for RATFOR—sort of. One was able to obtain the invaluable index, and often the RATFOR routines typeset just fine. But there were constant annoying glitches because of the inevitable differences of syntax. After a few years of experience, it became clear that one really wanted a proper FORTRAN version of WEB. Also, the absence of a FORTRAN macro preprocessor was causing us major annoyance. Although

Knuth's original WEB had a rudimentary preprocessor, Levy had deleted it for CWEB, since C has its own preprocessor. I decided there needed to be one built in that would serve all languages equally well. Thus, the major design goals were clear: a new FORTRAN WEB that also supported RATFOR and C, with a macro preprocessor. So I learned how Knuth and Levy had implemented the syntax production rules, sat down with a FORTRAN reference manual, and began to write some FORTRAN productions. I thought this would take a couple of weeks. That was actually about right. Ultimately, however, my estimate of the total amount of work involved in realizing all of my ideas about an "industrial strength" WEB easily usable in the scientific environment was off by at least two orders of magnitude. (Well, I'm a *theoretical* physicist.)

Some years later, I'm still upgrading FWEB. Had I been left to my own devices, I probably would never have made it publicly available, since the conflicts between trying to complete the research I had intended to do and maintaining a reasonable robust code are almost overwhelming. But my colleague Charles Karney, a true expert both in TEX and scientific computation, began distributing codes written in FWEB and convinced me that a public announcement was the best way of both achieving some sort of standardization and accomplishing the debugging task. So an announcement was timidly made, and the response has been amazing. Most incredible has been the continuing patience of the many users who are faced with a utility that is still not bug-free and that they might have designed differently. FWEB is far from perfect, but it's much better than my original attempt because of the many well-thought-out suggestions and questions that I continue to receive. My profound thanks.

(Incidently, in the midst of all this a workable version of the DIA code was finally finished with the completion of John Bowman's Ph.D. thesis [2] at Princeton University. To John fell the terrible task of working with the early prototypes of FWEB. The demands of his project stressed the system to its limits, sometimes beyond, and elaborate kludges were sometimes necessary in order for him to continue in a timely way with his research. But John's excellent, properly skeptical remarks also measurable improved FWEB, and in the final analysis it's still not clear to me that we could have accomplished so much in a period of just a few years had we not had the principles of WEB and literate programming to help and guide us. So I'll both apologize to John for the early days as well as thank him for his patience and valuable suggestions. Although it's not uncommon for recent Ph.D.'s to want never to think about the thesis research again, I hope John will be an exception, since his contributions to both physics and FWEB have been outstanding.

1.3 Why is this *!@%*#@! manual so large?

As stated above, FWEB evolved from the excellent, tidy CWEB code of Levy and Knuth. Without CWEB and Knuth's original WEB, there would be no FWEB, so my debt to those authors is very large indeed. So I attempted to retain the original manual that came with CWEB, adding to it as necessary. That worked in the beginning, but as various features were added and, in particular, as I continued to try to explain WEB programming to a scientific user community that was strongly oriented toward vanilla FORTRAN, the manual accreted very much new material—too much for the present organization.

From at least two points of view, it is time for a complete rewriting of the manual. First, the appearance of Knuth's new book [2] on *Literate Programming* means that much background material can be omitted from the manual. Second, I'm probably past the experimental stage of FWEB; the functional design has mostly stabilized and I've accumulated enough experience to know where the troublesome points are. So there's just the issue of time to worry about. Unfortunately, that's nontrivial, so for now one is stuck with the present monstrosity.

However, the size and detail of this manual are alleviated in two ways. First, Marcus Speh [3] moderates an FWEB literate programming question and answer bulletin board and information service, from which elementary information can be easily obtained. Second, some of the appendices to this manual can be printed separately as a self-contained user guide. Third, emacs users can obtain online help from an fweb.info file that is distributed with the FWEB release.

2. The PHILOSOPHY of WEB

The WEB system consists of two processors, WEAVE and TANGLE. These are *software tools* in the sense of Kernighan and Plauger [4]. Together, these processors make important contributions to the two principle facets of software design: how to efficiently write a program that works and that can be easily maintained (TANGLE); and how to document what you have done so that both you and others can understand, appreciate, and later modify the code (WEAVE).

"The WEB system consists of two processors, WEAVE and TANGLE. These are *software tools*..."

2.1 The purpose of the processors

If one is not interested in obtaining documentation, the TANGLE processor can be used stand-alone, as a powerful preprocessor for any of the source languages. The WEAVE processor must be used in conjunction with TEX in order to obtain the documentation. Here documentation refers both to prose exposition of the logic and algorithms of the code, as well as to a typeset listing of the code itself. Note that, strictly speaking, you don't actually have to know TEX in order to document your code with WEAVE. Just typing straight text at the appropriate places will produce documentation whose quality will be far in excess of what can be achieved with comment lines embedded in the source code. The code itself is typeset automatically in a visually appealing format by using special TEX macros about which the user need not generally be concerned. However, the more features of TEX you employ, the higher quality your documentation will be. This is particularly important when the algorithms you are explaining are based on complicated mathematical formulas. The ability to typeset intricate mathematics right next to the code that implements the algorithm (or even as a comment *within* a line of source code), instead of trying to spell out the symbols in words in a series of comment lines, is so extraordinarily useful that it becomes difficult to understand how one could ever have gotten along without it. Compare, for example, the standard Fortran commenting style

```
C Perform the integral over v parallel from alpha to beta of the Maxwellian:
    call integrate(x,alpha,beta,fM)
```

with what FWEAVE can achieve:

call *integrate*(x, α, β, f_M) // Perform $\int_\alpha^\beta dv_\parallel\, f_{\rm M}(v_\parallel)$.

Here one needs to know something about TEX (not very much) in order to use math mode in the text of the comment; however, the typesetting of the equation itself was performed automatically. (If you want to know precisely how this was done, read about "Overloading identifiers" below.) Time spent learning the fundamentals of TEX will be amply repaid in dramatically increased productivity—and it's not hard to learn to do even rather powerful operations.

2.2 Top-down programming and structured design

The fundamental logic of the WEB system encourages "top-down" programming and "structured" design. Quoting from Kernighan and Plauger, "Top-down design and successive refinement attack a programming task by specifying it in the most general terms, then expanding these into more and more specific and detailed actions, until the whole program is complete. Structured design is the process of controlling the overall design of a system or program so the pieces fit together neatly, yet remain sufficiently decoupled that they may be independently modified. ... Each of these disciplines can materially improve programmer productivity and the quality of code produced."

"The fundamental logic of the WEB system encourages 'top-down' programming and 'structured' design."

The WEB system encourages you to work top-down by giving you the ability to break up your code into independent segments (called "*modules*"). Often, modules correspond to individual subroutines. However, this is not necessary and the WEB system goes further by strongly encouraging you to also subdivide long subroutines into additional modules that can be given names for readability, put into a separate place, and explained as logic dictates. WEAVE prints out your explanations (and associated code fragments) in the *logical* order in which you have decided to explain the code. TANGLE strips off the explanations and rearranges the code into the *physical* order in which the compiler should see it. For example, in FORTRAN all **common** declarations must physically be placed at the beginning of a subroutine, before executable code. However, various parts of such declarations may be quite unrelated, and should logically be explained in separate, not necessarily adjacent, sections of the documentation. As another example, in C code function prototypes should appear at the beginning if they are to be useful; however, it's distracting to find a very long list of prototypes at the beginning of the documentation. WEB allows one to both place the prototypes at the end of the documentation and to tell the compiler to read them first. Finally, it might be desirable to explain the output routines of a code before the input routines, since one might then better know exactly what should be input.

Though top-down programming is easy with WEB, bottom-up coding is not excluded either. It's perfectly possible to write very low-level routines first, if that's convenient, and hook them into the web. The flexibility to mix top-down and bottom-up programming in a superior documentation environment means that one gets the best of all possible worlds!

Although WEB is simple in conception, it is quite sophisticated in practice because WEAVE uses the full power of TEX to typeset not only the expository documentation but also the actual code. (Note that automatically formatting source code is nontrivial, because WEAVE must understand a good deal about the language syntax in order to emphasize keywords, indent loops, and so forth.) Such visual enhancements help one to write good code, because it can be more clearly and logically explained and, therefore, better understood in the future.

Furthermore, the FWEB version of the WEB system goes significantly beyond this. In fact, FTANGLE is also a *macro preprocessor* and a *statement translator*. By allowing symbolic macro abbreviations for commonly used constructions, FTANGLE allows you to write significantly more concise and therefore more readable code. Furthermore, in its RATFOR mode it endows the FORTRAN language with a richer, C-like syntax that provides much more logical, flexible, and readable loop and conditional constructions. These constructions are translated directly into FORTRAN code. Although it is impossible to fully correct for the many design deficiencies of FORTRAN (except by writing in a better language such as C, which is highly recommended whenever possible), in many situations the RATFOR syntax is so close to that of the C language that users have complained that they can't remember in which language they're programming. FWEB deliberately adopts the point of view that the language syntaxes should be as close as possible because it is a common practice to mix languages, and uniform syntax should reduce the frequency of errors. When arbitrary choices of syntax have arisen, the design of FWEB has been strongly influenced by the solution adopted by the recent ANSI standard for C. For example, the macro preprocessor behaves like C's, not like m4 or Knuth's original preprocessor for his Pascal WEB.

"When arbitrary choices of syntax have arisen, the design of FWEB has been strongly influenced by the solution adopted by the ANSI standard for C."

2.3 Knuth's original description of WEB

We begin with Knuth's original description of WEB.

"The philosophy behind WEB is that an experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like TEX for formatting, and a language like C for programming. Neither type of language can provide the best documentation by itself; but when both are appropriately combined, we obtain a system that is much more useful than either language separately.

"The structure of a software program may be thought of as a "web" that is made up of many interconnected pieces. To document such a program, we want to explain each individual part of the web and how it relates to its neighbors. The typographic tools provided by TEX give us an opportunity to explain the local structure of each part by making that structure visible, and the programming tools provided by languages such as C or FORTRAN make it possible for us to specify the algorithms formally and unambiguously. By combining the two, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into a working software system that matches the documentation.

"Since WEB is an experimental system developed for internal use within the TEX project at Stanford, this report is rather terse, and it assumes that the reader is an experienced programmer who is highly motivated to read a detailed description of WEB's rules. Furthermore, even if a less terse manual were to be written, the reader would have to be warned in advance that WEB is not for beginners and it never will be: The user of WEB must be familiar with both TEX and the source language in which he is writing. When one writes a WEB description of a software system, it is possible to make mistakes by breaking the rules of WEB and/or the rules of TEX and/or the rules of the source language. In practice, all three types of errors will occur, and you will get different error messages from the different language processors. In compensation for the sophisticated expertise needed to cope with such a variety of languages, however, experience has shown that reliable software can be created quite rapidly by working entirely in WEB from the beginning; and the documentation of such programs seems to be better than the documentation obtained by any other known method. Thus, WEB users need to be highly qualified, but they can get some satisfaction and perhaps even a special feeling of accomplishment when they have successfully created a software system with this method."

"The structure of a software program may be thought of as a 'web' that is made up of many interconnected pieces."

2.4 How to use WEB

To use WEB, you prepare a file called COB.WEB (say), and then you apply a system program called WEAVE to this file, obtaining an output file called COB.TEX. When TEX processes COB.TEX, your output will be a "pretty printed" version of COB.WEB that takes appropriate care of typographic details like page layout and the use of indentation, italics, boldface, etc.; this output will contain extensive cross-index information that is gathered automatically. You can also submit the same file COB.WEB to another system program called TANGLE, which will produce for example, a file COB.FOR that contains the FORTRAN code of your COB program. The FORTRAN compiler will convert COB.FOR into machine-language instructions corresponding to the algorithms that were so nicely formatted by WEAVE and TEX. Finally, you can (and should) delete the files COB.TEX and COB.FOR, because COB.WEB contains the definitive source code. **Are you paying attention? Did you hear what I just said?** Examples of the behavior of WEAVE and TANGLE

Extensive cross-index information is gathered automatically.

are appended to this manual.

When you are using FWEB you should use the commands FWEAVE and FTANGLE to avoid confusion with the original Pascal WEB processors WEAVE and TANGLE, which are still supplied with the TEX distribution.

Let us review what has been said so far. Given a FORTRAN code that has been prepared in the WEB format in file test.web, to get compilable code you say (using upper case solely for emphasis to distinguish the system command from the file name)

```
FTANGLE test
```

This produces the file test.for, which can be compiled, linked, and executed. To get the documentation of your code, you say

```
FWEAVE test
```

This produces the file test.tex, which can be processed with either Plain TEX or LATEX.

A worry that arises when one is deciding whether to use the WEB system is whether the overhead of using it will be annoying. Experience shows that the answer is generally "No". The TANGLE processor is quite fast; the pass through TANGLE typically takes a small fraction of the time to compile and link, especially for large codes. WEAVE is slower (it does a *lot* of work!), and furthermore one must run the output of WEAVE through TEX. However, when one is developing a code he typically tangles it many times for each time he weaves it, and furthermore the clarity and completeness of the resulting documentation are very much worth any additional overhead. The shift from heavily time-shared mainframes to very fast, individual workstations will further reduce any annoyance with overhead.

Besides providing a documentation tool, FWEB enhances the source languages by providing a relatively sophisticated, C-like macro capability together with the ability to permute pieces of the program text, so that a large system can be understood entirely in terms of small modules and their local interrelationships. Furthermore, it can even understand syntactical constructions absent from the source language and translate those into valid, compilable code; for example, it can translate the RATFOR dialect of FORTRAN directly into standard FORTRAN. The TANGLE program is so named because it takes a given web and moves the modules from their web structure into the order required by the compilers; the advantage of programming in WEB is that the algorithms can be expressed in "untangled" form, with each module explained separately. The WEAVE program is so named because it takes a given web and intertwines the TEX and code portions contained in each module, then it knits the whole fabric into a structured document. (Get it? Wow.) Perhaps there is some deep connection here with the fact that the German word for "weave" is "*web*", and the corresponding Latin imperative is "*texe*"!

"FWEB enhances the source languages by providing a relatively sophisticated, C-like macro capability together with the ability to permute pieces of the program text..."

2.5 History and design influences (Knuth)

> It is impossible to list all of the related work that has influenced the design of WEB, but the key contributions should be mentioned here. (1) Myrtle Kellington, as executive editor for ACM publications, developed excellent typographic standards for the typesetting of Algol programs during the 1960s, based on the original designs of Peter Naur; the subtlety and quality of this influential work can be appreciated only by people who have seen what happens when other printers try to typeset Algol without the advice of ACM's copy editors. (2) Bill McKeeman introduced

a program intended to automate some of this task [Algorithm 268, "Algol 60 reference language editor," *CACM* **8** (1965), 667–668]; and a considerable flowering of such programs has occurred in recent years [see especially Derek Oppen, "Prettyprinting," *ACM TOPLAS* **2** (1980), 465–483; G. A. Rose and J. Welsh, "Formatted programming languages," *SOFTWARE Practice & Exper.* **11** (1981), 651–669]. (3) The top-down style of exposition encouraged by WEB was of course chiefly influenced by Edsger Dijkstra's essays on structured programming in the late 1960s. The less well known work of Pierre-Arnoul de Marneffe ["Holon programming: A survey," Univ. de Liege, Service Informatique, Liege, Belgium, 1973; 135 pp.] also had a significant influence on the author as WEB was being formulated. (4) Edwin Towster has proposed a similar style of documentation in which the programmer is supposed to specify the relevant data structure environment in the name of each submodule ["A convention for explicit declaration of environments and top-down refinement of data," *IEEE Trans. on Software Eng.* **SE–5** (1979), 374–386]; this requirement seems to make the documentation a bit too verbose, although experience with WEB has shown that any unusual control structure or data structure should definitely be incorporated into the module names on psychological grounds. (5) Discussions with Luis Trabb Pardo in the spring of 1979 were extremely helpful for setting up a prototype version of WEB that was called DOC. (6) Ignacio Zabala's extensive experience with DOC, in which he created a full implementation of TEX in Pascal that was successfully transported to many different computers, was of immense value while WEB was taking its present form. (7) David R. Fuchs made several crucial suggestions about how to make WEB more portable; he and Arthur L. Samuel coordinated the initial installations of WEB on dozens of computer systems, making changes to the code so that it would be acceptable to a wide variety of Pascal compilers. (8) The name WEB itself was chosen in honor of [Knuth's] wife's mother, Wilda Ernestine Bates.

The appendices to this report contain various examples of WEB programming. Complete WEB programs for the FWEAVE and FTANGLE processors are available in the source files provided with the release of FWEB. A study of these examples, together with an attempt to write WEB programs by yourself, is the best way to understand why WEB has come to be like it is.

3. SIMPLE EXAMPLES

Before we plunge into the depths of the WEB system, it may be useful to introduce several simple examples, even though few of the concepts have been yet introduced.

3.1 A simple C program organized with FWEB

The first example gives the outline of a simple C program that has been organized with FWEB. (If you are not familiar with C, you should not worry. Examples from both FORTRAN and RATFOR will be given shortly, and almost all of the important WEB features are language-independent.) First, we present a verbatim listing of the source code. Then, we show how FWEAVE typesets the documentation.

```
@z --- demo0.web ---

This file is part of FWEB.  It and its tangled output demo0.tex are
included into the user manual fwebman.tex.  All web source files should
have header information such as this.

Author:  J. A. Krommes
Version: 1.23
Date:    April 1, 1992

@x---------------------------------------------------------------------------
```

```
@c @% This command (invisible on output) sets the global language to C.
@* EXAMPLE. The ‘\.{@@*}’~begins a major module or section (one for which an
entry is made in the table of contents). We are now in the \TeX\ part of
the section, in which we can type arbitrary \TeX\ to explain what goes on in
the remainder of the section.

The next statement introduces the definition part of this section.
\modlabel{FirstMod} % Attach an identifying name to this section.

@m PRINT(word) printf("%s, world.\n",#word) /* An example of a \.{WEB}
      macro def’n.  They are compatible with ANSI~C, but have extensions
      (not illustrated here) that will be explained later.
      Long comments that extend over more than one line can be written
      like this, in the C commenting style. */

@a @% The @a command (invisible on output) introduces the code part of
   @% this (unnamed) module.
main()
{
PRINT(Hello); /* Bullets as subscripts indicate that the name is defined in
      the current section. */
init(); /* Example of a function call.  The subscript is inserted
      automatically, and indicates in which section the function is
      defined. */
@<Do the computations@>; /* Use of named modules makes the code readable.
      Again, the section number where this module is defined is inserted
      automatically. */
PRINT(Goodbye);
}

@ The ‘\.{@@\ }’~begins a minor section.  (No entry is made in the table of
contents.)  Here the named fragment |@<Do...@>| used in the previous module
is actually defined.  (The definition part of this section is empty.) The
name of this module is ‘Do the computations’; it can be abbreviated (using
an ellipsis) for simplicity in the source because it appeared earlier in
full; however, when it’s printed the full name will be used for
readability.

@<Do the comp...@>=
{
/* Put arbitrary C code here. */
}

@ In general, function names don’t carry as much information as do named
modules, since a module name can be arbitrarily long and complicated.  But
function calls have their place as well, as described later.

This function is actually accreted to the unnamed module begun in
\WEBsection{FirstMod}.  Examine the source listing to see how the section
number in the last sentence was generated automatically through the use of
\FWEB’s \TeX\ macros \.{\\modlabel} and \.{\\WEBsection}.
```

```
@a
void init(void)
{}

@* INDEX.  It's customary to make the index the last major module.
```

It is hoped that this example is mostly self-explanatory. Its various facets will be explained in great detail below. Here, we just observe that the source file is broken up into *modules* or *sections*, which in turn are divided into parts, with the aid of simple commands or *control codes* beginning with '@'. In each section there is space for TEX'd documentation, macro definitions (and other stuff to be explained later), and the code itself. The code can be broken up into named fragments, and these can be defined elsewhere. Therefore, each section can be short and its purpose and logical structure can be easily captured by the eye. The other nuances of this example will be explained later.

FWEAVE typesets this example as follows. (For brevity, the last few pages of the output from FWEAVE are omitted here; those include the index and the table of contents. Those items are some of the most important features of the WEB system. To see how they appear for this example, you can run FWEAVE yourself on the source code for this demo, which is called demo0.web.)

---

1. EXAMPLE. The '@*' begins a major module or section (one for which an entry is made in the table of contents). We are now in the TEX part of the section, in which we can type arbitrary TEX to explain what goes on in the remainder of the section.

The next statement introduces the definition part of this section.

@m *PRINT*•(*word*) ***printf***("%s, world.\n", #*word*)
/* An example of a WEB macro def'n. They are compatible with ANSI C, but have extensions (not illustrated here) that will be explained later. Long comments that extend over more than one line can be written like this, in the C commenting style. */

main•( )
{
PRINT•(*Hello*);
/* Bullets as subscripts indicate that the name is defined in the current section. */
init$_3$( ); /* Example of a function call. The subscript is inserted automatically, and indicates in which section the function is defined. */
⟨ Do the computations 2 ⟩; /* Use of named modules makes the code readable. Again, the section number where this module is defined is inserted automatically. */
PRINT•(*Goodbye*);
}

2. The '@␣' begins a minor section. (No entry is made in the table of contents.) Here the named fragment ⟨ Do the computations 2 ⟩ used in the previous module is actually defined. (The definition part of this section is empty.) The name of this module is 'Do the computations'; it can be abbreviated (using an ellipsis) for simplicity in the source because it appeared earlier in full; however, when it's printed the full name will be used for readability.

```
⟨ Do the computations 2 ⟩ ≡
    {   /* Put arbitrary C code here. */
    }
```

This code is used in section 1.

3. In general, function names don't carry as much information as do named modules, since a module name can be arbitrarily long and complicated. But function calls have their place as well, as described later.

This function is actually accreted to the unnamed module begun in section 1. Examine the source listing to see how the section number in the last sentence was generated automatically through the use of FWEB's TEX macros \modlabel and \WEBsection.

```
void init•(void)
  {}
```

4. INDEX. It's customary to make the index the last major module.

(Index and remaining material skipped.)

(Page break skipped.)

Note how module numbers are used both in module names and as subscripts to identifiers to help one find his way around the documentation. (When an identifier is used in the same section in which it is defined, it is subscripted with a bullet.) These features will be discussed at length later.

3.2 Converting a FORTRAN program to WEB

We continue to "learn by doing" by considering how to convert a very simple FORTRAN code to WEB. Consider the following elementary example:

```
C --- f0to_web.src---

C A simple Fortran example (that does nothing at all).
      program main

C Do the computation of alpha.
      call compute
      end

C --- COMPUTATIONAL ROUTINES ---

      subroutine compute
      call compute1
      end

      subroutine compute1
      end
```

The code consists of one main program and several related subroutines. To convert such a code to WEB, the standard procedure is to make each program unit a separate WEB section. This is done by prefacing each program unit by (1) an '@*' or '@␣' command, signifying major or minor sections, respectively; (2) explanatory TEX text about the purpose and general logic of the program unit, and (3) an '@a' command to signal the start of the actual code. Since the explanatory text can be absent, the quickest way of converting a FORTRAN code is to preface each subroutine with '@␣@a'. (Actually, one could just preface the entire code with '@␣@a', thereby making the entire code into just one huge section. It would still weave and tangle correctly. However, in this case one loses one of the most important feature of WEB—namely, a useful index. With everything in one big section, every index entry would refer to section 1!) Thus, the WEB code one constructs from the present FORTRAN source will look something like this:

```
@z --- f0to_web.web ---

A simple Fortran example (that does nothing at all).

@x
@n
@* The MAIN PROGRAM.  {\it (Explain the purpose of the code here, using the
full power of \TeX\ to help you.)}
@a
      program main

C Do the computation of~$\alpha$.
      call compute
      end

@* COMPUTATIONAL ROUTINES.  {\it (Explain the general philosophy of the
computational algorithm here.)}
@a
      subroutine compute
      call compute1
      end
```

```
@ {\it (Other computational routines should be minor sections, all grouped
under the major section called ``\.{COMPUTATIONAL ROUTINES}''.)}
@a
      subroutine compute1
      end

@* INDEX.  {\it (An index is produced automatically.  It's customary to make
the index a separate major section.)}
```

Notice that in the conversion we removed the comments that preceded the beginning of the program units. Such comments are best incorporated into the names of major sections or expanded into a more leisurely prose that becomes the TEX text. However, we left a comment internal to the main program (almost) intact. However, we took the opportunity to change the word "`alpha`" into the more meaningful TEX form "`$\alpha$`". This is not necessary when one is making a first cut at converting a working code, but in the long run it greatly improves the readability. Thus, the present example weaves to

---

1. The MAIN PROGRAM. *(Explain the purpose of the code here, using the full power of TEX to help you.)*

program *main*•

// Do the computation of α.
call *compute*$_2$
end

2. COMPUTATIONAL ROUTINES. *(Explain the general philosophy of the computational algorithm here.)*

subroutine *compute*•
call *compute1*$_3$
end

3. *(Other computational routines should be minor sections, all grouped under the major section called "*`COMPUTATIONAL ROUTINES`*".)*

subroutine *compute1*•
end

4. INDEX. *(An index is produced automatically. It's customary to make the index a separate major section.)*

(Index and remaining material skipped.)

(Page break skipped.)

This leads us to a very important point: WEAVE does not just transcribe the contents of comments literally, it treats them as TEX to be typeset. Thus, if your comments weren't written with TEX in mind but contain characters that are special to TEX, such as '$' or '_', TEX may complain or typeset the comments in weird ways. Users converting large codes may find this quite annoying. However, note the annoyance should not arise when writing a code in WEB from the start since one should be thinking about and writing in TEX from the very beginning.

A related point is that authors of pre-WEB codes have sometimes gone to considerable lengths to align fields within comments in readable, meaningful ways. For example, one might see something like this in a FORTRAN code:

```
C The method is as follows:
C    (1) Specify the initial condition.
C    (2) Integrate forward one step.
```

Unfortunately, by default WEAVE will undo most of this painstaking work. Unless one does something special, the previous comments will come out looking something like this:

/* The method is as follows: (1) Specify the initial condition. (2) Integrate forward one step. */

The pretty alignment has been totally lost! This is the price one pays for having the full power of TEX available. In TEX, one has the powerful \halign macro and other mechanisms to create alignments. Had one been writing in WEB from the beginning, he would have used that naturally. The best advice, then, is: Write all your codes in WEB from the very beginning. It's easier to do that than to convert them later—and the results are generally spectacular.

Actually, there is a mechanism that will transcribe arbitrary material verbatim to the output. This is the *meta-comment*: Any material enclosed between the commands '@(' and '@)' (each of which should begin in column 1 and be on a line by itself) will be appropriately enclosed in a verbatim environment (slightly different depending on whether TEX or LATEX is used). Thus, spaces and alignment are preserved in the following example:

```
@(
0 2 4
 1 3 5
@)
```

The meta-comment can be used for purposes other than comments; see the more detailed discussion of the '@(' command below. In general, it is not recommended that one use meta-comments in lieu of standard WEB comments whose contents are valid TEX.

In Appendices A–E we describe more complicated instances of WEB programming. Many further examples can be found throughout the text. As we proceed, we will learn more methods and guidelines for writing or converting to high-quality WEB code. A more complete review of the typical conversion procedure will be given later in the section "Usage Tips and Suggestions" below

4. GENERAL RULES

In this section we describe the syntax rules that must be followed in preparing a WEB source file. Although the syntax is ultimately straightforward, the issue itself is somewhat complex because of the need to simultaneously process source codes written in several different languages. We shall therefore proceed in stages, first introducing the logic, later discussing nuances and differences between languages.

4.1 Text

The original documentation for WEB stated the following:

> "A WEB file is a long string of text that has been divided into individual lines. The exact line boundaries are not terribly crucial, and a programmer can pretty much chop up the WEB file in whatever way seems to look best as the file is being edited; but string constants and control texts must end on the same line on which they begin, since this convention helps to keep errors from propagating. The end of a line means the same thing as a blank space."

Unfortunately, the situation becomes intrinsically more complicated when one desires to support a column-oriented language such as FORTRAN–77. However, this detail can be dealt with later. For C and RATFOR free-form syntax is indeed appropriate, and once you've used it you'll probably hate going back to FORTRAN's clumsy conventions. Note that FORTRAN–90 allows a free-form mode, which should be used for new codes. (However, as we will explain below, RATFOR–90 is still easier to use than free-form FORTRAN–90.)

Two kinds of material go into WEB files: TEX text and code text. A programmer writing in WEB should be thinking both of the documentation and of the program that he or she is creating; i.e., the programmer should be instinctively aware of the different actions that WEAVE and TANGLE will perform on the WEB file. TEX text is essentially copied without change by WEAVE, and it is entirely deleted by TANGLE, since the TEX text is "pure documentation." Code text, on the other hand, is formatted by WEAVE and it is shuffled around by TANGLE, according to rules that will become clear later. For now the important point to keep in mind is that there are two kinds of text. Writing WEB programs is something like writing TEX documents, but with an additional "code mode" that is added to TEX's horizontal mode, vertical mode, and math mode.

"Writing WEB programs is something like writing TEX documents, but with an additional 'code mode' that is added to TEX's horizontal mode, vertical mode, and math mode."

4.2 Modules

A WEB file is built up from units called *modules* or *sections* that are more or less self-contained. Each section has three parts:

1) A **TEX part**, containing explanatory material about what is going on in the module.

2) A **definition part**, containing (1) macro definitions that serve as abbreviations for code constructions that would be less comprehensible if written out in full each time, (2) preprocessor commands that allow one to selectively process the macro definitions, (3) format statements that tell WEAVE how to deal with identifiers such as macro names that are not in its vocabulary, and (4) miscellaneous commands related to operator overloading, limbo text, etc.; these will be explained later.

3) A **code part**, containing a piece of the program that TANGLE will produce. *This code should ideally be about a dozen lines long, so that it is easily comprehensible as a unit and so that its structure is readily perceived.* (Italics added.) Preprocessor commands may also appear in the code part, allowing one to selectively include or delete fragments of code.

The three parts of each module must appear in this order; i.e., the TEX commentary must come first, then the definitions, and finally the source code. Any of the parts may be empty.

4.3 Beginning a module

A module begins with the pair of symbols '`@␣`' or '`@*`', where '␣' denotes a blank space. A module ends at the beginning of the next module (i.e., at the next '`@␣`' or '`@*`'), or at the end of the file, whichever comes first. The WEB file may also contain material that is not part of any module at all, namely the text (if any) that occurs before the first module. Such text is said to be "in limbo"; it is ignored by TANGLE except for any embedded language-switching commands and copied essentially verbatim by WEAVE, so its function is primarily to provide any additional formatting instructions that may be desired in the TEX output. Indeed, it is customary to begin a WEB file with TEX code in limbo that loads special fonts, defines special macros, changes the page sizes, and/or produces a title page. You should also place a global language command such as '`@n`' somewhere in limbo; see below.

A module begins with either '@ ' or '@*'.

If a source file begins with '`@z`' as the *very first* two characters, all text between the '`@z`' and the end of a subsequent line begun with '`@x`' is completely ignored by both processors. This material is intended to include commentary such as author, date, version, etc. See the demo programs scattered throughout this manual for simple examples of such commentary.

Modules are numbered consecutively, starting with 1; these numbers appear at the beginning of each module of the TEX documentation, and they appear as bracketed comments at the beginning of the code generated by that module in the source program.

Fortunately, you never mention these numbers yourself when you are writing in WEB. You just say '`@␣`' or '`@*`' at the beginning of each new module, and the numbers are supplied automatically by WEAVE and TANGLE. As far as you are concerned, a module has a name instead of a number; such a name is specified by writing '`@<`' followed by TEX text followed by '`@>`'—for example, '`@<Read $\alpha$ and~$\beta$@>`'. When WEAVE outputs a module name, it replaces the '`@<`' and '`@>`' by angle brackets and inserts the module number in small type—for example, '⟨ Read α and β 5 ⟩'. Thus, when you read the output of WEAVE it is easy to locate any module that is referred to in another module.

For expository purposes, a module name should be a good description of the contents of that module, i.e., it should stand for the abstraction represented by the module; then the module can be "plugged into" one or more other modules so that the unimportant details of its inner workings are suppressed. A module name therefore ought to be long enough to convey the necessary meaning. Unfortunately, however, it is laborious to type such long names over and over again, and it is also difficult to specify a long name twice in exactly the same way so that WEAVE and TANGLE will be able to match the names to the modules. Therefore a module name can be abbreviated after its first appearance in the WEB file, by typing '`@<`α`...@>`', where α is any string that is a prefix of exactly one module name that appears in the file. For example, '`@<Clear the arrays@>`' can be abbreviated to '`@<Clear...@>`' if no other module name begins with the five letters '`Clear`'. Module names must otherwise match character for character, except that consecutive blank spaces and/or tab marks are treated as equivalent to single spaces, and such spaces are deleted at the beginning and end of the name. Thus, '`@< Clear  the arrays @>`' will also match the name in the previous example.

We have said that a module begins with '`@␣`' or '`@*`', but we didn't say how it gets divided up into a TEX part, a definition part, and a code part.

4.4 The definition part

The TEX part ends and the definition part begins with the first appearance in the module of one of a set of commands. These are distinguished by the attribute that they do not produce compilable code but rather tell one or the other processor to do or remember something. Those commands will be explained in

detail later; briefly, they are:

`@d` — Define an "outer macro" whose definition will be copied to the very beginning of the tangled output file.

`@f` — Format an identifier to behave like some other identifier.

`@l` — Specify TEX text for `FWEAVE` to output at the beginning of the limbo section.

`@m` — Define an "inner" or "`WEB` macro" to `FTANGLE`.

`@v` — Tell `FWEAVE` how to "overload" an operator.

`@W` — Tell `FWEAVE` how to "overload" an identifier.

`@#` — Begin a `WEB` preprocessing command such as `@#if`.

4.5 The code part

The definition part ends and the code part begins with the first appearance of '`@a`' or '`@<`'. The latter option '`@<`' stands for the beginning of a module name, which is the name of the module itself. An equals sign (`=`) must follow the '`@>`' at the end of this module name; you are saying, in effect, that the module name stands for the code text that follows, so you say '⟨ module name ⟩ = code text'. Alternatively, if the code part begins with '`@a`' instead of a module name, the current module is said to be *unnamed*. Note that (generally) module names cannot appear in the definition part of a module, because the first '`@<`' in a module signals the beginning of its code part. Any number of module names might appear in the code part, however, once it has started.

"The code part begins with the first appearance of '@a' or '@<'."

(Actually, in `FWEB` module names *can* appear in the definition part in certain circumstances. First, a module name may appear immediately after a format command; see the discussion of '`@f`' below. Second, module names may appear in `FWEB` macro definitions if they are begun by '`#<`' instead of '`@<`'; see the discussion of macros below. Finally, module names may appear inside of the vertical bars that signify a shift into code mode.)

4.6 How `TANGLE` makes compilable programs out of modules

The general idea of `TANGLE` is to make a compilable program (or programs, if one is mixing languages) out of these modules in the following way: First all the code parts of unnamed modules are copied down, in order; this constitutes the initial approximation T_0 to the text of the program. *(There should be at least one unnamed module, otherwise there will be no output.)* Then all module names that appear in the initial text T_0 are replaced by the code parts of the corresponding modules, and this substitution process continues until no module names remain. Then all macros defined by '`@m`' are replaced by their equivalents, according to certain rules that are explained later. The resulting code may have pieces in any or all of the several supported languages C, C++, FORTRAN, RATFOR, and TEX. This code is run through an *output driver* appropriate for the language in question; the driver writes files with the appropriate compiler extension—'`.c`', '`.cpp`' ('`.c++`' for UNIX), '`.for`' ('`.f`' for UNIX), '`.rat`' ('`.r`' for UNIX), or '`.x`'—and syntax. For example, the FORTRAN–77 output driver ensures that statements begin in column 7 and are correctly continued if they extend beyond column 72. All comments will have been removed from these programs except for the verbatim comments (either begun explicitly by '`@/*`' or '`@//`' or implicitly selected by the command-line option `-v`) and the meta-comments delimited by '`@(`' and '`@)`', as explained below, and except for the module-number comments that point to the source location where each piece of the program text originated in the `WEB` file.

"There should be at least one unnamed module, otherwise there will be no output."

If the same name has been given to more than one module, the code text for that name is obtained by putting together all of the code parts in the corresponding modules. This feature is useful, for example, in a module named '`Global variables`', since a C programmer can then declare global variables in whatever modules those variables are introduced, but be sure that they will all be grouped together at the beginning of the code. A similar application arises in FORTRAN when one is describing and defining **common** declarations. When several modules have the same name, `WEAVE` assigns the first module number as the number corresponding to that name, and it inserts a note at the bottom of that module telling the reader to 'See also sections so-and-so'; this footnote gives the numbers of all the other modules having the same name as the present one. The code text corresponding to a module is usually formatted by `WEAVE` so that the output has an equivalence sign in place of the equals sign in the `WEB` file; i.e., the output says '⟨ module name ⟩ ≡ code text'. However, in the case of the second and subsequent appearances of a module with the same name, this '≡' sign is replaced by '+≡', as an indication that the code text that follows is being appended to the code text of another module.

"If the same name has been given to more than one module, the code text for that name is obtained by putting together all of the code parts in the corresponding modules."

The previous paragraph is in Knuth's original words. Note that he uses "module" and "section" interchangably. A more precise distinction might have been to use "section" for the distinct numbered entities that begin with '`@␣`' or '`@*`', and "module" for the concatenations of all sections with the same names. Then one could say things like "The unnamed module consists of sections 1, 2, and 5." In any event, the user will soon get used to how things work. In this article, we have generally chosen to retain Knuth's original usage because it makes it easier to quote just what Knuth said.

As `TANGLE` starts and leaves modules, it writes down the line number of the original `WEB` file. When the language is C, this is done in the form of the C preprocessor **#line** command. This means that when the compiler gives you error messages, or when you debug your program, the messages refer to line numbers in the `WEB` file, rather to ones in the C file. In most cases, you can even forget about the C file altogether. For other languages, the '`#`' character is changed to a comment character. Unfortunately, in FORTRAN and RATFOR there is no compiler feature analogous to **#line** that resets the line number, so for those languages compiler error messages will refer to the output file, not the `WEB` file.

4.7 How `WEAVE` makes a TEX file containing documention

The general idea of `WEAVE` is to make a `TEX` file from the `WEB` file in the following way: The first line of the `TEX` file will generally be output as '`\input fwebmac.sty`'; this will cause TEX to read in the macros that define `FWEB`'s documentation conventions. (If you don't want this line to be first for some tricky reason, turn it off with the command-line option '`-w`'. However, you must then say yourself '`\input fwebmac.sty`' somewhere in limbo.) Next may be inserted special TEX material generated automatically by the '`@l`' or '`@v`' commands; see below. The next lines of the file will be copied from whatever TEX text is in limbo before the first module. Then comes the output for each module in turn, possibly interspersed with end-of-page marks. Finally, `WEAVE` will generate a cross-reference index that lists each module number in which each code identifier appears, and it will also generate an alphabetized list of the module names, as well as a table of contents that shows the page and module numbers for each "starred" module.

4.8 Starred (major) modules

What is a "starred" module, you ask? A module that begins with '`@*`' instead of '`@␣`' is slightly special in that it denotes a new major group of modules. The '`@*`' should be followed by the title of this group, followed by a period. Such modules will always start on a new page in the TEX output, and the group title

will appear as a running headline on all subsequent pages until the next starred module. The title will also appear in the table of contents, and in boldface type at the beginning of its module. Caution: Do not use TEX control sequences in such titles, unless you know that the `fwebmac` macros will do the right thing with them. The reason is that these titles are converted to uppercase when they appear as running heads, and they are converted to boldface when they appear at the beginning of their modules, and they are also written out to a table-of-contents file used for temporary storage while TEX is working; whatever control sequences you use must be meaningful in all three of these modes.

Starred sections have associated level numbers, where 0 denotes the most significant level, 1 denotes a subsection, 2 denotes a subsubsection, and so on. You can indicate the level number in several ways. First, if the command '`@*`' is not immediately followed by a digit, then the level is 0. (This was the only possibility in the original `WEB` design.) Next, if '`@*`' is followed by a positive digit, then that digit indicates the level. (Be default, that digit must be ≤ 4.) Thus, you can say

```
@* MAJOR. (The level is 0.)
@*3 Subsubsubsection.  (The level is 3.)
```

Note that entries are made in the table of contents for all starred sections, including those whose level is greater than 0.

Level numbers are processed by `fwebmac` macros, not by code hard-wired into `FWEAVE`. The level number is supplied as an argument to various `fwebmac` macros, and one can redefine those macros to achieve various special effects. For example, entries in the table of contents are formatted by the macro `\WZ` (which is defined inside the table-of-contents macro `\Wcon`), whose first argument is the level number n. By default, the entry is just indented by n ems. However, it is straightforward to achieve more sophisticated effects (such as various fonts or case) by enhancing the definition of `\WZ` to include an `\ifcase` construction. (See how this is done in `fwebman.tex` for the table of contents for this manual.) Similarly, by default major sections get a page break in the woven output, whereas subsections do not; however, that can also be changed by redefining the `fwebmac` macro `\Wsectionbreak`.

The TEX output produced by `WEAVE` for each module consists of the following: First comes the module number (e.g., '`\WM123.`' at the beginning of module 123, except that '`\WN`' appears in place of '`\WM`' at the beginning of a starred module). Then comes the TEX part of the module, copied almost verbatim except as noted below. Then comes the definition part and the code part, formatted so that there will be a little extra space between them if both are nonempty. The definition and code parts are obtained by inserting a bunch of funny-looking TEX macros into the source program; these macros handle typographic details about fonts and proper math spacing, as well as line breaks and indentation.

(The original `WEB` employed control sequences such as '`\M`' instead of '`\WM`'. This makes the `tex` file somewhat shorter and more readable and makes the input phase of TEX run a tiny bit faster. However, it seems undesirable to deny the user most of the single-character upper-case control sequences; the present `FWEB` opts for user convenience.)

4.9 Code mode

When you are typing TEX text, you will probably want to make frequent reference to variables and other quantities in your code, and you will want those variables to have the same typographic treatment when they appear in your text as when they appear in your program. Therefore the `WEB` language allows you to get the effect of source code editing within TEX text, if you place '`|`' marks before and after the code material. For example, suppose you want to say something like this while documenting FORTRAN code:

The **integer** variable *itype* is used in such statements as **if**$(itype \neq m)\ldots$

The TEX text would look like this in your WEB file:

```
The |integer| variable |itype| is used in such statements as |if(itype␣!=␣m)|...
```

Note that the WEB file contained the construction '!=' instead of FORTRAN–77's standard (and archaic) '.ne.'. In fact, you could have typed either, but WEB gives you the flexibility to type the more modern symbols if you wish, thereby helping you to write clearer code. The options will be explained in detail later; as much as possible, they follow the conventions for C. Also, observe that either construction is woven into the more meaningful symbol '≠'.

Incidentally, the cross-reference index that WEAVE would make, in the presence of documentation like this, would include the current module number as one of the index entries for *itype*, even though *itype* might not appear in the code part of this module. Thus, the index covers references to identifiers in the explanatory comments as well as in the program itself; you will soon learn to appreciate this feature. However, the identifiers **integer**, **if**, and m would not be indexed, because WEAVE does not make index entries for reserved words or single-letter identifiers. Such identifiers are felt to be so ubiquitous that it would be pointless to mention every place where they occur.

Although a module begins with TEX text and ends with code text, we have noted that the dividing line isn't sharp, since code text can be included in TEX text if it is enclosed in '|...|'. Conversely, TEX text also appears frequently within code text, because everything in comments (i.e., between '/*' and '*/', or between '//' and the next newline) is treated as TEX text. Furthermore, a module name consists of TEX text; thus, a WEB file typically involves constructions like 'if(x == 0) @<Empty the |buffer| array@>' where we go back and forth between code and TEX conventions in a natural way.

In the original WEB design, a module name (@<...@>) was not allowed between the vertical bars, because the first occurrence of the name signified the beginning of the code part. Beginning with version 1.30, this restriction has been removed. Thus, one can include in his TEX documentation statements like "See section |@<A@>| for ...". Since code mode is allowed inside module names, it is now also possible to have module names within module names, as in "@<A = |@<B@>|@>". (Obviously, it's possible to abuse this flexibility.)

4.10 Fortran demo program

To summarize and illustrate some of what has just been said, here is an (incomplete) example of a FORTRAN program that is intended to read some data, process it, and graph the results. This uses several features that have not yet been explained, namely macro definition, preprocessor commands, and format commands, but these are either fairly obvious and/or can be ignored for now. This example tries to demonstrate how one should liberally use named modules to enhance the logical structure of each module and to keep the length of each module quite short, and how to use vertical bars to intersperse code mode with TeX'd documentation.

```
@z --- demo1.web ---

This file is part of FWEB.  It and its woven output demo1.tex is included
into the user manual fwebman.tex.

Author:  J. A. Krommes
Version: 1.23
Date:    April 1, 1992

@x---------------------------------------------------------------------------
```

```
\def\WEB{\.{WEB}} % A TeX macro definition in limbo.

@n/ @% The slash tells Fortran to recognize '//' as the start of a short
    @% comment rather than the conventional concatenation symbol.
@* DEMO. This example demonstrates the use of named modules, and some other
features of~\WEB.

@#ifndef N
      @m N 100 /* If you define this macro from the command line by
                  saying ``\.{-mN=200}'', that definition will
                  override this one. */
@#endif

@a
      program main
      @<Common declarations@>
      @<Read data@>
      @<Process data@>
      @<Graph results@>
      end

@ Accreting and inserting |common| declarations is an interesting use of
named modules.

@f @<Com...@> common

@<Com...@>=
      real a(N)
      common a

@ Possibly after many more modules that aren't shown here, we get to code
related to graphics. Here we accrete more information into the common
declarations. The discretized abscissa data is held in the array~|xx|; the
ordinate is in~|yy|.
@<Com...@>=
      real xx(N), yy(n) // Example: |call curve(xx,yy,N,1)|
      common/graphs/ xx,yy

@* INDEX.
```

FWEAVE typesets this example as follows.

---

1. DEMO. This example demonstrates the use of named modules, and some other features of WEB.

@#ifndef *N*•
  @m *N*• 100 /∗ If you define this macro from the command line by saying "-mN=200", that definition will override this one. ∗/
@#endif

program *main*•

```
    ⟨ Common declarations 2 ⟩

    ⟨ Read data 0 ⟩
    ⟨ Process data 0 ⟩
    ⟨ Graph results 0 ⟩
  end
```

2. Accreting and inserting **common** declarations is an interesting use of named modules.

```
  @f ⟨ Common declarations 2 ⟩ common
```

```
⟨ Common declarations 2 ⟩ ≡
    real a(N_1)
    common a
```

See also section 3.

This code is used in section 1.

3. Possibly after many more modules that aren't shown here, we get to code related to graphics. Here we accrete more information into the common declarations. The discretized abscissa data is held in the array xx; the ordinate is in yy.

```
⟨ Common declarations 2 ⟩ +≡
    real xx(N_1), yy(n)    // Example: call curve(xx, yy, N_1, 1)
    common /graphs/ xx, yy
```

4. INDEX.

(Index and remaining material skipped.)

(Page break skipped.)

4.11 Modules versus functions

Clearly the concept of WEB modules leads to great flexibility and readability in the construction of source codes. However, modules can also be misused or, in particular, *overused*, and these difficulties are not uncommon for beginning programmers in WEB.

4.11.1 When to use named modules

The problem is that although in many ways modules are functionally quite similar to functions or subroutines, they are not identical. Of course, modules do not take arguments, but that is not so much the point. Of more concern is that it is in principal possible to create a very large code with just one main program (consider the above example), even though no section is more than about a dozen lines long, just by nesting named modules to arbitrary levels. That this is undesirable is usually emphasized when one attempts to debug such a code; he may find that it is not easy to set a breakpoint at a module, whereas had each module been a function call one could have set many breakpoints and obtained a detailed picture of the control flow. Also, many compilers have limitations on the maximum length of a function. A partial solution to this is discussed in a later subsection on debugging, but it is nevertheless true that one should often use function calls instead of modules. Named modules should be used when the overhead of a function call is unacceptable, such as, for example, in a computationally bound inner loop. They also find an important use in making constructions such as large **switch** statements readable; each case of the **switch** can be a separate module. However, very large blocks of code, especially in outer sections of the program, should probably be accessed with function calls, if possible. Note, though, that certain blocks of code, such as **common** declarations in FORTRAN, cannot be replaced by function calls. For such cases, the use of named modules is almost always desirable.

"Named modules should be used when the overhead of a function call is unacceptable, such as, for example, in a computationally bound inner loop."

As an example of this discussion, it would actually be better to code the main program of the preceding example as follows:

```
program main
@<Common declarations@>
call input
call process
call graph
end
```

Since the main program is not inside a critical inner loop, the slight extra overhead of the subroutine calls will be imperceptible, and one can easily set a debugging breakpoint at each of the fundamental subroutines *input*, *process*, and *graph*. A named module is properly used here to insert the common declarations, which cannot be replaced by a subroutine call.

♣ **4.11.2** Self-documentation and cross-referencing for named modules and identifiers

One of the great virtues of employing named modules is that the names are *self-documenting*. For example, it is much more illuminating to see "⟨ Check that θ lies in the range $-\pi \le \theta < \pi$; issue a warning message otherwise 98 ⟩" than to have to remember what the statement "**call** *check*" does. Furthermore, in the original WEB design it was not immediately obvious where *check* was defined. One had to turn first to the index, look up *check*, then turn back to the desired section. The extra step of accessing the index is unnecessary with module names, which have the relevant section number built in.

However, beginning with FWEB version 1.20, certain identifiers such as function names and macro definitions also carry the section name in which they were defined, displayed as a subscript—e.g., *check*$_{98}$. Although this does not solve the problem of self-documentation, it does expedite finding one's way around the code. This mechanism is discussed further in the section on "Forward referencing" below.

In fact, it's possible to the virtues of both self-documentation and function calls if one is willing to do a bit of extra typing. Consider the following:

```
@<Check that $\theta$ lies in the range $-\pi \le \theta < \pi$;
issue a warning message otherwise@>=
    call check
```

With this trick, the documentation shows clearly what is really happening, but one can still set a breakpoint at *check* for convenient debugging.

4.11.3 WEB programming and UNIX

In summary, both function calls and named modules can be used to good advantage in WEB programming. If they are used properly, the resulting product can be amazingly easy to understand and maintain.

Nevertheless, the possibility of indiscriminate use of modules has led some programmers, particularly those used to UNIX, to suggest that WEB programming is a step backwards and defeats the utility of other tools such as the make utility. Those programmers will delegate each function to a separate file, then use the make utility to keep them updated as necessary. However, although this keeps the amount of compiling to a minimum, this extreme limit can also be criticized on the grounds that it is very difficult to maintain a sensible, coherent documentation, which was the primary goal of the WEB system. More reasonably, the ideal solution lies somewhere between the two extremes. Generally it is reasonable and efficient to combine at least several functions into one source file. Furthermore, all but the very shortest functions can benefit from being broken into named modules, and common WEB macros can be easily included with each separate file. WEB and UNIX are not in conflict; they both provide powerful tools that can be simultaneously applied to complicated programming problems. Both systems provide the programmer with a great deal of power. It is up to him to use that power wisely and with discipline. As Knuth admits, the users of WEB must be sophisticated, but they reap significant rewards in return.

"WEB and UNIX are not in conflict."

5. The PHASES of WEB

It is not necessary to know in great detail just how WEB accomplishes its various tasks in order to use it effectively. However, to fully understand some of the material to follow, especially topics related to macro processing, it is necessary to appreciate that WEB processes its input in several distinct phases: two for TANGLE; three for WEAVE. These phases are explained briefly here.

First, we must remark that the WEB processors are best viewed as consisting of three parts: an *input driver*; the *processor proper*; and an *output driver*. Each source language has (at least in principle) a distinct input driver. The role of the input drivers is to preprocess the incoming text into a uniform syntax that can be understood by the processor proper. For example, idiosyncratic commenting styles such as FORTRAN–77's column 1 convention are converted by the input driver to the standard C style that the innards of the processors understand. For WEAVE, the output driver is common to all languages; it creates the tex file. For TANGLE, the output driver is more or less the inverse of the input driver; it creates c, for, rat, or tx files. In the following discussion, "input" is best thought of as the input to the processor proper, or equivalently as the preprocessed output of the input driver.

5.1 Phase 1

For both processors, phase one involves *tokenization* of the input. If the input is TEX material, it is skipped by `TANGLE` or absorbed essentially unchanged by `WEAVE`. Code material is broken up into identifiers, numerical constants, character strings, etc., and these are represented by special codes. For ease in working with `WEB` macros, the concept of *identifier* is generalized somewhat from the syntax of any of the supported source languages: identifiers are character sequences of arbitrary length that contain either an alphabetic character (`A-Z` or `a-z`), a digit (`0-9`), an underscore ('`_`'), a dollar sign ('`$`'), or, when the language is neither C, C++, nor FORTRAN–90, a per cent sign ('`%`'); they may not begin with a digit. Case is always significant. Thus, examples of unique identifiers are

```
a, A, UPPER_and_lower_case, i5, SYS$TEST, $1, _reserved, %loc
```

Avoid defining macros or C identifiers that begin with an underscore; such identifiers may be used internally by the system. Dollar signs or (generally) per cent signs are permitted since some compilers use them to identify extensions to the standard languages. It is common to enter the source code entirely in lower case, thereby reserving upper case for possible use in macro definitions. In particular, the reserved words of C or FORTRAN, such as **for** or **dimension**, are understood only in lower case.

When an identifier or module name is recognized, both `TANGLE` and `WEAVE` store it in a table, along with its special code. Otherwise, however, the two processors do different things in phase one. `TANGLE` stores all of the tokenized code, filing it into the appropriate (unnamed or named) module. It also memorizes macro definitions. `WEAVE`, on the other hand, does *not* store the code during phase one. Rather, it merely processes any format (`@f`) definitions or operator overload (`@v`) commands (both explained later) and constructs a cross-reference table for the identifiers and module names. This preliminary pass is required in order that forward references to named modules and certain identifiers can be resolved.

"Avoid defining macros or C identifiers that begin with an underscore."

5.2 Phase 2

During phase two, `TANGLE` outputs the code it has stored. As explained earlier, it begins with the contents of the unnamed module. If at any point it encounters a reference to a named module, it takes a detour to dump out the contents of that module. Since modules may contain references to other modules, and such references can be nested to arbitrary depth, the output routine is recursive. As text is output, each identifier is examined to see whether it has been defined as a `WEB` macro. If so, the macro is expanded; that expansion procedure is also recursive. In the RATFOR mode, certain identifiers such as **for** or **switch** have special significance. These are not macros, exactly; rather, they are special keywords signifying that special actions should be taken on the text that follows. When one of these keywords is recognized, a special *statement translation function* is invoked. The input to that function is the raw output of `TANGLE`. The function may request further output, possibly storing up text and outputting it in a different order, possibly inserting additional statements such as **goto** into the output. The ultimate output from the statement translator will be the FORTRAN equivalent of the original RATFOR construction. Macros and module names will have been expanded properly during that output.

Thus, at the end of phase two of `TANGLE` all of the code will have been output in the appropriate order, in a form acceptable to a language compiler. (See the example in Appendix D.)

Phase two of `WEAVE` is more complicated. For each module, `WEAVE` reads the source file again. It copies the TEX material to the output essentially unchanged. It tokenizes the code and stores it along with the cross-reference information that it collected during phase one. At the end of the module, it then analyzes the code hunting for constructions it understands, such as expressions, loops, entire functions, etc. These are

processed and output into a series of TEX macros that will typeset the code in a useful, visually appealing way. (See the examples in Appendices B and E.)

5.3 Phase 3

TANGLE has no phase three.

Phase three of WEAVE sorts the cross-reference information, and writes out the index, alphabetized list of module names, and the table of contents. In order to expedite advanced editing and other tasks, both the index and list of module names are written to separate files, which by default are named INDEX.tex and MODULES.tex. (These can be changed with the style file; see below.)

♣ 6. LANGUAGES

This version of FWEB supports a variety of source languages: FORTRAN (both FORTRAN–77 and FORTRAN–90), RATFOR (both RATFOR–77 and RATFOR–90; to be explained shortly), C, C++, and TEX. (A future version may support the language MAKE, defined to be the syntax of UNIX make files.) These languages can be intermixed within one WEB run. In the simplest situation the programmer will use just one language. However, it is not uncommon to code the outer, control part of a large program in, say, the C language while writing the inner, computationally bound routines in FORTRAN. In such situations, FWEB's language facilities are quite useful, since it enables one to maintain the documentation for the entire code in one unified source file.

6.1 Selecting a language

It is very simple to tell WEB which language is in effect at any point.

6.1.1 Language abbreviations

Each language has an abbreviation that can be used to identify it. These are as follows:

C*	— 'c'	MAKE	— 'k'
C++*	— 'c++'	RATFOR–77*	— 'r'
FORTRAN–77*	— 'n'	RATFOR–90*	— 'r9'
FORTRAN–90*	— 'n9'	TEX	— 'x'

(Strictly speaking, only the first character is the abbreviation; any subsequent text is an optional argument, as discussed in more detail below. Also, note the 'n' for fortraN; an 'f' conflicts with the format command '@f'.) Each of the starred languages can be invoked by its own command-line option or control code (these are explained below in more detail). For example, C can be invoked from the command line by the option '-c', and from within the web source by '@c'. Other languages, such as TEX or (in the future) MAKE, must be invoked by the general language command '-L*l*' or '@L*l*', where *l* is one of the language symbols listed above. (Case is significant; '-l' means something entirely different.) For example, one can invoke the TEX language by '@Lx'. This general command also works for the starred languages, so that '@n' is equivalent to '@Ln'.

6.1.2 Global language

FWEB has the concept of a *global language*. The global language is defined to be the language in force when the first module is encountered; namely, at the very end of the limbo stage. It is used as the starting language of the unnamed module and of the TEX parts of each section. The default global language is FORTRAN–77, a reluctant concession to the physics world. However, *it is not recommended that you use* FORTRAN–*77 for*

new code, as you will lose a good deal of the statement-processing abilities of `FTANGLE`. First, you should consider switching to C or C++, which are choices superior to FORTRAN for many applications. However, if you choose to write in a FORTRAN dialect, then *it is strongly recommended that you use* RATFOR, as described elsewhere in this document. Your code will be easier to write (it will be logically clearer and will involve fewer keystrokes), the result will be substantially more readable, and, therefore, you will ultimately be more productive.

One can override the default global language of FORTRAN–77 in two ways. First, although it is *not recommended* except for special situations, one can set it from the command line by using one of the options '`-c`', '`-r`', '`-n`', or '`-L`*l*'. Second, you can insert one of the language switching commands '`@c`', '`@r`', '`@n`', or '`@L`*l*' anywhere in the limbo section. Note that if you have such an explicit language command in your file, it will override anything that was said on the command line. (You will be warned.) The best practice is to put the global language command at the very beginning or the very end of the limbo section. (If you put it at the beginning, it must at present follow any `@z`...`@x` ignorable commentary.) If you're programming only in FORTRAN–77, you don't need the `@n` command. However, as a matter of style, and for compatibility with future releases, it's best to always insert a language command explicitly.

"It is not recommended that you use Fortran–77 for new code."

6.1.3 Changing languages within modules

All modules, both named and unnamed, have a language. The TEX part of each section will begin in the global language. Now although language doesn't matter for TEX text, it does matter if you shift into code mode by using vertical bars. Unless told otherwise, code between vertical bars will be in whatever language is currently in force (generally, this will be the global language). If, however, you want that code to be interpreted according to some other language, you can put a language command immediately after the opening vertical bar, as in '`|@n real a(0:n)|`' or '`|@r repeat {i=f(i);} until(i > 10);|`'; that language switch will be local to the barred material.

Except for this local use of language commands, it is safest (because of certain restrictions in the FORTRAN mode) and logically and visibly clearest that language commands begin in column 1, on a line by themself.

You can also place a language command anywhere in the TEX part, not just between bars, to reset the language for the rest of that part, but this is neither recommended nor usually necessary.

When the command '`@a`', which signifies the start of the unnamed module, is encountered (either for the first or for subsequent times), the language reverts to the global language. You can change languages in the unnamed module, but that change is local to one section; it is cancelled by the next '`@*`' or '`@␣`'. `FTANGLE` sorts out the languages and deflects the code to the appropriate output files. `FWEAVE` prints the language commands as a marginal note, and it identifies the language of any module name not in the global language with a superscript, as in '`@<C code@>`'$^{\mathrm{C}}$.

"You can change languages in the unnamed module, but that change is local to one section."

The code parts of named modules inherit the language in force *when that name was first encountered*. That language attribute propagates through all levels of nesting, so that if you have a named module in, say, the C language, all named modules referenced for the first time in that module will automatically

"The code parts of named modules inherit the language in force when that name was first encountered."

be interpreted in C; you don't need to preface each of those (possibly many) modules with an explicit '`@c`'.

However, if, for example, the global language is FORTRAN, there must be at least one '`@c`' somewhere in your file in order to mix in any C routines at all. The most painless way to do this is to switch languages in the unnamed module. Put a language command just before a reference to a named module that you want to be interpreted in C. That one command is all you need to have everything connected with that named module also be understood to be in C.

When the language is FORTRAN*–77, begin all '@' commands in column 1.* This restriction is imposed in order to help simplify the job of the FORTRAN input driver, which converts the column-oriented syntax of traditional FORTRAN to the free-form syntax that the innards of the `WEB` processors understand. It's not a bad programming style in any event.

6.2 Demo program with two languages

As an example, here's how you might handle a main program in FORTRAN and some subroutines written in C, all of which are here put into the unnamed module.

```
@z --- demo2.web ---

This file is part of FWEB.  It and its woven output demo1.tex are
included into the user manual fwebman.tex.

Author: J. A. Krommes
Version: 1.23
Date:   April 1, 1992

@x---------------------------------------------------------------------------

\def\FWEB{\.{FWEB}} % A tex macro definition in the limbo section.

@n/ @% Set the global language to Fortran--77, and allow short comments.
@* MIXING LANGUAGES. In \FWEB, languages can be mixed with a minimum of
effort.
@a
      program main /* In Fortran, a \&{program} statement should always
be used.  */
      call CRTN // We'll actually do the work in a C~routine.
      end

@c @% Temporarily change the language of the unnamed module to C.
@<C code@>; // Link into the web a named module in~C.

@ This is the start of the next section. At this point, the language has
reverted to the global language of Fortran. However, after the equals sign on
the next line the language is~C until the next section is encountered.
@<C code@>=
void CRTN(void)
{
/* The code parts of the following named modules will be understood to be
in~C, since that is the current language when those names are first
referenced. */
@<Compute@>;
```

```
@<Graph@>;
}

@* INDEX.
```

FWEAVE typesets this example as follows. Note the marginal notation @Lc, which marks the language change '@c'. Also notice how module names not in the global language are superscripted with a language symbol.

---

1. MIXING LANGUAGES. In FWEB, languages can be mixed with a minimum of effort.

```
program main•    /* In Fortran, a program statement should always be used. */
  call CRTN    // We'll actually do the work in a C routine.
end

@Lc:    ⟨ C code 2 ⟩^C;    // Link into the web a named module in C.
```

2. This is the start of the next section. At this point, the language has reverted to the global language of Fortran. However, after the equals sign on the next line the language is C until the next section is encountered.

⟨ C code 2 ⟩^C ≡

```
void CRTN•(void)
  {    /* The code parts of the following named modules will be understood to be in C, since that is
       the current language when those names are first referenced. */
  ⟨ Compute 0 ⟩;
  ⟨ Graph 0 ⟩;
  }
```

This code is used in section 1.

3. INDEX.

(Index and remaining material skipped.)

(Page break skipped.)

When the definition of module @<C code@> is encountered, FWEAVE will properly format it in C. FTANGLE will also do the right thing, spitting the main program out into a FOR file but the @<C code@> into a C file. Furthermore, the modules @<Compute@> and @<Graph@> will also be properly interpreted in C, the language that they inherit from @<C code@>.

6.3 Language commands in the definition part

Language commands may also be inserted in the definition section, and are sometimes necessary. This situation arises because identifiers also have language attributes; this includes the special cases of reserved words and intrinsic functions, and it implies that formats and outer macros (see below) are interpreted according to one particular language. Although possibly a bit complicated, this allows one to handle the important case where an identifier is a reserved word in one language but not in another. For example, **dimension** is a reserved word in FORTRAN and RATFOR, but not in C. The only time when you have to worry about this explicitly is when you're formatting a new identifier with '`@f`'. (Formatting is explained below.) You must be in the language in which you intend that identifier to be used. This means you may have to switch languages in the definition section, even if the identifier you're formatting is used in a named code section immediately below whose language is already known. This is necessary because modules start off in the global language and the language of the named module isn't known until the module name is encountered, *after* the definition section. Had one been thinking about multiple languages when the `WEB` system was first designed, he might have devised a somewhat more elegant scheme, but it's too late now. In practice, the present scheme does not seem to lead to too many annoyances.

"Identifiers also have language attributes."

Two built-in `WEB` functions, `_LANGUAGE` and `_LANGUAGE_NUM`, are sometimes useful in writing conditional macros. See the discussion about built-in functions below for their definition. The `WEB` built-in function `_LANGUAGE` (built-in functions are explained below) expands to either '`_C`', '`_CPP`', '`_N`', '`_N90`', '`_R`', '`_R90`', or '`_X`', depending on the language currently in effect.

♣ 6.4 Optional arguments to language commands

Some of the language commands may have optional arguments. For example, FORTRAN–90 is treated as a dialect of the fundamental language FORTRAN. It can be invoked by the commands '`-n9`' or '`@n9`'. Here the text "`9`" is the optional argument. An other example of such a command is '`@r9`', which sets the language to RATFOR–90. The latter means that RATFOR syntax is understood along with FORTRAN–90 keywords, *and* that the RATFOR is translated into FORTRAN–90 rather than FORTRAN–77. All the possible arguments relating to languages are detailed in the section below on command-line options, and in Appendix L.

"Language control codes may always be optionally followed by text enclosed by square brackets."

The preceding examples are special cases of a more general mechanism. Language control codes may always be optionally followed by text enclosed by square brackets. One may put inside the square brackets (almost) any option that may be put on the command line, allowing parameters to be reset at each language change. As a useful shorthand, if a language control code is followed immediate by text (non-white space) that is not begun by a left bracket, the text is automatically prefaced by a hyphen and the control code, then enclosed by brackets. That is,

$$\texttt{@n}\mathit{text} \equiv \texttt{@n[-n}\mathit{text}\texttt{]}$$

so we see, for example, that

$$\texttt{@n/} \equiv \texttt{@n[-n/]}$$

(It may be interesting to know that C++ is handled in this same way—i.e., "`@c++` ≡ `@c[-c++]`", although one will generally not need to be concerned with this.) Parameters in force for a given language are saved when leaving that language and restored when returning to it. Thus, one can mix modules with different dialects of FORTRAN, as in

```
@n9
@
```

```
@A
    @<Fortran--90 stuff@>
    .
    .
@n7
    @<Fortran--77 stuff@>

@ The code part of this section will be in the global language
of Fortran--90.
@<Fortran--90...@>=
    .
    .
@ This section will be in Fortran--77.
@<Fortran--77...@>=
    .
    .
```

Although it would sometimes be convenient, one cannot at present use shorthand such as '`@n9&`'; you must say '`@n9[-n&]`'. This restriction is necessary because some arguments expect additional characters to follow. Thus, the form '`@n9&`' is, according to the rules above, equivalent to '`@n[-n9&]`', so the ampersand would be interpreted as an (invalid) subargument to "`-n9`".

Certain options such as macro definitions are forbidden as optional parameters following a language code. These are specified in Appendix L.

The bottom line about languages is that if one uses only one, one needs to do nothing more than place one language command somewhere in the limbo section. If one mixes languages, one needs to place at least one additional language command somewhere in the unnamed module; one has to be slightly more alert, but not very much so. The intent is that languages should work the way one expects them to, without having to do much of anything.

♣ 7. MACROS

Macro processing—definition of shorthand, readable symbols for otherwise obscure, tedious, or repetitive constructions—is one of the most effective ways to enhance one's coding productivity and the ultimate readability of a code. For example, virtually no numerical constants should appear explicitly in a well-written source code; they should be defined symbolically in terms of macros instead. Say '`YES`' instead of '`1`', or '`EPS`' instead of '`1.0e-6`'. If later you decide that ϵ should be 10^{-7} instead of 10^{-6}, you need to change just the single macro definition line rather than to edit many lines of the source code; furthermore, a well-chosen symbolic name carries much more meaning than a raw number. The C language has a built-in preprocessor, and that is one of the major virtues of C. Unfortunately, FORTRAN lacks a macro preprocessor. It does contain the **parameter** statement. However, that statement is highly restrictive: its scope is restricted to each individual subroutine, and arguments are not allowed. Since often more flexible macro processing is useful, many people adopt the strategy of running their FORTRAN code through a preprocessor such as UNIX' `m4`. This extra processing step is annoying at the very least, and also requires one to learn the syntax of the preprocessor. Although these steps cannot be entirely avoided, their difficulty can be minimized. Thus, `FWEB` has its own built-in macro preprocessor. This is patterned after the ANSI C preprocessor, so that users who are either already familiar with C or who are using `FWEB` to program in both C and FORTRAN are presented with a syntax that is essentially language-independent. In practice, this turns out to be an important consideration. The design and operation of the C preprocessor are also arguably more convenient than `m4` for many applications, although this is certainly a matter of taste to some extent.

"Say 'YES' instead of '1', or 'EPS' instead of '1.0e-6'."

♣ 7.1 WEB macros

WEB macros, sometimes called "internal" or "inner" macros, are defined by the command '@m'. Temporarily, let us assume that these definitions are made in the definition section (as they should always be, except for special considerations described below). In general, WEB macros are expanded when the code is being *output* in phase two. The exception to this is when a macro is encountered in a WEB preprocessor statement (beginning with '@#'; see below); then, it is expanded immediately. As defined for ANSI C, there are two macro forms: "object-like," and "function-like."

WEB macros are defined by '@m'.

7.1.1 Object-like macros

Object-like macros have no arguments; they have the form

@m *identifier replacement-text [optional C-style comment]*

Here and elsewhere, *replacement-text* is an arbitrary string of symbols (except that module names are not allowed). The *replacement-text* may be continued on subsequent lines. Unlike the C preprocessor, which requires continued lines to end with a backslash, no backslashes are needed (or allowed) for macros continued in the definition section. The reason for this is that the end of the definition can be determined from the context; it occurs when the next definition ('@m' or '@d'), format command ('@f'), preprocessor command ('@#'), limbo text command ('@l'), operator overloading command ('@v'), identifier overloading command ('@W'), unnamed module command ('@a'), module name ('@<'), or new module command ('@*' or '@␣') is encountered. The definition of *identifier* is memorized during phase one. Then, whenever the *identifier* is encountered as the code text is being output in phase two, it is replaced by the *replacement-text* (which is then rescanned for further macro substitutions).

Simple examples of object-like macros are

```
@m NO 0 // An object-like macro (having no arguments).
@m YES 1 // It is conventional to use 1 for true, 0 for false.
@m PI 3.14159 // Most constants should be given readable symbolic names.
@m ARG_LIST x,kx,ky,kz
@m DCL_ARGS real x;
       integer kx,ky,kz // Notice how easily this macro was continued.
@m BLANK // A null macro, with no replacement text.
```

Object-like macros are closely related, though not identical, to FORTRAN's **parameter** statement. The differences are that a WEB macro is known throughout the entire code and will appear in expanded form in the tangled output, whereas a **parameter** declaration is local to a subroutine and is expanded by the FORTRAN compiler, rather than by TANGLE. This means that your tangled output may be more readable if you use **parameter** statements instead of WEB macros. However, although this may be adequate in simple cases, it will often not be a viable option. For example, to effectively use the preprocessor facility to be described shortly one must use WEB macros. Furthermore, one can supply arguments to WEB macros, a very useful feature that we describe now.

7.1.2 Function-like macros

The syntax of function-like macros is

@m *identifier*(*argument-list*) *replacement-text [optional C-style comment]*

The *argument-list* is a comma-separated list of parameter names. (The list may be empty.) If the *argument-list* is terminated by an ellipsis (...) instead of a parameter name, the macro is allowed to have a variable

number of arguments (unlike ANSI C, which permits only macros with a fixed number of arguments). In simple cases things work the way one would expect: the actual arguments are determined and the parameters in the *replacement-text* are replaced by those arguments. The general mechanism is slightly complex and is explained below. First, however, here are simple examples of function-like macros. (In some of the examples to follow, the result of the macro expansion is shown in a comment, enclosed by single left and right quotes. These quotes are not to be confused with FORTRAN's character strings; they would not be part of the actual macro expansion.)

```
@r
@* FUNCTION-LIKE MACROS.
@m MULT(a,b) ((a)*(b)) // A macro with 2 args.  $a$ and $b$ are dummies.
@m EAT(x) // Gobble up the argument.

@f LOOP for
@m LOOP() do i=0,N;
    do j=0,N

@A
z = MULT(x-5,y); // Expands to ‘z = ((x-5)*(y));’.
if(z < 0.0) return NO; // Uses the object-like macro |NO| defined above.
LOOP()
    {...}
```

Several remarks can be made about the previous examples. First, a standard remark: the parentheses in the replacement text of *MULT* are absolutely essential; consider what would have happened if *a* had not been parenthesized. Second, one may ask why *LOOP* was defined as a function-like macro with no arguments instead of as an object-like macro. `FTANGLE` would create the same expanded code in either case. However, it is useful to define things as we did in order to help `FWEAVE` format the macro properly. The format command tells `FWEAVE` to treat *LOOP* in the same way that it treats **for**. In RATFOR the keyword **for** expects a parenthesized expression to follow; our definition of *LOOP* accomodates that. Thus, in the output it will actually appear in boldface, and the body of the **LOOP** will be properly indented.

♣ 7.1.3 Extensions to WEB macro syntax

There are two special (and partly experimental) variants of the ‘`@m`’ command. The construction ‘‘`@m*`␣...’’ means that the macros is allowed to be recursive. *(However, recursive macros are not implemented in the present version.)* The construction ‘‘`@m[`*letters*`]`␣...’’ says that this macro is associated with automatic insertion material. See the subsequent discussion of RATFOR for more details.

♣ 7.1.4 Stringizing

Several special tokens beginning with ‘`#`’ may appear in the replacement text of `WEB` macros. The most important of these are ‘`#`’ and ‘`##`’, which are borrowed from the ANSI C preprocessor. In the ANSI usage, the token ‘`#`’ (“stringize”) must appear before a macro parameter (dummy argument). It and the parameter will be replaced by a string literal (appropriate for the language in force at the moment of expansion) constructed out of the corresponding actual (unexpanded) argument. (In FORTRAN, strings are delimited by single quotes; in RATFOR and C, they are delimited by double quotes.) For the simplest possible example,

“Several special tokens beginning with ‘#’ may appear in the replacement text of WEB macros.”

```
@r
@* STRINGIZING.
```

```
@m S(s) #s
@A
S(Hello); // Expands to ‘"Hello";’.
@n
    S(Hello) // In Fortran, the same macro expands to ’Hello’.
```

What happens when the argument to a stringize operation is already a string? The answer, consistent with ANSI C, is that the original quotes are escaped appropriately and the whole thing is surrounded by quotes. Therefore, `"hello"` stringizes to `"\"hello\""` in C. This may not be what you want; see the discussion of the `#*` operator below.

♣ 7.1.5 Making single- and double-quoted strings

If one stringizes a parameter in C, the parameter will always be enclosed in double quotes. In cases involving manipulations of single characters, one may instead desire single quotes. To stringize a parameter and force it to be enclosed in a particular kind of quote, use the `#’` or `#"` commands, which are extensions to ANSI. Thus, if in C one defines "`@m A(c) #’c`", then "`A(x)`" expands to "`’x’`".

♣ 7.1.6 Token pasting

The token ‘`##`’ ("paste") merges together the things on either side of it. If possible, a new identifier is created. For example, `pas##te` becomes the new identifier `paste`; however, nothing effectively happens if one says ‘`pas##(`’. Note that the token-pasting operation can create new identifiers that may themselves be macros; these will be expanded when the result is rescanned. For a simple example,

```
@* PASTING.
@m PASTE(a,b) a##b
@m paste 1
@A
PASTE(pas,te) // Expands to ‘1’.
```

♣ 7.1.7 Macro expansion

Macro expansion is intrinsically recursive, and one must be aware of the order of expansion of various objects. The complete expansion of a macro proceeds through several steps. The intent is that the procedure emulate that of the ANSI C preprocessor.

First, the identifier is recognized and the expected number n of arguments is determined. If the identifier is object-like, a parenthesized, comma-delimited list of actual arguments is expected (except that if $n = 0$, there should be no arguments inside the parentheses). It is an error if the parentheses aren't found or if the actual number of arguments does not agree with the original definition. Within the parentheses, commas protected by balanced parentheses do not count in determining the end of the argument. Thus, the second argument to the three-argument macro call ‘`MACRO(a,(b,c),d)`’ is ‘`(b,c)`’.

“Macro expansion is intrinsically recursive.”

Next, the replacement text associated with the macro is examined. If that text contains instances of parameters preceded by ‘`#`’, both ‘`#`’ and the parameter are replaced by a string built out of the actual argument (which is not expanded). If there are parameters preceded or followed by ‘`##`’, those parameters are replaced by their corresponding actual arguments; again, these arguments are *not expanded*. Otherwise, parameters are replaced by the actual arguments *after those arguments have been exhaustively scanned for further macro expansions*.

After argument substitution, any paste operations are performed. Then the result is rescanned for any further macros, which will be, in general, expanded. Rescanning continues until nothing was expanded on the last pass. For example, given the definitions

```
@c
@m A 1
@m B 2
@m AB 3
@m C(a,b) a + a##b + #b
```

the macro call `C(A,B)` becomes after argument substitution and pasting '`1 + AB + "B"`'; this is rescanned to yield the final result '`1 + 3 + "B"`'. Note that macros inside strings are not expanded, and that macro rescanning does not simplify numeric expressions such as $1 + 3$. (Such simplifications can be accomplished by the `_EVAL` built-in function described below.)

In one important situation macros are not expanded. If a macro is being expanded and its name is encountered again during that expansion, then it is not expanded. This prevents various kinds of infinite recursion, the simplest candidate being '`@m A A`'—this expands to '`A`', not an infinite loop. Also, consider the previous example of `PASTE(a,b)` and experiment to find out what happens when you say '`PASTE(PAS,TE)`'. (This construction is illegal.)

```
@c
@ Here are further examples of stringizing and token-pasting:

@m P(s) #s = s // Displays the result of a macro expansion.
@m VAR(i) var##i // Makes a new identifier name.
@m RECURSE(a,b) a##b(a,b) // Possibility for recursion here.
@m X 1
@m Y 2
@m Z ZZ
@m XY Z
@A
P(VAR(2)) // -> '"VAR(2)"=var2'
P(RECURSE(X,Y)) // -> '"RECURSE(X,Y)"=ZZ(1,2)'
P(RECURSE(RE,CURSE)) // -> '"RECURSE(RE,CURSE)"=RECURSE(RE,CURSE)'
```

Note that the last example did not get hung in an infinite loop.

The macro facilities that have been described so far are intended to emulate those of ANSI C. A wide variety of tasks can be accomplished with them. Although in the following paragraphs we describe various `FWEB` extensions to the ANSI C preprocessor, *it is recommended that you employ these only as a last resort*. The sparser the constructions and features that you deal with, the less likelihood there will be for programming errors.

7.1.8 Including a comma in a macro argument

Since the arguments of macros are delimited by commas, it is not immediately apparent how to introduce a comma as part of a macro argument. To do so, use the '`#,`' token. For example,

To include a comma in a macro argument, use the '#,' token.

```
@m A(x,y) [x][y]
@m B(x) A(x)
@a
B(3#,2) // -> '[3][2]'
```

Had one tried here to say "`B(3,2)`", he would have elicited an error message, since `B` is defined with only one argument.

7.1.9 Concatenating strings

In ANSI C, adjacent strings, possibly separated by white space, are concatenated on input. Thus, "`"a"␣"bc"`" is equivalent to "`"abc"`". `FWEB` does not do such concatenation when reading `WEB` source code; however, the `FWEB` macro preprocessor does do so. Thus, the expansion of the macro definition "`@m␣A␣"1""23"`" will be "`123`". This facility is useful because the preprocessor does not expand macros inside quoted strings. Thus, with

The FWEB macro preprocessor concatenates adjacent strings.

```
@c
@
@m PW sexylady
@m MSG "The password is '"_STRING(PW)"'."
```

MSG expands into "`"The password is 'sexylady'."`". This feature should be particularly useful in FORTRAN or RATFOR for easily constructing readable error messages.

♣ **7.1.**10 Quoting macros

In some situations, it is desirable to prevent a macro from being expanded. Enclosing a macro with special characters to prevent expansion is sometimes called "quoting" the macro. In other preprocessors such as `m4`, often left and right square brackets are used to prevent expansion; each time a pair of brackets is encountered, the outer brackets are stripped off. For example, with the definition '`@m N 10`' the FORTRAN source statement '`[N] = N`' would expand to '`N = 10`'. The `m4` preprocessor provides the **changequote** command to set the quoting characters. No such quoting facility exists in ANSI C, and experience shows that one is generally better off without it. For example, the previous example is better handled with the facilities of `FWEB` by recalling that the macro processor is case-sensitive while FORTRAN is not, so one could simply say '`n = N`'. However, in a few situations quoting is necessary or convenient. For example, it is difficult to include a comma unprotected by parentheses in a macro argument without quoting it. Unfortunately, the number of special characters available in `FWEB` as candidates for quote characters is severely limited; C has usurped almost all of them (including brackets, in particular) for its own purposes. Presently, in `FWEB` one level of quoting can be accomplished by enclosing the text that is not to be expanded with left single quotes—for example, '`'N' = N`'. (This style of programming is definitely not recommended.) As another example, consider a C example that aborts by printing the name of a buffer that was overrun by a string print operation. The example defines a three-argument macro `SPRINTF`, whose last argument is intended to actually be a comma-delimited *list* of arguments. Left quotes can be used to force that list to be treated as one argument (although see the discussion of variable arguments below for a better way). The example also illustrates the stringizing operation and an appropriate usage of outer macros (explained below) vs. `WEB` macros.

```
@c
@
@d N 1000
@m SPRINTF(buf_name,nmax,args)
    if(sprintf(buf_name,args) >= nmax) overflow(#buf_name)
@A
char temp[N];
SPRINTF(temp,N,‘"x = 0.2e\n",x‘);
```

In general, if you encounter a situation in which you think you need to quote something, pause and think again; very often there's a better way.

♣ **7.1.**11 Passing quoted strings unchanged through stringize

As noted, when the single token '`#`' is followed by a macro parameter, the parameter is made into a string. When '`#`' is followed by something other than a macro parameter, other actions are taken by `FWEB`.

Sometimes you would like an existing quoted string to be passed unchanged through the stringize operation. This can be accomplished with the `#*` operator. Thus, consider

```
@c
@* STRINGIZING REDUX.
@m S(s) #s
@m T(s) #*s
@A
S("hello"); // -> ‘"\"hello\"";’.
T("hello"); // -> ‘"hello";’.
@n
    S(’hello’) // -> ’’’hello’’’.
    T(’hello’) // -> ’hello’.
```

♣ **7.1.**12 Automatic statement numbering

For FORTRAN programmers, the most important of the extensions is related to automatic statement numbering. There are two distinct possibilities, both begun by '`#:`'. The construction '`#:0`' expands *at the time the definition is being stored* (in phase one) to a unique statement number. That same number will be used in all expansions of that macro throughout the program. Thus, it is possible to completely abandon numeric statement labels in FORTRAN simply by "declaring" alphanumeric labels as `WEB` macros, as in the following example:

"Numeric statement labels are unreadable anachronisms."

```
@n
@* STATEMENT LABELS.
@m START #:0
@m DONE #:0
@a
    program main
START: continue
DONE: end
```

Note that alphanumeric labels must be followed by a colon, just as in C. If you are a FORTRAN programmer, *it is strongly encouraged that you make use of this mechanism.* Numeric statement labels are an unreadable anachronism.

If '`#:`' is followed by a positive integer n, it expands into a unique statement number (the current automatic statement number plus n) *at the time the macro is being expanded* (in phase two). That number can appear more than once in the macro's replacement text. It will expand into the same statement number during a given expansion of the macro, but will generate a different unique number in each subsequent expansion. This feature is useful in defining macros such as error procedures that contain labelled code. For example,

```
@n
@m CHECK(i,err_num) if(i >= 0) goto #:1;
        call errprint(err_num);
            return;
    #:1 continue@;
```

(Of course, this particular example could have been accomplished without statement numbers at all by using an **if**...**end if** construction.)

The starting value *nnnnn* of the automatic statement number can be set by the command-line option '`-:`*nnnnn*'. Any user statement number must be less than that value. (`FWEB` does not check that this restriction is obeyed.) Again, proper use of the '`#:0`' and '`#:`n' options should entirely eliminate the need for explictly-defined user statement numbers in FORTRAN. *Note:* Automatic statement numbers are not intended to replace **if**-, **case**-, or **do**-construct names in FORTRAN–90. For example, in the legitimate FORTRAN–90 construction

```
iterate:  do
    ...
end do iterate
```

the label *iterate* should not be declared as an automatic statement number.

♣ **7.1.**13 Preventing macro expansion

Next, we have the "don't expand" option '`#!`'. As stated earlier, usually actual macro arguments are exhaustively expanded before they are substituted into macro replacement text. To prevent this, preface a macro parameter by '`#!`'. Consider, for example,

```
@c
@* PREVENTING MACRO EXPANSION.
@m A 1
@m S(s) #s
@m R(expr) S(expr)
@m Q(expr) S(#!expr)
@A
S(A) // -> ‘"1"’
R(A) // -> ‘S(1)’ -> ‘"1"’
Q(A) // -> ‘S(A)’ -> ‘"A"’
```

♣ **7.1.**14 Module names in macro definitions

Occasionally one may want to include a module name as part of a macro definition. The original design of `WEB` precluded this possibility, since encountering an '`@<`' construction while in the definition section signaled the start of a named code section. Therefore, within a macro definition the special construction `#<...@>` may be used to represent a module name. Thus, you can make definitions such as

```
@m SHORTHAND #<First stuff@>@; #<Last stuff@>
```

Of course, you lose the readability of the module names in the code proper if you do things this way. Generally, the module name would be only a small part of a much longer macro definition.

♣ **7.1.**15 Macros with variable numbers of arguments

We now discuss macros with variable numbers of arguments. These are indicated by an argument list that ends with an ellipsis (`...`). Certain built-in functions such as *_IFCASE* accept a variable number of arguments. You can also define them yourself, using some special tokens beginning with '`#`' to help.

WEB macros may have a variable number of arguments.

The construction '`#0`' expands into the number of variable arguments (those arguments that replace the ellipsis). Note that an empty argument is not the same as no arguments. For example, `#0` counts as follows:

```
@m A(...)  #0
@a
A => 0
A(x) => 1
A() => 1
A(x,y) => 2
A(x,) => 2
A(,y) => 2
A(,) => 2
```

The construction '`#`n', where $n > 0$, expands into the n^{th} variable argument (counting from 1).

The constructions '`#{0}`' and '`#{`n`}`' are just like '`#0`' and '`#`n' except that the argument inside the braces may be an macro expression [anything that you could put inside `@#if(...)`] that is known at output time. This is often useful in conjunction with the `_DO` built-in macro. For example,

```
@n9
@
@m BFORALL(nmin,nmax,mmin,mmax,...)
        forall(i=nmin:nmax,j=mmin:mmax) #{I}(i,j) = #{I+1}(j,i)
@a
_DO(I,1,6,2)
    {
    BFORALL(1,5,0,20,v1,v2,v3,v4,v5,v6)
    }
```

The construction '`#[0]`' is just like '`#{0}`' except that it counts the fixed arguments as well. The construction '`#[`n`]`' is just like '`#{`n`}`' except that the counting begins with the first fixed argument. Thus, in the above call to *BFORALL*, one has `#0` = 6, `#{0}` = 6, `#[0]` = 10, `#[2]` = 5, and `#[6]` = `#{2}` = `v2`.

Finally, the construction '`#.`' expands into a comma-separated list of all the variable arguments.

As examples,

```
@m V(x,y,...)  x + y + f(#0,#.)  + #3
@m V1(x,y,...)  x + y _IFCASE(#0,,+g(#.))
@a
V(a,b,x1,x2,x3) => a + b + f(3,x1,x2,x3) + x3
V1(a,b) => a + b
```

```
V1(a,b,c) => a + b + g(c)
```

The *V1* example shows that the conditional builtins can often be used to advantage in conjunction with variable arguments.

We can use variable arguments to reconsider the previous example of the SPRINTF macro:

```
@m SPRINTF(buf_name,nmax,...)
    if(sprintf(buf_name,#.)  >= nmax) overflow(#buf_name)
@a
char temp[N];
SPRINTF(temp,N,"x = 0.2e, y = 0.2g\n",x,y); // 2 fixed args, 3 variable ones.
```

This works just fine. However, a special feature of this example is that valid syntax demands that there be at least one variable argument (the string template). Suppose instead we had defined

```
@m SPRINTF(buf_name,nmax,strng,...)
    if(sprintf(buf_name,strng,#.)  >= nmax) overflow(#buf_name)
```

Now there's potential difficulty when there are no variable arguments. For example, one might expect "`SPRINTF(temp,N,"hello")`" to expand to "`if(sprintf(temp,"hello",)` ..."; however, the missing argument is invalid C syntax. Since this kind of construction seems to arise fairly frequently, we have the following special *experimental* rule: *if* `#.` *is empty and it immediately follows a comma in the macro expansion, the comma is deleted.* (Whether this is a good idea remains to be seen. More elaborate alternatives are possible; complain if you don't like this.)

♣ 7.1.16 Debugging macros

If one wishes to debug an errant macro, there are several possibilities. First, one could simply dump its current expansion into the output file. For example,

```
@m DUMP(...)  [#.]
@a
DUMP(A,B(1),C(x,y,z))
```

writes a comma-separated list of the expansions of three macro calls, surrounded by brackets. Alternatively, one can write information to the terminal with the built-in function `_DUMPDEF`. The format is the same as above; just replace `DUMP` by `_DUMPDEF`. For each macro call in its argument list, `_DUMPDEF` writes two lines. The first is a representation of the original macro definition; the second is its current expansion.

♣ 7.2 Outer macros

As we have said, WEB macros are expanded by TANGLE as it creates the compilable output file during phase two. This means that the output file, which is what you will actually be debugging, need not be very readable if the macros were at all sophisticated (and, therefore, very useful). For FORTRAN programming, one has no choice. However, for C programming it is desirable to defer macro expansion to the C preprocessor so that your output file remains readable. This can, of course, be done with no special effort just by using the C preprocessor `#define` command as part of your code fragment. Usually, you should insert these in the code section of the module in which they are first used so that the flow of the logic is clearest. However, what you usually really want is for your preprocessor definitions to appear at the top of your file, no matter where you actually defined them. (There are exceptions to this.) To help you achieve this effect while maintaining logical clarity, the WEB system supports the concept of "*outer*

To defer macro expansion to an external preprocessor, use the '@d' command.

macros". These are defined by the command '`@d`', which is allowed in the definition section only. `WEB` does not expand these; rather, it just collects them and copies them at the start of phase two to the beginning of the output file in a format appropriate for the relevant language compiler. For symmetry, the command '`@u`' is also provided to undefine an outer macro.

The format of outer macros depends on the language in force. If the current language is C, the outer macros should, of course, be in the format appropriate to the C preprocessor; an outer C definition of the form '`@d A 1`' will appear in the beginning of the C output file as '`#define A 1`'. Otherwise, the outer macro should be in the format appropriate to the `m4` preprocessor. In languages other than C, the definition '`@d (B,2)`' will appear in the beginning of the output file as '`define(B,2)`'. (With the advent of `FWEB`'s macro processor, the need for outer macro definitions in languages other than C should virtually disappear.)

For example, when the language is C, the statements

```
@
@d A 1
@d B 2
@
@u A
@d A 2
```

will be output as follows:

```
#define A 1
#define B 2
#undef A
#define A 2
```

A C program maintained with `WEB` should almost exclusively contain the outer macro, '`@d`' commands. The internal, '`@m`' commands should be used only when the `WEB` system provides a macro feature not included in the C preprocessor. Those features mostly include stringizing and token pasting (included in the ANSI C standard, but not in many extant compilers). If you are using an ANSI C compiler, your need for `WEB` macros will be slight, although a few of the extensions such as `#!` or variable arguments may prove useful.

♣ 7.3 Deferred macros

Returning now to the internal `WEB` macros, their order of evaluation requires further discussion. It is important to understand that *all* text, including macro definitions, is collected, tokenized, and stored during `TANGLE`'s phase one. However, with certain exceptions (involving the preprocessor) to be described, *no macros are actually expanded during phase one*. The reason for this is that the actual source text in which the macros are embedded and on which they must operate is not known until all the text has been collected and placed into the proper modules. Therefore, it is during phase two, when the completed code text is output, that macros are expanded. However, in the original design of `WEB` this led to an annoying difficulty—namely, `WEB` macro definitions could be *retroactive*.

The situation arises as follows. In the original design, macros were allowed only in the definition section of a module. Since all such macros definitions are collected during phase one, they are all known by the time code is output during phase two. (Effectively, they are all placed at the top of the unnamed module.) Thus, a definition made in the definition section of, say, module 100 could lead to a macro being expanded in module 1 if the code in module 1 contained a reference to that macro name. Although this is usually the desired effect, it may not be in all cases. Consider, for example, the following example (see the discussion of preprocessor commands, below):

```
@ Here is a peculiar example of retroactive macro definition.
```

```
@m A 1
@#if(A==1)
    x = A;
@#else
    x = B;
@#endif

@ Another section.
@m A 2
.
.
```

Here the generated code will be "`x = 2;`", not "`x = 1`" or "`x = B;`". Note also that since the sections, hence the definition parts, are not required to appear in any particular order, the order of encountering `WEB` macro definitions is somewhat indefinite. This can lead to confusion and hard-to-understand bugs if macros are redefined and sections are subsequently moved around.

For complete flexibility, therefore, `FWEB` introduces the notion of *deferred macros*. Deferred macros are nothing more than `WEB` macros defined (again with '`@m`') in the *code* section rather than in the definition section. A deferred macro definition, although it is stored away in a safe place during phase one, becomes known to the `WEB` macro processor only at the point in the code where the definition is made, when the code is being output during phase two. Thus, the order in which the deferred macro definition is made is unambiguous (it is determined by the intrinsic structure of the code, not by the order in which things were explained) and the deferred definition cannot be inadvertently expanded retroactively. Remembering the previous discussion of C preprocessing, we see a complete analogy: deferred `WEB` macros are analogous to the use in C of `#define`; both are used in the code section. All definitions made in the definition part behave similarly in that they are placed at the beginning of the appropriate place: outer macro definitions ('`@d`') are placed at the beginning of the output file; `WEB` macro definitions in the definition part ('`@m`') are effectively placed at the beginning of the unnamed module.

Don't define WEB macros in the code part unless you absolutely have to.

Most of the time, it will be adequate to define a `WEB` macro in the definition part, thereby placing it at the beginning of the unnamed module, and it is recommended that you do so whenever possible. Reserve deferred macro definitions for those relatively rare instances when the order of defining the macro really does matter.

Although deferred `@m` commands work as specified, presently preprocessor commands (see below) such as `@#if` or `@#undef` do not work as one might expect when referring to deferred macros. Presently, such commands are executed during phase one (on input), whereas the deferred macros become known only during phase two (on output). Luckily, there is an an alternative way of handling preprocessing during output, namely to use built-in functions such as `_IF`. See the discussion of built-ins below.

♣ 7.4 Language dependence of macros

`WEB` macros are by and large not sensitive to language when they are being memorized, and with a few exceptions they will expand in the same way no matter what language is current during output. (An example of an exception is the behavior of the stringizing operation, which must build a string using either a single or double quote depending on the language. Another example is the `_ROUTINE` built-in function which at presently is defined only for RATFOR.) Generally such uniform expansion is desirable. For example, here is how one can supply a macro definition in C and a **parameter** statement in FORTRAN with a common value:

```
@n
@ Web macros are known to all languages.
@m NN 100 // One can affect all languages by changing this one number.
@c
@d N NN
@A
    parameter(N=NN) // A fragment of a Fortran code.
@c
@<C functions@>@;
@ A simple C routine.
@<C...@>=
f()
{
return N;
}
```

If language-sensitive behavior is desired, it can be achieved by using the `_LANGUAGE` built-in function in conjunction with an `_IFELSE`, or the `_LANGUAGE_NUM` built-in in conjunction with an `_IFCASE`. See further discussion below.

`WEB` macros can also be defined from the command line by using the command-line option '`-m`'. (See the detailed explanation in the section about command-line options.) Command-line macros are processed at the beginning of the first definition section. Defining macros from the command line is an efficient way to customize a code without re-editing it. For example, you can supply fixed array bounds to a FORTRAN program employing one of the two equivalent statements

WEB macros can be defined from the command line with the option '-m'.

```
FTANGLE test -mN=1000
FTANGLE test -m"N 1000"
```

which is equivalent to saying '`@m N 1000`' at the beginning of the first definition section, then including statements such as '`integer x(0:N)`' in your code. The facility is particularly useful in conjunction with the `FWEB` preprocessor, to be described next.

♣ 7.5 Preprocessing

A common situation arises when one is designing a code to run on more than one machine. One might want to define a macro with machine-dependent values, or one might want to selectively include one or another piece of code. In addition to macro definitions, WEB has a complete preprocessing language to help in this respect.

The WEB preprocessor is patterned after that for ANSI C. All preprocessing commands begin with '@#'. As in ANSI C, they need not, in general, begin in in column 1 (they must do so, at present, in FORTRAN's fixed-format mode). Unlike ANSI C, however, there may be no white space between '@#' and the command name. (WEB interprets an isolated '@#' as a command to insert a blank line.) In C and RATFOR they may be continued to subsequent lines by using a backslash; in FORTRAN, they are presently restricted to end on the same line (including any optional comment). They are

"The WEB preprocessor is patterned after that for ANSI C."

```
@#define macro_name replacement_text
@#undef macro_name
@#ifdef macro_name
@#ifndef macro_name
@#if expression
@#elif expression
@#else
@#endif
```

The '@#define' command is entirely equivalent to '@m'. One can undefine a macro definition with '@#undef'. (All previous definitions are completely lost; definitions are not stacked.) '@#ifdef' and '@#ifndef' test whether a macro is or is not defined. (These commands are special cases of '@#if'; see the discussion of the 'defined' operator below.) The '@#if...@#elif...@#else...@#endif' construction evaluates *expression*, which must consist of symbols known to the preprocessor at the current point of the input scan, then takes action depending on whether *expression* is true (nonzero) or false (0). All preprocessing commands may appear in either the definition part or the code part. In the definition part, FTANGLE will not process macro definitions that are between the false part of the preprocessor conditionals. In the code part, FTANGLE will not store or output code that is between the false part. The conditionals may be nested.

Note that all preprocessor commands, including in particular those in the code section, are expanded during phase one. This is a design flaw (maybe it will be corrected someday) that means that the preprocessor commands will not work correctly with deferred macros, which become known only during phase two. If you need to use deferred macros with preprocessor commands, use the built-in forms of the preprocessor commands such as _IF. Built-ins are discussed below.

Use of the WEB preprocessor is entirely analogous to that of C. If you are a C programmer, you will need the WEB preprocessor commands relatively rarely; operations on your source code are usually better done with the C preprocessor. If you are a FORTRAN programmer, however, you should find the preprocessor facility extremely useful.

It is conventional to end long preprocessor constructions as follows, to enhance the quality of the documentation:

```
@#if(CRAY)
...
@#endif // |CRAY|
```

♣ 7.6 Expression evaluation

The *expression* in constructions such as `@#if` *expression* is evaluated by a built-in expression evaluator that can also be used for other purposes, such as in macro expansion. Its behavior is again motivated by expression evaluation in ANSI C; it is not quite as general, but should be more than adequate. It supports both integer and floating-point arithmetic (with type promotion from integer to floating-point if necessary), and the ANSI '`defined`' operator. Operators with the highest precedence (see table below) are evaluated first; as usual, parentheses override the natural order of evaluation. The unary operator '`defined`' has the highest precedence; all the other unary operators have the next highest (and equal) precedence; then come the binary operators. When the operator exists in C, the action taken by `FWEB` is precisely that that the C compiler would take. Arithmetic is done in either **long** or **double** variables, as implemented by the C compiler that compiled `FTANGLE`. (This was the easy choice, not necessarily the most desirable one.)

The operators, listed from highest precedence to lowest, are as follows:

Unary operators:

`defined` — '`defined`' is a unary operator that acts on identifier tokens. '`defined id`' or equivalently '`defined(id)`' evaluates to 1 (true) if the identifier is defined as a `WEB` macro; to 0 (false) otherwise. The construction '`@#if defined A`' works the same way as `@#ifdef A`, but you can use '`defined`' in expressions, as in

```
@#if defined(A) || defined(B).
```

(The parentheses around the macro names are optional.) With the advent of '`defined`', the `WEB` preprocessor statements `@#ifdef` and `@#ifndef` become redundant, but are often useful shorthands.

`-` — Unary minus.

`!` — Logical `NOT`. `!`*expression* evaluates to 0 if *expression* is nonzero, and evaluates to 1 if *expression* is 0.

`~` — One's complement of an integer. For example, `~0` $= -1$.

Binary operators:

`^^` — Exponentiation (all languages). `2^^3` $= 8$.

`^`, `**` — Exponentiation (FORTRAN or RATFOR).

`*`, `/`, `%` — Multiplication, division, and modulus: a `%` b means $a \bmod b$; for example, `5 % 3 = 2`.

`+`, `-` — The usual plus and minus.

`<<` — a `<<` b means shift integer a left b bits. $1 \ll 3 = 8$.

`>>` — As above, but right-shift. $7 \gg 2 = 1$.

`<`, `<=`, `>`, `>=` — Evaluates to 1 if the inequality holds, to 0 otherwise. E.g., `(2.0 < 3.0)` evaluates to 1.

`==`, `!=` — a`==`b (a`!=`b) evaluates to 1 (0) if a equals b; evaluates to 0 (1) otherwise.

`&` — Bitwise `AND`. The truth table is `0b1100 & 0b1010 = 0b1000`.

`^` — Bitwise `EXCLUSIVE OR` (C). (For FORTRAN, use '`.xor.`'.) The truth table is `0b1100 .xor. 0b1010 = 0b0110`.

`|` — Bitwise `OR`. The truth table is `0b1100 | 0b1010 = 0b1110`.

`&&` — Logical `AND`. a `&&` b evaluates to 1 if both a and b are true (nonzero).

`||` — Logical `OR`. a `||` b evaluates to 1 if either a or b are true.

Note in particular the use of the single caret, which is language-dependent: it is an exponentiation operator for FORTRAN (just as in TEX), but the `EXCLUSIVE OR` operator for C. Also, note that the bitwise operators should almost never be used. For logic, almost always you will be using `==`, `!=`, `&&`, and `||`.

The preprocessor commands are "active" only for FTANGLE. FWEAVE will format them in a reasonable way, but you can't, for example, comment out some active WEAVE operation such as '@f' with '@#if...@#endif'. Furthermore, you can't use preprocessor commands to comment out the '@i' include statement. (Include statements are processed by the input driver, so the '@i' command never gets to the innards of the WEB processors. A single '@i' command can be commented out with '@%', since '@%' is also processed by the input driver.)

Here are some suggested uses for the preprocessing facility. To comment out code, say

```
@#if(0) // You can also say ``@#if 0''.
    code // This text will not appear in the tangled output.
@#endif
```

To conditionally define a macro, say something like

```
@m CRAY // You might define this macro from the command line.

@#ifdef CRAY
    @m CRAY 1
    @m VAX 0
@#else
    @m CRAY 0
    @m VAX 1
@#endif // |defined CRAY|

@#if CRAY
    @m INIT cray_code
@#else
    @m INIT vax_code
@#endif // |CRAY|
```

To conditionally select a block of code, say something like

```
    program main
@#ifdef CRAY
    @<Cray stuff@>
@#else
    @<Vax stuff@>
@#endif // |defined CRAY|
    end
```

♣ 7.7 Built-in macro functions

In certain circumstances a macro function is desired that cannot be accomplished by a user definition, or is so ubiquitous that it seems useful to save the user the trouble of defining it himself. Therefore, for enhanced flexibility, FWEB contains a small number of *built-in* functions. Each of these functions begins with an underscore and is fully in upper case. These functions behave just like WEB macros. Therefore, *do not define any macros of your own that begin with underscores,* even if they don't correspond to the name of a built-in function. This restriction is intended to prevent conflict with certain auxiliary macros used internally by WEB in the course of expanding its built-ins.

"FWEB contains a small number of built-in functions."

The following list describes the built-in functions in a (somewhat) logical order. They are summarized alphabetically in Appendix L.

♣ 7.7.1 _EVAL

The built-in function _EVAL evaluates its argument using the preprocessor expression evaluator described above. (If an error is encountered during the evaluation, a message is printed and the argument is returned without any evaluation.) For example,

```
@m A 2
@m B 3
@A
_EVAL(defined A && defined(B)) // ‘1’
_EVAL(A + 5.0*B) // ‘17.0’ (Note the promotion to floating point.)
_EVAL(A==B) // ‘0’
_EVAL(A==1 || A==2) // ‘1’
_EVAL(x) // ‘x’
array[_EVAL(B+1)] // ‘array[4]’
```

The last example above shows a common usage of _EVAL, namely in providing more readable tangled code. That is, although many C compilers accept the construction array[3+1], in some circumstances it may be more meaningful to see array[4] while you're debugging.

♣ 7.7.2 _DEFINE, _M, _IFDEF, _IFNDEF, _UNDEF

A built-in form of deferred macro definition is achieved via _DEFINE or _M. These two names are synonyms; they are used by enclosing in parentheses all the text that would normally follow an ‘@m’, such as

```
_M(N␣5) // Equivalent to ‘@m N 5’.
_M(X(a,b) ((a)-(b)))
```

A subtlety occurs with a definition of the form "_M(foo␣(bar))". Because spaces are removed early in the tokenization process, this definition is equivalent to "@m foo(bar)" (a one-argument macro with null replacement text) instead of what one probably intended, "@m foo␣(bar)" (a zero-argument macro). To get the latter effect, say "_M(foo=(bar))".

The companion macros _IFDEF, _IFNDEF, and _UNDEF are also provided. The syntax of the first two is, e.g, _IFDEF(id,true_text,false_text). Remember, these are macros, so their arguments must be enclosed in parentheses—e.g., _UNDEF(X).

♣ 7.7.3 _DO

A powerful means of *repetitively* defining a macro is provided by _DO. The syntax is

```
_DO(I,Imin,Imax[,dI])
{
text
}
```

This command is FWEB's version of the FORTRAN **do** loop. It behaves as though one said "**do** I = $Imin, Imax[, dI]$ {*text*}", where the equals sign means a macro definition. The loop index should not be used as a WEB macro for any other purposes, since its previous definition will be destroyed.

(For symmetry, there ought to be a _FOR statement, but that's not in place yet.)

♣ 7.7.4 _INCR, _DECR

The macros `_INCR` and `_DECR` increment or decrement by 1 a macro that expands to a number. For example, if the `WEB` macro `N` expands to `5`, then `_INCR(N)` redefines `N` to be `6`. These are included for convenience; they are defined in terms of `_M` and `_EVAL`.

♣ 7.7.5 _IF

The conditional `_IF(expr,true_text,false_text)` expands to `true_text` if `expr` is true, or to `false_text` otherwise. *Note that in almost all situations the preprocessor commands* `@#if...@#else...@#endif` *provide a better, more readable alternative to* `_IF`*; use this built-in only as a last resort.* One case in which you must use `_IF` is when `expr` contains a deferred macro.

♣ 7.7.6 _ABS, _MAX, _MIN

A few other constructions built out of `_IF` have been included for convenience: `_ABS(a)`, `_MAX(a,b)`, and `_MIN(a,b)`.

♣ 7.7.7 _IFCASE

The `_IFCASE` macro provides conditional evaluation based on the value of a control integer. It is analogous to TEX's `\ifcase` macro or to FORTRAN's computed **goto**. The format is

```
_IFCASE(m,case_0,case_1,...,case_n,default)
```

Here m is an expression that evaluates to an integer in the range $0 \le m \le n$, and the macro expands into the text `case_m`. If m is not in the above range, then the default text is returned. (This is an example of a macro that uses a variable number of arguments.)

♣ 7.7.8 _IFELSE

The `m4`-like conditional `_IFELSE(macro1,macro2,true_text,false_text)` expands each of `macro1` and `macro2`, then does a character-by-character comparison. If the results are identical, `true_text` is returned, otherwise, `false_text` is returned. *Whenever possible use* `@#if` *instead.* Again, if `macro1` or `macro2` involve a deferred macro, you must use `_IFELSE`.

♣ 7.7.9 _LEN

The macro `_LEN(s)` interprets its argument as a character string (without expanding it if it is a macro) and returns the length of that string. For example, if you say "`@m N 56`", then `_LEN(N) -> 1`. Note that the string does not need to be in quotes; that's done for you. As a more complicated example, here's one way to create a self-counted Hollerith string:

```
@m HOL(s) _LEN(#!s)##H###!s // HOL(abc) -> '3Habc'
```

Note that we had to be alert for the possibility that the argument to *HOL* might inadvertently be a macro name, so we prevented expansion with the `#!` operator.

♣ 7.7.10 _POW

The exponentation macro `_POW(x,y)`, which expands x and y and returns x^y, is included for convenience; it is equivalent to `_EVAL((x)^^(y))`.

♣ 7.7.11 _TRANSLIT

The macro `_TRANSLIT(s,from,to)` interprets each of its arguments as strings (without expanding anything). Then s is modified by replacing any of the characters found in *from* by the corresponding characters in *to*. If *to* is shorter than *from*, then the excess characters in *from* are deleted from s. As a limiting case, if *to* is empty, then all the characters in *from* are deleted from s. For example, `_TRANSLIT(`s`,aeiou,12345)` replaces the vowels in s by the corresponding digits, and `_TRANSLIT(s,aeiou,)` deletes all the vowels. The backslash may be used to escape a character, as in ANSI C. For example, `_TRANSLIT("a\"\\d","d\\a\"","D,A'")` translates into ‘ `A',D`’. Here we had to explicitly enclose strings involving '`\"`' in double quotes in order to avoid a complaint about an unterminated string.

♣ 7.7.12 _A

The macro `_A` is the built-in equivalent of the `@'` or `@"` command. The construction `_A('x')` functions essentially the same as the command “`@'x'`”, and “`_A("xyz")`” is essentially the same as `@"xyz"`. (We say “essentially” because the `@'` and `@"` commands are expanded on input, whereas the built-in function `_A` is expanded on output.) The need for such a built-in is evident from the following example, in which the quantity to be converted to ASCII is constructed via macro expansion:

```
@m OCTAL(n) OCTAL0(\@&n)
@m OCTAL0(n) _A(#'n) // OCTAL(123) -> _A('\123') -> 0123.
```

Here one couldn't say something like “`@m OCTAL(n) @'n'`” for two reasons: first, `@` commands are expanded on input; second, the n parameter wouldn't be expanded because it's inside a quoted string. (Do you understand the function of the `@&` command in the definition of *OCTAL*?)

♣ 7.7.13 _STRING

The macro `_STRING(macro)` expands its argument, then makes a string out of it (using the `#*` operator). That is, it provides one extra level of expansion over the basic stringize operation. Compare, for example, the following expansions:

```
@c
@m A 1
@m S(s) #s
@A
S(A) // ‘"A"’
_STRING(A) // ‘"1"’
```

This macro is included for convenience, as the user can define it himself.

♣ 7.7.14 _UNQUOTE, _P

For certain special effects, one has the `_UNQUOTE` built-in function (formerly called “`_VERBATIM`”). If one says “`_UNQUOTE("`*string*`")`”, the macro returns *string* without the surrounding quotes. For example, this provides a way of getting the special character '`#`' into the output from within a macro. For example, the following macro creates C preprocessor keywords:

```
@m PP(keyword) _UNQUOTE("#")##keyword
```

For the user's convenience, the macro `_P` is a synomym for `_UNQUOTE("#")`. However, there is a subtlety here. Do you understand why if one says

```
@m QQ(keyword) _P##keyword
```

one gets the following?

```
PP(define) // -> ‘#define’
QQ(define) // -> ‘_Pdefine’
```

To answer this question, carefully study the rules about macro expansion and pasting. The proper way of defining QQ is to protect _P with left quotes:

```
@m QQ(keyword) ‘_P‘##keyword
```

♣ 7.7.15 _L, _U

Occasionally it is useful to change the case of a string parameter to a macro. This can be accomplished by means of the built-in functions _L (change argument to lower case) and _U (change to upper case). The arguments to these functions must be quoted strings; they return the case-converted string, also quoted. For example, “_L("ABC")” returns “"abc"”.

An application in which such actions might be useful concerns mixed programming in C and FORTRAN. Suppose that in one's C program the FORTRAN functions to be called had been distinguished by writing them in upper case. The code is supposed to be portable, and on at least one machine the case-sensitive linker understands things just as they are. However, suppose that on some other machine the FORTRAN library was compiled by a compiler that produced lower-case names—and, for good measure, had appended an underscore to the name. (This is precisely what happens on Sun workstations.) Here is how to macro up the situation to make everyone happy:

The built-in functions ‘_L’ and ‘_U’ change the case of their arguments.

```
@c
@ In the \WEB\ program \Fortran\ functions will be in upper case,
C functions will be in lower case.
@#if VAX
    @m F(name) name
@#else
    @m F(name) _UNQUOTE(_L(#name))##_ // Stringize, convert to l.c.,
               remove quotes, append ’_’.
@#endif
@a
cfcn();
F(FORFCN)(); // Either |FORFCN()| or |forfcn_()|, depending on machine.
```

♣ 7.7.16 _COMMENT

The macro _COMMENT(string) generates a comment in the output file. This is useful for macro definitions in which comments are desired; the usual C-style comments are stripped off as the definition is being absorbed. Thus, in C, the definition ‘@m TEST a;_COMMENT("Hello,␣world")b;’ expands into ‘a;/*␣Hello,␣world␣*/b;’. Note that the string argument must be in quotes. (Actually, this is essential only when the argument contains commas; otherwise, the commas will be interpreted as argument delimiters and an error message about the wrong number of arguments will ensue. However, it is good practice to always use the quotes.) If you had wanted the comment to be on a separate line, you could have inserted newlines into the string, as in "\nHello,␣world\n". In FORTRAN, the comment is automatically put on a separate line.

♣ 7.7.17 _ASSERT

Error mechanisms are provided by several macros, `_ASSERT` and `_ERROR`. The `_ASSERT(`*expression*`)` macro evaluates *expression*. If that expression is false, an error message is sent (to both the terminal and the output file) and processing is aborted immediately. This feature, motivated by ANSI C, is intended to help one verify certain conditions that must be true in order for processing to proceed. For example, suppose you always intended to define the macros *N1* and *N2* from the command line. To make sure you didn't forget, you could say at the beginning of your code

```
_ASSERT(defined N1 && defined N2)
```

Like all of the built-ins, `_ASSERT` is expanded on output, during processing of the *code part*. It will have no affect if it appears in the definition part.

♣ 7.7.18 _ERROR

The `_ERROR(string)` macro sends `string` to the standard error message facility, but does not abort. Thus, `_ERROR("That didn't work")` generates an error message giving the line number and the file in which the macro was expanded. Note that the `string` argument must be in quotes.

♣ 7.7.19 _DUMPDEF

The `_DUMPDEF` built-in is used for debugging `WEB` macros. It takes an arbitrary number of macro calls, separated by commas, as in

```
_DUMPDEF(A,B(1),C(x,y,z))
```

where it is assumed that A, B, and C are macros with 0, 1, and 3 arguments, respectively. For each macro, two lines are written to the terminal. The first is a representation of the original macro definition, the second is the expansion.

♣ 7.7.20 _LANGUAGE, _LANGUAGE_NUM

The two built-in functions `_LANGUAGE` and `_LANGUAGE_NUM` provide ways of endowing a `WEB` macro with a language attribute. The `_LANGUAGE` macro expands into an *identifier* (*not* a macro, unless you define it yourself) such as '`_C`'. (See the table below.) It is intended to be used in an `_IFELSE` statement. The `_LANGUAGE_NUM` macro expands into an integer (see the table below) and is intended to be used as the first argument to an `_IFCASE`.

Language	_LANGUAGE	_LANGUAGE_NUM
C	_C	0
C++	_CPP	1
FORTRAN–77	_N	2
FORTRAN–90	_N90	3
RATFOR–77	_R	4
RATFOR–90	_R90	5
TEX	_X	6

For example, here is a way of defining a macro to be the "natural" lower array index in either of C or FORTRAN, assuming that one programs in only those two languages.

```
@m LOWER _IFELSE(_LANGUAGE,_C,0,1)
```

If you also sometimes use C++, however, you need a three-way switch; the simplest way of achieving that is to use _LANGUAGE_NUM:

```
@m LOWER _IFCASE(_LANGUAGE_NUM,0,0,1)
```

This macro still isn't ideal, however. It returns an explicit value for the languages C, C++, and FORTRAN–77, but would return nothing for any other language. If you started to use FORTRAN–90 someday, you'd get into trouble, and this kind of error can be hard to track down. It's best to be very explicit:

```
@m LOWER _IFCASE(_LANGUAGE_NUM,0,0,1,1,1,1,0)
```

♣♣ 7.7.21 _STUB

If a reference to an undefined module is encountered, it is replaced automatically by a call to the built-in _STUB, with the module name as argument. The form of the call is language-dependent. Thus, if the module @<Absent@> is never defined, then a reference to @<Absent@> will tangle to the expansion of "_STUB(Absent)", which in C is by default "{missing_mod("Absent");}". In FORTRAN, the same reference would tangle to "call nomod('Absent')". If you don't like the default actions, you can redefine _STUB yourself with @m. The purpose of this macro is to help one compile and debug a code that is not fully developed.

♣ 7.7.22 _GETENV, _HOME

The built-in _GETENV(*ENV*) returns the value of the environment variable *ENV*, using the C library call *getenv*. The built-in _HOME is equivalent to _GETENV(HOME).

♣ 7.7.23 _VERSION

The built-in _VERSION expands into a string containing the WEB version number, such as "1.13".

♣ 7.7.24 _MODULE_NAME, _SECTION_NUM

Two built-ins provide information about where one presently in the web. _MODULE_NAME expands to the name of the current module that is being expanded; _SECTION_NUM is the number of the current section.

♣ 7.7.25 _MODULES, _SECTIONS

Two built-ins provide some statistics about the structure of the program being tangled or woven. _MODULES expands into an integer that is the total number of *independent* module names, plus 1 for the unnamed module. _SECTIONS expands into the maximum section number, as WEAVE would compute it. These numbers could be used as array bounds for various esoteric purposes. For an example of the use of these macros, see the demo program breakpt.web.

♣ 7.7.26 _DATE, _DAY, _TIME

Finally, there are various date and time macros that expand into strings containing the relevant information: _DATE, _DAY, and _TIME. These macros are intended primarily for internal use; they are used in generating the comments output at the beginning of the tangled output and at the end of the woven output. However, they are available to the user as well. These macros expand as follows:

```
_DATE → "August 15, 1989"
_DAY  → "Monday"
_TIME → "23:59"
```

The built-in functions are summarized alphabetically in Appendix L.

♣ 8. OVERLOADING OPERATORS and IDENTIFIERS

For special effects in the woven output, there are special commands to help one change the appearance of operators and identifiers.

♣ 8.1 OVERLOADING OPERATORS

A feature common to both C++ and FORTRAN–90 is *operator overloading*, the ability to extend or redefine the definition of an operator such as '`.FALSE.`' or '`=`'. FORTRAN–90 even allows one to define new "dot" operators—for example, one might define the operator "`.IN.`" to test for inclusion in a set. In a nontrivial extension of the original design, `FWEAVE` allows one to define how overloaded operators should appear on output—for example, it is much more readable to read "**if**(*x* ∈ *set*)" than "**if**(x `.IN.` *set*)". Indeed, this feature can be used even when the language does not permit overloading in order to customize the appearance of the woven output.

One can change the printed appearance of an operator with the '@v' command.

The '`@v`' control code is used to change the appearance of an operator. The format is

`@v` *new_operator_symbol_or_name* `"`*TEX material*`"` *old_operator*

This means "Display the new operator according to the *TEX material*, but treat it like the old operator—e.g., unary or binary—for formatting purposes. The quoted TEX material is treated just like a C string, so for example if you want to include a backslash you must escape it with another backslash. For example, we can make an equals sign display on output as a large left arrow by saying

```
@v = "\\Leftarrow" =
```

Then, if you say "`x = y;`", it will display as "$x \Leftarrow y$;". Two FORTRAN examples are

```
@v .FALSE. ".FALSE." .FALSE.
@v .IN. "\in" +
```

which makes the operator `.FALSE.` display as "`.FALSE.`" instead of the default $\mathcal{F}$ (but still behave in the default way—i.e., like an ordinary expression), and makes the operator `.IN.` display as "∈" (and behave like a binary operator).

This feature can go a long way toward enhancing readability of the woven output, particularly when operators are actually being overloaded. It can also lead to arbitrarily bizarre output that no-one else will understand. As usual, restraint is advised.

Examples of operator overloading can be found in the sample C++ and FORTRAN–90 code in Appendix E.

♣ 8.2 OVERLOADING IDENTIFIERS

Although operator overloading is quite useful, it does not allow one to change the appearance of identifiers. In its most general form, such a facility becomes quite complicated; one must endow FWEAVE with a macro-processing facility analogous to that of FTANGLE. This has not been done yet (probably it will be someday). In the meantime, one has the command '@W, which provides a restricted form of such a facility. *This command, new with version 1.30, is experimental, and not firmly established. Changes in usage and/or syntax may be made in future versions.*

One can change the printed appearance of an identifier with the '@W' command.

The most general form of the '@W command is

@W *identifier* "*replacement text*"

This means: Replace any references to *identifier* in the woven output with the *replacement text*.

A more restrictive form is

@W *identifier* \newmacro

which replaces references to *identifier* with a call to \newmacro. (Note that there are no quotes in this form.)

The shortest form is

@W *identifier* .

which replaces references to *identifier* with a call to \identifier. For example, the identifier x normally appears in woven output as "\Wshort{x}". If one says

```
@W x .
```

one will instead get the macro reference "\x", which could be defined to give a variety of special effects.

It should now be clear how the previous "**call** *integrate*" example was formatted. One simply said

```
@n
@
@W alpha .
@W beta .
@W fM "f_\\rm M"
@a
    call integrate(x,alpha,beta,fM)
```

One of the important uses of this facility is to expedite special formatting of array references. This subject is discussed separately below in the section on "Special array formatting," where an example is given.

9. RATFOR

Closely related to macro preprocessing is the notion of *statement translation*. In statement translation, FWEB recognizes a special keyword or construction that is not part of the source language, and automatically translates that construction into valid compilable code. It is a more general operation than macro expansion, although it reduces to that in the simplest cases.

In the present version of FWEB, statement translation is active only when the language is RATFOR. The RATFOR language was introduced by Kernighan and Plauger in *Software Tools* as a tool for explaining good programming practice; it is also a vastly superior way of writing understandable FORTRAN code. In their words, "bare Fortran is a poor language indeed for programming or for describing programs. So we have written all of our programs in a simple extension of Fortran called 'Ratfor' (short for Rational Fortran). RATFOR provides modern control flow statements..., so we can do structured programming properly. It is easy to read, write, and understand, and readily translates into Fortran... or similar high-level languages." Unfortunately, extant RATFOR processors essentially convert the new statements into pre-FORTRAN–77 code, with many **goto**'s, so the resulting code can be somewhat messy and hard to read. If your code worked perfectly the first time, this would not be an issue since you would never have to see the FORTRAN file. However, if it is necessary to work with a debugger (and it very frequently is!), then it is the generated FORTRAN code at which one will be looking. It is desirable to make that code as readable (and as efficient) as possible.

"Ratfor provides modern control flow statements..., so [one] can do structured programming properly."

Thus, FTANGLE has been taught the RATFOR language and, by default, will translate it directly to FORTRAN code. Translations into both FORTRAN–77 and FORTRAN–90 are supported.

(Certain existing codes use a now-archaic mode of FTANGLE in which RATFOR code was just TANGLEd into a RAT file that was intended to be passed through a separate RATFOR processor. *This mode has died; it should not be used for new codes.* To tell FTANGLE to *not* translate RATFOR statements, you must use the command-line option '-q'.)

The syntax of the RATFOR language understood by FWEB is as C-like as possible. It features the following:

- The syntax is totally free-form.
- Declarations and statements are ended by semicolons. (An auto-semi mode can fill these in for you; see below.)
- Program units should be begun by either the **program**, **subroutine**, **function**, or **blockdata** statement. Use **program main** instead of an unnamed main program; the latter may confuse FWEAVE.
- The body of the subroutine is delimited by braces (as are groups of statements that form the bodies of keywords such as **else** or **do**).
- Functions are declared in the same way as the "old" style of argument declaration in C. (There is no concept of function prototyping.) Thus, the outline of a RATFOR function is

```
function f(x,y)
    real x,y;
{}
```

  (Actually, as far as the FORTRAN code that is generated is concerned, it makes no difference whether the function arguments are declared before or after the opening brace. However, it is useful logically and typographically to set them off as suggested.)
- No **end** statement should be used; that is filled in automatically when FTANGLE processes the closing brace.

- Function values can be returned with a **return** statement of the form '`return` *expression*`;`'.
- Numeric statement labels, if used, should be followed by a colon.
- Comments should be C-style: '`/*...*/`' or '`//...`'.
- When one is inside a recognizable **program**, **subroutine**, **function**, or **blockdata** unit, the built-in function *_ROUTINE* expands to the name of the program unit. (This is true only for RATFOR; `FTANGLE` is not so intelligent in the other languages.)
- The special operators '`++`', '`--`', '`+=`','`-=`', '`*=`', and '`/=`' are allowed in restricted places; see the discussion in the section "Additional features."
- RATFOR character strings must be enclosed in double, not single, quotes (consistent with C, but unlike FORTRAN–77). As in C, single characters enclosed by single quotes are interpreted as character constants and are translated into the ASCII integer equivalent; for example, `'a'` → `97`.

Thus, an example of a RATFOR function is as follows:

```
@r/
@m ERROR call error(_ROUTINE)
@a
real function f(x)
    real x; // We should have $x \ge 0$.
{
real y;

if(x >= 0) y = sqrt(x);
else
    {
    ERROR;
    y = 0.0;
    }

return y;
}
```

RATFOR statements can be easily "stacked". For example, the following is valid RATFOR:

```
if(do_it)
    for(i=1; i<100; i *= 2)
        while(k < i)
            do j=0,k;
                a(i,j,k) = 1.0;
```

Note the simplicity of the construction: no **end** statements are required to terminate the loops.

An important notion in the C language whose essence is retained in RATFOR is the *compound statement*. Here, a compound statement is any group of statements delimited by braces. Compound statements may be used wherever a simple statement may be. Thus, in the above example any or all of the last four lines could be replaced by a compound statement—for example,

```
if(do_it)
    {
```

```
x = dx;
for(...)
    while(...)
        do ...
            {
            a...;
            b...;
            }
x *= alpha;
} // |do_it|
```

Here, as in C, the use of braces creates a quite concise and readable code.

[Some people disagree. Indeed, again quoting Kernighan and Plauger, "It is truly remarkable how much heated debate can result from such trivial questions as whether braces are better than **begin** and **end**..." In this regard, it is very important to observe that in many situations neither braces nor keywords are required to terminate loops; loops whose bodies are simple statements terminate automatically, just as in C. The principle argument that appears to be raised against braces is that if they enclose long blocks of code they can get lost or misplaced. However, note that at the user's discretion braces can be labelled (the previous example shows one possible style); furthermore, such long blocks are strongly discouraged by the philosophy and design of `WEB`. As you know by now, the proper use of named modules leads to a programming style in which no modules need be longer than, ideally, about a dozen lines. For modules that short, it's very difficult to lose braces; rather, the braces tend to enhance the logical structure and we believe that they are more pleasing to, and more readily captured by, the eye than verbose keywords. In any event, our design goal of syntactical consistency with C is obviously better achieved by using braces rather than keywords. Therefore, within the framework of the present philosophy, keywords such as **end do** or **end while** seem to represent a definite step backwords and are not used.]

"It is truly remarkable how much heated debate can result from such trivial questions as whether braces are better than begin and end..."

9.1 Ratfor–77 commands

Here are the RATFOR constructions recognized and expanded by `TANGLE`. In the following, the construction `{...}` stands for either an arbitrary number of statements enclosed by braces (a compound statement), or a simple statement terminated by a semicolon. (Recall that the RATFOR syntax is free-form, so the newlines and spacings in the examples are inserted entirely for readability.)

9.1.1 if

The **if** statement looks exactly like that of C (the **else** clauses are optional):

```
if(condition)
    {...}
else if(another condition)
    {...}
else
    {...}
```

There may be multiple `else if`'s. Functionally, this statement is identical to FORTRAN's **if**...**then**...**else if** ...**then**...**else**...**end if** construction (into which it is expanded), but the keywords have been abolished.

The simple FORTRAN **if**, such as '`if(x < 0.0) x = 0.0;`', is a special case of the general construction. For simplicity, RATFOR also expands this into an '**if**...**then**...**endif**'.

9.1.2 while

The **while** statement is also equivalent to that of C:

```
while(condition)
    {...}
```

The condition is checked. If it is true, the body of the **while** is executed. Then the process repeats. If the condition is false, control passes to the first statement after the body of the **while**. This means that if the condition is false initially, the loop is not executed even once.

9.1.3 for

The **for** statement is equivalent to that of C. It is a souped-up version of the **while** that includes initialization and reinitialization:

```
for(initialization; condition; reinitialization)
    {...}
```

The initialization (which must be a *single* FORTRAN statement; C programmers beware!) is performed. The condition is tested. If it is false, processing terminates. If is is true, then the body of the **for** is executed. At the bottom of the loop, the reinitialization (which must also be a *single* FORTRAN statement) is performed; then the condition is tested again and the processing iterates. For example,

```
for(i=0; i<10; i++) a(i) = i;
```

is equivalent to FORTRAN–90's

```
do i=0,9
    a(i) = i
end do
```

However, it is important to note that for arbitrary bodies even this simple **for** is *not* equivalent to the FORTRAN **do**, since FORTRAN does not allow one to tamper with the loop index within the loop but there is no such restriction in RATFOR or C. Therefore, the construction is translated into an **if** and a **goto**; no attempt is made to optimize it into a **do**. RATFOR programmers working with vectorizing compilers should employ **do**'s instead of simple **for**'s for critical loops.

9.1.4 repeat—until

The **repeat**–**until** construction executes the body before the condition is tested, so it is guaranteed to be executed at least once; it corresponds to the **do**–**while** construction of C:

```
repeat
    {...}
until(condition);
```

Unlike the corresponding loop in C, the **until** clause is optional. If it is omitted, one gets an infinite loop that must be broken out of by a **break** or **goto**. That loop is equivalent to C's `while(1) {...}` or FORTRAN–90's `do {...}`.

9.1.5 **do**

The RATFOR **do** statement is fundamentally FORTRAN's:

```
do index=lower,upper[,increment]
    {...}
```

or

```
do index=lower,upper[,increment];
    simple statement;
```

Functionally, these are equivalent to

```
for(index=lower; index<=upper; index += increment)
    {...}
```

but the **do** may be implemented more efficiently by the compiler. Note the semicolon after *increment* in the second example. This is annoying, as it looks different than the syntax for the **for** statement. Unfortunately, the standard FORTRAN syntax is simply incompatible here with the uniform RATFOR syntax. (If a simple statement follows the **do**, some terminator is required in order to separate the increment from the beginning of the statement. The carriage return at the end of the physical line cannot be used because the syntax is free-form.) If you would like to make things look more symmetric, you could define a `WEB` macro:

```
@m DO(i,min,imax) do i=imin,imax;
```

or, more simply and also more generally,

```
@m DO(i,...)  do i=#.;
```

[If you are using the auto-semi mode (see below), you should omit the semicolon; it will be inserted for you.] Note that the RATFOR syntax makes multiple **do** loops look very clean; neither braces nor **end** keywords are required in the following example:

```
do i=1,10;
do j=1,10;
    a(i,j) = 0.0;
```

9.1.6 **break**, **next**

In the bodies of the above loops (**if** statements are not loops), the special commands **break** and **next** are allowed. The **break** command exits the loop immediately. The **next** command is functionally equivalent to the **continue** statement of C. Following Kernighan and Plauger, "**next** goes to the *condition* part of a **while**, **do**, or **until**, to the top of an infinite **repeat** loop, and to the *reinitialize* part of a **for**." (Note that other statement processors may implement this statement differently.)

Only one level of looping can be broken out of by **break**. This restriction is deliberate, not fundamental. A `break␣`*n* statement, though very feasible to code, is not allowed partly because C does not and partly (equivalently?) because it is felt that its presence would make certain coding errors more likely. To conveniently break out of several levels of looping, use **goto**; this is one of its few legitimate uses [4].

9.1.7 **switch**

Finally, we have the **switch** statement, again functionally equivalent to that of C:

```
switch(expression)
```

```
    {
    case integer_expression:
        .
        .
        break; // Use this to prevent falling through to the next case.
    .
// More case statements.
    .
    default:
        .
        .
        break; // Optional here, but still a good idea.
    }
```

The *expression* is evaluated at run time and converted to an integer. If any of the listed cases correspond to that integer, control is passed there. Otherwise, if the optional **default** statement is present, control is passed there. (It is not necessary that the **default** be last in the **switch**.) Otherwise, the entire **switch** body is skipped.

The arguments of the cases should be or expand to single integers. Lists of cases, as in '`case 2-5:`', are not allowed. If the cases are all pure numbers or macros known at `TANGLE` time, then in certain circumstances the RATFOR–77 **switch** will be implemented using a computed **goto**. This will occur if the list of cases is sufficiently dense, with few gaps, or if the number of cases is sufficiently large, provided the spread in the case values is not too great. Otherwise, in RATFOR–77 the **switch** is expanded into a series of **if** statements. In this case, if the **switch** is lengthy, it will pay to put the most frequently expected case first. In RATFOR–90 the **switch** is translated into a "**select case**" construction.

Use break to terminate a case.

More precisely, the decision of whether to use a computed **goto** is based on three parameters—r, m, and s. Let the number of cases in the **switch** be n and the spread of case values, plus 1, be Δv. The computed **goto** is chosen if both $n > m$ and $\Delta v < s$ or, failing that, if $\Delta v/n \leq r$. By default, $m = 5$, $s = 128$, and $r = 2.0$. If you absolutely don't want the computed **goto** for some reason (maybe because it doesn't vectorize), use $r = 0.0$. These parameters can be set by the user through the command-line option "`-rg`". The format is, for example, "`-rgm5s128r2.0`". The `m`, `s`, and `r` keyletters may appear in any order, or may be absent. *(Now that the style file exists, these parameters should really be defined there. That will be done at some point.)*

In the **switch**, the **break** statements are optional. If they are missing, control just continues to the next case. This is in accord with C's behavior, but *is in disagreement with the original* RATFOR *design.*

Note that a **next** statement appearing inside a **switch** has nothing to do with that **switch**. (The **switch** is not a loop.) Rather, it is related to whatever loop encloses the **switch**.

Here is an example of a **switch**. It assumes that the single character c has been read. It should be either '`y`' or '`n`', in either upper or lower case. If it is '`y`', the *execute* subroutine is called; if it is '`n`', nothing is done; if it is anything else, an error routine is called. Note how the lower-case cases fall through to the upper-case ones, and how the **break** statement is used to terminate the processing of a group of cases.

```
switch(ichar(c)) // Use intrinsic function to return integer value.
    {
    case 'y':
    case 'Y':
        call execute;
```

```
            break;
        case 'n':
        case 'N':
            break;
        default:
            call error(c);
            break;
        }
```

♣ 9.2 Ratfor–90 commands

In RATFOR–90, additional constructions are supported. In general, where FORTRAN–90 has a compound construction that ends with an **end** statement, such as **type person**...**end type person**, RATFOR abolishes the **end** statement and encloses the statements with braces. It will expand the construction into valid FORTRAN–90, including the optional labelled form of the **end** statement, such as **end module**. Furthermore, FORTRAN–90 allows optional symbolic labels on such compound constructions. RATFOR–90 permits these as well, as in

```
test:  if(x)
    {...}
```

This construction will be expanded into code that ends with "**end if test**". Such symbolic labels should *not* be declared with the automatic statement numbering facility; that is, do *not* say "`@m test #:0`".

9.2.1 module

The **module** statement is analogous to the **class** statement of C++:

```
module module_name
    {...}
```

As an example,

```
module work_arrays
    {
    integer n;
    real, allocatable, save ::  A(:), B(:,:), C(:,:,:);
    }
```

9.2.2 type

The **type** statement is analogous to the **struct** statement of C:

```
type type_name
    {...};
```

As an example,

```
type line
    {
    real, dimension(2,2) ::  coord; // $x_1$, $y_1$, $x_2$, $y_2$.
    real ::  width; // Line width in centimeters.
    integer ::  pattern; // 1 for solid, 2 for dash, 3 for dot.
```

```
};
```

Note the terminating semicolon, which is consistent with the **struct** statement of C and C++.

9.2.3 **interface**

The **interface** statement is FORTRAN–90's way of overloading operators or procedures. The corresponding RATFOR statement has one of the three following forms:

```
interface procedure_name
    {...}

interface operator(operator)
    {...}

interface assignment(assignment operator)
    {...}
```

As an example,

```
interface operator(*)
    {
    function boolean_and(b1,b2)
        {
        logical ::  boolean_and(size(b1));
        logical, intent(in) ::  b1(:), b2(size(b1));
        }
    }
```

9.2.4 **where**

The **where** statement is supported (the **else** clause is optional):

```
where(condition)
    {...}
else
    {...}
```

9.2.5 **contains**, **private**, **sequence**

The keywords **contains**, **private**, and **sequence** should be followed by colons; for example,

```
subroutine outer
    {
    ...
contains:
    subroutine inner(b)
        {...}
    }
```

♣ 9.3 Additional features of RATFOR

A few additional topics are useful to discuss.

♣ 9.3.1 RATFOR's automatic comments

As FWEB translates RATFOR statements, it typically writes comment lines to the output file to help one correlate the source statement with the translated output. (Look at some sample output for examples.) If one wishes to suppress such comments, he may use the command-line option '-k'. See the detailed description under 'Command-line options'.

♣ 9.3.2 Automatic insertion material

It is often desirable to include common material at the beginning of each program unit, such as the phrase "implicit none". Since RATFOR is aware when a program unit begins, it is feasible to automate such insertions. The material to be inserted is indicated by a special notation that extends the syntax of a WEB macro definition, as follows:

@m[*ctrl-list*] *macroname macro text*

Here the characters in the *ctrl-list* between the square brackets signify for which kind of program unit the material is to be inserted; they may be one or more of the following:

b — **block data**
f — **function**
i — **interface**
m — **module**
p — **program**
s — **subroutine**
* — All of the above.

Thus, if you say

```
@m[pfs] AUTO implicit none
```

the text "implicit none will be inserted on the lines immediately following every **program**, **function**, and **subroutine** declaration. (There is at present no way to prevent this insertion for a particular program unit; it is either all or none.)

At present, you cannot stack automatic material for a particular program unit. Thus, after the statements

```
@m[pfs] AUTO1 implicit none
@m[f] AUTO2 implicit integer[i-n]
```

functions will begin with the text "implicit integer[i-n]" while the main program and subroutines will begin with "implicit none." (In the future, such stacking may be allowed.)

Note that there is nothing special about the macro names used in these extended definitions. Thus, in the above example one could have replaced *AUTO1* by *X* and *AUTO2* by *Y* and achieved precisely the same effect.

♣ 9.3.3 Semicolons

We now return to the issue of semicolons. *It is highly recommended that the complete free-form syntax, with semicolons as statement terminators, be used for all new code.* However, it is recognized that an intermediate step may expedite conversion of existing FORTRAN codes. One can initiate an *auto-semi* mode with the command-line option '-;'. In this mode, the expected RATFOR syntax is midway between FORTRAN's and C's.

The auto-semi mode is almost free-form syntax, except that carriage returns end statements when the line is not "obviously continued". By definition, a statement is obviously continued if it ends with any of the following characters between double quotes: "`+-*={}^&|(:><,`". In this case, `WEB` just goes on to the next line and continues to read. Otherwise, the input driver terminates the statement with a semicolon, then ships it off to the innards of `WEB`. (This clarifies why the character '`}`' is part of the previous list: "Obviously continued" really means "don't end this line with a semicolon.") Continuation can be forced by ending a line with a backslash; the backslash is discarded. (For compatibility with previous RATFOR implementations, a trailing underscore will also continue a line except when it is part of an identifier; however, its use is definitely not recommended.) One implication of these rules is that in the auto-semi mode no semicolon should end the first line of a **do**; say

```
do i=1,n
    a(i) = i
```

(Neither statement is obviously continued; the input driver will supply semicolons for both.)

According to these rules, lines such as '`for(...)`' will be terminated by a semicolon. In free-form syntax, such a semicolon would be incorrectly interpreted as a null statement. In the auto-semi mode, however, a special check is made for such a supplied semicolon, which is discarded if present. Thus, the RATFOR syntax behaves consistently in both syntax modes.

In the auto-semi mode, an additional commenting style is allowed: anything between '`#`' and the end of the line denotes a comment. This is changed by the input driver into a standard C-style comment. However, it is recommended that this RATFOR commenting style be avoided; *it will probably be allowed to die with the next major release.*

♣♣ 9.3.4 `FWEB` *sans* Ratfor

RATFOR is a self-contained subset of `FWEB`. It is possible to create `FWEB` processors that do not contain most of the code associated with RATFOR and are therefore smaller. This may be relevant for users of personal computers. See the installation notes in `INSTALL_FWEB.tex` for more information.

Once again, it is recommended that if you are a FORTRAN programmer you seriously consider RATFOR. It will make your life simpler and your documentation superior.

The RATFOR commands are summarized in Appendix L.

10. ADDITIONAL LANGUAGES

In addition to the principal supported languages of C, C++, FORTRAN, and RATFOR, several other languages are supported at various levels of experimental development:

10.1 TEX mode

As an experiment, `FWEB` offers restricted support for tangling and weaving TEX code. This provides a superior way of maintaining and documenting TEX input files, for example macro packages such as `fwebmac`. One selects the TEX language with the command '`@Lx`'. `FTANGLE` will write its TEX-language output into a file with the extension `.sty`; that can be overridden by the style-file option "`suffix.TEX`". `FWEAVE` will, as usual, create a file with the extension `.tex`. As an example, `fwebmac` itself is now maintained with `FWEB`. `FTANGLE` has been used to create the "executable" (i.e., TEX-compatible) file `fwebmac.sty` from the

"`FWEB` offers restricted support for tangling and weaving TEX code."

master file `fwebmac.web`. This explains why `FWEAVE`'s `tex` output files begin with the command "`\input fwebmac.sty`" instead of just "`\input fwebmac`". The latter would incorrectly read in `fwebmac.tex`, which is `FWEAVE`'s output intended for typesetting.

In TEX mode, TEX-like comments (begun with '`%`', or more precisely begun with any character whose category code is 14) are displayed in the standard C-style format, except that if the last character on the line is '`%`' it is displayed as is. If several adjacent lines consist solely of TEX-like comments, they are concatenated into one long comment.

Actual spaces in the source are displayed as '␣', with two exceptions. First, tabs in the source are turned into invisible spaces. Second, spaces after multi-character control sequences are also printed as invisible spaces. If this is not done, the output tends to become extremely dense with the '␣' symbol.

Note that a full-featured implementation of `WEB` processing for TEX is quite difficult because TEX can, for example, change category codes on the fly from within very complicated macro constructions. A truly general `WEB` for TEX should probably be completely integrated into TEX itself. This daunting possibility is far beyond the scope of the present `FWEB` project. However, it is possible to give `FWEB` some help and thus deal with a variety of circumstances. In particular, one can force `FWEB` to change the category code of a character. When `FWEB` starts up, it has assigned default category codes to each ASCII character—for example, the default category code of '\' is 0; that of '`A`' is 11; that of '`%`' is 14. To change the category code, say in the definition section

`@f ‘` *TEXchar new_cat_code*

where here, as when changing category codes in TEX, *TEXchar* has one of the forms '*c*', '*c*', or '`^^`*c*', *c* being a visible ASCII character. For example, the command

```
@f ‘!  14
```

will make the exclamation point also function as a comment character.

At this early stage of development, the *only* thing that is guaranteed is that `fwebmac.sty` will be created correctly from `fwebmac.web`. Feel free to experiment with `FWEB`'s TEX language, but don't expect perfection yet. However, suggestions are welcome.

A sample of woven output from the TEX mode may be found in `fwebmac.tex`, which is typeset in Appendix F.

10.2 MAKE mode

Someday `FWEB` may also understand the syntax of UNIX `make` files.

11. CONTROL CODES

We have seen several magic uses of '`@`' signs in `WEB` files, and it is time to make a systematic study of these special features. A `WEB` *control code* is a two-character combination of which the first is '`@`'. (The only exceptions are the three-character combinations '`@/*`' and '`@//`'.)

Here is a complete list of the legal control codes. (Some of these codes can be changed by the user, although this is not recommended; see the section below on the style file.) The abbreviations L, T, C, M, Co, S, and/or H following each code indicate whether or not that code is allowable in limbo (L), in TEX text (T), in code text (C), in module names (M), in comments (Co), in strings (S), and/or in change files (H). A bar over such a letter means that the control code terminates the present part of the `WEB` file;

for example, $\overline{L}$ means that this control code ends the limbo material before the first module. (A shorter summary of the control codes is presented in Appendix L.)

11.1 @@ (the character '@')

$[Co, L, M, C, S, T]$ A double @ denotes the **single character '@'**. This is the only control code that is legal in comments and in strings. For example, to get the output '"Is there a missing @z?"', one must type in the WEB source file '"Is there a missing @@z?"'. This is also one of the few control codes that is legal in limbo (the only others being the language commands '@c', '@r', '@n', and '@Ll', and the ignorable commentary commands '@z' and '@x'.)

11.2 @| (literal vertical bar [TEX text])

$[Co, T]$ In TEX text, this affords a way of inserting the **vertical bar** '|'. This is particularly useful within LATEX's verbatim environment. For example,

```
\begin{verbatim}
Consider the expression @|x != y@|.
\end{verbatim}
```

Without the @s, WEAVE would translate the bars and the enclosed material into the TEX macros appropriate for code mode.

In code text, this command is instead an optional line break in an expression. See the discussion below.

11.3 @␣ (begin unstarred module)

$[\overline{L}, \overline{C}, \overline{T}]$ This denotes the beginning of a **new (unstarred) module**. A tab mark or end-of-line (carriage return) is equivalent to a space when it follows an @ sign.

11.4 @* (begin a starred module)

$[\overline{L}, \overline{C}, \overline{T}]$ This denotes the beginning of a **new starred module**, i.e., a module that begins a new major group. The title of the new group should appear after the '@*', followed by a period. This title will be entered in the table of contents. As explained above, TEX control sequences should be avoided in such titles unless they are quite simple. When WEAVE and TANGLE read an '@*', they print to the terminal an asterisk followed by the current module number, so that the user can see some indication of progress. The very first module should be starred.

If the group title is immediately followed by a non-negative integer n, the title is considered to be of level n, where $n = 0$ corresponds to a major section, $n = 1$ corresponds to a primary subsection, and so on. The fwebmac macros can be defined to treat these levels in different ways—for example, the appearance of a subsection entry in the table of contents can be modified by appropriate macro definitions.

11.5 @A (begin code part of unnamed module)

$[\overline{C}, \overline{T}]$ **The code part of an unnamed module** begins with '@A' (or '@a'; see below). *(This is a change from the original* WEB*, which used '@p' (for Pascal) and from* CWEB*, which used '@c'. In designing* FWEB*, it was felt most logical to reserve these commands for actually switching into a particular language; this is an operation distinct from beginning the unnamed module.)* This causes TANGLE to append the following code to the initial program text T_0 as explained above. The WEAVE processor does not cause an '@A' to appear explicitly in the TEX output, so if you are creating a WEB file based on a TEX-printed WEB documentation you have to remember to insert '@A' in the appropriate places of the unnamed modules.

11.6 @a (begin code part of unnamed module; mark first non-reserved word)

$[\overline{C}, \overline{T}]$ Equivalent to "@A@[". That is, the first non-reserved word following the '@a' will be marked as defined in this module. This convention helps circumvent the forward-referencing problem; see the discussion about "Forward references" below.

11.7 @b (insert breakpoint command)

$[C]$ When the debugging mode is turned on (which means when the _BP macro has been defined from the command line; see discussion below), the '@b' command means **insert a breakpoint command**. When debugging is off, this command is ignored.

11.8 @c (set language to C)

$[L, C, T, H]$ The '@c' command means **set the current language to C**. *It does not mean begin the unnamed module, as it did in Levy's* CWEB. (See the detailed discussion of language commands for more information.)

11.9 @c++ (set language to C++)

$[L, C, T, H]$ The '@c++' command means **set the current language to C++**.

11.10 @D (define outer macro)

$[\overline{C}, \overline{T}]$ **Definitions of outer macros** begin with '@D' (or '@d'; see below), followed by the code text for the macro syntax appropriate for the language currently in force. These definitions must be made in the definition part, which consists of any number of macro definitions (beginning with '@d' or '@m'), format definitions (beginning with '@f'), preprocessor commands (beginning with '@#'), limbo text definitions (beginning with '@l'), operator overloading instructions (beginning with '@v'), identifier overloading instructions (beginning with '@W'), and language commands, intermixed in any order.

For FORTRAN, the syntax should be that for the m4 preprocessor; for C, it should be that of the C preprocessor. For RATFOR, outer macros should never be used, since TANGLE's macro-processing capabilities are intended to provide a self-contained means of getting directly from the WEB source to compilable FORTRAN. Outer macros are simply copied to the beginning of the appropriate output file. If the text of the macro contains an identifier that is a WEB macro, *that macro will be expanded*.

The companion command '@u' undefines an outer macro.

11.11 @d (define outer macro; mark macro name defined)

$[\overline{C}, \overline{T}]$ Equivalent to "@D@[". See discussion about "Forward references" below.

11.12 @f (format identifier)

$[\overline{C}, \overline{T}]$ **Format definitions** begin with '@f'; they cause WEAVE to treat identifiers or module names in a special way when they appear in code text. The general form of a format definition is '@f' l r, followed by an optional comment, where l is an identifier or a module name and r is an identifier; WEAVE will subsequently treat l as it currently treats r. The formatting is *language-specific*; it only applies to identifiers used in the language in force at the point the format statement is encountered. This feature allows a WEB programmer to invent new reserved words and/or to unreserve some reserved identifiers. Note that module names can be formatted. By default, module names are interpreted as expressions. However, sometimes you use them in

other contexts, such as replacing a bunch of type specifications. In this situation, you should say something like "`@f @<Common blocks@> common`".

An extension of the '`@f`' command is used when the language is `TEX` to change the category code of a character. The format is ''`@f` ` *TEXchar new_cat_code*'. See the discussion of TEX mode for more details.

11.13 @i (include a file)

[*webfile*] The web file itself can be a combination of several files. When `WEAVE` or `TANGLE` are reading a file and encounter the control code '`@i`' at the beginning of a line, they interrupt their reading and start reading the file named after the '`@i`', much as the C preprocessor does when it encounters an **#include** line. After the included file is done, they go back to the next line of the original file. The file name following '`@i`' can be surrounded by double quotes or not; it should be made up of visible ASCII characters only, not including double quotes. Include files can nest, with level 0 being the primary level associated with the source `WEB` file. Optionally, a second file name also may be given. Just as on the command line, this names the change file associated with the new include file. This name is in effect for all higher levels of nesting, but is forgotten when the include file ends and control reverts to the next lower level. If no change file is specified at any level, the one in effect at the time of the include continues to be used. Automatic file-name completion is done when the command-line option '`-e`' is in effect; otherwise, you must give the complete name of the file, including extensions such as '`.web`'.

`FWEAVE` will print the name of the current include file at the beginning of each section. However, note that there is no need that a file included by '`@i`' consist of a complete module. Include files may be included anywhere, in either the TEX part, the definition part, and/or the code part; they may, in principle, consist of arbitrary fragments of code. Whenever possible, however, it is best to stick to complete modules for included files.

By default, include files are searched for in the current directory. However, if the environment variable *FWEB_INCLUDES* is defined, then its contents are interpreted as a colon-delimited list of paths to be searched for the include file. (The same format as UNIX' *PATH* variable is used.) In addition, whether or not that variable is defined, one can append to the include path list by means of the '`-I`' command-line option.

(The following is rendered obsolete by the previous paragraph; the feature will probably be deleted in a future release.) File names in '`@i`' commands may begin with a prefix followed by a colon, as in "`LN:file_name`". The intention is that `LN` behaves like a logical name under VMS. In fact, under VMS `LN` is just left alone. However, under UNIX `LN` is assumed to be an environment variable and is expanded. Thus, if one says "`setenv LN /fweb`" then "`LN:file_name`" will expand to "`/fweb/file_name`" (note that the last slash was inserted automatically). This mechanism is intended to enhance portability of `WEB` sources between various operating systems.

11.14 @I (optionally include a file)

[*webfile*] This command is like '`@i`', except that in special cases `FWEAVE` will not process it completely. The special cases are when the command-line options '`-i`' or '`-i!`' are used. In particular, when '`-i`' is in effect, files included via '`@I`' will be processed but not printed in the woven output. (This helps save trees.) See the description of the command-line options below for more information about this *experimental* feature.

11.15 @L (set language)

[L, C, T, H] The command '`@L`l' sets the language to l, where $l \in \{\texttt{c}, \texttt{n}, \texttt{r}, \texttt{x}\}$. Optional arguments to this command enable one to invoke C++, FORTRAN–90, or RATFOR–90. See the detailed discussion of languages

above.

11.16 @l (specify limbo text)

$[\overline{C}, \overline{T}]$ This command specifies **limbo text**. It must be followed by a double-quoted string, in which special characters are escaped just as in C. The contents of that string are written out verbatim by FWEAVE just before the limbo section is copied to the output. Thus, if there was just one limbo text command of the form

```
@l "\\def\\greeting{Hello}\n\\def\\done{Goodbye}"
```

the output tex file would begin with the lines

```
\input fwebmac.sty
% --- Limbo text definitions from @l ---
\def\greeting{Hello}
\def\done{Goodbye}
```

Note the use of C-like escape sequences such as '\\' or '\n'.

11.17 @M (define a WEB macro)

$[C, \overline{T}]$ The '@M' (or '@m'; see below) command denotes a WEB **macro**. WEB macros have exactly the same syntax as C macros (including arguments), but they are expanded when FTANGLE outputs the code. Just as in C, the construction is expanded again and again until nothing remains to be expanded. [The ANSI C constructions '#' ("stringize") and '##' ("token paste") are supported, as are certain extensions such as the ability to handle variable numbers of arguments.] On input, the preprocessor understands all macros that have been defined up to the current point in the file, so they can be used in statements such as '@#if(MACRO) *block_of_code* @#endif' to selectively reject or retain fragments of code. Usually, @m commands should appear in the definition section, but they are also permitted in the code section, where they are called *deferred macros*. See the section on 'Macros' for more details.

11.18 @m (define a WEB macro; mark macro name defined)

$[C, \overline{T}]$ Equivalent to "@M@[". See the discussion about "Forward references" below.

11.19 @n (set language to FORTRAN)

$[L, C, T, H]$ The '@n' command means **set the current language to** FORTRAN. (See the more detailed discussion of language commands for more information.) One may question the choice of symbol here. Unfortunately, 'f' was already in use, denoting formatting; hence, we use the *last* letter of the language name: fortraN, C, ratfoR. With no argument, FORTRAN–77 is selected. The command '@n9' selects FORTRAN–90.

11.20 @O (open new output file with global scope)

$[C]$ The '@O' command **changes the name of the output file** for tangled code, during the output in phase 2. At present, it may appear only in the code section. White space is skipped after the '@O'. The next run of non-white characters is interpreted as a complete file name, including extension. Any remaining text on the line is skipped. The upper-case form of this command has *global* scope—that is, the name change remains in effect until the next '@O' command, if any. (The lower-case form has local scope and is probably more useful; see below.) At present, this command has no effect for FWEAVE. The purpose of this command is to allow very large codes to be tangled into several output files in cases where the compiler may have limitations on the size of the source file it can process.

Note that since '@d' commands are collected during phase 1, they are at present oblivious to any '@O' or '@o' commands; '@d' definitions will always be written into the first file open for a particular language. (In the future, this may be generalized.)

11.21 @o (open new output file with local scope)

[*C*] The '@o' command **changes the name of the output file** for tangled code, during the output in phase 2. The rules are the same as for the '@O' command described above, except that '@o' has *local* scope. That is, the command behaves in the same way as does a local language change: output is diverted to the new file *for the duration of the current section only.* When the next section begins, output reverts to the global output file, as known at the beginning of the first module. The purpose of this command is to generate several different kinds of files from a single source file. For example, let the WEB file be called test.web. Then one could say

```
@c
@ Demonstration of the \.{@o} command.
@a
@o test.h
int i; // This code goes into \.{test.h}.
@ The next code goes into \.{test.c}.
@a
main()
{}
```

11.22 @r (set language to RATFOR)

[*L*, *C*, *T*, *H*] The '@r' command means **set the current language to** RATFOR. With no argument, RATFOR–77 is selected. The command '@r9' selects RATFOR–90. (See the detailed discussion of language commands for more information.)

11.23 @u (undefine an outer macro)

[$\overline{C}$, $\overline{T}$] This command **undefines an outer macro**. It should be used in the definition part only. See the discussion of '@d' for more discussion.

11.24 @v (overload an operator)

[$\overline{C}$, $\overline{T}$] The '@v' command allows one to give FWEAVE information about **operator overloading**, an important feature of both C++ and FORTRAN–90. The syntax of this command is "@v *new_op* "*replacement* *TEX␣text*" *old_op*" and is described in detail in the section on "Operator overloading" above.

11.25 @W (overload an identifier)

[$\overline{C}$, $\overline{T}$] The '@W' command allows one to change the appearance of identifiers. This command has several variants: "@W *identifier* "*replacement* *TEX␣text*"", "@W *identifier* *newmacro*", and "@W *identifier* .". These are described in detail in the section on "Identifier overloading" above.

11.26 @x (terminate commentary section; begin old material in change file)

[*L*, *H*] This terminates the opening commentary section of a file. See the discussion of '@z' below.

This command is also used in change files to begin the old material; see the separate discussion of change files.

11.27 @y (terminate old material in change file)

[*H*] This is used in change files to terminate the old material and to begin the new material. See the separate discussion of change files.

11.28 @z (begin commentary section; end changed material)

[*L*, *H*] If a file begins with '@z' in its very first two positions, everything up to and including a line beginning with '@x' is skipped. This feature allows one to insert commentary such as date, author, etc. that will not be printed or otherwise processed.

This command is also used in change files to end the changed material; see the separate discussion of change files.

11.29 @' (convert character to ASCII integer)

[*C*] A construction of the form "@'*c*'" **converts the single character** *c* **to an integer representing its ASCII value**. The character can be represented in any of the standard forms: for example 'a', '\141', and '\x61' are all equivalent. In C and C++, the character is converted into octal—e.g., @'a' → 0141. In the FORTRAN-like languages it is converted into an integer of base 10—e.g., @'a' → 97.

11.30 @" (convert string to ASCII)

[*C*] A construction of the form "@"$c_1c_2\ldots c_n$"" **converts each character in the string to an integer representing its ASCII value**, returning a construction appropriate for initializations in the current language. In C or C++, the result is a string with the characters replaced by the appropriate octal values—e.g., @"a\376b\n" → "\141\376\142\12". For FORTRAN-like languages, the implementation is experimental and subject to change. Presently, it works as follows. First, the '@' is stripped away, leaving the string. If the value of the style-file field ASCII_fcn is the null string, then the string is unchanged. If ASCII_fcn is not null, this field is used as the name of a function to be called with the ASCII string as argument. Thus, the default value of ASCII_fcn is "ASCIIstr". Unless this is changed in the style file, then in FORTRAN the command "@"a\376b\n"" will tangle to "ASCIIstr('a\376b\n')".

11.31 @[(mark next identifier as defined here)

[*C*, *H*] This command has two uses. First, if it appears in a change file in the first two positions of the line, it signifies a shift into code mode; see the separate discussion of change files.

Otherwise, it says that the next *non-reserved* identifier should be marked as being defined in this section. Identifiers thus marked are subscripted in the woven output with the number of the section in which they were defined. Usually, the identifier is a function name. When FWEAVE's syntax analyzer recognizes a function in phase 2, it marks the function name automatically. However, only subsequent references to that function are marked. Additional mechanisms exist to handle the problem of forward referencing. For example, the commands '@a', '@d', and '@m' issue implicit '@['s. See the more detailed discussion about "Forward references to identifiers" in the section on "Additional Features" below.

11.32 @] (shift out of code mode)

[*H*] This command also has two uses. If it appears in a change file in the first two positions of the line, it signifies a shift out of code mode; see the separate discussion of change files.

(Otherwise, its use is experimental and not supported yet.)

11.33 @‘ (*reserved*)

[C, H] *This command is experimental. Please do not use.*

11.34 @< (begin a module name)

[$C, \overline{T}$] A **module name** begins with ‘@<’ followed by TEX text followed by ‘@>’; the TEX text should not contain any WEB control sequences except ‘@@’, unless these control sequences appear in code text that is delimited by |...|. The module name may be abbreviated, after its first appearance in a WEB file, by giving any unique prefix followed by ‘...’, where the three dots immediately precede the closing ‘@>’. No module name should be a prefix of another. Module names may not appear in the definition part of a module (since the appearance of a module name ends the definition part and begins the code part). There are two exceptions to this last rule: first, a module name may appear immediately after an ‘@f’ command, thereby allowing module names to be formatted. Second, you may use the construction #<...@> in a WEB macro definition to stand for a module name.

11.35 @/* (begin long verbatim comment)

[C] This denotes a long **verbatim comment** (terminated by ‘*/’). The idea is that while TANGLE generally throws away all comments, stripping down the output to a bare minimum, retaining some comments in the output may be helpful for debugging purposes. Therefore, one is allowed to preface ordinary C-style comments with an ‘@’; such comments will be passed through to the output. For C, the comment is literally just passed along; nothing else happens. For the other languages, care is taken to generate a valid comment line. This is particularly annoying for FORTRAN–77, which does not have the notion of a trailing comment. There, trailing verbatim comments are moved to the next line (essentially the inverse of what the input driver does). Incidentally, module numbers are automatically inserted as verbatim comments into the program, in order to help correlate the outputs of WEAVE and TANGLE (see Appendix C). To make all comments verbatim, use the command-line option ‘-v’ or put the command ‘+v’ into your .fweb file.

11.36 @// (begin short verbatim comment)

[C] A short **verbatim comment** (terminated by an end-of-line).

11.37 @% (ignorable comment)

[L, T, C] **Ignorable comments** are FWEB’s analogs to TEX comments: Everything from the ‘@%’ command to and including the next newline is ignored, for both FWEAVE and FTANGLE. Thus, this comment does not appear as part of the FWEAVE’s typesetting, in either the TEX, definition, or code part. It can be useful for hiding text that might be used in conjunction with special editors. It can also be used to suppress unwanted newlines that sneak into module definitions. Those generally begin and end with a newline, as indicated:

```
@<...@>=newline
    .
    .
last line of modulenewline
```

Especially in FORTRAN-77, those newlines may prevent the use of the module name as a component of a longer line, for example as an argument list of a function:

```
function f(@<Args@>)
```

To ensure that this example tangles correctly, say

```
@<Args@>=@%
a,b,c@%
```

11.38 @? (compiler directive)

[*C*] This command begins a one-line **compiler directive**. FTANGLE constructs an output line by first outputting the contents of the style field `cdir_start.`l, where l is the identifier character for the current language. Then it copies everything between the '`@!`' and the next newline. Consider the C language as an example. By default, `cdir_start.C` is set to `"#pragma␣"`. Then the command "`@?help`" is output as "`#pragma␣help`".

11.39 @! (compiler directive)

[*C*] This is an obsolete form of '`@?`'. It differs in the way that the text following the command is processed. For '`@!`', that text is just treated as one big string. For '`@?`', the text is broken up into tokens. The advantage of this is that argument substitution can occur when '`@?`' is used in a WEB macro definition.

11.40 @((begin meta-comment)

[*C*] The beginning of a "**meta-comment**," i.e., commented-out code-section material that is supposed to appear in the output file, is indicated by '`@(`' in the WEB file. (Place this command in column 1.) **Note that the behavior of this command has been changed beginning with version 1.30.** For both processors, this command provides a verbatim channel directly to the output; it behaves as a generalized comment. Its operation is controlled by several style-file parameters. For FTANGLE, these are `meta.top.`l, `meta.prefix.`l, and `meta.bottom.`l, where l is a language symbol such as '`N`' for FORTRAN. By default, these are defined to generate valid comments for each particular language. For example, the C defaults are `meta.top.C = "/*"`, `meta.bottom.C = "*/"`, and `meta.prefix.C = ""`. If they are defined in the style file, then the body of text between '`@(`' and '`@)`' is preceded by the contents of `meta.top` and followed by the contents of `meta.bottom`. Each line of the body of text will begin with the contents of `meta.prefix`. Experiment with an example to see just what happens. For FWEAVE, the parameters are `meta.begin` and `meta.end`; these are defined by default to set up appropriate verbatim environments to surround the body of text. For example, in LaTeX (when the '`-PL`' option is used) they bracket the text with `\begin{verbatim}` and `\end{verbatim}`.

This command may be useful when converting pre-existing FORTRAN codes with comments designed without regard for TeX's conventions. The verbatim environments will not destroy vertical alignment, nor will they complain about the use of special symbols such as '`$`'. However, please use this command only as a last resort. For pretty alignments, use TeX's alignment features (e.g., `\halign`) inside a standard WEB command. If you want to temporarily comment out a section of code, it is best to use the preprocessor commands '`@#if`' and '`@#endif`'.

11.41 @) (end meta-comment)

[*C*] The end of a "meta-comment" is indicated by '`@)`'. (Place this command in column 1.)

11.42 @{ (suppress breakpoint comment)

[*C*] In debugging mode (which means when the `_BP` macro has been defined from the command line; see discussion below), this command is used to replace the opening brace of a module, and means to suppress the default insertion of a breakpoint command just after that brace. This is necessary in C code when the brace is followed by declaration statements. To insert the breakpoint later, use '`@b`' at the desired location.

11.43 @& (join two items)

[*C*] The @& [*join*] **operation** causes whatever is on its left to be adjacent to whatever is on its right, in the code output. No spaces or line breaks will separate these two items. However, the thing on the left should not be a semicolon, since a line break might occur after a semicolon. (See also the '@+' command.)

The *join* operation should be distinguished from the macro processor's *paste* operation (##). Pasting abuts two things, then *retokenizes* the result to obtain a single new identifier. Joining simply prevents a space from appearing between two objects on output. The difference can be very significant in macro processing, where the new identifier that results from the paste might be a macro subject to further expansion. In general, the join operation should not be used within WEB macro definitions.

11.44 @^ (index entry in Roman type)

[*C*, *T*] The "**control text**" that follows, namely everything up to the next '@>', will be entered into the index together with the identifiers of the program; this text will appear in Roman type. For example, to put the phrase "system dependencies" into the index, you can type '@^system dependencies@>' in each module that you want to index as system-dependent. A control text must end on the same line of the WEB file as it began. Furthermore, no WEB control sequences are allowed in a control text, not even '@@'. (If you need an '@' sign you can get around this restriction by typing '\AT!'.)

11.45 @. (index entry in typewriter type)

[*C*, *T*] The "**control text**" that follows will be entered into the index in typewriter type; see the rules for '@^', which is analogous.

11.46 @9 (user-defined index entry)

[*C*, *T*] The "**control text**" that follows will be entered into the index in a format controlled by the TEX macro '\9', which the user should define as desired; see the rules for '@^', which is analogous. The reference would be made as follows: @9wildcard reference@>. If you wanted your wildcard reference to appear in sans serif type, you would define \9 like this: \def\9#1{{\tenss#1}}. (In earlier versions of WEB, this command was called '@:'. However, that is now the pseudo-colon, and although @9 may not look as pretty, it should be easier to remember.)

11.47 @t (format control text)

[*C*] The "**control text**" that follows, up to the next '@>', will be put into a TEX \hbox and formatted along with the neighboring program. This text is ignored by TANGLE, but it can be used for various purposes within WEAVE. For example, you can make comments that mix code and classical mathematics, as in '$size < 2^{15}$', by typing '|size < @t2^{15}@>|'. A control text must end on the same line of the WEB file as it began, and it may not contain any WEB control codes.

11.48 @= (verbatim control text)

[*C*] The "**control text**" that follows, up to the next '@>', will be **passed verbatim** to the program.

11.49 @_ (underline index entry)

[C, T] The module number in an index entry will be underlined if '@_' immediately precedes the identifier or control text being indexed. This convention is used to distinguish the modules where an identifier is defined, or where it is explained in some special way, from the modules where it is used. A reserved word or an identifier of length one will not be indexed except for underlined entries. An '@_' is implicitly inserted by WEAVE when an identifier is being defined or declared—for example, when WEAVE recognizes a function, or in type specification statements such as **integer** i, j. It is also inserted implicitly just after @d, @m, and @f. Because of these implicit insertions, one should rarely need to use @_ explicitly.

11.50 @- (delete index entry)

[C] An identifier that immediately follows an '@-' will not appear in the index.

11.51 @, (insert a thin space)

[C] This control code inserts a **thin space** in WEAVE's output; it is ignored by TANGLE. Sometimes you need this extra space if you are using macros in an unusual way, e.g., if two identifiers are adjacent.

11.52 @/ (line break)

[C] This control code causes a **line break** to occur within a program formatted by WEAVE; it is ignored by TANGLE. Line breaks are chosen automatically by TEX according to a scheme that works most of the time, but sometimes you will prefer to force a line break so that the program is segmented according to logical rather than visual criteria. Caution: '@/' should be used only after statements or clauses, not in the middle of an expression; use '@|' in the middle of expressions, in order to keep WEAVE's parser happy.

11.53 @| (optional line break in expression [code text])

[C] In code text, this control code specifies an **optional line break** in the midst of an expression. For example, if you have a long condition between **if** and **then**, or a long expression on the right-hand side of an assignment statement, you can use '@|' to specify breakpoints more logical than the ones that TEX might choose on visual grounds.

In TEX text, this command is instead a literal vertical bar. See the discussion above.

11.54 @# (line break plus white space)

[C] This control code forces a **line break**, like '@/' does, and it also causes a little extra white space to appear between the lines at this break.

WEB automatically inserts this extra space between functions, between external declarations and functions, and between declarations and statements within a function. Furthermore, *this command is essentially equivalent to a blank line in your source file*, unlike the original convention of WEB. You should have to use this command very rarely, since blank lines are so much easier and prettier looking in the source file. It is good practice to insert blank lines liberally in your source file anyway for readability. For example, it usually looks best if you set off things like **if** statements by blank lines.

Note that '@#' is also the prefix for the preprocessor commands. No confusion arises, however; WEB interprets things like @#endif as single units.

11.55 @+ (cancel line break)

[*C*] This control code cancels a line break that might otherwise be inserted by WEAVE. For example, you might use this to force two statements to appear on the same line, as in the FORTRAN line

```
a = b; @+ c = d
```

It is ignored by TANGLE.

11.56 @; (pseudo-semicolon)

[*C*] This control code, sometimes called a *pseudo-semi*, is treated like a semicolon, for formatting purposes, except that it is invisible. You should use it, for example, after a module name when the code text represented by that module name represents a complete statement, since in the absence of an explicit format statement WEAVE thinks module names are just expressions.

Some examples of the use of pseudo-semis are in order. The principal use of them is after module names in certain situations. Consider the following situation:

```
@c
@ Here is an example of the use of pseudo-semicolons.
@A
if(debug) @<Test@>@; // Use pseudo-semi here because @<Test@> ends with '}'.
else @<Compute@>; // Use semi here because @<Compute@> does NOT end with ';'.

@
@<Test@>=
{...}

@
@<Comp...@>=
x = 1.0@; // Ends with a pseudo-semi so WEAVE will format it properly.
```

The goal is to simultaneously fool WEAVE into thinking that you have written down a complete statement and to avoid introducing a spurious semicolon where it doesn't belong. In particular, it would be wrong (and would lead to a compiler error about an unmatched **else**) to follow @<Test@> with a semicolon instead of a pseudo-semi, because the definition of @<Test@> is already syntactically complete.

You may also need pseudo-semicolons to terminate macro definitions that are not complete statements (but become one when they are terminated by a semicolon in actual use). Some users feel annoyed at typing such pseudo-semis; they feel they should be inserted automatically. Unfortunately, they are not always necessary, and inserting extra ones will typically at least insert an extra blank line in the output, possibly worse. Nevertheless, the command-line option '-m;' has been provided; this will append a pseudo-semi to all WEB macro definitions. However, use of this option is not recommended.

11.57 @e (pseudo-expression)

[*C*] The philosophy of the *pseudo-expression* is similar to that of the pseudo-semi. The pseudo-expression is treated like an expression (an identifier is a simple expression) for formatting purposes, except that it is invisible. It finds use in certain macro constructions that the syntax analyzer would not otherwise recognize. For example, in C the analyzer has the rules $*expr \rightarrow expr$ and $(expr) \rightarrow expr$, but it doesn't understand the construction '(*)', which occurs only very rarely in ordinary C syntax (in certain casts) but certainly might appear in unusual macro usage. The cure is to use the pseudo-expression; WEAVE will be happy if you say '(*@e)'.

11.58 @: (pseudo-colon)

[*C*] The *pseudo-colon* is philosophically similar to both the pseudo-semicolon and the pseudo-expression: It's treated just like a colon, except that it's invisible. It is useful in formatting certain **case** constructions. For example, consider

```
@<Special cases@>=
case 1:
case 2@:  @;
```

The pseudo-colon is used so that one can later say "@<Special cases@>:" when the module name is actually used. The pseudo-*semi*colon is used to terminate the module so that the syntax analyzer can understand the entire construction as a statement and thus format it properly.

The last eight control codes (namely '@,', '@/', '@|', '@#', '@+', '@;', '@e', and '@:') have no effect on the program output by TANGLE; they merely help to improve the readability of the TEX-formatted code that is output by WEAVE, in unusual circumstances. WEAVE's built-in formatting method is fairly good, but it is incapable of handling all possible cases, because it must deal with fragments of text involving macros and module names; these fragments do not necessarily obey the syntax of the source languages themselves. Although WEB allows you to override the automatic formatting, your best strategy is not to worry about such things until you have seen what WEAVE produces automatically, since you will probably need to make only a few corrections when you are touching up your documentation.

"WEAVE's built-in formatting method is fairly good, but it is incapable of handling all possible cases..."

Because of the rules by which every module is broken into three parts, the control codes '@d', '@f', '@l', '@v', and '@a' are not allowed to occur once the code part of a module has begun. Note that '@m' *is* allowed (unlike in the original WEB design); it signifies a deferred macro definition.

The WEB control codes are summarized in Appendix L.

♣ 12. ADDITIONAL FEATURES and CAVEATS

Here we collect a miscellany of features, warnings, and suggestions.

12.1 Extended character sets

In Knuth's original memo, this remark was about extended character sets. See the documentation for common.web for more information.

12.2 It's best to use ASCII characters

If you have an extended character set, all of the characters listed in Appendix C of *The TEXbook* can be used in strings. But you should stick to standard ASCII characters if you want to write programs that will be useful to all the poor souls out there who don't have extended character sets.

12.3 Numerical constants

Hexadecimal, octal, and binary constants are allowed in all languages, generalizing the C-style format (which does not allow binary constants). The syntax is 0x*hhh* for hexadecimal, 0*ooo* for octal, or 0b*iii* for binary. Here *hhh* stands for a sequence of hexadecimal digits (0–9 and *A*–*F*), *ooo* stands for a sequence of

octal digits (0–7), and *iii* stands for a sequence of binary digits (0 or 1). Of course, C already recognizes hexadecimal and octal constants, so for C these are just passed through to the output unchanged. In the other cases, the constants are converted (during tokenization in phase one) to the appropriate integer. For example, in RATFOR or FORTRAN each of `0xF`, `017`, and `0b1111` is replaced by 15.

12.4 Special assignment and increment operators

In FORTRAN and RATFOR, the post-increment operators '`++`' and '`--`' and the compound assignment operators '`+=`', '`-=`', '`*=`', and '`/=`' are allowed in restricted contexts—namely, in simple assignment statements. These operators translate as follows, where x is anything allowed on the left-hand side of an equals sign—for example, a subscripted expression:

```
x++; // -> ‘x = x + 1;’
x--; // -> ‘x = x - 1;’
x += expr; // -> ‘x = x + (expr);’
x -= expr; // -> ‘x = x - (expr);’
x *= expr; // -> ‘x = x*(expr);’
x /= expr; // -> ‘x = x/(expr);’
```

They are very useful at increasing the readability of your code. They can also be used in the RATFOR **for** statement. For example,

```
for(i++; i<100; i*= 5)
```

They cannot, however, be used in ways such as '`a(k++) = b(i++);`', even though the analogous construction in C is both legitimate and often highly valuable. Furthermore, one must say '`i++`', not '`++i`', even though these would be identical in C when used stand-alone. If one does not want these constructions to be recognized, he may turn them off by the command-line option '`-+`'.

12.5 Strings

Strings are delimited by single quotes (FORTRAN), double quotes (C and RATFOR), or (in the case of certain built-in functions), by parentheses. In order to continue quoted strings to another line, all lines but the last must end with a backslash (just as one would define a lengthy C macro). Parenthesized strings should not be explicitly continued, unless the command-line option '`-(`' is used.

By default, the continuation of the string begins in column 1 of the next line. FORTRAN–90 introduces a different convention, in which the continuation of the string may begin in an arbitrary column, but *must* be preceded by the same continuation character (an ampersand in FORTRAN–90) that was used at the end of the previous line. This is neither required nor allowed in `FWEB`'s default continuation mode. However, if the FORTRAN–90 convention is desired, it may be turned on by the command-line option '`-\`'. When turned on, it operates for all languages, not just FORTRAN–90. As an example, the following two lines of code are equivalent, both dealing with the string `"12345"`.

```
@n9[-n&]
@ The global language is Fortran--90, with free-form syntax.
FWEB’s default continuation convention is used.
@A
x = "123&
45"
// Now tell \FWEB\to use Fortran--90’s continuation convention.
@n9[-n& -\]
x = "123&
    &45"
```

12.6 Breaking long strings

FWEAVE will break very long strings if necessary after embedded commas, or after every so many characters. When it does so, it inserts a backslash; this generally causes no confusion.

12.7 Breaking TEX output lines

The TEX file output by WEAVE is broken into lines having, by default, at most 80 characters each. The algorithm that does this line breaking is unaware of TEX's convention about comments following '%' signs on a line. When TEX text is being copied, the existing line breaks are copied as well, so there is no problem with '%' signs unless the original WEB file contains a line more than eighty characters long or a line with code text in |...| that expands to more than eighty characters long. (You may not be likely to create such lines yourself, but certain processors that create WEB code automatically have been known to generate such long lines.) Such lines should not have '%' signs. If you run into trouble here, you might try increasing the length of the output line by using the command-line option "-yll*nn*", where $nn < 255$.

12.8 Comments

In all languages, the preferred commenting style is that of C or C++: /*...*/ can be extended across newlines; //... is terminated by a newline. In principle, these can be placed anywhere; however, see suggestion a) in the following item. Note that standard FORTRAN uses the '//' operator for concatenation. Therefore, in FWEB the short comment is not recognized unless the one of the command-line options '-n/' (FORTRAN), '-r/' (RATFOR), or '-/' (both FORTRAN and RATFOR) is used. To allow both concatenation and short comments, the operator '\/' has been introduced as an alternative symbol for concatenation. Also, FORTRAN–90 introduces the exclamation point to begin a single-line comment. However, this conflicts with FWEB's standard use of the point for the logical NOT. By default, therefore, FWEB will not recognize the exclamation point as related to comments in FORTRAN–90 unless one of the command-line options '-n!', '-r!', or '-!' is used; it will, however, always recognize the construction '!!' as the beginning of a short comment. Thus, each of the following four lines produces the same output for both FTANGLE and FWEAVE.

```
@n9[-n& -!]
@
@A
x = y // z !  This is a comment.
x = y // z !!  This is a comment.
@n9[-n& -n/]
x = y \/ z // This is a comment.
x = y \/ z !!  This is a comment.
```

12.9 Translation of code text

Code text is translated by a "bottom up" procedure that identifies each token as a "part of speech" and combines parts of speech into larger and larger phrases as much as possible according to a special grammar that is explained in the documentation of WEAVE. It is easy to learn the translation scheme for simple constructions like single identifiers and short expressions, just by looking at a few examples of what WEAVE does (see the following paragraph), but the general mechanism is somewhat complex because it must handle much more than code itself. Furthermore the output contains embedded codes that cause TEX to indent and break lines as necessary, depending on the fonts used and the desired page width. For best results it is wise to adhere to the following restrictions:

a) Comments in code text should appear only after statements or clauses; i.e., after semicolons, after reserved words like **then** and **do**, or before reserved words like **end** and **else**.

b) Don't enclose long code texts in `|...|`, since the indentation and line breaking codes are omitted when the `|...|` text is translated from code to TEX. Stick to simple expressions or statements.

One can watch the translation in action by using the option '`-2`' on `FWEAVE`'s command line. This will send to the terminal a detailed list of the actions `FWEAVE` takes while combining parts of speech into phrases. For example, the simple C program

```
main()
{
x = y;
}
```

will produce output something like the following:

```
Tracing after l.  8 (language = C):
121:  *expr +expr+ +{+ expr...  =="\{" "\\{x}"
5:  *+expr+ +{+ expr...  =="\{" "\\{x}"
1:  *+fn_decl+ +{+ expr +binop+...  =="\\{x}" "="
3:  *+fn_decl+ +{+ +expr+ ;...  =="\\{x}" ";"
6:  +fn_decl+*+{+ +stmt+ +}+ -ignore- =="\}" "[force]"
131:  *+fn_decl+ -stmt+ -ignore- \"[force]${}[[5]]$[force]${}[[15]]
$[force]${}[[10]]" "[force]"
71:  *+function- -ignore- =="\\{main}" "[force]"
0:  *+function- =="\254" "\\{main}"
```

Each line begins with the number of the rule that was just executed. For example, rule 121 recognizes the combination '`()`' as an expression, and rule 71 recognizes that a function declaration ('`main()`') followed by a complete statement ('`{...}`') is a function. The asterisk indicates the starting point for the next search for a matching rule. The leading and trailing plus and minus signs indicate whether that particular part of speech begins or ends with math mode (plus) or ordinary text mode (minus). The last two quoted expressions on a line are strings whose contents are the current translations of the last two explicit parts of speech shown on the line. (Some of the details of the printed expressions are really intended for the developer of `FWEB` and need not be explained here.) If all goes well, the last line will consist of a single part of speech such as *function*; however, if there is a rule missing or incorrect or if the source code syntax is invalid, the final line may consist of a multitude of unreduced parts of speech. Option '`-1`' will print only such unreduced scraps. Such debugging can be turned on selectively by the commands '`@1`' or '`@2`', and turned off again by '`@0`'. (The '`@0`' should be placed in a section separate from the other debugging commands.)

12.10 Code within vertical bars

Comments and `WEB` macro definitions are not permitted in `|...|` text. After a '`|`' signals the change from TEX text to code text, the next '`|`' that is not part of a string or control text ends the code text. Thus, you can say "`|@c x || y|`" but not "`|@c x | y|`". To handle the intent of the last example, you may say "`|@c x ||| y|`".

12.11 Braces in comments

A comment must have properly nested occurrences of left and right braces, otherwise `WEAVE` will try to balance the braces to keep TEX from fouling up too much. Unfortunately, the scanning mechanism that is used here is optimized for speed, not perfection, so `WEAVE` will complain about the legitimate construction "`/* ...$\{$...  */`", for example. You may need to introduce TEX macros to avoid such warnings.

12.12 Reserved words

Reserved words of the individual languages, such as **int** or **double precision** must appear entirely in lowercase letters in the `WEB` file; otherwise their special nature will not be recognized by `WEAVE`. You could, for example, have a macro named *DIMENSION* and it would not be confused with FORTRAN's **dimension**.

12.13 FORTRAN keywords

However, FORTRAN also has another class of words, namely keywords such as `BLOCKSIZE` or '`ERR` that are used only inside I/O statements. In FORTRAN and RATFOR, the upper-case versions of these words are treated as reserved and will be formatted in a special way that makes the I/O statements look appealing.

12.14 Formatting identifiers

The `@f` feature allows you to define one identifier to act like another, and these format definitions are carried out sequentially. However, a given identifier has only one printed format throughout the entire document (and this format will even be used before the `@f` that defines it). The reason is that `WEAVE` operates in two passes; it processes `@f`'s and cross-references on the first pass and does the output on the second. (Note that in C one has the possibility of defining additional types via the **typedef** command. This essentially acts like an implicit '`@f`'.)

12.15 Formatting module names

In fact, as we have already stated, one can also format a *module name*. By default, module names are interpreted as expressions, but this is not always what is desired. For example, one can say

```
@f @<Common blocks@> common
```

and the parser will understand the use of `@<Com...@>` as a type specification. Only the first slot of `@f` is allowed to be a module name; the second slot must always contain an identifier.

12.16 New reserved words

You may want some `@f` formatting that doesn't correspond to any existing reserved word. In that case, `WEAVE` could be extended in a fairly obvious way to include new "reserved words" in its vocabulary. For example, `WEAVE` has already been taught to understand the identifier ***Real*** as an intrinsic function in Fortran. Note that a way of teaching `WEAVE` your own formats without recompiling is to include via `@i` a file including those format commands.

12.17 Special array formatting

Both FORTRAN and C have a somewhat vanilla-flavored syntax for array elements—e.g., `f(i,j)` or `f[i][j]`. In some physics applications certain indices are preferred—e.g., they might be covariant or contravariant—-and it is useful to see this explicitly in the woven output. Also, some users are annoyed (with good reason) by the dual use of parentheses in FORTRAN to denote both function calls and array elements; they prefer to see something like `f[i,j]` for a FORTRAN array element. Several features are available to help in this regard.

Usually the parentheses (FORTRAN) or brackets (C) denoting array elements are just passed through to the output by both `FTANGLE` and `FWEAVE`. However, when the '`-W[`' option is used, `FWEAVE` inserts a special call to the TEX macro `\WXA` , with the array elements as arguments. By redefining the behavior of `\WXA`, a variety of special effects can be achieved. To be more precise, one shouldn't generally modify the definition

of \WXA itself, which is rather complicated; rather, one should redefine \WARRAY, which is called by \WXA. See the following example.

Redefining \WARRAY affects all array references. An even more general possibility is to use the '@W' command to overload specific identifiers in different ways. See the description of '@W' and the following example.

---

1. ARRAY PROCESSING. This example demonstrates two ways of beautifying array references in FORTRAN and C. Parenthesized references can be overloaded with the '@W' command, as follows:

```
@W x \x    // Replace references to x by the macro \x.
@W y \y
@W z \z

@l "\\def\\x(#1){x^{#1}}"    // Contravariant index.
@l "\\def\\y(#1){y_{#1}}"    // Covariant index.
@l "\\def\\z(#1,#2){z^{#1}{}_{#2}}"    // Mixed indices.
```

2. Bracketed array references are activated by the '-W[' command. (In FORTRAN FTANGLE always automatically replaces brackets by parentheses.) One can redefine the \WARRAY macro to get special effects.

```
@l "\\let\\WARRAY\\WSUB"    /* Subscript bracketed indices. (\WSUB is defined in fwebmac.web.) */
```

3. In the following test, carefully note the difference in type size between the results of parenthesized subscripts and bracketed ones. To fully understand why this occurs, study the definition of the `\WXA` macro in `fwebmac.web`.

program *main*•

```
  /* Test of overloaded identifiers. */
x^i
y_j
z^i_j

  /* Bracketed indexing. */
A_i
B_j_par
C_index,j_par
D_index j_par+1
E_1+2*i

  /* Brackets aren't active inside strings. */
'a[b]c(d)'
```

end

```
@Lc:    /* Now, an example from C. */
a_1,2,k;    // In the source, this is "a[1][2][k]".
```

4. INDEX.

(Index and remaining material skipped.)

(Page break skipped.)

12.18 Forward references to identifiers

We have learned that `FWEAVE` makes two passes through the source file in order to handle forward references to module names—i.e., the situation in which a module name is used before it is defined. It is desirable to extend this feature to various identifiers such as function names and macros. The convention is to automatically *subscript* such identifiers with the number of the module in which they are defined. (The format can be changed by refining the `fwebmac` macro `\WIN`; see below.) If the identifier has been defined in the current section, then the numerical subscript is replaced by a small bullet.

Implementing such a mechanism poses a difficult problem in general. For example, function names are processed by `FWEAVE`'s syntax parser during phase 2, so unless some special action is taken only function references occurring *after* the definition will be subscripted—in other words, forward referencing would not work. The most straightforward way of solving this problem would be to make two passes through the parser; however, this has been rejected as being too slow. Nevertheless, some partial solutions can be given.

First, one can force `FWEAVE` to understand that an identifier is a function name by prefacing it with the command '`@[`'. This command is executed during phase 1; it marks the name as being defined in the current module. Actually, it marks the first identifier it can find that is not already known to be a reserved word, so the '`@[`' may occur at the very beginning of a function declaration. This allows a further convenience. By

default, the command ‘@a’ is equivalent to ‘‘@A@[’’, where ‘@A’ begins the unnamed module but does nothing else. (Previously in this manual, we have used ‘@a’ to begin the unnamed module; we now see that that was a little white lie.) Thus, the following three function definitions are completely equivalent:

```
@ Explicitly mark the function name.
@A
void @[f()
{}

@ Explicitly mark the entire function name.
@A@[
void f()
{}

@ Rely on ‘@a’ to issue an implicit ‘@[’.
@a
void f()
{}
```

Since the marks are executed during phase 1, forward referencing is not a problem. Thus, in the following code the references to functions f and g in module 1 will be properly subscripted in the woven output.

```
@c
@ An example of a function with forward referencing.
@a
main()
{
f();
g();
}

@ Forward references to functions declared in the unnamed module are
handled with ease.
@a
void f()

@ For named modules, one must mark explicitly if that is appropriate.
@<Special functions@>=@[
void g()
```

Note that at present section names do *not* automatically issue an ‘@[’ command, so we had to insert one explicitly in the above example. This restriction may be removed in future releases.

Macro references can also be subscripted. The commands ‘@d’ and ‘@m’ issue implicit ‘@[’s by default, so you should have to do nothing explicitly to have macro references properly identified. If you never want a particular macro reference to be subscripted, use the upper-case commands ‘@D’ or ‘@M’.

In C-like languages additional types can be created via the **typedef** command. In the original CWEB design, **typedef**s were processed during phase 2, so there was again a problem with forward referencing. The implicit ‘@[’ command now issued by ‘@a’ aggravates the difficulty, as seen in the following example:

```
@c
@
@A
@<Typedefs@>@;

@ The user-defined type |MY_TYPE| isn’t understood here yet when |typedef|s
```

```
are processed in phase 2.
@a
MY_TYPE f()
{}
@
@<Typedefs@>=
typedef int MY_TYPE;
```

Not only is the **typedef** name **MY_TYPE** not understood when it is discussed in the text, the implicit '`@[`' issued by '`@a`' will incorrectly mark *MY_TYPE* instead of *f* as a function defined in the current module. Therefore, **typedef**s are now partly processed during phase 1 to the extent that the identifier being **typedef**ed is now marked as a reserved word. Since the subscripting of identifiers is done during the output in phase 2, the above example will now work correctly.

Here is a typeset illustration that contains forward references to function names, macro references, and **typedef**s:

---

1. FORWARD REFERENCING. Here is a nonsense program illustrating forward referencing for identifiers. Although there are various instances of forward references, there was need for just one explicit '`@[`' command (in module 3). Identifiers that are used in the same section as they are defined are subscripted with a bullet. Note how the module-number subscripts are set in different type for different kinds of identifiers.

Here we see that one can refer to the macros D_2 and $W_{\underline{2}}$ as well as the user-defined type $\mathbf{PTR_4}$ in advance of their definition, yet they will be subscripted properly.

int $main_\bullet$( )
{
⟨ Typedefs 4 ⟩
⟨ Special stuff 3 ⟩

$x = fcn_2(D_2(outer\_test))$;
$py = g_{\underline{3}}(W_{\underline{2}}(WEB\_test))$;
}

2. Examples of definitions of an outer macro, a `WEB` macro, and a function.

@d $D_\bullet(name)$ #*name*
@m $W_\bullet(arg)$ ∗*arg*++

int $fcn_\bullet$(**char** ∗*name*)
{}

3. In the following, we had to say "`PTR␣@[g...`" in order to make forward referencing to $g_\bullet$ work. Although one can say things like "`@[int␣g...`", it wouldn't work here to say "`@[PTR␣g...`" because $\mathbf{PTR_4}$ isn't known yet as a special type, even in phase 1.

⟨ Special stuff 3 ⟩ ≡

```
PTR4 g•(int i)
  { }
```

This code is used in section 1.

4. The compiler will see this **typedef** before any statements that use the type $\mathbf{PTR_\bullet}$.

⟨ Typedefs 4 ⟩ ≡

```
typedef char *PTR•;
```

This code is used in section 1.

5. INDEX.

(Index and remaining material skipped.)

(Page break skipped.)

When many macro references and/or **typedef** statements are in use, one may in some cases actually be annoyed by the clutter of subscripts. Several mechanisms are provided to selectively turn on or off the subscripts. First, all subscripts can be turned off from the command line by the option '`-f`'. Second, style file entries (see section on "Advanced Features" below) serve as switches to enable or disable subscripting for certain types of identifiers, according to the following table:

`mark_defined.generic_name` — Something explicitly marked by '`@[`'. [0]
`mark_defined.fcn_name` — Function names. [1]
`mark_defined.WEB_macro` — `WEB` macro (`@m`). [2]
`mark_defined.outer_macro` — Outer macro (`@d`). [3]
`mark_defined.exp_type` — Something explicitly marked by `@``. [4]
`mark_defined.typedef_name` — A **typedef**-like statement in C-like languages. [5]

By default, `mark_defined.generic_name` and `mark_defined.exp_type` are set to 1 (on); the others are set to 0 (off).

Each of the above types of identifiers has a number, indicated in brackets. This number is the first argument to the subscripting macro `\WIN`; the second argument is the number of the module in which the identifier was defined. Thus, if the function *fcn* was defined in module 38, the output `tex` file produced by `FWEAVE` will contain the phrase "`\\{fcn}\WIN1{38}`". The identifier number is used as an argument to an `\ifcase` statement, so the output format can be different for different types of identifiers and can be controlled by the user. The default `fwebmac` macro `\WIN` subscripts all kinds of identifiers with a bullet if they were defined in the current module. Otherwise, the module number subscripts are typeset in the same style as the bracketed numbers in the above table. You may wish to experiment to find an output format that is more informative or pleasing to your eye.

12.19 Spacing and macros

Macros place unusual demands on WEAVE. For example, suppose one defined

```
@m PLUS +
```

then attempted to use that macro in the expression "x␣PLUS␣y". This would tangle correctly and do what you want, but WEAVE's output would look like "*xPLUSy*", whereas what you really would like is "*x PLUS y*". This occurs because WEAVE treats all normal identifiers as expressions, and its rules tell it to simply concatenate expressions (with no intervening white space). To help in situations like this, FWEAVE has a few extra reserved words that invoke special rules that insert extra spaces. These words are $_*BINOP*_, $_*COMMA*_, $_*EXPR*, $_*EXPR*_, $*EXPR*_, and $*UNOP*_. The underscores show where spaces will be inserted. You can format your unusual macros to these words, as in

```
@f PLUS $_BINOP_
```

and your macro will then be treated as the proper part of speech—either a binary operator, a comma, an expression, or a unary operator.

12.20 M4 built-in commands

By default, FORTRAN and RATFOR do *not* understand the special m4 preprocessor built-in commands: **changequote**, **define**, **divert**, **divnum**, **dnl**, **dumpdef**, **errprint**, **ifdef**, **ifelse**, **include**, **incr**, **index**, **len**, **maketemp**, **sinclude**, **substr**, **syscmd**, **translit**, and **undivert**. To make them understand m4, use the command-line option '-m4'. Because these commands are essentially macros, which need not obey the FORTRAN syntax, WEAVE has a very difficult time formatting arbitrary m4 constructions. To help WEAVE out, the underlined commands in the above list have their arguments interpreted as strings (delimited by balanced parentheses), and WEAVE will format them as such, in typewriter type. These strings are treated just like other strings, except that if they don't end on the same line, they don't have to be continued by a backslash. (However, you have the option of changing the default. For parenthesized strings only, if you use the command-line option '-(', then those strings must be continued by backslashes.)

Similar considerations about string arguments apply to the FWEB built-in commands **_COMMENT**, **_ERROR**, **_IF**, **_IFELSE**, and **_LEN**.

12.21 More general spacing

Sometimes it is desirable to insert spacing into code that is more general than the thin space provided by '@,'. The @t feature can be used for this purpose; e.g., '@t\hskip 1in@>' will leave one inch of blank space. Furthermore, '@t\4@>' can be used to backspace by one unit of indentation, since the control sequence \4 is defined in webmac to be such a backspace.

12.22 Change file

WEAVE and TANGLE are designed to work with two input files, called *web_file* and *change_file*, where *change_file* contains data that overrides selected portions of *web_file*. The resulting merged text is actually what has been called the WEB file elsewhere in this report.

Here's how it works: The change file consists of zero or more "changes," where a change has the form '@x⟨old lines⟩@y⟨new lines⟩@z'. The special control codes @x, @y, @z must appear at the beginning of a line; the remainder of such a line is ignored. The ⟨old lines⟩ represent material that exactly matches consecutive lines of the *web_file*; the ⟨new lines⟩ represent zero or more lines that are supposed to replace the old.

Whenever the first "old line" of a change is found to match a line in the *web_file*, all the other lines in that change must match too.

Between changes, before the first change, and after the last change, the change file can have any number of lines that do not begin with '`@x`', '`@y`', or '`@z`'. Such lines are bypassed and not used for matching purposes, except for the following special codes.

In addition to '`@x`', '`@y`', and '`@z`', the language commands '`@c`', '`@r`', '`@n`', and '`@L`*l*' are also allowed in the change file, as are the commands '`@[`' and '`@]`'. All these commands must begin in column 1. The command '`@[`' means switch into code mode; '`@]`' means switch out of code mode. These latter commands are necessary only in FORTRAN, but are crucial there. Although modes switch back and forth automatically between free-form and column syntax as the input driver reads sequentially through the *input* file, that can't happen automatically for the *change* file, which consists of isolated fragments of text that could match any line of the input file, in either TEX mode or code mode. If you do not help out the change file, the input driver will probably not interpret the syntax of the change file in the same way as it does the source file, and lines that look identical in the source files nevertheless won't match.

For example, if a FORTRAN–77 source file contains the line

```
call open('datafile')
```

the following change file will replace that line with two new lines:

```
@n
@[
@x
    call open('datafile')
@y
/* Here are the replacement lines:  */
    call open('newdatafile')
@z
```

This dual-input feature is useful when working with a master `FWEB` file that has been received from elsewhere (e.g., `FTANGLE.WEB` or `FWEAVE.WEB`), when changes are desirable to customize the program for your local computer system. You will be able to debug your system-dependent changes without clobbering the master `web` file; and once your changes are working, you will be able to incorporate them readily into new releases of the master web file that you might receive from time to time.

You specify the name of the change file on the command line; it is the second command-line argument that the parser can recognize as a file name—i.e, that does not begin with a hyphen. A default extension of '`.ch`' is implied. If you just say '`FTANGLE test`', you are talking about the web file '`test.web`' with the null change file. If you say '`FTANGLE test test`', you are talking about the web file '`test.web`' and the change file '`test.ch`'. If you say '`FTANGLE alpha.fweb beta.fch`', you are calling for the web file '`alpha.fweb`' and the change file '`beta.fch`'. It is almost never necessary to deal with nondefault file extensions. (The range of default extensions can be increased by entries in the style file. See the following discussion of the style file and of the command-line option '`-e`'.)

13. INPUT

It doesn't take much time attempting to fit the column-oriented FORTRAN–77 syntax into the `WEB` framework to realize the virtues of languages such as C, RATFOR, or FORTRAN–90 with free-form syntax. The present `FWEB` design solves the problem of multiple, incompatible input syntaxes by introducing the concept of *input drivers*. These are front ends that convert the incoming syntax as quickly as possible to

a uniform syntax that the innards of FWEB can understand. Individual lines from the WEB file are filtered through the *input driver* appropriate to whatever language is in force at that point. For C, that filter is essentially a unit operator; it presents to FWEAVE and FTANGLE the same input line that it read in. At the other extreme, however, for FORTRAN–77 the filter does much more work. Because of the possibility of *continuation lines*, the filter must read ahead. If it finds a continuation line, it concatenates it to the stuff already in the filter's buffer. Finally, when it finds no more continuations, it *appends a semicolon* to the logical line it has constructed, then presents this line to the innards of FWEAVE or FTANGLE. The semicolon serves as the end-of-statement delimiter, just as it does in C. Thus, the input line presented to the processors has a standard form for all supported languages, so it can be parsed in a standard way.

In fact, the FORTRAN and RATFOR drivers do even more things, especially related to comments. These details will be described in the following sections. Here, the point to remember is that your input file can pretty well follow the conventions of whatever mode you're in at the moment: C, C++, FORTRAN, TEX, or whatever.

13.1 FORTRAN–77 input

It is recommended that you do not use bare FORTRAN *for new codes; use* RATFOR *instead.* Another option is to use the free-form mode of FORTRAN–90.

Although the preferred commenting style is C-style, the FORTRAN column-1 convention is retained for compatibility with existing code. These two styles are fundamentally incompatible, and the FORTRAN style is definitely not recommended! Note that trailing comments beginning with an exclamation point *will not work*!

```
x = y !This doesn't work in Fortran--77.
```

This is a VAX extension that conflicts with the standard WEB use of the exclamation point, as it is the input symbol for the logical NOT. (See below.) If your source file already contains such comments, you must change them into C-style ones. With the aid of a good editor, this can be done with just one command repeated automatically.

Following the conventions of FORTRAN–77, lines that begin with 'C', 'c', or '*' are (short) comments (terminated by the end-of-line). When the option '-n/' is in effect, these lines behave identically to ones beginning with '//'. Consecutive comments of the same kind (long or short) are concatenated by the input driver. Thus, the construction

```
/* A */ /* B */
C C
// D
* E
```

will be typeset into the two lines

/* A B */
// C D E

Since the input driver concatenates successive lines of the same type, very long effective input lines can be created. Sometimes the associated buffer overflows and FWEB terminates with an error message. Since that buffer is allocated dynamically, one can use the '-y' option to increase its size; the error message explains how.

A comment that appears on the line immediately after a statement or continuation line will be attached

to the previous line as a trailing comment. Generally, this is not what you want; a comment beginning in column 1 probably belongs with the next statement. One solution is to insert a blank line before the comment. This puts a blank line in your woven output. If you don't want that, omit the blank line and preface the comment by '`@/`'.

The commenting rules are such that both `FTANGLE` and `FWEAVE` will handle compiler directives appropriately. Typically a compiler directive begins with the character '`C`' so it will be ignored by a compiler that doesn't understand the directive. Thus if `cdir` is a compiler directive, one can say

```
    x = y

cdir Text of the directive
    (more code)
```

and the compiler directive will be executed. Note the important blank line before the directive. If it were not there, the directive would be treated as a trailing comment attached to the previous line of code. (It is instructive to try this example with the blank line both present and absent. With the blank line absent, you will find that the '`d`' of `cdir` gets eaten. This occurs because of a quirk of the input driver that will be fixed someday.) However, for new codes a much better solution for compiler directives is to use the '`@?`' command; see the discussion in the section on control codes.

The FORTRAN–77 input driver recognizes the end of a valid statement when the next line is neither a continuation nor a comment. At this point, the input driver performs a scan over everything it has collected so far and concatenates any adjacent comments of the same type. It then *appends a semicolon* to the end of the compilable part of the statement (i.e., just before the last comment), because the semicolon is used as a universal statement terminator in the innards of `FWEAVE` and `FTANGLE`. Logically, the semicolons should show up in the woven output, since they are `FWEB`'s way of terminating statements. However, this makes the FORTRAN output look a little strange, and, logic notwithstanding, people have understandably found this annoying. Therefore, by default the semicolons are actually appended as *pseudo-semicolons*, so they will be invisible on output. To make the semicolons visible, use the command-line option '`-np`'. `FTANGLE`, of course, always throws the semicolons away during its output phase, since they would not be understood by the compiler.

"By default the semicolons are actually appended as *pseudo-semicolons*, so they will be invisible on output."

If you think the FORTRAN input driver isn't working the way it's supposed to (for example, if it doesn't concatenate continuation lines properly), you might want to look at the line-by-line output from the driver. You can turn on this debugging feature with the command-line option '`-l`'. If this confirms that something is wrong, by all means report the bug. The FORTRAN driver is much too complicated for its own good! (Often, bugs show up in incorrect continuation lines. These can often be circumvented by using the `-#` option to remove the line- and module-number comments.)

If all goes well and `WEAVE` is able to recognize a complete main program, function, or subroutine, the body of that program unit is indented. To be consistent, you should use the **program** statement for main programs; otherwise, the indentation may appear strange. (You may call this a bug, but it's really a design "feature".)

FORTRAN contains such archaic constructions as '`.ne.`'. `WEB` has two ways of helping you to deal with these things. You are allowed more flexibility in input, and `WEAVE` substitutes prettier constructions such as '$\neq$' as it typesets the documentation. (The exact form of the output is controlled by a TEX macro; study `fwebmac.web`. Thus, if you don't like the way exponentiation is typeset, for example, you can easily change it without recompiling `FWEB`.) Here is a table of what you can type on input, and what `WEAVE` will typeset.

The first entry is standard FORTRAN; the parenthesized material is an allowable input alternative. (In most cases, the pretty input alternatives follow C's convention.)

`.lt. (<)`	$\rightarrow <$	`.or. (||)`	$\rightarrow \vee$
`.le. (<=)`	$\rightarrow \leq$	`.neqv.`	$\rightarrow \not\equiv$
`.eq. (==)`	$\rightarrow \equiv$	`.xor.`	$\rightarrow \not\equiv$
`.ne. (!=,<>)`	$\rightarrow \not\equiv$	`.eqv.`	$\rightarrow ?=$
`.gt. (>)`	$\rightarrow >$	`.not. (!)`	$\rightarrow \neg$
`.ge. (>=)`	$\rightarrow \geq$	`**` (`^`)	$\rightarrow$ `(a+b)^(c+d)` $\rightarrow (a+b)^{c+d}$
`.and. (&&)`	$\rightarrow \wedge$	`//` (`\/`)	$\rightarrow \|\|$

These same conventions are allowed in RATFOR mode. Note that in FORTRAN and RATFOR '`//`' is interpreted by default as the concatenation symbol, not the start of a short comment. To override that default, use one of the command-line options '`-n/`', '`-r/`', or '`-/`', or use a language-changing command of the form "`@n/`".

A few restrictions or modifications of FORTRAN's rules have been made in the interest of simplicity. First, `FWEB` recognizes identifiers in a slightly different way than does FORTRAN. As an extension compatible with RATFOR and C, identifiers may contain underscores and dollar signs and may be of arbitrary length. (`FTANGLE` can translate these to garden-variety FORTRAN identifiers; see the discussion of the `-t` option. On the other hand, as a restriction there may not be any white space within identifiers. (However, `FWEAVE` will correctly understand **go to**, **else if**, **end if**, and **end do**.) Secondly, there are a few cases in FORTRAN where the same identifier is allowed to be used in two entirely different ways. Perhaps the most prominent case is the identifier `real`, which is used both as a type specification and as an intrinsic function. `FWEAVE`, not being as intelligent as the FORTRAN compiler, does not understand enough about the syntax to properly distinguish these two uses; specifically, it understands `real` only as a type specification. In cases such as this, one can often make `FWEAVE` happy by recalling that `FWEAVE`'s reserved words are all in lower case, but that FORTRAN is case-insensitive. Thus, FORTRAN doesn't care whether you type `real` or `Real`, but `FWEAVE` will not confuse *Real* with a type specifier. In fact, for this particular case `FWEAVE` has been taught to understand that *Real* is an intrinsic function, so what you will really get is **real** and ***Real***. As a matter of good programming style, you just shouldn't get into this situation if you can avoid it. Don't, for example, use *Integer* as the name of a function, even though FORTRAN allows that and `FWEAVE` will treat it as an ordinary identifier.

Standard FORTRAN–77 does not understand the unnumbered block **do** construction, which is a VAX extension. `FTANGLE` understands such constructions, and can optionally convert them into standard numbered **do**s. See the description of the command-line option '`-d`'. (This mode is slow, and is not recommended. Use RATFOR or FORTRAN–90 instead.)

An excellent application of named modules is to define things like **common** declarations that you wish to insert at the beginning of every subroutine. By using a named module, you can keep the code for the common declarations in the same file as the rest of your code; a separate include file is unnecessary.

One of the annoying FORTRAN constructs is the "dot constant"—e.g., '`.false.`' or '`.eq.`'. In some cases `FWEB` gets confused by the periods, which are used for other purposes as well. For example, a VAX extension uses periods to separate components of structures. (FORTRAN–90 uses '`%`' for this purpose.) If you want, you can change the delimiters of such dot constants via the style-file options `dot_constant.begin` and `dot_constant.end`. For example, if you say in `fweb.sty`

```
dot_constant.begin '['
dot_constant.end ']'
```

then you can say in your code things like "`if(a[eq]b)...`". However, generally it is better to say "`if(a==b)...`". The command-line option '`-.`' tells `FWEB` to not attempt to identify dot constants (even when the delimiters have been changed as above). Use this option if your compiler uses periods for something else.

13.2 FORTRAN–90 input

FORTRAN–90 permits two input syntax styles: column format, as in FORTRAN–77, and free-form format. The column format behaves as in FORTRAN–77; see the previous subsection. The free-form format is a great advance, and should be used for new code. The following remarks pertain to free-form input.

In free-form input, lines are continued by a character at the *end* of the line instead of one in column 6 of the next line. The continuation character is selected by a command-line option, which also serves to turn on the free-form input mode. You may choose either '`-n&`' or '`-n\`', allowing one to continue lines with either an ampersand (FORTRAN–90's rule) or a backslash (compatible with C and other `WEB` conventions). (If you select '`-n\`', `FTANGLE` will convert the backslash into an ampersand on output.) For example, here is how to begin a `web` source code written with free-form FORTRAN–90:

```
@n9[-n& -n/]

@* INTRODUCTION. The following code will be in free-form Fortran--90.
(It may take a lot of CPU time.)
@a
program main
integer i

i =&
0 // Illustration of line continuation.
iterate:  do
    i++
    end do iterate

end
```

13.3 RATFOR input

By default, the auto-semi mode is not used. In this case the RATFOR syntax is completely free-form and identical to that of C. In the auto-semi mode the input driver will turn the RATFOR-style comment (anything between '`#`' and the end-of-line) into a C-style comment. ('`#`' characters embedded in comments will just be copied, and paste tokens ('`##`') in `WEB` macro definitions will be properly understood.) As described earlier, in that mode it also handles the RATFOR style of obvious continuation, and supplies semicolons in the appropriate places.

With the exception of remarks specific to FORTRAN's line-and-column-oriented syntax, all the other remarks about FORTRAN apply to RATFOR as well.

13.4 C and C++ input

This is the bare-bones driver; it does nothing but pass the free-form syntax along to the innards of the processors. This driver is also used for RATFOR.

Beginning with version 1.21, **typedef**ed variables can be marked automatically with the number of the module in which they are defined. See the previous discussion of "Forward References."

One formatting annoyance is concerned with constructions of the form

```
typedef struct weird
    {
    ...
    } weird;
```

The C language permits this construction, in which *weird* is used in two ways. However, WEAVE is not as clever as a compiler; it only understands one interpretation of a variable, and will likely become slightly confused by the above situation. It's not really good programming practice to use an identifier in more than one way anyway; replace the last *weird* by *WEIRD*, or the first *weird* by *weird0*.

14. COMMAND-LINE OPTIONS

Various information of interest to FWEAVE and/or FTANGLE may be entered from the command line. Command-line arguments are delimited by spaces; if you need a space inside an argument itself, surround the argument with quotes (single or double, depending on your operating system or shell). Since the arguments are case-sensitive, remember that under VAX/VMS arguments are translated into lower case unless they are protected by double quotes. Under UNIX, certain characters such as "`&|;<>(){}*?[]`" may mean special things to the shell and may need to be escaped with a backslash or protected with quotes. Thus, while under VMS a valid option is '`->`', under UNIX you must say '`-'>'`' or '`-\>`'. (Since this is cumbersome to type, the synonym '`-=`' has been provided.)

Command-line arguments may appear in any order; they are processed from left to right. If a command-line argument does not begin with a hyphen, it is understood to be a file name. The first file name found is the name of the WEB source file; optionally, if a second file name is found, it is the name of the change file. If the name (excluding a possible path) contains a period, then it is processed exactly as specified. Otherwise, WEB supplies an extension. Precisely how this is done depends on whether the '`-e`' option is in effect. If it is not in effect, then a default extension is used: "`.web`" for the WEB file, "`.ch`" for the change file. If it is in effect, then extensions taken from the relevant style file entry (`ext.web` or `ext.change`) are applied in turn until one matches. A lone hyphen '`-`' is understood to stand for the special file `stdin` (standard input), which is usually the terminal. Hyphenated options should have no space between the option and its parameter, if one is required. Thus, an example of a command line is

"Command-line arguments may appear in any order."

```
fweave test test -wmy_macros
```

which means "Weave the file `test.web` using the change file `test.ch`; print '`\input my_macros`' at the beginning of `test.tex`." It would be an error if `test.web` were missing. However, if one had said in the style file "`ext.web = "web␣wb"`" and used the '`-e`' option, then in the absence of `test.web` the system would attempt to open `test.wb`. This same mechanism works for file names specified on `@i` lines (see the discussion of the control code '`@i`', except that if the '`-e`' option is not in effect there are no default extensions. The associated style-file entries are `ext.hweb` for the included web file and `ext.hchange` for the associated change file.

Certain command-line options can be negated by including an extra hyphen. For example,

```
fweave test --x
```

means "Do the opposite of option '`-x`'." Since '`-x`' means suppress all cross-reference information, '`--x`' means "*do* write all cross-reference information." This example is essentially pedagogical, since that information is normally written anyway by default. The most common reason to negate an option is when one wishes to override an option that was set in the initialization file `.fweb` (discussed below). For example, `.fweb` might contain the command '`+v`', which tells FTANGLE to make all comments verbatim and pass them along to the output file. If you want to turn off those comments for one particular run, say '`--v`' on the command line.

14.1 Options to language commands

Whenever a language is changed via a control code such as '`@c`', that code may be optionally followed by

text, optionally enclosed in square brackets. This was explained in the section on languages. The optional text is essentially parsed as though it were a command line; this affords a way of selecting different parameters for each language. For example, the following are valid commands:

```
@c++ @% Select c++.
@r[-k*] @% Select Ratfor, and suppress comments about keyword expansions.
```

However, some options such as `-x` are truly global and are not reasonably changed in the midst of the `web` source. Those global commands that may *not* be used inside brackets are marked in the following list with asterisks.

14.2 List of options

Here are the options (brackets mean optional arguments; case is significant):

`-` — The file name "`stdin`", which signifies "standard input".

`-0` — Turn off `WEAVE`'s debugging output. (The command '`@0`' should be placed in a different module from '`@1`' or '`@2`'.)

`-1` — Turn on limited debugging for `FWEAVE`.

`-2` — Turn on full verbose debugging for `FWEAVE`.

`-A` — Turn on translations to **ASCII**. (On **ASCII** machines, translations to the internal **ASCII** representation are redundant, so are turned off by default. This flag is used for debugging. On non-ASCII machines, the translations are done regardless of the setting of this flag.)

`-b` — Number **do** and **if** blocks in woven FORTRAN and RATFOR output. This is helpful when the blocks are highly nested and/or long. The form of this numbering is defined by the `fwebmac` macro `\Wblock`, which by default prints as ' // Block 99'.

* `-c` — Set the global language to C. *(Will be overridden by a language command in limbo.)*

`-c++` — Set the global language to C++.

`-D`*[letters]* — Display information about reserved words of the current (command-line) language (beginning with *letters* if present). For example, to see all the reserved words of C, say "`ftangle -c -D`".

`-d`*[nnnnn]* — In FORTRAN–77, this tells `TANGLE` to convert unnumbered '**do**...**enddo**' constructions to numbered ones of the form '**do** 10000 ... 10000 **continue**'. The constructions are numbered uniquely, without regard for subroutine boundaries. By default, the numbering starts with a large number such as 90000. If you wish to change that starting number, say '-d*nnnnn*', where *nnnnn* is an integer number sufficiently less than 99999. Any statement numbers that you use yourself should be less than that starting number. (This feature is a kludge and is slow; use RATFOR instead.)

`-E`*c* — Change the delimiter of a file-name extension from the default '`.`' to '*c*'. This option is included because on some (weird) systems the period is used as a directory delimiter.

`-e` — Turn on automatic file-name completion. If any file name contains a period, then it is assumed that this is the complete name of the file to be opened. If it does not contain a period, then trial names are constructed by appending in turn each extension from the relevant style-file extension list: one of `ext.web`, `ext.change`, `ext.hweb`, or `ext.hchange`. The first trial name that matches a file in the user's directory is opened. For example, if one says in the style file "`ext.hweb = "hweb␣hw"`" and uses an include line of the form "`@i␣my_includes`", the system will search for the include files `my_includes.hweb` and `my_includes.hw`, using the first one it finds.

`-f` — Turn off the module reference subscripts for all identifiers.

* `-h` — Get help from the command line. *(Not implemented yet; just refers the user elsewhere and quits.)*

`-i` — This option is intended to help cut down on the volume of woven output. It tells `FWEAVE` to read include files named by the '`@I`' command, but to not print their contents. (Include files named by '`@i`' will always be processed fully, regardless of any command-line options.) This command will only work properly if the included text is a complete module, or has no new-module commands in it at all.

`-i!` — Tells `FWEAVE` to not even read include files named by the '`@I`' command.

`-I` — Append a directory to the list of directories to be search for include files. (That list begins with the contents of the environment variable *FWEB_INCLUDES* if that is defined.) The argument of this option can also be a colon-delimited list of directories.

`-k`*[letters]* — This option turns off the comment lines generated by `FTANGLE` during RATFOR statement translation. The '`k`' must be followed by a list of single lower-case characters (no quotes are required). If any of those characters is an asterisk, or if the list is empty, then all comment lines are suppressed. Otherwise, the characters serve as abbreviations for which RATFOR keyword the comment should be suppressed, as follows:

`b` — **break**	`n` — **next**
`c` — **case**	`p` — **repeat, until**
`t` — **default**	`r` — **return**
`d` — **do**	`s` — **switch**
`f` — **for**	`h` — **where**
`i` — **if**	`w` — **while**

For example, '`-kbn`' will suppress comments about the **break** and **next** statements.

`-K`*[letters]* — As above, except that the list of abbreviations specifies which comments to include, rather than to suppress.

`-l`*[mmmm[:nnnn]]* — Echo the input line that the input driver has constructed, between lines *mmmm* and *nnnn*. Useful for debugging (especially for `FWEB` itself), but rather slow. The command '`-l`*nnnnn*' echos all lines beginning with line *nnnnn*; if you just say '`-l`', the echo starts with the first line.

* `-L`*l* — Select language *l*, where $l \in \{\texttt{c}, \texttt{n}, \texttt{r}, \texttt{x}\}$.

* `-m`*id[=text]* — The construction '`-m`*string*', where *string* does not begin with '`4`', allows one to define `WEB` macros from the command line. To define the lower-case macro name '`test`' with null replacement text, say '`-mtest`'. To give `test` a value, say '`-mtest=5`'. (No spaces are allowed before or after the equals sign.) This is equivalent to saying at the beginning of the definition section of the first module '`@m test 5`'. (The equals sign in lieu of a space between macro name and replacement text is allowed *only in definitions made from the command line*, but allows you to avoid having to type quotes to ensure that the space is counted as part of the definition. For example, although allowable, it's more cumbersome to type '`-m"test 5"`'.) Under VMS, if your macro name is upper case (a common convention), you must enclose the definition in quotes, otherwise the name will be turned into lower case by VMS.

`-m4` — Tells `WEAVE` to understand the `m4` built-in commands. Also makes the output extension '`m4`' instead of '`rat`', or '`n4`' instead of '`for`'.

`-m;` — Tells `WEAVE` to automatically append a pseudo-semicolon to the end of `WEB` macro definitions. *(Not recommended; insert pseudo-semis explicitly where necessary.)*

* `-n` — Set the global language to FORTRAN-77.

`-n9` — Set the global language to FORTRAN–90.

`-n;` — For FORTRAN–77, supply semicolons automatically. (This is done automatically by default.)

`-nb` — Number **do** and **if** blocks in woven FORTRAN output. This is helpful when the blocks are highly nested and/or long. The form of this numbering is defined by the `fwebmac` macro `\Wblock`, which by default prints as ‘ // Block 99’.

* `-np` — Print semicolons in woven FORTRAN output. (Don’t confuse with ‘`-n;`’, described above.)

`-n\` — Use free-form syntax for FORTRAN–90; continue lines with a backslash.

`-n&` — As above, but continue lines with the ampersand.

`-n/` — In FORTRAN, make ’`//`’ denote the start of a short comment instead of concatenation. (One can always use ’`\/`’ for concatenation.)

`-n!` — In FORTRAN, make ‘`!`’ denote the start of a short comment instead of the logical `NOT`.

`-o` — Turn off `FWEAVE`’s mechanisms for overloading operators.

* `-P`*[letter]* — Inform `FWEAVE` about which processor, TEX or LATEX, will be used to process the `.tex` file. The default is ‘`-PT`’ (TEX); to select LATEX, say ‘`-PL`’. The empty command ‘`-P`’ is equivalent to ‘`-PT`’.

`-p`*cmd* — This absorbs a style-file command, which will be processed just before the local style file is read. This option is useful in the initialization file `.fweb`; style-file options common to all jobs can be put there, leaving only things that customize individual runs to the local style file.

`-q` — Turn off translation of RATFOR commands into FORTRAN. (This command is no longer supported.)

* `-r` — Set the global language to RATFOR–77.

`-r9` — Set the global language to RATFOR–90.

`-rb` — Number **do** and **if** blocks in woven RATFOR output. This is helpful when the blocks are highly nested and/or long. The form of this numbering is defined by the `fwebmac` macro `\Wblock`, which by default prints as ‘ // Block 99’.

`-rg`*params* — This option sets the parameters for the decision between the computed **goto** and **if** statements for expanding the RATFOR **switch**. See the discussion of RATFOR for a detailed discussion of this command.

* `-r;` — For RATFOR, make the input driver supply the semicolons automatically and use the “obviously continued” syntax. (It’s recommended that this option not be used. When the programmer supplies the semicolons, RATFOR looks quite close to C.)

`-r/` — In RATFOR, make ’`//`’ denote the start of a short comment instead of concatenation. (One can always use ’`\/`’ for concatenation.)

`-r!` — In RATFOR, make ‘`!`’ denote the start of a short comment instead of the logical `NOT`.

* `-s` — Print statistics about memory usage at the end of the run.
* `-sm`*[nnn]* — As above, but also print each dynamic memory allocation as it is made. By default, only allocations $\geq 10,000$ bytes are shown. However, if you say "`-sm`*nnn*", then allocations $\geq$ *nnn* bytes are displayed. (On personal computers, this command also displays some indication of how much memory is available at the beginning and end of the run.)
* `-t`*ln*[{...}] — Truncate identifiers of language *l* to length *n*. If braces are present, the dots signify a list of forbidden characters that are removed from the identifier before truncation. (For example, '`-tn6{_}`' removes any underscores, then truncates the result to 6 characters.) If this process results in non-unique identifiers, these will be listed.
* `-u`*id* — This undefine command is the inverse of '`-m`'. Saying '`-u`*macroname*' undefines a macro name either predefined by `WEB` or defined earlier on the command line. Use this command with discretion. Don't undefine built-in macros such as `_IF` since other things may also stop working because they use that built-in macro internally.

`-v` — Tells `FTANGLE` to make all comments verbatim; that is, retain them all in the output.

`-W`*letters* — Flag-setting commands that apply only to `FWEAVE`. Here *letters* may be one or more of the following:

- `[` — Turn on special processing of bracketed array indices.
- `f` — Don't print format statements (`@f`) in woven output.
- `l` — As above, but for limbo statements (`@l`).
- `m` — As above, but for macro definitions (`@m`).
- `v` — As above, but for operator overloading (`@v`).
- `w` — As above, but for identifier overloading (`@W`).

* `-w`*[file_name]* — With no argument, do not print "`\input fwebmac.sty`" as the first line of the `.tex` output file. (This may be useful if you have some other macro package that needs for some tricky reason to be loaded before `fwebmac`.) With an argument, print instead "`\input` *file_name*".

* `-x`*[letters]* — Reduce or eliminate printed cross-reference information. The optional letters refer to which piece of information should *not* be printed; they can be one or more of '`c`', '`i`', '`m`', or '`*`', referring respectively to the table of contents, index, module list, or all cross-reference information. The command '`-xi`' means "do not print the index, but print the module list and table of contents". The command '`-xim`' means "print only the table of contents, as does '`-Xc`' (see the next option) or even '`--xc`'. The command '`-x`' is equivalent to '`-x*`', which means "print nothing".

* `-X`*[letters]* — Print selected cross-reference information; equivalent to '`--x`'. To suppress printing of the index and module list, but retain the table of contents, say '`-Xc`'. The default is '`-X*`', which means print everything.

* `-y`*a[a]nnnn* — Allocate dynamic memory. The format is `-y`*aannnnn*, where *aa* stands for a one- or two-character abbreviation for the relevant array and *nnnnn* is the number of elements in the array. See the section below on dynamic memory allocation for more details.

`-Z`*[letters]* — Print default value of style-file parameters. If *letters* are absent, information about all parameters is displayed. If *letters* are present, information about only parameters beginning with *letters* is displayed.

`-z`*[name]* — Specify a new name for the style file, as in `-z`*newstyle*. (The default name is `fweb.sty`.) If no name is given, the null file is assumed.

* `-.` — In FORTRAN or RATFOR, do not recognize "dot constants" such as '`.eq.`'. (Use the modern `WEB` alternatives such as '`==`'.)

* `-\` — Explicitly escape continued strings. Without this option, continued strings must begin in the first column of the next line. With this option, the continuation line may begin with white space followed by an escape character; the string continues after the escape character. In C, the escape character is the backslash; in FORTRAN or RATFOR, it is selected by '`-n&`' or '`-n\`'.

* `-(` — Continue parenthesized strings (arguments of certain built-in commands) with backslashes.

* `-:`*nnnnn* — The command '`-:`*nnnnn* sets the starting automatic statement number for FORTRAN and RATFOR.

`->`*[l=][name]* — Redirect output. The command '`->`' (or its synonym '`-=`') with no argument redirects output to the terminal. The most general form is '`->`*l*`=`*name*', which redirects output for language *l* to the file *name*. See the section on output redirection for more details.

`-=`*[l=][name]* — Redirect output; same as '`->`'. Useful under UNIX, because '`>`' has special significance to the shell.

* `-#` — Turn off commands about line numbers and module names in tangled output.

* `-+` — In FORTRAN or RATFOR, don't allow the compound assignment operators '`++`', '`--`', '`+=`', '`-=`', '`*=`', or '`/=`'.

`-/` — In both FORTRAN and RATFOR, make '`//`' denote the start of a short comment instead of concatenation. (One can always use '`\/`' for concatenation.)

`-!` — In both FORTRAN and RATFOR, make '`!`' denote the start of a short comment instead of the logical `NOT`.

♣ 14.3 Initialization file (.fweb or fweb.ini)

Commonly used options can be placed into an initialization file instead of repeated each time on the command line. The name of this file is determined as follows. On systems that support environment variables (logical names under VMS), if the environment variable FWEB_INI is defined, then that is the name of the ini file. As examples,

```
setenv FWEB_INI .fweb      (UNIX)
define FWEB_INI .fweb      (VMS)
```

This should just be the raw file name, such as .fweb; no path should be specified, since the initialization file must be placed into the user's *home directory* (*not* the current one). If FWEB_INI is not defined, then if the operating system allows a file name to begin with a period the name is ".fweb". (It begins with a dot so it will be invisible by default on UNIX systems.) Otherwise, as on the IBM-PC, the name is instead "fweb.ini". Any argument that can be placed on the command line can be placed into .fweb. (Presently, they cannot be continued across newlines.) For options that on the command line would begin with a hyphen, one can either use the hyphen or, alternatively, a plus sign. Ini options beginning with a plus sign are processed *before* the command-line arguments; ini options beginning with a hyphen are processed *after* the command-line arguments. Entries beginning with anything other than a hypen or plus sign are interpreted as file names, and are processed after any file names found on the command line. Usually one uses the plus sign for ini file options, so that one can override them on the command line if desired.

Comments may be used in the ini file. They follow the same format as those for TEX: namely, anything beginning with a per cent sign is ignored to the end of the line.

Thus, a sample initialization file is

```
% Sample FWEB initialization file.
+PL % LaTeX will be used.
+v % Retain all comments in tangled output.
+n9 % Assume Fortran--90,
+n\% with free-form syntax.
```

♣ 15. ADVANCED FEATURES

Knowledge of the following features is not necessary for simple uses of the WEB system. However, they can be very useful in unusual applications. It is particularly helpful to learn how to use the style file.

♣♣ 15.1 Input and output redirection

As we have explained, the FWEB processors read their input by default from a file with extension .web. FWEAVE's TEX output goes by default into a file with extension .tex; FTANGLE's compilable output goes by default into .sty, .c, .rat, or .for files. (Under UNIX, the latter two are .r and .f.) Occasionally, one may wish to override these defaults. It is most useful to redirect the output. This can be done with the command-line option '->' or its synonym '-='. The bare command '->' redirects all output to the terminal. The command "->*file_name*" redirects all output to the file *file_name*. (For FTANGLE, this kind of redirection is not very useful if you're working with more than one language.) The command "->*l*=*file_name*" redirects output intended only for the file associated with language *l*, where *l* is one of x, c, r, or n. (Use n if you're working with RATFOR; the symbol refers to the output file that is being created, not the language from which things are being produced.) If *file_name* contains the symbol '#', that symbol is expanded into the name of the web file. Thus, for example, if you say "ftangle test ->#.out", all output is redirected into the file test.out.

One should not confuse FWEAVE's redirection with the redirection provided by UNIX shells. The default FWEB processors cannot be used as UNIX filters: by default they neither take their input from the standard input nor write it to the standard output; they take it from, and write it to, files. (Information and error messages are always written to the standard output.) The FWEB command '->' makes FWEB write all output to the standard output; that output could then be redirected again by the UNIX shell. Thus, the UNIX command "ftangle test -= >test.out" puts all output, both information and compilable code, into test.out. If you didn't use the '-=' option, you would just get the information messages in test.out. Incidently, it should now be clear why the command '-=' was introduced as a synonym for '->'. UNIX would interpret the '>' as a redirection command to the shell, and it's cumbersome to type '-\>' when you're in a hurry.

Input redirection is also possible; just replace the web file name by "stdin" or, equivalently, a single hyphen. Thus, the command "ftangle stdin" or equivalently "ftangle -" reads from the standard input (usually the terminal). (The analogous command for FWEAVE doesn't work in a reasonable way, since FWEAVE makes two passes over the input file.) Thus, you can force FTANGLE to be a UNIX filter by saying "ftangle stdin -=". For example, if you say "ftangle <test.in stdin -= >test.out", input will be taken from test.in and written to test.out (along with the information messages). Note that "ftangle <test.in stdin" is equivalent to just "ftangle test.in"; however, "ftangle test.in -= >test.out" is not quite equivalent to "ftangle test.in >test.out". The former puts everything into test.out; the latter just puts the information messages into test.out.

♣ 15.2 Customizing FWEB: The style file fweb.sty

It is possible to customize various aspects of FWEB's behavior. The original motivation was to provide some control over the appearance of the index; the same mechanism can also be used to change the definitions of many of FWEB's '@' commands and to customize other facets of FWEB's behavior. (Customizing the production rules is not presently implemented, but may be in the future.)

The solution that has been adopted follows that for the utility program makeindex [5]: namely, a *style file* can be supplied. If present, the style file is read in after the command line is parsed. This file is located in the directory specified by the environment variable FWEB_STYLE_DIR, if that is defined. Otherwise, it is assumed that the style file is in the current directory. The default name of the style file is fweb.sty; that can be overridden by using the command-line option '-z'. (If you say "-z" with no argument, the style file is null; specify an alternate name by saying "-z*name*".) Its contents consists of a sequence of keywords and keyword arguments, in essentially free-form format. The keywords and arguments are separated by white space, and/or optionally an equals sign. Also, periods in keywords are equivalent to underscores. Thus, the following two lines are equivalent:

One can customize behavior with the style file fweb.sty.

```
lethead_flag 1
lethead.flag = 1
```

(The form with the period is preferred; the intent is to emulate structure references in C.) The arguments can be either single characters (surrounded by single quotes), character strings (surrounded by double quotes), boolean (0 for *NO*, 1 for *YES*), or integers (optionally followed by a trailing L for **long**). As in C, characters can be escaped with a backslash. Thus, for example, to include a double quote and a newline inside a string, say "...\"...\n...".

Comments may be included in the style file. They follow the TEX format: everything from a per cent character to the end of line is ignored.

At present, there can be just one style file. No include command is allowed within a style file. (Someday these restrictions may be removed.)

In the following discussion, the style-file vocabulary is presented in functional groups. For an alphabetical listing of all of the vocabulary entries, see the index entry "style file, vocabulary". Also, to save space, the actual list of keywords associated with each group is not given here, but is rather presented in Appendix M. Here, we content ourselves with a summary of the available features and some examples of their use.

♣ 15.2.1 Customizing the index, etc.

First there are keywords related to customizing the index, module list, and miscellaneous features that are recognized in the style file. (In conjunction with these keywords, you should also study the \Wcon macro in fwebmac.web.) Note that during its processing FWEB uses several temporary files to accumulate the index, list of modules, and the table of contents. By default, these are called INDEX.tex, MODULES.tex, and CONTENTS.tex. When working with more than one file at a time, it is desirable to change these names to include the file name. This can be done by incorporating the character '#' into the style-file field; this character is translated into the root name of the web file. For example, a typical style file might contain the entries

```
index.tex "#.ndx"
modules.tex "#.mds"
contents.tex "#.cts"
```

As an example, here is how one might beautify FWEAVE's index (see Appendix M, "Customizing FWEAVE's index"):

```
% --- SAMPLE STYLE FILE fweb.sty ---

limbo "\\input fweb" % Input fweb.tex automatically just after fwebmac.sty.

% If the source file is test.web, make the name of the index file test.ndx.
index.tex "#.ndx"

% Separate index entries by \letter{...}.  See fweb.tex for \letter.
lethead.prefix "\\letter{"
lethead.suffix "}\n"
lethead.flag -1 % Use lower-case letters.
```

The file fweb.tex contains TEX macros used for producing the woven output of the FWEB sources themselves. In it, one finds the definition of \letter, which in part looks like

```
\def\letter#1{\hbox{...\kern2em--- {\tt #1} ---}\smallskip}
```

With this definition, each index entry will be preceded by a line that contains a construction such as "— a —" to denote the start of each new letter group.

For further information about customizing the module list and table of contents, see App. M, "Customizing FWEAVE's module list" and "Customizing FWEAVE's table of contents."

FWEB can attach cross-reference information as subscripts to various entities such as function names. FWEB will do this by default for some entities, but not for others. To change the defaults, see App. M, "Customizing cross-reference subscripts."

When FWEAVE writes the .tex file, it emits the names of various macros that are used by the FWEB's macro package fwebmac.sty. In some cases it is desirable to have some control over these names without recompiling FWEB. For further information, see App. M, "Overriding or completing definitions in fwebmac.sty."

There are various miscellaneous customization commands for FWEAVE. One important one is the limbo

command, which specifies TEX material that FWEAVE should print at the beginning of the limbo section. For example,

```
limbo "\input fweb"
```

writes the line "\\input fweb" somewhere soon after the first line of the output file, which is generally "\\input fwebmac.sty". This provides a way of including a user's macro file in addition to the default one of FWEB. For more information, see App. M, "Miscellaneous customization commands for FWEAVE."

A variety of parameters are useful only for FTANGLE. These include the cdir_start commands, which specify default text that should be output at the beginning of every compiler directive (initiated by '@?'). Also, one can specify the extension for the output file for each language with the suffix command. For more information, see App. M, "Customizations for FTANGLE."

For a few miscellaneous commands, see App. M, "Miscellaneous customizations for both FTANGLE and FWEAVE.

♣ **15.2.**2 Automatic file name completion

When the '-e' option is in effect, file names that contain no period will be completed automatically by using extensions from the style-file entries described in App. M, "Automatic file-name completion." The entries such as ext.web are strings that contain a blank-delimited list of extensions. Complete file names are created by applying each of these extensions in turn to the original name; the first full name that matches a directory entry is used. Thus, if one says "ext.change = "ch chng"", then if the '-e' option is in effect the command line "ftangle test test" means to tangle test.web with the change file test.ch or, if that does not exist, with the change file test.chng if that exists.

♣♣ **15.2.**3 Custom colors

FWEB supports a *restricted, experimental, and incomplete* color enhancement mode. Color output is not considered to be essential, since FWEB is not interactive and usually prints only a small amount of information to the terminal. Nevertheless, color can be usefully used for emphasis—e.g., for error messages, file names, etc.—and those who have spent considerable amounts of money on a color monitor like to feel that they're getting their money's worth. Unfortunately, color manipulations are not fully standardized. The X Terminal System standardizes one mode of manipulating color, and someday there may be an X version of FWEB. Meanwhile, FWEB can be taught to translate color commands into VT100 escape sequences, so various highlighting modes such as underlining, double intensity, etc., can be used with standard terminal emulators such as xterm.

"Color is obviously a frill. Have fun..."

Here's how it works. The information output by FWEB has been classified into various types; each type can be assigned a different color. For example, referring to the list below, error messages have the type color.error. Each type has a default color that can be overridden in the style file, as in

```
color.error = "red"
```

Ultimately, these colors will correspond to R–G–B triplets. However, in this restricted implementation they are mapped to VT100 escape sequences. There are also defaults for these, or they can be set in the style file, as in

```
color.red = "md mr"
```

(Note the space separating multiple escape sequences.) The abbreviations such as "md" for the escape sequences are those standardized by termcap files; in UNIX, see the man page for termcap, or in emacs see

the `termcap` menu item. Thus, the previous example means that red information fields will be displayed as double intensity reverse video.

Whether any kind of color manipulation is operative is specified by the variable `color.mode`. When `color.mode = 0`, no color manipulations are done. When `color.mode = 1`, a "bilevel", essentially dual-intensity palette is selected, with the default sequences as below. When `color.mode = 2`, true color is supposed to be selected; however, at present, that also corresponds to a particular mapping between colors and escape sequences. Experiment to see what happens.

Color is obviously a frill. Have fun playing with it, but don't forget to do some real work too. For the list of available commands, see App. M, "Colors".

♣♣ 15.2.4 Customizing control codes

Now we describe how to customize `FWEB`'s control codes. *This feature is still experimental, and is recommended for advanced users only.* (No, cancel that; *it's not recommended at all.*) Generally, the default codes are adequate. However, occasionally one might want to change the code that maps to a particular operation. The format is the keyword listed below, followed by a (double-quoted) character string containing all the symbols that you wish to map onto the operation associated with the keyword. For example, if you want to allow the use of any of '`@a`', '`@A`', or '`@j`' to denote the start of the unnamed module, you would use a style file entry

```
begin_code "aAj"
```

In the above, replace the string `"aAj"` by `"j"` if you want to use only '`@j`'.

There may be no overlaps—i.e, one can't map the same character onto two different operations. Also, because of design decisions built into the original `WEB`, some commands with different meanings are all mapped onto the same fundamental operation, such as the commands '`@␣`' and '`@*`'. At present, such commands cannot be remapped.

For more information, see App. M, "Customizing `FWEB`'s control codes."

♣♣ 15.3 Dynamic memory allocation

`FWEB`'s internal buffers and arrays are allocated dynamically (at run time, rather than at compile time). Although default lengths are built into `FWEB` and will be adequate for most applications (in particular, for running the `FWEB` processors on themselves and each other), the facility does exists to change those. (One might need to enlarge certain arrays for an unusually large job, or one might need to shrink things to make them fit on a personal computer.) Dynamic memory allocation is done with the command-line option '`-y`'. This is followed by a one- or two-character abbreviation for the array in question, followed in turn by the number of array elements (not bytes) to allocate, as in '`-yb150000`'. These abbreviations are described more fully in Appendix N.

In general, *you probably shouldn't fiddle around with dynamic memory allocation unless you understand the inner workings of* `FWEB` *in considerable detail.* However, someday one of the processors may complain that it ran out of memory for one reason or another; then you may wish to try overriding the default. Usually the error message will tell you which abbreviation to use as well as its maximum allowed value. (Beware: if it's the macro buffer that overflowed, there's probably something wrong with the macro, or there's a bug in `FTANGLE`.) Sometimes, however, the message will simply tell you that it ran out of memory, but it won't deign to advise you which array to shrink. In this case, proceed as follows. First, determine the default allocations by using the '`-y`' option with no arguments. (Query just one option by saying '`-y`*a*[*a*]'.) Alternatively, one can run a very small test code with the '`-s` option; that reports statistics about the principal buffers at the

end of the run. (If you want even more information, use the option '`-sm`' to obtain a detailed accounting of the dynamic memory allocations as they occur.) Then, attempt to shrink one or more of the largest buffers by using the appropriate '`-y`*a[a]nnnnn*' option; refer to Appendix N.

After one determines an appropriate set of allocations, one typically puts the allocation commands into the ini file `.fweb`, since the same parameters would likely be used for many runs.

♣♣ 15.4 Debugging

There is no denying that the use of named sections (in this discussion we explicitly distinguish between modules and sections: modules are the concatenations of all sections with the same name) significantly helps one to write structured, readable code. In many cases, one can use a named section in place of a function call, with all the readability and helpful cross-reference information of the former but without the overhead of the latter. Unfortunately, the absence of an explicit function call means that the job of debugging is somewhat more annoying. One might want to set a breakpoint at the beginning of the section. Without a function call one must set a breakpoint at a line number, but this is the line number of the output file as understood by the compiler, not the `WEB` file; that line number may be difficult to determine.

`FWEB` provides a partial, experimental solution to this problem. If the user defines *from the command line* the *breakpoint* macro `_BP(num,name)`, then `FTANGLE` will insert at the beginning of every section that begins with a left brace the expansion of `_BP`, with *num* replace by the section number and *name* replaced with the (possibly truncated) name (surrounded by quotes) of the module to which the section belongs. Thus, the user has complete flexibility to develop his own tracing/breakpointing system. For example, suppose the C user issues the command-line option '`-m_BP(num,name)=trap(num,name);`'. Then each section that begins with a left brace will begin with a call to the function *trap*, which the user can define as he pleases. He might call another function directly from the debugger to add or remove a section number from a list of breakpoints, then arrange to have *trap* pause when only the turned-on sections are encountered. Exactly how this would work depends in some detail on the debugger available, but the philosophy itself is quite general. Casual users of `WEB` will probably not bother with all this, but for those who work with large codes this feature may be essential.

C and Ratfor users will have no trouble understanding the restriction to sections beginning with a left brace; the debugging call needs to be tucked away inside a compound statement so that it does not inadvertently appear to be a single statement following a loop and thereby change the program logic. Fortran users ordinarily do not need to use braces; however, if they do not, this breakpointing mechanism will not function. Happily, straight Fortran will expand braces just fine—it just ignores them on output—so things will work consistently in all languages if all sections that are to be interpreted as "call-less functions" are delimited by braces.

However, C users will object that one can't always insert an executable statement immediately after a left brace, because a declaration statement might be coming up. The solution to this is to replace the left brace by '`@{`'. This expands into a left brace, but also prevents the default insertion. To insert a breakpoint later in such a section, use the command '`@b`' at the desired place. For example,

```
@ An example of inserting breakpoint commands for debugging.
@<Test@>=
@{
int i;

@b
executable code;
@b @% Note that one can insert a breakpoint at more than one place.
}
```

For an example of section breakpointing functions suitable for use with C programs, see the example file `breakpt.web`. It would be easy to adapt this scheme for FORTRAN programs as well. That demo also illustrates the use of the built-in functions `_SECTIONS` and `_MODULES`.

16. USAGE TIPS and SUGGESTIONS

In this section we collect various tips and suggestions to help one make full use of the `WEB` system. *(There's more to come here!)*

16.1 Converting an existing code to `FWEB`

In summary, to convert an existing code to `FWEB`, you should do the following. (The following simple procedure assumes that you put all the subroutines into the unnamed module. However, other more elaborate schemes are possible.)

1. Place invisible commentary about the author, version, etc. at the beginning of the source file by bracketing it with '`@z`...`@x`'. The '`@z`' must be the first two characters of the file.

2. Next, set the language by including a command such as '`@n`'.

3. Place an '`@a`' command before each program unit (e.g., main **program**, **subroutine**, or **function**).

4. Before each '`@a`', place an '`@*`' or '`@␣`' command, followed by TEX documentation about that particular section of code.

5. If you have program units longer than about twelve lines, either make them function calls, if you can afford the overhead and can impart sufficient information via the function name, or break them up into shorter fragments by using named modules. Insert the command '`@<`*Name of module*`@>`' in place of the fragment you're replacing, then put that fragment somewhere else, prefaced by '`@␣`' and '`@<`*Name of module*`@>=`'.

6. Make sure your comments are valid TEX. (You can't have things like raw underscores or dollar signs in comments, since those cause TEX to take special actions.)

7. Beautify and clarify your documentation by using code mode (enclosing stuff between vertical bars) liberally within your TEX.

8. After you've seen the woven output, you may need to go back and format a few identifiers or section names so that `FWEAVE` understands them properly, or you may need to insert some pseudo-semicolons, pseudo-expressions, or pseudo-colons.

9. Consider using the built-in macro preprocessor to make your code more readable—for example, replace raw numerical constants by symbolic names.

10. If you are a FORTRAN user, for ultimate readability consider converting to RATFOR. The initial annoyance is getting rid of column 6 continuations. With the aid of a good editor, this can be done simply. For example, in `EMACS` one can replace the regular expression [carriage return, five spaces, something not equal to space, tab, or 0] with [backslash, carriage return, six spaces]:

```
M-x replace-regexp RET
C-q C-j ␣␣␣␣␣[^␣tab 0]RET
```

```
\\C-q C-j ␣␣␣␣␣␣RET
```

Get rid of the keywords such as **then** or **end if** in favor of braces. Change singly-quoted character strings to doubly-quoted ones.

16.2 Programming tips and other suggestions

This section will be enlarged in the future! Meanwhile, please feel free to contact krommes@princeton.edu for help and advice, and to suggest items to include here.

1. Periodically check lyman.pppl.gov:/pub/fweb/READ_ME for bug reports and other news.

2. Most options in .fweb should begin with '+' so they can be overridden by command-line options for the job itself.

3. Put standard command-line options into .fweb. Also put there standard style parameters—e.g.,

```
+pindex.tex "#.ndx"
+pmodules.tex "#.mds"
+pcontents.tex "#.cts"
```

4. Learn how to use the style file.

5. Use the info options '-D', '-y', and '-Z' to find out about various internal FWEB tables (reserved words, memory allocations, and style-file parameters).

6. Begin all FWEB sources with invisible commentary bracketed by @z...@x.

7. Always include an explicit language-setting command in the limbo section.

8. Keep sections quite short. Knuth suggests a dozen lines. That's quite hard to achieve sometimes, but almost never should a section be more than a page long.

9. It's easy to define macros from the command line to expedite conditional preprocessing.

10. Use the preprocessor construction @#if 0...@#endif to comment out unwanted code.

11. For logical operations with the preprocessor, use '||', not '|'.

12. It's conventional to identify the ends of long preprocessor constructions as follows:

```
@#if A
.
.
@#endif // |A|
```

13. To debug an errant WEB macro, use the built-in function _DUMPDEF.

14. Use '@?' for compiler directives ('@!' is obsolete). Use the style-file parameters cdir_start.*l* to specify information that will be written out at the beginning of the line.

15. Stick to the standard FWEB commenting style /*...*/ or //.... Don't use alternatives such as FORTRAN's column 1 convention; these may not work or may not be supported someday.

16. The meta-comment feature `@(...@)` provides a poor-man's alignment feature. But that's not in the spirit of TEX; learn to use `\halign` or the LATEX alternatives.

17. In FORTRAN, use `#:0` to declare readable alphabetic statement labels.

18. When mixing languages, define the language of a module at the highest possible level—e.g., in the unamed module, not after '`@<...@>=`'.

17. PRESENT STATUS and the FUTURE

The goal of `FWEB` v. 1.0 was to achieve functionality, particularly regarding the macro preprocessor and RATFOR. Having done so, it is now time to rework certain parts of the code in order to achieve portability, optimal speed, and/or eloquence. This will be done as time permits. Note that the author works with large codes written with `FWEB`, so there is a powerful incentive to maintain and improve it. The best way to ensure that `FWEB` evolves as you would like it is to use e-mail liberally to make suggestions, report bugs, etc.—send them to `krommes@princeton.edu`.

18. ACKNOWLEDGEMENTS

FWEB has evolved from the CWEB code written by Silvio Levy of Princeton University, who graciously provided the author with version 0.5 of the CWEB source code. Henry Greenside greatly fostered the author's initial appreciation of software tools, particularly RATFOR. The author is extremely grateful to Charles Karney for providing many incisive suggestions about FWEB and enormous help with all aspects of TEX and computing in general. A number of beta-testers suffered patiently through the initial debugging stage and made many valuable suggestions (usually not obscene): Cris Barnes, John Bowman, Charles Karney, and Chang–Bae Kim. (Patience of some of those beta-testers flagged somewhat when FWEB stopped working a few days before a major meeting of the American Physical Society, but that is perhaps understandable.) Arnold Kritz generously provided both precious time and the use of his excellent, super-enhanced IBM-PC system for developing the original PC version. Barbara Kritz made a seminal contribution by locating the key PC disk on Arnold's desk. Thorsten Ohl provided expert, trans-Atlantic assistance and advice in debugging first the PC version, then the ASCII translations necessary for non-ASCII machines. Many users have reported bugs and provided suggestions, all of which are greatly appreciated. Charles Karney and Bart Childs contributed nice demos of the use of FORTRAN FWEB (see adj.web and series.web, respectively). Bart Childs furnished a variety of insightful remarks about literate programming that influenced aspects of the design of FWEB. Tom McCurdy and Clarissa Wilson furnished the short pedagogical introduction Intro.tex to WEB programming that is included with the FWEB release.

"Many users have reported bugs and provided suggestions, all of which are greatly appreciated."

This work was supported by U.S. D.o.E. contract DE–AC02–76–CHO–3073.

19. REFERENCES

[1] D. E. Knuth, *Literate Programming* (Center for the Study of Language and Information, Leland Standard Junior University, 1992).

[2] J. Bowman, Ph.D. thesis, Princeton University, 1992.

[3] Marcus Speh, marcus@x4u.desy.de. For further information, see the files in the official FWEB public area: lyman.pppl.gov:/pub/fweb/faq.

[4] B. W. Kernighan and P. J. Plauger, *Software Tools* (Addison-Wesley, Reading, Massachusetts, 1976).

[5] The makeindex utility is available from ftp.math.utah.edu in /pub/tex/pub/makeindex/2-11. The extension .trz is equivalent to .tar.Z.

20. APPENDICES

The basic ideas of WEB can be understood most easily by looking at examples of "real" programs. Appendix A shows the FWEB input for the simple FORTRAN to FWEB demo program f_to_web; Appendix B shows the corresponding TEX code output by FWEAVE; Appendix C shows excerpts from the finished woven product typeset from f_to_web.tex; and Appendix D shows the corresponding code output by FTANGLE. Appendix E shows the woven output from the sample f90_cpp.web, which mixes C++ and FORTRAN–90 code.

The complete webs for FWEAVE and FTANGLE are available in the files fweave.web and ftangle.web. It's very instructive to study these programs, since WEAVE and TANGLE contain several interesting aspects, and since an attempt has been made in these codes to evolve a style of programming that makes good use of the WEB language.

Appendix F is the 'fwebmac' file that sets TEX up to accept the output of FWEAVE; Appendix G discusses how to use some of its macros to vary the output formats; Appendix H summarizes how this FWEB code differs from its immediate predecessor CWEB. Appendix I is a question-and-answer session about FWEB, and Appendix J discusses what needs to be done when FWEAVE and FTANGLE are installed in a new operating environment.

Appendix K lists and explains the error messages produced by FWEB. Finally, Appendix L contains a summary of all the FWEB syntax, commands, and idiosyncracies.

O, what a tangled web we weave
When first we practise to deceive!
—SIR WALTER SCOTT, *Marmion* 6:17 (1808)

O, what a tangled WEB we weave
When TEX we practise to conceive!
—RICHARD PALAIS (1982)

20.1 APPENDIX A: A SIMPLE DEMO PROGRAM: f_to_web.web

Appendices A–D use a simple FORTRAN–77 program to demonstrate the various facets of the WEB system. The demonstration begins with the file f_to_web.src, provided with the FWEB distribution but not shown here. This appendix is a verbatim listing of a web file based on f_to_web.src. Appendix B shows the output f_to_web.tex from FWEAVE, Appendix C shows part of the finished typeset product, and Appendix D is a verbatim listing of the output from FTANGLE.

```
@z --- f_to_web.web ---

This file is part of FWEB.  It and its various pieces of processed output are
included into the user manual fwebman.tex.

Author:  J. A. Krommes
Version: 1.23
Date:    April 1, 1992

@x---------------------------------------------------------------------------

@n/ @% Set the language to Fortran, and allow short comments.

\def\title{--- F\_TO\_WEB ---}

@* INTRODUCTION. This demo shows you how to convert the file
\.{f\_to\_web.src} (look at that file with your editor now) into a valid
\.{FWEB} file, namely this file \.{f\_to\_web.web}. Each subroutine should
be placed into a separate module, begun with~‘\.{@@*}’ or~‘\.{@@\ }’. After
those symbols, you should explain what the subroutine does, using \TeX\ in
all its glory. Then follows the code, introduced by~‘\.{@@a}’ or
‘\.{@@<...@@>=}’. Code sections should be short---about 12 lines, according
to Knuth. If they’re too long, break them up into named fragments that are
explained separately. For new code, comments should be C-style---namely,
‘‘\.{/*...*/}’’ or ‘‘\.{//...}’’.  (In order to use the latter form, you
must use the command-line option~‘\.{-n/}’, and then you must use the
symbol~‘\.{\\/}’ for concatenation.)

Before proceeding, let us note that standard Fortran is {\it not}
recommended for new code. Use the Ratfor mode instead. Standard Fortran
mode is intended primarily to support conversion of existing code.

Notice how forward references to named modules and function names are
handled in the woven output.
@a
        program main
        @<Common blocks@>

/* Initialize values.  Long comments should be done in standard
C style and may be continued across lines.  */
        x = -3.14159e-11
        i = 1

        call see  // Print results.  (Short comments can be done like this.)
```

```
        end

@ Code fragments can be defined anywhere, even after they are used.

@f @<Com...@> common /* Use a format statement to tell \.{WEAVE} how to handle
                 this module name. */

@<Com...@>=
        integer i
        real x
        common/test/ x,i

@ Notice how the common block information is handled. You don't need to
include such stuff from a separate file.

@M NFMT #:0 /* This preprocessor command is a handy way of replacing numeric
      statement labels by symbolic ones. Numeric statement labels will
      never have to be used. */

@a
        subroutine see
        @<Com...@> /* You can abbreviate the name if it has already
 appeared in full. */

        write(6,NFMT) x,i
NFMT:   format(' x = ',1pe10.2,', i = ',i2)
        return
        end

@ In fact, the Ratfor language is recommended for new Fortran codes. It's
best to do everything in Ratfor, but you can also work on a
module-by-module basis. Here's an example. Examine the listing of the
output file in Appendix~D to see how this is translated into standard
Fortran.

@a
@r/ @%* Set the language to Ratfor-77, for this section only. */
integer function f(a,b,n)
        integer n;
        real a(0:n-1),b(0:n-1);
{
integer k;

/* You can (and should) use a |do| loop for the following, but the |for|
construction is more flexible in general, so we use it to demonstrate. */
for(k=0; k<n; k++)
        {
        a(k) = k;
        b(k) = k^2; /* In Ratfor and Fortran, you can use pretty
alternatives for archaic Fortran constructions such as~\.{.lt.} or~\.{**}. */
        }
```

```
return n;  // It's easy to return values from functions.
}

@* INDEX.
```

20.2 APPENDIX B: WOVEN OUTPUT f_to_web.tex

The following output f_to_web.tex results from the command "fweave f_to_web".

```
% FWEAVE v1.30 (May 15, 1993)

\input fwebmac.sty

\Wbegin[]{article}{1em}{1em}{f_to_web.cts}{{\&}{\|}{\\}{\\}{\\}{\\}{\@}{\.}{\.}}

% --- Beginning of user's limbo section ---



\def\title{--- F\_TO\_WEB ---}


\WN1.  INTRODUCTION. This demo shows you how to convert the file
\.{f\_to\_web.src} (look at that file with your editor now) into a valid
\.{FWEB} file, namely this file \.{f\_to\_web.web}. Each subroutine should
be placed into a separate module, begun with~``\.{@*}'' or~``\.{@\ }''. After
those symbols, you should explain what the subroutine does, using \TeX\ in
all its glory. Then follows the code, introduced by~``\.{@a}'' or
``\.{@<...@>=}''. Code sections should be short---about 12 lines, according
to Knuth. If they're too long, break them up into named fragments that are
explained separately. For new code, comments should be C-style---namely,
``\.{/*...*/}'' or ``\.{//...}''.  (In order to use the latter form, you
must use the command-line option~``\.{-n/}'', and then you must use the
symbol~``\.{\\/}'' for concatenation.)

Before proceeding, let us note that standard Fortran is {\it not}
recommended for new code. Use the Ratfor mode instead. Standard Fortran
mode is intended primarily to support conversion of existing code.

Notice how forward references to named modules and function names are
handled in the woven output.
\WY\WP \&{program} \1\\{main}\WIN1{0}\2\1\6
\WX2:Common blocks\X \X\1\2\7
\WC{ Initialize values.  Long comments should be done in standard    C style
and may be continued across lines.  }\7
${}\|x={-}\WO{3.14159\^E-11}$\6
${}\|i=\WO{1}{}$\7
\&{call} \\{see}\WIN1{3}\5
\Wc{ Print results.  (Short comments can be done like this.)}\2\7
\&{end}\WY\par
```

```
\fi % End of module 1

\WM2. Code fragments can be defined anywhere, even after they are used.

\WY\WP\WF::\WX2:Common blocks\X \X\ \\{common}\5
\WC{ Use a format statement to tell \.{WEAVE} how to handle this module name. }%
\WY\par
\WY\WP\4\4\WX2:Common blocks\X \X${}\WS{}$\6
\&{integer} \1\|i\2\6
\&{real} \1\|x\2\6
\&{common} \1 ${}{/}\\{test}{/}$ \|x, \|i\2\WY\par
\WU sections~1 and~3.\fi % End of module 2

\WM3. Notice how the common block information is handled. You don't need to
include such stuff from a separate file.

\WY\WP\WMD$\\{NFMT}$\5
\NC $\WO{0}{}$\5
\WC{ This preprocessor command is a handy way of replacing numeric statement
labels by symbolic ones. Numeric statement labels will never have to be used. }%
\WY\par
\WY\WP \&{subroutine} \1\\{see}\WIN1{0}\2\1\6
\WX2:Common blocks\X \X\1\2\5
\WC{ You can abbreviate the name if it has already     appeared in full. }\7
${}\&{write}\,(\WO{6},\39\\{NFMT})$ \|x, \|i\6
\llap{\\{NFMT}\Colon\  }${}\&{format}\,(\.{'\ x\ =\ '},\39\WO{1}\\{pe10.2},\39%
\.{'\1\ i\ =\ '},\39\\{i2})$ ;\6
\&{return}\2\6
\&{end}\WY\par
\fi % End of module 3

\WM4. In fact, the Ratfor language is recommended for new Fortran codes. It's
best to do everything in Ratfor, but you can also work on a
module-by-module basis. Here's an example. Examine the listing of the
output file in Appendix~D to see how this is translated into standard
Fortran.

\WY\WP \LANGUAGE{R}\&{integer} \&{function} \1\|f\WIN1{0}(\|a,\39\|b,\39\|n)\2%
\1\1\6
\&{integer} \1\|n;\2\2\2\1\1\6
\&{real} \1\|a${}(\WO{0}:\|n-\WO{1}),$ \|b${}(\WO{0}:\|n-\WO{1});$\2\2\2\1\6
${}\{$\6
\&{integer} \1\|k;\2\7
\5
\WC{ You can (and should) use a \WCD{ \&{do}} loop for the following, but the %
\WCD{ \&{for}} construction is more flexible in general, so we use it to
demonstrate. }\7
${}\&{for}\,(\|k=\WO{0};$ ${}\|k<\|n;$ ${}\|k\PP)$ \1\6
${}\{$\6
${}\|a(\|k)=\|k;$\6
${}\|b(\|k)=\|k\EE{\WO{2}};{}$\5
\WC{ In Ratfor and Fortran, you can use pretty alternatives for archaic Fortran
```

```
constructions such as~\.{.lt.} or~\.{**}. }\6
${}\}$\2\7
\&{return} \|n;\5
\Wc{ It's easy to return values from functions.}\6
${}\}$\2\WY\par
\fi % End of module 4

\WN5.  INDEX.
\fi % End of module 5

\input f_to_web.ndx
\input f_to_web.mds

\Winfo{"fweave ./f\_to\_web -zdemos.sty -=f\_to\_web.tex"}  {"./f\_to%
\_web.web"} {(none)}
 {FORTRAN}

\Wcon
```

20.3 APPENDIX C: The FINISHED PRODUCT f_to_web

Here is part of the typeset documentation produced by saying "`tex f_to_web.tex`" and printing the output `f_to_web.dvi`.

---

1. INTRODUCTION. This demo shows you how to convert the file `f_to_web.src` (look at that file with your editor now) into a valid `FWEB` file, namely this file `f_to_web.web`. Each subroutine should be placed into a separate module, begun with '`@*`' or '`@␣`'. After those symbols, you should explain what the subroutine does, using TEX in all its glory. Then follows the code, introduced by '`@a`' or '`@<...@>=`'. Code sections should be short—about 12 lines, according to Knuth. If they're too long, break them up into named fragments that are explained separately. For new code, comments should be C-style—namely, "`/*...*/`" or "`//...`". (In order to use the latter form, you must use the command-line option '`-n/`', and then you must use the symbol '`\/`' for concatenation.)

Before proceeding, let us note that standard Fortran is *not* recommended for new code. Use the Ratfor mode instead. Standard Fortran mode is intended primarily to support conversion of existing code.

Notice how forward references to named modules and function names are handled in the woven output.

program *main*•
⟨ Common blocks 2 ⟩

/∗ Initialize values. Long comments should be done in standard C style and may be continued across lines. ∗/

$x = -3.14159 \cdot 10^{-11}$
$i = 1$

call *see*$_3$ // Print results. (Short comments can be done like this.)

end

2. Code fragments can be defined anywhere, even after they are used.

```
@f ⟨ Common blocks 2 ⟩ common
        /* Use a format statement to tell WEAVE how to handle this module name. */

⟨ Common blocks 2 ⟩ ≡
    integer i
    real x
    common /test/ x, i
```

This code is used in sections 1 and 3.

3. Notice how the common block information is handled. You don't need to include such stuff from a separate file.

```
@M NFMT #:0    /* This preprocessor command is a handy way of replacing numeric statement labels
        by symbolic ones. Numeric statement labels will never have to be used. */

    subroutine see•
      ⟨ Common blocks 2 ⟩    /* You can abbreviate the name if it has already appeared in full. */
      write (6, NFMT) x, i
NFMT: format ('␣x␣=␣', 1pe10.2, ',␣i␣=␣', i2) ;
      return
    end
```

4. In fact, the Ratfor language is recommended for new Fortran codes. It's best to do everything in Ratfor, but you can also work on a module-by-module basis. Here's an example. Examine the listing of the output file in Appendix D to see how this is translated into standard Fortran.

```
@Lr:    integer function f•(a,b,n)
          integer n;
          real a(0 : n − 1), b(0 : n − 1);
        {
        integer k;

          /∗ You can (and should) use a do loop for the following, but the for construction is more flexible
           in general, so we use it to demonstrate. ∗/

        for (k = 0; k < n; k++)
          {
          a(k) = k;
          b(k) = k^2;     /∗ In Ratfor and Fortran, you can use pretty alternatives for archaic Fortran
               constructions such as .lt. or **. ∗/
          }

        return n;     // It's easy to return values from functions.
        }
```

5. INDEX.

(Index and remaining material skipped.)

(Page break skipped.)

20.4 APPENDIX D: TANGLED OUTPUT `f_to_web.f`

The following output results from the command "`ftangle f_to_web`".

```
C FTANGLE v1.30, created with UNIX on "Tuesday, May 11, 1993 at 10:55."
C COMMAND LINE: "ftangle ./f_to_web -zdemos.sty -=f_to_web.f"
C RUN TIME: "Thursday, June 10, 1993 at 12:39."
C WEB FILE:    "./f_to_web.web"
C CHANGE FILE: (none)
C* 1: *
*line 35 "./f_to_web.web"
      program main
C* 2: *
*line 54 "./f_to_web.web"
      integer i
      real x
      common/test/x,i
C* :2 *
*line 37 "./f_to_web.web"
```

```
C* Initialize values.  Long comments should be done in standard    C style and m
Cay be continued across lines.
      x=-3.14159e-11
      i=1
      call see
C Print results.  (Short comments can be done like this.)
      end
C* :1 *
C* 3: *
*line 66 "./f_to_web.web"
      subroutine see
C* 2: *
*line 54 "./f_to_web.web"
      integer i
      real x
      common/test/x,i
C* :2 *
*line 69 "./f_to_web.web"

C You can abbreviate the name if it has already     appeared in full.
      write(6,90000)x,i
90000  format(' x = ',1pe10.2,', i = ',i2)
      return
      end
C* :3 *
C* 4: *
*line 82 "./f_to_web.web"
C* 4: *
*line 83 "./f_to_web.web"
      integer function f(a,b,n)

        integer n
        real a(0:n-1),b(0:n-1)

        integer k

C* You can (and should) use a |do| loop for the following, but the |for|
Cconstruction is more flexible in general, so we use it to demonstrate.
        CONTINUE
C ---  "for(k=0; k<n; k++)" ---
        k=0
90001   IF(k.LT.n)THEN
          a(k)=k
          b(k)=k**2
C* In Ratfor and Fortran, you can use pretty
Calternatives for archaic Fortran constructions such as~\.{.lt.} or~\.{**}.
C --- Reinitialization of "for(k=0; k<n; k++)" ---
          k=k+1
          GOTO 90001
        ENDIF

        CONTINUE
```

```
C ---  "return n" ---
        f=n
        RETURN
C/ It's easy to return values from functions.
      END

C* :4 *
```

20.5 APPENDIX E: EXAMPLE of C++ and Ratfor--90 CODE

This example demonstrates some sample RATFOR–90 and C++ code that involves operator overloading. Note how the '@v' command is used to change the default printed form of the overloaded operators.

---

1. INTRODUCTION. This example demonstrates operator overloading in both Ratfor–90 and C++. It also shows how brackets may be used instead of parentheses for FORTRAN array indices.

```
    @l "\\let\\WARRAY\\WSUB"    // Subscript FORTRAN indices.

@Lc++:  ⟨ Ratfor–90 2 ⟩
        ⟨ C++ 5 ⟩C++
```

2. The following example is excerpted from the ANSI Draft S8, Version 112 for the Fortran–90 language.

```
@v .IN. "\\in" <    /* Make .IN. display as '∈' and be treated as a relational operator. */
@v ≤ "\\subset" ≤   /* Make .LE. display as '⊂' and be treated as a relational operator. */
```

⟨ Ratfor–90 2 ⟩ ≡

```
module integer_sets•
  {
  integer, parameter :: max_set_card = 200;

  type set
    {
  private:
    integer card;
    integer element max_set_card;
    };

  interface operator (∈ ≡ .IN.)
    {
    module procedure element;
    };

  interface operator (⊂ ≡ .LE.)
    {
    module procedure subset;
    };

  ⟨ Union function 3 ⟩
  ⟨ Subset function 4 ⟩
  }
```

This code is used in section 1.

3. This function uses structure elements. We overload the element operator '`%`' to make the resulting code look more like C.

```
@v % "." .    // Make % print as '.' and also be treated as '.'.
```

⟨ Union function 3 ⟩ ≡

```
function union•(A,B)
    type(set) A, B;
  {
  type(set) UNION;
  integer j;

  UNION = A;
  do j = 1, B.card;
    if (¬(B.element_j ∈ A))
      if (UNION.card < max_set_card)
        {
        UNION.card += 1;
        UNION.element_UNION.card = B.element_j;
        }
      else ;    // Maximum set size exceeded...
  }
```

This code is used in section 2.

4. We claim that this is much more visually appealing than a raw listing.

⟨ Subset function 4 ⟩ ≡

```
logical function subset•(A,B)
    type(set) A, B;
  {
  integer i;

  subset• = A.card ⊂ B.card;      /* In the source, this is "subset␣=␣A%card␣<=␣B%card;" */
  if (¬subset•) return;

  do i = 1, A.card;
    subset• = subset• ∧ (A.element_i ∈ B);
  }
```

This code is used in section 2.

5. Here is a short example of operator overloading in C++. Note that since the global language is RATFOR–90, we must explicitly insert a language command in the following definition section in order that the star be overloaded in the proper language.

```
@v^C++ * "\\times" *
```

⟨ C++ 5 ⟩^C++ ≡

```
class complex
  {
  double re, im;

public:
  complex(double r, double i)
    {
    re = r; im = i;
    }
  friend complex operator +(complex, complex);
  friend complex operator ×(complex, complex);
  };

z = x × y;     /* An example of a statement using the overloading multiplication operator. */
```

This code is used in section 1.

6. INDEX.

(Index and remaining material skipped.)

(Page break skipped.)

20.6 APPENDIX F: The FWEBMAC MACROS

If this appendix is not here, you can say `\typesetfwebmactrue` near line 39 of `fwebman.tex` to create a larger, self-contained manual. Alternatively, you can weave `fwebmac.web` separately.

20.7 APPENDIX G: HOW TO USE FWEB MACROS

The macros in `fwebmac.sty` (produced by tangling `fwebmac.web`) make it possible to produce a variety of formats without editing the output of `FWEAVE`, and the purpose of this appendix is to explain some of the possibilities. *(NOTE: Although* `FWEB` *now works with* LaTeX*, the original discussion in this appendix was for Plain* TeX*, and it may have not been completely updated yet.)*

Before proceeding, it is important to stress that it is sometimes not possible to satisfactorily change the appearance of the woven output just by modifying macros in `fwebmac`. Occasional difficulties may arise since certain TeX commands are hard-coded into `FWEAVE`'s output routines. In unusual situations, one may need to recompile `FWEAVE` to achieve the desired effect. (But don't do this unless you really have to; it can be an endless sink of time.) However, many features can be customized via the style file. (Please feel free to suggest additional features that should be customizable.)

20.7.1 Additional fonts

Several fonts have been declared in addition to the standard fonts of PLAIN format: You can say '{\SC STUFF}' to get STUFF in small caps, or '{\Csc Stuff}' to get Stuff; and you can select the largish fonts \titlefont and \ttitlefont in the title of your document, where \titlefont gives one **titlefont** and \ttitlefont is a typewriter style of type giving one `ttitlefont`.

20.7.2 Typesetting comments

Comments are typeset in the font \cmntfont, which is \let to \tenrm by default. You can redefine \cmntfont in the limbo section; for example, "\let\cmntfont\Csc".

20.7.3 Typesetting identifiers

When you mention an identifier in TEX text, you normally call it '|identifier|'. But if that identifier is not a reserved word, you can also say '\\{identifier}'. The output will look the same in both cases, namely '*identifier*', but the second alternative doesn't put *identifier* into the index, since it bypasses WEAVE's translation from code mode. For one-character identifiers, you should say '\|i' to get '*i*'. Also, for a reserved word you can say '\&{reserved}' to get '**reserved**'. If you're talking about an intrinsic function, you can say '\@@{intrinsic}' to get '***intrinsic***'. (Note that the macro name itself is '\@'.)

Note that WEB identifiers may contain the characters '$' and '_'. These must be preceded by a backslash when using them with the macros \\, \&, and \@. This is not necessary when enclosing such identifiers between vertical bars; WEB handles the escaping for you. Thus, say "|id_code|" but "\\{id_code}".

LATEX users (see the section below on Using LATEX) will object that '\\' means something very different to them. Actually, the true situation is slightly more complicated than that described in the last paragraph. In fwebmac, the macro that formats an ordinary identifier is really called '\Wid', not '\\'. The macro '\\' becomes associated with '\Wid' because the style-file entry 'format.identifier' has '\\' as its default value. That value is transmitted to TEX or LATEX via the '\Wbegin' macro that is emitted automatically just after the "\input fwebmac.sty" command at the beginning of the output file. (Study a sample of woven output, for example Appendix B.) Similar equivalences are set up for the other formatting macros according to the following table:

Type of argument	fwebmac *macro*	*Style-file entry*	*Default value* -PT (-PL)
character string	\Wtypewriter	format.typewriter	\.
reserved word	\Wreserved	format.reserved	\&
single-character identifier	\Wshort	format.short_identifier	|
ordinary identifier	\Wid	format.identifier	\\ (\>)
outer macro	\WidD	format.outer_macro	\\ (\>)
WEB macro	\WidM	format.WEB_macro	\\ (\>)
intrinsic function	\Wintrinsic	format.intrinsic	\@
FORTRAN keyword	\Wkeyword	format.keyword	\.

[Note that when the LATEX processor is specified ('-PL' option), a few of the defaults are changed for convenience.] Note that by default macro names are treated the same way as ordinary identifiers. To cause macro names to be formatted in a distinctive way, you must use the style-file entries format.outer_macro and/or format.WEB_macro. (If more than one of these has the same value, the equivalences to the fundamental definitions in fwebmac.sty are made in the order format.WEB_macro, format.outer_macro, then format.identifier.) However, you must *also* supply a new definition for the fwebmac macros \WidD

and/or `\WidM`, since by default these are merely `\let` equal to `\Wid`. You could introduce such new definitions by placing them into a file that is `\input` automatically at the beginning of each run (see the style-file entry `limbo`), by creating a new personal version of `fwebmac.sty` (use a change file in conjunction with `fwebmac.web`), or by using the '`@l`' command to place the definition directly into the limbo section of your code.

20.7.4 Typewriter type

To get typewriter-like type, as when referring to '`WEB`', you can use the '`\.`' macro (e.g., '`\.{FWEB}`'). In the argument to this macro you should insert an additional backslash before the following characters enclosed by double quotes: "␣`\#%$^{}~&_`". A '`\␣`' here will result in the visible space symbol; to get an invisible space following a control sequence you can say '`{␣}`'.

The original form of the '`\.`' macro surrounded the string with an `\hbox`, which unfortunately prevented very long strings from being broken across lines. The present `FWEB` form does *not* surround the string with an `\hbox` and allows strings to be broken, either after commas or every so many characters. To accomplish this, `FWEAVE` inserts into strings special control sequences that are treated essentially like discretionary hyphens. Automatically broken strings are marked by a backslash at the end of the line.

Actually, the `fwebmac` macro for typewriter type is called '`\Wtypewriter`'. Its equivalence to '`\.`' is established by the same mechanism as described above for formatting identifiers, via the style-file entry '`format.typewriter`'.

20.7.5 Page dimensions

The three control sequences `\pagewidth`, `\pageheight`, and `\fullpageheight` can be redefined in the limbo section at the beginning of your `WEB` file to change the dimensions of each page. The standard settings

```
\pagewidth=6.5in
\pageheight=8.7in
\fullpageheight=9in
```

were used to prepare the present report; `\fullpageheight` is `\pageheight` plus room for the additional heading and page numbers at the top of each page. If you change any of these quantities, you should call the macro `\setpage` immediately after making the change.

20.7.6 Page heads

The macro `\identicalpageheads` is normally false, which means that by default the page numbers and module numbers will alternate left and right on even- and odd-numbered pages. If you want all the page heads to be formatted identically, say "`\identicalpageheadstrue`".

20.7.7 Shifting pages left or right

The `\pageshift` macro defines an amount by which right-hand pages (i.e., odd-numbered pages) are shifted right with respect to left-hand (even-numbered) ones. By adjusting this amount you may be able to get two-sided output in which the page numbers line up on opposite sides of each sheet.

20.7.8 Page title

The `\Wtitle` macro will appear at the top of each page in small caps. This macro is null by default, but you can define it in the limbo section.

20.7.9 Page numbering

The first page usually is number 1; if you want some other starting page, just set `\pageno` to the desired number—e.g., '`\pageno=16`'.

20.7.10 Paragraph breaks

By default, paragraph breaks in TEX mode are spaced out with an extra blank line. If you don't want that, say '`\pardimen=0pt`'.

20.7.11 Magnifying the output

If you want your output to be bigger than usual, use `\magnify` instead of `\magnification`; say, for example, '`\magnify{\magstep1}`'.

20.7.12 Table of contents

The macro `\iftitle` will suppress the header line if it is defined by '`\titletrue`'. The normal value is `\titlefalse` except for the table of contents; thus, the contents page is usually unnumbered. If your program is so long that the table of contents doesn't fit on a single page, or if you want a number to appear on the contents page, you should reset `\pageno` when you begin the table of contents.

Two macros are provided to give flexibility to the table of contents: `\topofcontents` is invoked just before the contents info is read, and `\botofcontents` is invoked just after. For example, Appendix D of Knuth's original manual was produced with the following definitions:

```
\def\topofcontents{\null\vfill
  \titlefalse % include headline on the contents page
  \def\rheader{\mainfont Appendix D\hfil 15}
  \centerline{\titlefont The {\ttitlefont WEAVE} processor}
  \vskip 15pt \centerline{(Version 2.5)} \vfill}
```

Redefining `\rheader`, which is the headline for right-hand pages, suffices in this case to put the desired information at the top of the page.

20.7.13 Customizing the table of contents

Data for the table of contents is written to a file that is read after the indexes have been TEXed; there's one line of data for every starred module. For example, one might obtain a file `CONTENTS.tex` containing

```
\WZ {0}{ Introduction}{1}{16}
\WZ {0}{ The character set}{11}{19}
```

and similar lines. Here the first argument of `\WZ` is the level number of the module, the second argument is the title, the third is the module number, and the fourth is the page number. The `\topofcontents` macro could redefine `\WZ` so that the information appears in another format. (The default name `CONTENTS.tex` can be changed by means of the style-file parameter "`contents.tex`".

20.7.14 Date and time

The macro `\Date` gives the present date in the form "June 17, 1993". The macro `\Time` gives the time in the form "14:40". By default, these are printed automatically at the bottom of the title page, via the `\botofcontents` macro.

20.7.15 Subdividing output

Sometimes it is necessary or desirable to divide the output of `WEAVE` into subfiles that can be processed separately. For example, the listing of TEX runs to more than 500 pages, and that is enough to exceed the capacity of many printing devices and/or their software. When an extremely large job isn't cut into smaller pieces, the entire process might be spoiled by a single error of some sort, making it necessary to start everything over.

Here's a safe way to break a woven file into three parts: Say the pieces are α, β, and γ, where each piece begins with a starred module. All macros should be defined in the opening limbo section of α, and copies of this TEX code should be placed at the beginning of β and of γ. In order to process the parts separately, we need to take care of two things: The starting page numbers of β and γ need to be set up properly, and the table of contents data from all three runs needs to be accumulated.

The `webmac` macros include two control sequences `\contentsfile` and `\readcontents` that facilitate the necessary processing. We include '`\def\contentsfile{CONT1}`' in the limbo section of α, and we include '`\def\contentsfile{CONT2}`' in the limbo section of β; this causes TEX to write the contents data for α and β into `CONT1.TEX` and `CONT2.TEX`. Now in γ we say

```
\def\readcontents{\input CONT1 \input CONT2 \input CONTENTS};
```

this brings in the data from all three pieces, in the proper order.

However, we still need to solve the page-numbering problem. One way to do it is to include the following in the limbo material for β:

```
\message{Please type the last page number of part 1:  }
\read-1to\\ \pageno=\\ \advance\pageno by 1
```

Then you simply provide the necessary data when TEX requests it; a similar construction is used at the beginning of γ.

This method can, of course, be used to divide a woven file into any number of pieces. (One problem with it is that each piece will have its own index. This problem will be addressed in the future.)

20.7.16 Special index entries

Sometimes it is nice to include things in the index that are typeset in a special way. For example, we might want to have an index entry for 'TEX'. `WEAVE` provides only two standard ways to typeset an index entry (unless the entry is an identifier, an intrinsic function, or a reserved word): '`@^`' gives roman type, and '`@.`' gives `typewriter` type. But if we try to typeset 'TEX' in roman type by saying, e.g., '`@^\TeX@>`', the backslash character gets in the way, and this entry wouldn't appear in the index with the T's.

The solution is to use the '`@9`' feature, declaring a macro that simply removes a sort key as follows:

```
\def\9#1{}
```

Now you can say, e.g., '`@9TeX}{\TeX@>`' in your `WEB` file; `WEAVE` puts it into the index alphabetically, based on the sort key, and produces the macro call '`\9{TeX}{\TeX}`' which will ensure that the sort key isn't printed.

A similar idea can be used to insert hidden material into module names so that they are alphabetized in whatever way you might wish. Some people call these tricks "special refinements"; others call them "kludges".

20.7.17 Module number

The control sequence `\modno` is set to the number of the module being typeset.

20.7.18 Symbolic names of modules

The macros `\modlabel` and `\module` afford a way of assigning symbolic names to modules (whose absolute numbers one doesn't know). Say `\modlabel{alpha}` somewhere in the module in question. If somewhere else you want to discuss that particular module, say something like '`discussed in \module{alpha}`'. With Plain TEX, forward references do not work; you must label the module before you reference it with `\module`. (However, the scheme could be generalized in standard ways, using the I/O features of TEX.) When LATEX is in use, forward references *do* work; the definitions of the module labels are stored in the `.aux` file, using LATEX's `\label` macro. In fact, under LATEX the definitions of `\modlabel` and `\module` are equivalent to simply

```
\def\modlabel#1{\label{MOD#1}}
\def\module#1{module~\ref{MOD#1}}
```

If you want to say "section" instead of "module", say "`\WEBsection`" instead of "`\module`". (LATEX usurps "`\section`".) For symmetry, "`\WEBmodule`" is equivalent to "`\module`".

20.7.19 Listing modules that have been changed

If you want to list only the modules that have changed, together with the index, put the command '`\let\maybe=\iffalse`' in the limbo section before the first module of your `WEB` file. It's customary to make this the first change in your change file.

20.7.20 Loading the macro package

The macro package `fwebmac.sty` `\let`s the macro `\FWEBisloaded` to be `\relax`. This macro can be used in various ways by macros you write that may need to behave differently depending on whether `fwebmac` has been loaded or not. See the Dirty Tricks section of the TEXbook.

20.7.21 Redefined macros

The macro package `fwebmac` usurps certain macro names for its own use. To find a complete list of the `fwebmac` macros, look in the index to `fwebmac.tex`. The original definitions of some of the more common ones are stored away under other names, as follows:

```
\let\amp\&
\let\at\@
\let\bslash\\
\let\caret\^
\let\dollar\$
\let\dstar\*
\let\equals\=
\let\leftbrace\{
\let\period\.
\let\rightbrace\}
\let\vertbar|
\let\PM\#
\let\PC\%
```

20.7.22 Using FWEB with LaTeX

The WEB system was originally designed for use with Plain TeX, and the design of the macros in fwebmac reflects this. However, with some restrictions it is also possible to use FWEB with LaTeX, thereby gaining the use of additional convenient macros, environments, etc. To use LaTeX instead of TeX to process the output test.tex that results from running FWEAVE on test.web, you should do two things at a minimum:

(1) Use the command-line option '-PL'. (If you will always be using LaTeX, place the command '+PL' into your .fweb file.) This command affects the default definitions of certain quantities defined through the style file; see the discussion below.

(2) Simply say "latex test" instead of "tex test".

Probably the principal price one pays is that FWEB has already usurped certain macro names for its own use. For example, the standard FWEB macro used to format an ordinary identifier is '\\'. See the table just above for alternative macro names. For example, a LaTeX user could use '\bslash' instead of '\\'. However, since the use of '\\' is quite common in LaTeX and it is cumbersome to type '\bslash' it is possible to dynamically reconfigure FWEAVE to use a different macro to format ordinary identifiers. This is done through the style file. As explained above in the section on "Typesetting identifiers," the names of the identifiers actually used in the tex file to format identifiers are mapped through an equivalencing procedure to the actual definitions in fwebmac, and those names can be changed without tampering with the definitions through entries in the style file. If no entries are made in the style file, then by default the macro '\\' is used to format both outer macro names, WEB macro names, and ordinary identifiers (with more than one character). When the '-PL' option is used, then that default is instead chosen to be '\>'. That default can be overridden by explicit style-file entries.

Just say "latex test" instead of "tex test".

Additional difficulties will arise with certain attempts to produce very clever output using some of the page layout facilities of LaTeX, since FWEB overrides the \output routine of LaTeX.

Also, LaTeX's \index macro is not correlated in any way to the index produced by FWEAVE.

On the positive side, the use of the aux file allows forward referencing to module names to be done conveniently. In particular, LaTeX's \label macro understands the current module number. (It does *not* understand anything about the level of starred modules.) See the discussion about "Symbolic names of modules" above.

Beginning with version 1.30 the fwebmac macros have been augmented (thanks to Charles Karney) to work with LaTeX's New Font Selection Scheme (NFSS). There hasn't been too much experience here; please report any difficulties.

20.8 APPENDIX H: SUMMARY of EXTENSIONS or CHANGES FROM CWEB

Here are some of the more significant differences between CWEB *and* FWEB *1.0. Many more features have been added in v. 1.1 and later; these are too numerous to describe here.*

Blank lines in the source are significant to WEAVE. The '@#' command will hardly ever have to be used.

The unnamed module is begun by '@a'.

Language switching: the '@c', '@r', and '@n' commands.

Long strings ('\.' macro) are broken with discretionary backslashes every so many characters, and after commas.

Verbatim comments: Preface comments you want TANGLE to keep by '@', as in "@/*␣Keep.␣*/". Make all comments verbatim by command-line option '-v'.

WEB macros, defined by '@m' and expanded by TANGLE, were reintroduced. Macro definitions are allowed in the code section.

A C-like preprocessing language was added. The preprocessor commands may appear in either the definition section or the code section.

FTANGLE translates RATFOR directly to FORTRAN.

One-character uppercase fwebmac macros have been changed to two-character ones starting with 'W'.

Because physicists sometimes like to use the asterisk for complex conjugation and therefore may define it to be an active character, FTANGLE will output an asterisk when it is in TEX mode, but puts out "\ast" when it is in code mode.

20.9 APPENDIX I: FWEB Q and A

This section is not complete! Please see /pub/fweb/faq for a more complete discussion [5].

Q. *What is the difference between* fwebmac.web, fwebmac.tex, *and* fwebmac.sty*?*

A. The macros read in by the *.tex files produced by FWEAVE are called fwebmac.sty; they are maintained in the source file fwebmac.web. Running FTANGLE on fwebmac.web produces fwebmac.sty; if you want to see a pretty listing of the macros, run FWEAVE on the source to get fwebmac.tex.

Q. *Why not incorporate Pascal as one of the supported languages, especially since Knuth's original processors supported that?*

A. This project evolved from Levy's CWEB, which was written in C for C. Since the present author does not personally use Pascal, it was considered too much of an effort to incorporate Pascal processing, especially since most modern codes are written in C.

Q. *What is the difference between* .fweb *and* fweb.sty*?*

A. Both are initialization files. However, .fweb is intended to be a global initialization file for all runs made by a particular author. One is supposed to put common command-line options into here. The style file fweb.sty is intended to allow customizations of particular runs or groups of runs. For example, the appearance of the index can be customized by setting parameters in the style file. Also, .fweb is read *before* the command line is processed, while fweb.sty is read *after* .fweb and the command-line are processed.

Q. *How does one force* FWEAVE *to align comments?*

A. One can't do that yet. In general, all complicated alignment issues relating to FWEAVE have been deferred to a future version. Sorry!

This list is not completed yet. Please suggest questions to be discussed in this space.

20.10 APPENDIX J: ERROR MESSAGES

Presently the error- message-processing facilities are a mixture of old and new: the original scheme of Knuth and Levy was restricted to function calls with fixed number of arguments, which tended to result in error messages with the bare minimum of information. Many, although not yet all, of the error messages have now been generalized to calls allowing variable numbers of arguments, which facilitates constructing rather elaborate error messages. Error processing will be refined still further in the future. *It is very important that you report difficulties with* `FWEB`*'s error processing facilities.* If `FWEB` ever issues an error not in the following list, please report that too.

There are five general classes of error messages: (1) messages common to both `FTANGLE` and `FWEAVE` (produced, for example, by command-line processing or one of the input drivers); (2) general messages from `FTANGLE`; (3) macro processing errors; (4) RATFOR errors; (5) general messages from `FWEAVE`. The class of the error is (usually) indicated in the output to the screen. There is as yet no numbering scheme. The messages are described here alphabetically within each class. Italics indicate variable fields that are filled in by the particular situation.

20.10.1 Messages common to both `FTANGLE` and `FWEAVE`

"Ambiguous prefix." A section name abbreviated with an ellipsis was not unique.

"Can't open include file "*name***"."** Possibly a misspelled file name; the `@i` command is skipped.

"`get_mem0`**: Can't request** *n* **units; used max of** *n*." Either you're asking for too much memory, or there's something wrong with the use of the ANSI *calloc* routine. If you think it's the latter, please report it.

"Change file ended after `@x`." There's something missing.

"Change file ended before `@y`." You must use the complete construction `@x`...`@y`...`@z`.

"Change file ended without `@z`." You must use the complete construction `@x`...`@y`...`@z`.

"Change file entry did not match." There's no text in the `WEB` file that matches the stuff between `@x` and `@y`.

"`get_mem0`**: Can't request** *n* **units; used max of** *m*." This is probably related to the difference between **int**s and **long**s on the local machine, and may signify a bug in the implementation.

"Hmm... some of the preceding lines failed to match." There was stuff left in the change buffer at the end of the run.

"`get_mem0`**: Can't request** *n* **units; used max of** *m*." This is probably related to the difference between **int**s and **long**s on the local machine, and may signify a bug in the implementation.

"Hmm... some of the preceding lines failed to match." There was stuff left in the change buffer at the end of the run.

"Input line too long; must be shorter than *n* **characters."** The input line length may be increased with the `-mbs` option.

"Invalid 'g' option: parameter type *c*.**"** . The valid parameters are 'r', 'm', or 's'.

"Invalid 'g' option: ....." You specified an invalid parameter for the RATFOR **switch**.

"Invalid language command "..."." You said '@L*l*', where *l* was not one of 'c', 'n', 'r', or 'x'.

"Missing id for 'm' option." To define a macro from the command line, you must say -mA=1 or -m"A␣1".

"Missing id for 'u' option." You must say -u*name*.

"Missing language character after @L." You must say '@L*l*'.

"No includes allowed in change file." You may not use the @i command within a change file.

"Style file name too long; must be less than *n* **characters."** There's something wrong with the -z option.

"Syntax error in output redirection command "->". Language must be one of 'c', 'r', or 'n', not '*x***'."** If you're redirecting output from a particular language, the form of the command is "->*l*=*file*".

"Too many nested includes; *n* **allowed."** @i commands can be nested to a level of *n*.

"WARNING: Command-line language *name* **overridden in limbo by** *new_name*.**"** A language command in the limbo section always takes precedence over a language specified on the command line.

"WEB file ended during a change." There's something incompatible about the change you're trying to make.

"Where is the matching @x?." In the change file, a @y or @z was encountered before an @x.

"Where is the matching @y?." A misplaced @x or @z was encountered.

"Where is the matching @z?." A misplaced @x or @y was encountered.

20.10.2 General messages from FTANGLE

"@d, @f, and @a are ignored in code text." @d and @f may appear only in the definition section. @a, which signifies the unnamed module, must be preceded by either @␣ or @*.

"ASCII string didn't end." ASCII constants such as @'a' must end on the same input line as they begin. (They can't even be continued by backslashes.)

"Can't continue comments on @i lines." You said something like "@i␣*filename*␣/*...".

"Code text flushed; = sign is missing." A section name of the form @<...@> was encountered at the end of a TEX or definition section, but it wasn't followed by the requisite equals sign that signifies the start of a named module.

"Compiler directives are allowed only in code." The compiler directive command @! was encountered while scanning through TEX text. This directive is allowed only in the code section.

"Continuation character '*c*' from '&' option is invalid; '*d*' assumed." FORTRAN's continuation character must be printable and not blank.

"Definition flushed; must start with identifier." An @d or @m command must be followed by a valid identifier.

"Expected @> after @<." A section name must be indicated by @<...@>.

"Expected '*c*' after language in "->"; command ignored." The syntax of the output redirection command is "->*l*=*filename*".

"Improper @ within control text." Other than @>, only @@ is allowed within control text.

"Include file name not given." In an include command, the file name must be on the same line as the @i.

"Incompatible section names." A section name is not uniquely distinguishable from another.

"Infinite recursion in definition of environmental variable "...")." Either a bug in your logic or in WEB's. If you think it's WEB, please report it.

"Input ended in mid-comment." A "/*" was not followed by a matching "*/".

"Input ended in middle of embedded comment." Embedded comments are C-style comments within strings that are delimited by parentheses. (Certain macros or **include** statements treat their arguments as strings.)

"Input ended in middle of string beginning with '*delimiter*'." Found end-of-file before end of string.

"Input ended in section name." The closing @> of a section name was not found.

"Input ended in verbatim comment." A '@/*' was not followed by a matching "*/".

"Inserted '*c*' at beginning of continued string." The option -\ is in effect, but the continuation character *c* that ended the last line does not appear as the first non-blank character on the present line.

"Invalid escape sequence '*c*' in ASCII constant; null assumed." In a construction of the form '*c*', *c* was not one of 0, \, ', ", ?, a, b, f, n, r, t, or v.

"Name does not match." A section name was used that was never defined.

"Nested named modules. Missing @?." A construction of the form '@<...@>=' was found in the code section of a module. Perhaps the equals sign is spurious or there's a missing @␣ or @*.

"Not present: <*section name*>." A section name of the form @<...@> was referenced but never defined.

"Output file name not given." You must say @o␣*filename*.

"Output file name too long; allowed only *n* characters." The file name following an @o command is too long.

"Section name didn't end." A new module command (@␣ or @*) was encountered in the middle of a

section name beginning with `@<`.

"Section name ended in mid-comment." An `@␣` or `@*` was encountered while inside a comment beginning with "`/*`".

"Should use double @ within ASCII constant." You should say "`'@@'`", not "`'@'`".

"String beginning with '*delimiter*' didn't end." Quoted strings must end on the same input line as they began, unless they are explicitly continued by a backslash as the very last character in the line.

"TeX line had to be broken." FTANGLE had to insert its own line break in a TEX input line. Usually this will be done satisfactorily, but it's best to check it out.

"Too many END DOs." Issued only when the `-d` option is used and the **do**...**end do** blocks don't match properly. *Note: This option is obsolete; use* RATFOR *instead.*

"Verbatim string didn't end." Verbatim strings (beginning with `@=`) must end on the same line as they began. (They can't be continued with backslashes.)

"WARNING: Code mode ended during unbracketed optional argument. Should there be white space after language command?." If you say "...`|@ninteger i|`...", the '`i`' is interpreted as a command-line type of argument to `@n`. Leave a space after the `@n`.

20.10.3 Errors related to preprocessing and macro processing

"Actual number of macro arguments (*m*) does not match number of def'n (*n*)." WEB macros with a fixed number of arguments (not defined with an ellipsis) must be called with precisely the same number of arguments. Macros with a variable number of arguments must be called with at least as many arguments as explicitly defined.

"Adjacent operators "*name1 name2*" not allowed in expression." For example, you can't say "`A␣||␣||␣B`".

"Argument *n* of _TRANSLIT must be a string." The arguments of this built-in must be enclosed in quotes.

"Argument *n* of _TRANSLIT doesn't begin with '*c*'." The delimiter characters for the string arguments of `_TRANSLIT` must all be the same.

"Auto insertion type must be one of "ibfmps"." The characters between brackets in `@m[...]` must be one of `ibfmps*`.

"Can't have @#elif after @#else." The order must be `@#if`...`@#elif`...`@#elif`...`@#else`...`@#endif`.

"Can't have more than 6 types of automatic insertion material; remaining ignored." You would get this message if you said `@m[*f]`..., since the `*` already counts as the six types `bifmps`.

"Can't negate type *name*." You can only apply the `!` operator to numeric types.

"Can't take one's complement of type *name*; operand converted to integer." You can only take the one's complement of an integer.

"'defined' ends prematurely." In WEB macros, `defined` must be followed by an identifier.

"`'defined'` must act on identifier, not type *name*." As above.

"Expected `')'` after ellipsis." Macros with variable numbers of arguments must be defined with the format "`@m␣A(x,y,...)␣`*text*"; no other arguments may follow the ellipsis.

"Expected constant after `"#:"`." For automatic statement labelling, you must say `#:`n, where $n \geq 0$.

"Found space instead of ']' after automatic insertion material." The syntax of an automatic insertion macro is `@m[...]␣`*name*`␣`*text*.

"Identifier "*name*" not allowed as binary operand." Possibly you forgot to define a macro. For example, in `_EVAL(A+B)`, both `A` and `B` must be `WEB` macros.

"Identifier must follow `#!`; command ignored." In `WEB` macro text, the construction `#!`*name* means substitute the macro parameter *name* without expanding it.

"Identifier must follow `#&`." The construction `#&`*name* is intended for internal use by the designer of `FWEB`; do not use it.

"Identifier must follow `@#undef`." You must say '`@#undef A`', where `A` is a macro name.

"Ignored out-of-order "*preprocessor cmd*" (mlevel = n)." There's something wrong with a preprocessor construction. The order must be `@#if`...`@#elif`...`@#elif`...`@#else`...`@#endif`.

"Internal function name "*name*" not defined." Do not use the construction `#&`; it's for internal use only.

"Invalid data type *name* in promotion." A crazy syntax error, or a bug.

"Invalid operand of exponentiate has type *name*." There's something wrong with an exponentiation operation (`^`) in a macro expression.

"Invalid operand of unary minus has type *name*." The unary minus (`-`) can only be applied to numeric types.

"Invalid preprocessor block structure (level n). Missing `@#endif`?." The definition section ended before an `@#if` statement was properly terminated.

"Invalid type *name* in bit operation. (Macro not defined?)." You can only apply bit operations such as `&` or `||` to integers.

"Invalid statement number offset (n) after `#:`; 1 assumed." In the construction `#:`n, n must satisfy $n \geq 0$.

"Invalid token `0x`*XX* (*'c'*) after `#`." In `WEB` macro text, only the constructions `#:` or `#!` were expected here.

"Invalid token *'c'* (`0x`*XX*) in expression." There's a syntax error in an argument to something like an `@#if`.

"Invalid type returned from _eval_." There's something wrong with the expression evaluator, or a syntax error in an expression.

"Macro after `#!` may not have arguments." The more general case is not implemented.

"Macro buffer full; *n* bytes requested for *reason*." The macro expansion buffer overflowed. The size can be increased with the `-mb` option. However, this message may also signify a bug in `FWEB`.

"Macro definition may not start with `'c'`; `-m` option ignored." There's trouble with a macro definition from the command line. Macro names must begin with an alphabetic character, an underscore, or a dollar sign.

"Macro inner recursion depth exceeded." Either a macro was too complicated, or a bug permitted an infinite recursion.

"Macro outer recursion depth exceeded." As above.

"Macro must start with identifier." There's something wrong with a `WEB` macro.

"Macro token `"#!"` must be followed by a parameter." In `WEB` macro text, the construction `#!`*name* means substitute the macro parameter *name* without expanding it.

"Macro token `"#*"` must be followed by a parameter." In `WEB` macro text, the construction `#*`*name* means stringize the macro parameter *name* without expanding it, and without adding an extra level of quotes if the parameter is already a quoted string.

"Missing argument to token-paste operation. Null assumed." The operator `##` is not allowed at the very beginning or end of a macro definition.

"Missing equivalence field while undefining "*name*"; this shouldn't happen!." There's trouble in paradise; a bug in `FWEB`.

"Missing internal function name after `#&`." The construction `#&`*name* is intended for internal use by the designer of `FWEB`; do not use it.

"Missing macro parameter in definition of macro "*name*". Token ... is invalid; can only have identifiers and commas between (...)." There's something wrong with the argument list of a `WEB` macro definition.

"Missing right paren in definition of macro "*name*"." Parentheses didn't match up while processing the argument list of a `WEB` macro.

"Missing `'('` in call to macro "*name*"." This macro was defined to have arguments, but is used without an argument list.

"No `')'` in call to macro "*name*"." In a call to a `WEB` macro, the closing parenthesis couldn't be found.

"Non-numeric type returned from eval (undefined macro?); assumed FALSE." The expression evaluator couldn't reduce an expression to 0 or 1.

"Non-numeric type returned from neval (undefined macro?); assumed 0." The expression evaluator couldn't reduce an expression to a number.

"Null expression encountered during expression evaluation; 0 assumed." You would get this message, for example, if you said "`@#if()`".

"Only one `@#else` allowed." Only one `@#else` is permitted in an `@#if` construction. Perhaps you meant to say `@#elif`.

"Overriding previous auto insertion type *c*." Auto insertions for the same type do not stack. You said something like `@m[f]` but type `f` had already appeared in a previous such construction (possibly implicitly, through a '`*`').

"Right operand of '*c*' is zero." You can't divide by 0.

"Section ended during scan for `"@#else"`, `"@#elif"`, or `"@#endif"`. Inserted `"@#endif"`. (elevel = *n*)." A `@␣` or `@*` was encountered before a `@#if` was properly closed.

"Sorry, @#pragma command not implemented yet." But you were clever to try.

"Too many macro arguments in definition of "*name*"; MAX_MARGS = *n*." A maximum of *MAX_MARGS* arguments are allowed for `WEB` macros.

"WARNING: Command-line language *language* overridden in limbo by *language*." You specified a language on the command line, but a different one in the file. Generally, put the language command into the file.

"WARNING: "*name*" is already undefined." You attempted to `@#undef` an identifier that was not defined as a `WEB` macro.

20.10.4 `Ratfor` errors

An annoying class of error messages is that which begins with "Output ended ...". If an expected delimiter is missing, the scan will, at present, proceed to the end of file. Really, it's possible to stop the scan before that, say at the beginning of the next function; that mechanism will be installed in future versions.

"Automatic statement number out of bounds; *n* assumed." An automatic statement number bigger than FORTRAN's maximum of 99999 was generated.

"Case value *val* of type double truncated to int." Cases should really be integer expressions.

"Can't return value from program or subroutine." The statement "`return` *expr*;" is allowed only in functions, not subroutines or main programs.

"Expected identifier after "*name*"." There's a missing name after **program**, **subroutine**, or **function**.

"Ignored '*r*' not matched with '*l*'." There were too many right delimiters *r* to be matched with the left delimiter *l*.

"Inserted '{'." A compound statement was expected here.

"Inserted '*c*' after "*name*"." For example, **default** wasn't followed by its mandatory colon.

"Invalid escape sequence '*letter*' in Ratfor character constant; null assumed." In a construction of the form '`\`*c*', *c* was not one of `0`, `\`, `'`, `"`, `?`, `a`, `b`, `f`, `n`, `r`, `t`, or `v`.

"Missing left paren after "*name*"; expansion aborted." Certain keywords such as **while** are expected to be followed by a left parenthesis.

"Missing opening delimiter '*c*'; text not copied." A construction beginning with the character *c* was expected here.

"Missing right brace (level *n*) at beginning of function; END statement inserted." A **program**, **subroutine**, or **function** statement was encountered before the body of a preceding program unit was properly terminated with a right brace.

"Misplaced keyword: "*name*" must appear inside loop." For example, **next** is allowed only inside loops such as **for**.

"Misplaced keyword: "*name*" must be used only inside "switch"." For example, **default** is allowed only inside a **switch**.

"Output ended after '{'." End of file was encountered before a matching right brace was found.

"Output ended at beginning of statement." A single or compound statement was expected here.

"Output ended during scan of simple statement." A semicolon to terminate the statement wasn't found.

"Output ended while copying to '*r*'." In some cases, RATFOR copies things directly to the output while searching for a closing delimiter. In this case, the closing delimiter wasn't found.

"Output ended while scanning for '*c*'." Some sort of list, such as the argument to a **for** statement, wasn't properly terminated.

"Output ended while skipping newlines." Probably a statement was expected here. Comments are also skipped while skipping over newlines.

"Ratfor character constant longer than one byte; extra characters ignored." In RATFOR, as in C, single quotes denote character constants whose length is one byte, so constructions such as 'ab' are not allowed. Perhaps you intended this to be a character string, which should be enclosed by double quotes: "ab".

"Ratfor is not loaded...." You linked on the dummy package ratfor0.o instead of ratfor.o. Therefore, you're not allowed to set the language to *RATFOR*.

"Shouldn't encounter top level here." There's something wrong with FWEB's stacking mechanism for expanding RATFOR keywords.

"Unexpected keyword "*name*" ignored." For example, an **until** was used without a preceding **repeat**.

20.10.5 General messages from FWEAVE

"@f line ends prematurely." For changing category codes, the syntax is @f␣‘a␣n̂.

"A string must follow @l." Where is the text of the limbo text definition?

"Braces don't balance in comment." This holdover from Knuth's Pascal WEB is sometimes spurious, and should be improved.

"Can't have vertical bars in @! compiler directives." A restriction of the implementation.

"Category code must be between 0 and 15." Just as in TEX.

"Control codes are forbidden in control text." If you need an @, you can say "\AT!".

"Control text didn't end." The terminating @> must be on the same line as the control text began.

"Double @ required outside of sections." If you intend for the character '@' to be here, you must double it. Otherwise, you're using a control code that isn't allowed here.

"Double @ should be used in strings." No control codes are allowed inside strings; for the '@' symbol you must say "@@".

"Identifier was already explicitly defined via @[in module n." Identifiers should be explicitly defined in only one place.

"Illegal use of @ in comment." No control codes are allowed in comments. If you intend the character '@', you must double it.

"Implicit declaration of '*name*' conflicts with previous declaration at module n." The syntax parser recognized a complete function (during phase 2), but either the function name had already been identified by an @[command in phase 1 or the function has been defined more than once.

"Improper format definition." There are too many or too few items in an @f command.

"Improper macro definition: expected ')' after ellipsis." Self-explanatory.

"Improper macro definition: expected identifier." Self-explanatory.

"Improper macro definition: unrecognized token in argument list." Self-explanatory, but the message should be made more explicit.

"Input ended in mid-comment." A "/*" was not followed by a matching "*/".

"Input ended in middle of string beginning with '*delimiter*'." Found end-of-file before end of string.

"Input ended in section name." The closing @> of a section name was not found.

"Inserted c at beginning of continued string." The option -\ is in effect, but the continuation character c that ended the last line does not appear as the first non-blank character on the present line.

"Invalid @f command: One of the representation 'a, '\a, or '^^M is required." A left quote appears after @f says that you want to change a TEX category code. Use the same syntax for the character whose catcode is to be changed as you would following TEX's \catcode command.

"Invalid category code." A numerical constant was expected here.

"Invalid op code n." There's something wrong with the operator being processed. If you can't figure this one out, please report it.

"Missing '|' after code text." Did you forget to switch out of code mode in the middle of TEX text?

"No op_macro name for "..." [*language*]; token ignored." There's something funny about an operator overload. If you can't figure this out, please report it.

"Operator after @v is invalid." There's something wrong with the operator after an @v command. Dot constants should be written *sans* dots; characters such as parentheses are not valid operators.

"Second argument (replacement text) of @v must be a quoted string." The syntax of an operator overload command is @v␣*newop*␣"*string*"␣*oldop*.

"Section ended in middle of Fortran-90 continuation." WEB thinks the last line was a continuation (ended by the continuation character '\' or '&'), but then reached end-of-file.

"Section name didn't end." A new module command (@␣ or @*) was encountered in the middle of a section name beginning with @<.

"String beginning with '*delimiter*' didn't end." Quoted strings must end on the same input line as they began, unless they are explicitly continued by a backslash as the very last character in the line.

"String must follow @l." Limbo text commands have the form "@l␣"abc\ndef"".

"TeX string should be in code text only." An @t command is misplaced.

"Verbatim string didn't end." Verbatim strings (beginning with @=) must end on the same line as they began. (They can't be continued with backslashes.)

"You can't do that in code text." Commands like @d or @f are not allowed here.

"You can't do that in TeX text." The following commands are not allowed in TEX text: @,, @|, @/, @#, @+, @&, @;, @e, @:.

"You need an = sign after the section name." Self-explanatory. One sometimes gets this error if FWEAVE has gotten confused about which part of the section it's in.

"You should say @@." Don't forget to double the @'s in WEB source code.

"You should say \@@." The single-character TEX macro \@ must be written in WEB source code as \@@.

20.11 APPENDIX K: GETTING WEB ONTO a NEW COMPUTER

The following paragraph, extracted from Knuth's original report, is included here mostly for entertainment.

> "If you have only the present report, not a tape, you will have to prepare files WEAVE.WEB and TANGLE.WEB by hand, typing them into the computer by following Appendices D and E. Then you have to simulate the behavior of TANGLE by converting TANGLE.WEB manually into TANGLE.C; with a good text editor this takes about six hours. Then you have to correct errors that were made in all this hand work; but still the whole project is not impossibly difficult, because in fact the entire development of WEAVE and TANGLE (including the writing of the programs and the manual) took less than two months of work." *[Note from J. A. Krommes:* (!!!)*]*

FWEB is available via anonymous guest ftp from Internet host lyman.pppl.gov, a Sun SparcStation

running UNIX. Use binary mode to transfer a compressed `tar` file with a name like `fweb-1.30.tar.Z`. See the file `/pub/fweb/READ_ME` for the current status, recent bug reports, and further instructions.

For those without `ftp` access, it is possible to emulate the function of `ftp` by sending a valid `ftp` session as a mail message to `bitftp@pucc.bitnet`. The requested files will be returned by mail, possible `uuencoded` and broken up into several parts if they are sufficiently long. A sample `ftp` session is

```
ftp lyman.pppl.gov
cd /pub/fweb
get READ_ME
quit
```

To learn more about the `bitftp` service, send a message containing the single word "`help`" to the above address.

To unpack the `tar` file, say

```
uncompress fweb-1.30.tar
tar -xvf fweb-1.30.tar
```

A brief summary of the installation procedure can be found in the `READ_ME.FWEB` file included in the subdirectory corresponding to the numbered release—e.g., `/pub/fweb/v1.30/READ_ME.FWEB`. UNIX users should do the following:

```
cd fweb-1.30
./configure
cd web
make bootstrap
make install
```

More detailed installation information can be found in the separate file `INSTALL_FWEB.tex` included with the `FWEB` release in the `manual` subdirectory.

20.12 APPENDIX L: SYNTAX SUMMARY

Here we summarize various of the important features of `FWEB`. This material is intended to be for reference; please refer to the body of the user's manual for full details.

20.12.1 The `FWEB` processors

`FTANGLE` — *Creates compilable code.*

Phase one:

- discards TEX documentation;
- tokenizes source;
- expands `@'...'`, `@"..."`, and `0b`*binary* (also `0`*octal* and `0x`*hex* in FORTRAN);
- stores code text in appropriate modules;
- memorizes macro definitions (`@d` and `@m`).

Phase two:

- outputs outer macro definitions (`@d`);
- outputs the unnamed module (`@a`);
- expands `WEB` macros (`@m`);
- expands build-in macros;
- translates RATFOR statements.

`FWEAVE` — *Typesets the documentation and code.*

Phase one:

- tokenizes and stores identifiers and module names;
- collects cross-reference information (including processing `@[` and `@``);
- stores limbo text definitions (`@l`);
- collects information about overloaded operators (`@v`) and identifiers (`@W`).

Phase two:

- outputs limbo text;
- outputs special TEX macros for overloaded operators;
- copies TEX material directly to output;
- treats material between vertical bars (`|...|`) as code to be typeset;
- tokenizes and stores contents of each code section;
- analyzes code syntax and converts to appropriate TEX macros.

Phase three:

- typesets index and module cross-references;
- writes table of contents.

20.12.2 Files

The FWEB system works with a variety of files. File names have the form *[path]/root[.ext]*. Here '/' is called the *PREFIX_END_CHAR*. Since it differs for various operating systems, it can be changed in custom.h. The character that initiates the file-name extension (normally a period) can be changed with the '-E' command-line option.

Input files:

null file — The system sometimes reads from the null file. This name varies from system to system; the default can be defined in custom.h, or it can be changed with the style-file parameter null_file.

.fweb (or fweb.ini) — Initialization file; always in the *home* directory. The basic file name can be overridden by the environment variable FWEB_INI. (It can also be changed in custom.h, although this is strongly discouraged.)

fweb.sty — Style file; in *current* directory unless overridden by environment variable FWEB_STYLE_DIR. The basic name can be changed by the '-z' option. (It can also be changed in custom.h, although this is strongly discouraged.)

name.web — Source code. (Alternative suffixes can be searched for automatically with the '-e' option and the ext.web style-file entry.)

name.ch — Change file. (Suffixes are treated as above, but see ext.change.)

name.hweb — Code included into web file with '@i'. Include files are searched for in the path set by the environment variable FWEB_INCLUDES and/or the -I option; if that path is empty, then the current directory is searched. (Suffixes are treated as above, but see ext.hweb.)

name.hch — Change file for include file. (Suffixes are treated as above, but see ext.hchange.)

Output files:

name.tex — Woven output; can be processed with either Plain TEX or LATEX.

CONTENTS.tex — Accumulates table-of-contents information written by FWEAVE. (This name can be overridden by the style-file option contents.tex.)

INDEX.tex — Stores indexing information written by FWEAVE. (This name can be overridden by the style-file option index.tex.)

MODULES.tex — Stores module list written by FWEAVE. (This name can be overridden by the style-file option modules.tex.)

name.ext — Compilable output file; see table below for the extension associated with each language.

When the '-e' option is in effect, input file names that include no period are completed automatically according to the style-file entries listed in the following table:

Type of file	*Style-file entry*	*Default*
WEB file	ext.web	.web
Change file	ext.ch	.ch
Include file	ext.hweb	.hweb
Change file for include file	ext.hch	.hch

The default extension for the file(s) output by FTANGLE sometimes depends on whether the operating

system is UNIX or not. (These defaults can be overridden by style-file parameters, as indicated.)

Language	*Style-file entry*	UNIX *default*	*Non*-UNIX *default*
C	suffix.C	.c	.c
C++	suffix.Cpp	.c++	.cpp
FORTRAN--77	suffix.N	.f	.for
FORTRAN--90	suffix.N90	.f	.for
MAKE	suffix.K	.mk	.mk
RATFOR--77	suffix.R	.f	.for
RATFOR--90	suffix.R90	.f	.for
TEX	suffix.X	.sty	.sty

20.12.3 Environment variables

Environment variables in UNIX and logical names in VMS behave in the same way.

FWEB_INCLUDES — Colon-delimited list (identical in format to the Unix C-shell PATH variable) of directories to search for include files. (One can append to this list by means of the -I option.)

FWEB_INI — Name of the initialization file—e.g., fweb.ini. If not defined, either .fweb or fweb.ini is chosen, depending on the machine. The initialization file always resides in $HOME.

FWEB_STYLE_DIR — Directory in which style file resides—e.g., $HOME/fweb. If not defined, the current directory is used.

HOME — Name of the home directory. This variable should be defined by UNIX.

TERM — The terminal type. Defined by UNIX.

To change an environment variable, say, for example (in UNIX)

```
setenv FWEB_INCLUDES .:$HOME/fweb:/usr/xxx/stuff
setenv FWEB_INI my_FWEB.ini
```

For VMS, substitute "define" for "setenv".

The built-in function _GETENV(*ENV*) expands to the value of the environment variable *ENV*.

20.12.4 Order of initial operations

FWEB begins its processing by performing the following operations:

1. Evaluate environment variables.
2. Read initialization file .fweb.
3. Execute .fweb options beginning with '+'.
4. Read and execute the command line.
5. Execute remaining .fweb options.
6. Read the style file fweb.sty.
7. Process the WEB file.

20.12.5 Command-line syntax

The command-line syntax is

$$\left\{\begin{array}{l}\texttt{FWEAVE}\\ \texttt{FTANGLE}\end{array}\right\}\ \textit{web\_file\_name}\ [\textit{change\_file\_name}]\ [\textit{options}]$$

where the command-line options, each of which must begin with a hyphen, are summarized below. (Actually, the options can appear before the file names, or can even be intermixed with them.) If an option has an argument, no space should precede the argument. (E.g., say "`-zmy.sty`", not "`-z␣my.sty`".) In the descriptions of the options, the letters `T` or `W` in brackets denotes to which processor the command applies; no brackets at all means it applies to both processor. Similarly, parenthesized `C`, `N`, `R`, or `X` denotes the applicable language. Italic brackets, as in `-d`*[nnnnn]*, indicate an optional argument; the brackets themselves shouldn't be typed. Stars mean the command is *not* allowed to be optionally changed along with a language change, according to the format "`@`*l*`[`*options*`]`" or "`@L`*l*`[`*options*`]`".

20.12.6 Command-line options `a`–`q`

	`-0`	— Turn off `WEAVE`'s debugging mode. [`W`].
	`-1`	— Turn on brief debugging mode. (Display irreducible scraps.) [`W`].
	`-2`	— Turn on verbose debugging mode. (Display detailed reductions of the scraps.) [`W`].
	`-A`	— Turn on **ASCII** translations.
	`-b`	— Number **do** and **if** blocks in woven FORTRAN and RATFOR output. [`W`].
*	`-c`	— Set the global language to C.
*	`-c++`	— Set the global language to C++.
	`-D`*[letters]*	— Display information about reserved words of the current language (beginning with *letters* if present).
	`-d`*[nnnnn]*	— Convert unnumbered '`do...enddo`' constructions to standard FORTRAN–77. [`T`]; (N).
	`-E`*c*	— Change the delimiter of a file-name extension from the default '`.`' to *c*.
	`-e`	— Turn on automatic file-name completion. (See discussion of the style file entries `ext.*`.)
	`-f`	— Turn off module references for identifiers. [`W`].
*	`-h`	— Get help from the command line. *(Not implemented yet.)*
	`-I`*directory*	— Append a directory or colon-delimited list of directories to the list of directories to be searched for include files.
	`-i`	— Read include files named by the '`@I`' command, but do not print their contents. [`W`].
	`-i!`	— Don't even read include files named by the '`@I`' command. [`W`].
*	`-L`*l*	— Select global language: $l \in \{\texttt{c}, \texttt{n}, \texttt{r}, \texttt{x}\}$.
	`-l`*[mmm[:nnn]]*	— Echo the input line constructed by the input driver between lines *mmm* and *nnn*.
*	`-m`*id[=text]*	— Define a `WEB` macro. [`T`].
	`-m4`	— Understand the `m4` built-in commands. [`W`].
	`-m;`	— Automatically append a pseudo-semicolon to `WEB` macro definitions. [`W`].
*	`-n`	— Set the global language to FORTRAN–77.
*	`-n9`	— Set the global language to FORTRAN–90.
	`-n;`	— For FORTRAN–77, supply semicolons automatically. (Default mode.)
	`-nb`	— In FORTRAN, number the **if**s and **do**s. [`W`]; (N).
	`-np`	— Print semicolons in woven FORTRAN output. [`W`].
	`-n\`	— Select free-form FORTRAN–90 syntax continued with a backslash.
	`-n&`	— As above, but continue with an ampersand.
	`-n/`	— In FORTRAN, make '`//`' denote the start of a short comment instead of concatenation. (One can always use '`\/`' for concatenation.)
	`-n!`	— In FORTRAN, make '`!`' denote the start of a short comment instead of the logical `NOT`.
	`-o`	— Turn off `FWEAVE`'s mechanisms for overloading operators. [`W`].
*	`-P`*letter*	— Selects the TEX processor for `FWEAVE`'s output. Say '`-PL`' for LATEX; the default is '`-PT`' for TEX. [`W`].
*	`-p`*styleentry*	— Buffer up a style-file entry, to be processed just before the local style file is read.
	`-q`	— Do *not* translate RATFOR commands into FORTRAN. (No longer supported.) [`T`]; (R).

20.12.7 Command-line options `r`–`z`

* `-r` — Set the global language to RATFOR–77.
* `-r9` — Set the global language to RATFOR–90.
`-rb` — In RATFOR, number the **if**s and **do**s. [`W`]; (N).
`-rg`*params* — Set the **goto** parameters for RATFOR's **switch**. [`T`]; (R).
`-rk`*[letters]* — Suppress particular comments about RATFOR statement translation. [`T`]; (R).
`-rK`*[letters]* — As above, but write out particular comments. [`T`].
`-r;` — Turns on RATFOR's auto-semi mode, and tells it to use the "obviously continued" syntax. (R).
`-r/` — In RATFOR, make '`//`' denote the start of a short comment instead of concatenation. (One can always use '`\/`' for concatenation.) (R).
`-r!` — In RATFOR, make '`!`' denote the start of a short comment instead of the logical `NOT`.
* `-s` — Print statistics about memory usage.
* `-sm`*[nnn]* — As above, but also display the dynamic memory allocations of size $\geq$ *nnn* as they occur.
* `-t`*ln[{...}]* — Truncate identifiers of language *l* to length *n*, after optionally filtering out the characters listed between the braces. [`W`].
* `-u`*id* — Undefine a predefined or command-line macro. [`T`].
`-v` — Make all comments verbatim. [`T`].
`-W`*letters* — Commands that apply only to `FWEAVE`. Here *letters* may be one or more of the following:
`[` — Turn on special processing of bracketed array indices.
`f` — Don't print format statements (`@f`) in woven output.
`l` — As above, but for limbo statements (`@l`).
`m` — As above, but for macro definitions (`@m`).
`v` — As above, but for operator overloading (`@v`).
`w` — As above, but for identifier overloading (`@W`).

* `-w`*[file_name]* — If *file_name* is absent, don't print '`\input␣fwebmac.sty`' as the first line of the `TEX` output file. Otherwise, print '`\input` *file_name*'. [`W`].
* `-X`*[letters]* — Print selected cross-reference information; the opposite of '`-x`'.
* `-x`*[letters]* — Reduce or eliminate cross-reference information. The optional letters can be one of '`c`', '`i`', '`m`', or '`*`', referring respectively to the table of contents, index, module list, or all cross-reference information. [`W`].
* `-y`*[a[a]][nnnn]* — Override default for dynamic memory allocation. If *nnnn* is omitted, then simply query the default instead of overriding it. The simple command '`-y`' with no argument queries everything.
`-Z`*[letters]* — Display default values of style-file parameters (starting with *letters* if present).

* `-z`*[name]* — Override default style-file name.

20.12.8 Command-line options (miscellaneous)

	`-.`	—	Don't recognize "dot constants" in FORTRAN or RATFOR.
	`-\`	—	Explicitly escape continued strings.
	`-(`	—	Continue parenthesized strings with backslashes.
	`-:`*[nnnnn]*	—	Set the starting automatic statement number. [`T`].
*	`->`*[l=][name]*	—	Redirect tangled output.
*	`-=`	—	Redirect tangled output. Synonymous with '`->`'. Easy to type under UNIX.
	`-#`	—	Turn off comments about line numbers and module names in tangled output. [`T`]
	`-+`	—	Don't interpret the compound assignment operators. [`T`]; (N,R).
	`-/`	—	In both FORTRAN and RATFOR, make '`//`' denote the start of a short comment instead of concatenation. (One can always use '`\/`' for concatenation.)
	`-!`	—	In both FORTRAN and RATFOR, make '`!`' denote the start of a short comment instead of the logical `NOT`.

These command-line options may also be put into the ini file `.fweb`. Options beginning with a plus sign are processed *before* the command-line options. Otherwise, they are processed after the command-line options.

20.12.9 Modules

- Module names are delineated by `@<...@>`.
- To reference a named module in a macro definition, say '`#<...@>`' instead of '`@<...@>`'.
- The unnamed module is begun by '`@A`' or '`@a`'. (The latter command inserts an implicit '`@[`'.)
- `FTANGLE` outputs the unnamed module. *(There must be at least one unnamed section, otherwise there will be no output!)*

`TEX part` — *Arbitrary TEX documentation.*

- Begins with '`@*`' or '`@␣`'.
- Change to code mode with `|...|`.

`definition part` — *Macro definitions, formatting, limbo text, operator overloading, etc.*

- Begins with '`@m`', '`@d`', '`@f`', '`@l`', '`@v`', or '`@#...`'.

`code part` — *The source code.*

- Begins with '`@a`' or '`@<`'.
- TEX mode inside comments.
- Ends with '`@*`' or '`@␣`'. (The end of file is like an '`@␣`'.)

20.12.10 The simplest WEB sources

The absolutely simplest nontrivial WEB source file is "@␣@a". The next simplest is

```
@* EMPTY.
@a
```

These have a null definition part as well as a null code part. The simplest FORTRAN–77 code is

```
@n
@* FORTRAN.
@a
    program main
    end
```

The same code in RATFOR is

```
@r
@* RATFOR.
@a
program main
{}
```

The corresponding code in C is

```
@c
@* C.
@a
main()
{}
```

20.12.11 Control codes allowed in WEB files

Literal control characters:

`@@` — Insert the single character '`@`'.
`@|` — Insert a vertical bar (TEX text only). In code mode, this command means an optional line break; see the "Spacing" commands below.

Beginning of module:

`@␣` — Begin a new minor (unstarred) module.
`@*`*[n]* — Begin a new major (starred) module of level n.

Beginning of code part:

`@<` — Begin a module name.
`@>` — End a module name.

`@A` — Begin the code part of an unnamed module.
`@a` — Equivalent to "`@A@[`".

Control codes `b`–`z`:

`@b` — Insert a breakpoint command in a WEB module.

`@c` — Set the language to C.
`@c++` — Set the language to C++.

`@D` — Define an outer macro.
`@d` — Equivalent to "`@D@[`".

`@e` — Invisible (pseudo-) expression.
`@f` — Format an identifier or module name.

`@I` — Include a file, but don't print it out if the `-i` option is used.
`@i` — Include a file.

`@L`*l* — Set language to l.
`@l` — Specify limbo text.

`@M` — Define a WEB macro.
`@m` — Equivalent to "`@M@[`".

`@n` — Set the language to FORTRAN–77.
`@n9` — Set the language to FORTRAN–90.

`@O` — Open new output file (global scope).
`@o` — Open new output file (local scope).

`@r` — Set the language to RATFOR–77.
`@r9` — Set the language to RATFOR–90.

20.12.12 Control codes allowed in WEB files, cont'd

@u — Undefine an outer macro.

@v — Overload an operator.
@W — Overload an identifier.

@x — Terminates ignorable material (begun by '@z' at beginning of source or include file).
@z — Begins ignorable material at beginning of source or include file.

Conversion to ASCII:

@' — Convert single character to ASCII.
@" — Convert string to ASCII. (In FORTRAN or RATFOR, generate a call to the function named by the style-file field ASCII_fcn.)

Markers for forward referencing:

@[— Mark the next identifier as defined in this module.
@] — *Reserved*; do not use.
@‘ — *Reserved*; do not use.

Comments:

@/* — Begin a long verbatim comment.
@// — Begin a short verbatim comment.
@% — An ignorable comment: Everything to the next newline is completely ignored.
@? — Begin a compiler directive.
@! — Begin a compiler directive (obsolete).
@(— Begin a meta-comment.
@) — End a meta-comment.

Special brace:

@{ — Suppress default insertion of breakpoint command.

Index entries:

@_ — Force an index entry to be underlined.
@- — Delete an index entry for the next identifier.

@^ — Make an index entry in Roman type.
@. — Make an index entry in typewriter type.
@9 — Make an index entry in a format controlled by '\9', which the user must define.

Control text:

@t — Put control text into a TEX \hbox.
@= — Pass control text verbatim to the output.

20.12.13 Control codes allowed in WEB files, cont'd

Spacing:

`@,` — Insert a thin space.
`@/` — Insert a line break.
`@|` — Insert an optional line break in an expression.
`@#` — Force a line break with some extra white space; very seldom necessary, since blank lines in the source are significant. Also begin a preprocessor command.
`@+` — Cancel a line break.
`@&` — Join left and right with no spaces or line breaks inbetween.

Pseudo (invisible) operators:

`@e` — Invisible (pseudo-) expression.
`@;` — Invisible (pseudo-) semicolon.
`@:` — Invisible (pseudo-) colon.

20.12.14 Special format for language changes

The most general form of a language command is

`@/`[`L`]*l text*[*options*]

where *l* is a language symbol, *text* is converted into a hyphenated option, and *options* have the same syntax as on the command line. For example,

```
@n9[-n&]
```

means set the language to FORTRAN–90 and use free-form syntax with the ampersand as the continuation character.

20.12.15 Control codes allowed in change files

The folllowing commands are allowed in change files. They *must* begin in column 1. Any line that does not begin with one of these commands is a comment.

`@x` — Begin a change file entry.
`@y` — End the old code; begin the replacement code.
`@z` — End the change file entry.

`@c` — Set language to C.
`@c++` — Set language to C++.
`@n` — Set language to FORTRAN–77.
`@n9` — Set language to FORTRAN–90.
`@r` — Set language to RATFOR–77.
`@r9` — Set language to RATFOR–90.
`@L`*l* — Set language to *l*.

`@[` — Switch into code mode. (Use for column-oriented language such as FORTRAN–77.)
`@]` — Switch out of code mode.

20.12.16 The null change file

When no change file is specified on the command line, the `WEB` processors attempt to open and read from the so-called "null file." (This file may or may not actually exist in the directory structure, depending on the system.) The name of this file, which is permanently empty, depends on the operating system and becomes the default value of the style-file option `null_file` according to the following table:

Operating system	*Name of null file*
IBM/PC	`nul`
IBM/MVS	`'NULLFILE'`
VAX/VMS	`nl:`
UNIX or other	`/dev/null`

Usually you must do nothing explicitly to access the null file. However, if `FWEB` can't find the default null file on your system, just create an empty file whose name is *null_name* and insert into the style file the line "`null_file "`*null_name*`"`".

20.12.17 Commenting modes

FWEB allows a variety of commenting styles. The visible comments are in the font \cmntfont, which defaults to \tenrm.

Invisible comments:

@z...@x — If a source or include file begins with '@z', then all material is skipped until and including a line beginning in column 1 with '@x'.

@% — All material until and including the next newline is completely ignored.

Visible comments:

/*...*/ — A long comment (may extend over several lines).

//... — A short comment (terminated by next newline). (In FORTRAN or RATFOR, this must be turned on explicitly with one of the command-line options '-n/', '-r/', or '-/'.)

!... — A short comment in FORTRAN or RATFOR. This must be turned on explicitly with one of the command-line options '-n!', '-r!', or '-!'.

!!... — A short comment in FORTRAN or RATFOR.

@(...@) — A meta-comment. The material between '@(' and '@)' is typeset in a verbatim environment, and is appropriately passed to the tangled output. (See the style-file parameters meta.*.)

20.12.18 Alternatives for 'dot' commands in FORTRAN and RATFOR

Although FORTRAN and RATFOR allow standard 'dot' commands such as '.LT.', they are considered to be obsolete; more modern alternatives are preferred. Here is a table of what you can type on input, and what WEAVE will typeset. The first entry is standard FORTRAN; the parenthesized material is an allowable input alternative. (In most cases, the pretty input alternatives follow C's convention.)

.lt. (<)	$\to <$	.or. (||)	$\to \vee$
.le. (<=)	$\to \leq$	.neqv.	$\to \not\equiv$
.eq. (==)	$\to \equiv$	.xor.	$\to \not\equiv$
.ne. (!=,<>)	$\to \neq$	.eqv.	$\to ?=$
.gt. (>)	$\to >$	.not. (!)	$\to \neg$
.ge. (>=)	$\to \geq$	** (^)	$\to$ (a+b)^(c+d) $\to (a+b)^{c+d}$
.and. (&&)	$\to \wedge$	// (\/)	$\to \|\|$

These same conventions are allowed in RATFOR mode. Note that in FORTRAN and RATFOR '//' is interpreted by default as the concatenation symbol, not the start of a short comment. To override that default, use one of the command-line options '-n/', '-r/', or '-/', or use a language-changing command of the form "@n/".

20.12.19 Considerations about formatting

The construction

`@f` *identifier old_identifier*

makes *identifier* behave like *old_identifier*.

The *old_identifier* may be one of the following special names, which insert extra spaces according to the positions of the underscores and behave as the part of speech indicated by the base names. These are useful for dealing with macro constructions.

```
$_BINOP_
$_COMMA_
$_EXPR
$_EXPR_
$EXPR_
$UNOP_
```

When the current language is `TEX`, the format command can be used to change a category code according to the format

`@f` ‘ *TEXchar new_cat_code*

20.12.20 Macro commands

Outer macros, defined by `@d`, are copied to the beginning of the output file. Inner `WEB` macros are defined by `@m`. `WEB` macro definitions in the definition section are collected at the beginning of the unnamed module. `WEB` macro definitions in the code section (deferred definitions) become known when they are encountered while the code is being output.

`WEB` macro definitions have one of the following three forms:

`@m` *name*`(`*args*`)` *text*
`@m*` *name*`(`*args*`)` *text*
`@m[bfimps*]` *name*`(`*args*`)` *text*

In the second form, the asterisk means that the macro may be recursive, although *this feature is not implemented yet*. In the third form, which is useful only for RATFOR, the brackets means that the contents of this macro are to be inserted automatically at the beginning of the type of program unit identified by the characters within the brackets. See the text for more information.

Macros with a variable number of arguments are indicated by an ellipsis , as in "`@m␣VAR(x,y,z,...)␣`*text*".

Adjacent strings in macro text are automatically concatenated.

The following special tokens can be used in the text of `WEB` definitions. Here *parameter* means a dummy argument in the argument list of a function-like macro.

ANSI C-compatible tokens:

`##` — Paste together tokens to left and right. (ANSI C-compatible.)
`#`*parameter* — Convert parameter to string, without expansion. (ANSI C-compatible.)

Extensions to ANSI C macro syntax:

`#*`*parameter* — As above, but pass a quoted string through unchanged.
`#!`*parameter* — Don't expand argument.
`#'`*parameter* — Convert parameter to a single-quoted string, without expansion.
`#"`*parameter* — Convert parameter to a double-quoted string, without expansion.
`#0` — The number of variable arguments.
`#`n — The n^{th} variable argument, counting from 1.
`#{0}` — Like `#0`, but the argument may be a macro or expression known at output time.
`#{`n`}` — Like `#`n, but the argument may be an expression.
`#[0]` — The total number of arguments (fixed plus variable). (The argument may be an expression.)
`#[`n`]` — The n^{th} argument (including the fixed ones), counting from 1. (The argument may be an expression.)
`#.` — A comma-separated list of all variable arguments.
`#:0` — Unique statement number (expanded during phase 1).
`#:`*nnn* — Unique statement number for each invocation of this macro (expanded during phase 2).
`#<` — Begin a section name. (Ends with '`@>`'.)
`#,` — An "internal" comma; does not delimit the end of an argument.

20.12.21 Preprocessor commands

`WEB` preprocessor commands may appear in either the definition or the code part. But *beware*: No matter where they appear, they are expanded during *input*, not output.

`@#define` *identifier* — Define a `WEB` macro; equivalent to '`@m`'.
`@#undef` *identifier* — Undefine a `WEB` macro.

`@#ifdef` *identifier* — Is `WEB` macro defined? Equivalent to '`@#if defined` *identifier*'.
`@#ifndef` *identifier* — Is `WEB` macro not defined? Equivalent to '`@#if !defined` *identifier*'.

`@#if` *expression*
`@#elif` *expression*
`@#else`
`@#endif`

20.12.22 Built-in FWEB macros (A–K)

Built-in macros are expanded during output while processing the code part. They all begin with an underscore and are in upper case. *User-defined macros should not begin with an underscore or a dollar sign.* In the following argument lists, *string* means a character string that should be surrounded by quotes. In a few cases the quotes are optional if the argument is a single alphanumeric identifier, but don't use this property unless you really have to.

_A(*string*)	—	The built-in equivalent to @'...' or @"...". (Note the extra parentheses required by the built-in.)
_ABS(*expression*)	—	Absolute value of *expression*.
_ASSERT(*expression*)	—	Evaluates *expression*; if false, prints an error message and aborts.
_COMMENT(*string*)	—	Generate a comment in the output file.
_DATE	—	A string consisting of the date in the form "August␣15,␣1989".
_DAY	—	A string consisting of the day of the week in the form "Monday".
_DECR(N)	—	Decrement a macro.
_DEFINE(defn)	—	Deferred macro definition.
_DO(*macro*,*imin*,*imax*[,Δi]{...}	—	Repetitively defines *macro* as would the FORTRAN **do** loop do *macro* = *imin*,*imax*,Δi.
_DUMPDEF(m_1,m_2,...)	—	Here m_1, m_2, etc. are macro calls (with argument list if appropriate). The macro definitions and their expansions are dumped to the terminal.
_ERROR(*string*)	—	Send string to the standard error message facility.
_EVAL(*expression*)	—	Evaluate a macro expression.
_GETENV(*name*)	—	Returns the present value of the environment variable *name*.
_HOME	—	The user's home directory; equivalent to _GETENV(HOME).
_IF(*expression*,t,f)	—	Evaluates *expression*. If true, returns t; otherwise, returns f.
_IFCASE(*expr*,*case_0*,...,*case_n*,*dflt*)	—	Evaluates *expr* to an integer m. If $0 \le m \le n$, then *case_m* is selected. Otherwise, the *dflt* is selected.
_IFDEF(*macro*,t,f)	—	If *macro* is defined, returns t; otherwise, returns f.
_IFNDEF(*macro*,t,f)	—	As above, but returns t if not defined.
_IFELSE(s1,s2,t,f)	—	Compares s1 to s2. If identical, returns t; otherwise, returns f.
_INCR(N)	—	Increment a macro.
_INPUT_LINE	—	Line number (in WEB source file) that begins current section.

20.12.23 Built-in FWEB macros (L–Z)

_L(*string*) — Changes *string* to lower case.

_LANGUAGE — An identifier such as '_C' depending on the current language. (See the table below under _LANGUAGE_NUM.) Intended to be used with an _IFELSE.

_LANGUAGE_NUM — An integer according to the following table; intended to be used with an _IFCASE.

Language	__LANGUAGE	_LANGUAGE_NUM
C	_C	0
C++	_CPP	1
FORTRAN–77	_N	2
FORTRAN–90	_N90	3
RATFOR–77	_R	4
RATFOR–90	_R90	5
TEX	_X	6

_LEN(string) — Length of (unexpanded) argument interpreted as a character string.

_M(defn) — Equivalent to _DEFINE.

_MAX(a,b) — Maximum of the two expressions a and b.

_MIN(a,b) — Minimum of a and b.

_MODULE_NAME — Name of present WEB module.

_MODULES — The total number of *independent* modules: namely, the total number of independent module names, plus 1 for the unnamed module.

_OUTPUT_LINE — Current line number of tangled output.

_P — The C preprocessor symbol '#'; a synonym for "_UNQUOTE("#")".

_POW(x,y) — Exponentiation: x^y.

_ROUTINE — In RATFOR mode, expands to a string built of the name of the current program, function, or subroutine; not useful for other languages, for which it expands to the empty string.

_SECTION_NUM — Number of current WEB section.

_SECTIONS — The maximum section number as understood by WEAVE.

_STRING(s) — Expands its argument, then stringizes it according to #*.

_STUB(name) — References to undefined modules are automatically replaced by a call to this macro, with the module name as argument.

_TIME — A string consisting of the local time in the form "19:59".

_TRANSLIT(*string*,*from*,*to*) — Interprets all arguments as character strings; replaces the *from* characters in s by the corresponding *to* characters.

_U(*string*) — Changes *string* to upper case.

_UNDEF(*macro*) — Undefine a macro.

_UNQUOTE(*string*) — Returns *string*, without the surrounding quotes.

_VERBATIM(*string*) — Obsolete name for _UNQUOTE.

_VERSION — A string built out of the FWEB version number—e.g., "1.30".

20.12.24 Ratfor commands

Select RATFOR–77 with '`@r`' or '`@r7`'; select RATFOR–90 with '`@r9`'. Disable RATFOR statement translation with command-line option '`-q`' (obsolete). In all cases, the construction {...} can be replaced by a simple statement terminated by a semicolon.

Ratfor–77 commands:

`break;`	— Exit loop or **switch** immediately.
`case` *i*`:`	— Used only inside **switch**.
`default:`	— Used only inside **switch**.
`do ...; {...}`	— FORTRAN's **do** statement. (The semicolon is required only when the **do** is followed by a simple statement; it is optional when followed by a left brace.)
`else {...}`	— Used in conjunction with **if**.
`for(`*a*`;`*b*`;`*c*`) {...}`	— Execute *a*. Test *b*. If true, execute body. Execute *c*. Test *b* again and iterate.
`if(`*condition*`) {...}`	— FORTRAN's **if**...**then**.
`next;`	— Go to bottom of loop.
`repeat {...} until(`*condition*`);`	— Execute body. If *condition* is true, iterate.
`return` *expression*`;`	— Return value from function.
`switch(`*expression*`) {...}`	— Select various cases. (Cases fall through unless terminated by **break**.)
`while(`*condition*`) {...}`	— If *condition* is true, execute body of loop.

Additional Ratfor–90 commands:

`contains:`	— Note the colon.
`interface` *name* `{...}`	— Used as in FORTRAN–90, but note the braces.
`interface operator(`*operator*`) {...}`	— As in FORTRAN–90.
`interface assignment(`*assignment*`) {...}`	— As in FORTRAN–90.
`module` *name* `{...}`	— As in FORTRAN–90.
`private:`	— Note the colon.
`sequence:`	— Note the colon.
`type` *name* `{...};`	— Note the semicolon.
`where(`*expression*`) {...}`	— FORTRAN–90 array operations; may be followed by an optional **else** clause.

Caviats and nuances about `FWEB` RATFOR:

1: Numeric statement labels must be followed by a colon; they should be first on their line.

2: The quoting convention for characters and strings follows that of C: Single-quote single characters, double-quote strings.

3: In a **switch**, cases fall through to the next case unless terminated by **break** (just as in C).

4: The **do** statement must be terminated by a semicolon if followed by a simple statement. (It's unnecessary if followed by a left brace that begins a compound statement.)

5: Use `&&` and `||` for the logical AND and OR.

6: Do not use an **end** statement at the very end of a program unit; it is added automatically when the closing brace is sensed.

20.12.25 Code mode and the principal `fwebmac` typesetting macros

The construction `|...|` signifies code mode; it may be used in TEX text (including comments and module names) to typeset code or identifiers between the bars. When code mode is used, entries are made in the index. Alternatively, the following macros may be used; these do not make entries in the index:

`\Wtypewriter{sample\_id}` — Typeset in typewriter type, such as "`sample_id`".
`\Wshort{x}` — Use for single-character identifiers, such as "*x*". (Do not use WEB's shorthand notation "`\|x`" for this purpose, as the bar gets confused with the entry into code mode.)
`\Wid{sample\_id}` — Ordinary identifiers, such as "*sample_id*".
`\Wreserved{sample\_id}` — For reserved words, such as "**sample_id**".
`\Wintrinsic{sample\_id}` — For intrinsic functions, such as "***sample_id***".

In the arguments of the above macros, you must precede the special characters "␣`\#%$^{}~&_`" by a backslash.

For brevity, the above macros are equivalenced to shorter macros, as follows:

Type of argument	`fwebmac` *macro*	*Style-file entry*	*Default value* `-PT (-PL)`
character string	`\Wtypewriter`	`format.typewriter`	`\.`
reserved word	`\Wreserved`	`format.reserved`	`\&`
single-character identifier	`\Wshort`	`format.short_identifier`	`|`
ordinary identifier	`\Wid`	`format.identifier`	`\\ (\>)`
outer macro	`\WidD`	`format.outer_macro`	`\\ (\>)`
WEB macro	`\WidM`	`format.WEB_macro`	`\\ (\>)`
intrinsic function	`\Wintrinsic`	`format.intrinsic`	`\@`
FORTRAN keyword	`\Wkeyword`	`format.keyword`	`\.`

[Note that when the LATEX processor is specified ('`-PL`' option), a few of the defaults are changed for convenience.]

20.12.26 Escape sequences

FWEB follows ANSI C in recognizing the following escape sequences within strings. (The corresponding **ASCII** code is in parentheses.)

`\'` (0x27) — *Literal apostrophe.*
`\"` (0x22) — *Literal quotation mark.*
`\?` (0x3F) — *Literal question mark.*
`\\` (0x5C) — *Literal backslash.*
`\a` (0x07) — *Alert*—ring the bell or print visual alert.
`\b` (0x08) — *Horizontal backspace.*
`\f` (0x0C) — *Form feed*—force output device to begin a new page.
`\n` (0x0A) — *Newline*—move to next line.
`\r` (0x0D) — *Carriage return*—move to beginning of line.
`\t` (0x09) — *Horizontal tab*—move to next tab mark.
`\v` (0x0B) — *Vertical tab*—move to next tab mark.
`\`*NNN* — *Octal number.*
`\x`*NN* — *Hexadecimal number.*

20.12.27 Conventions for FWEAVE's identifiers

Following are the interpretation of the various fonts used in the output produced by FWEAVE. Subscripts mean the number of the section in which the identifier was defined. In the index, underlined section numbers mean the identifier was defined there.

Automatically-generated entries:

italics	—	An ordinary identifier such as x or xyz.
mark$_{\underline{90}}$	—	An identifier explicitly marked with `@[`.
name$_{91}$	—	A function name such as *main* defined in section 98.
inner$_{\underline{92}}$	—	A WEB macro.
outer$_{93}$	—	An outer macro.
boldface	—	A reserved word such as **integer**.
newtype$_{\mathbf{95}}$	—	A new type created via **typedef**.
boldfaced italic	—	An intrinsic function such as ***sin***.
TYPEWRITER	—	A FORTRAN keyword such as BLOCKSIZE. (These must be in upper case.)

User-defined entries:

typewriter	—	`@....@>`
Roman	—	`@^...@>`
user-defined	—	`@9...@>`. For example, to make an index entry in sans serif type say "`\def\9#1{{\tenss#}}`".

20.12.28 Special array processing

In FORTRAN and RATFOR, FTANGLE replaces left and right square brackets (outside of strings) by left and right parentheses. Thus, brackets can be used for array subscripts if one desires.

When the option '`-W[`' is used, FWEAVE replaces square brackets by a special TEX macro. To change the appearance of array indices, redefine the macro `\WARRAY`. For example, to subscript indices, say "`\let \WARRAY\WSUB`".

20.13 APPENDIX M: CUSTOMIZING via the STYLE FILE

The default name of the style file is `fweb.sty`; change that with the '`-z`' command-line option. TEX-like comments (beginning with '`%`') may be included. An alphabetized list of the vocabulary commands may be found in the index under "`style file, vocabulary`". The command syntax is

keyword [=] value

For example,

```
LaTeX.options = "eqalign"
```

The style-file parameters are user-specific. The local style file is intended to be used for changes that are run-specific. (Contrast that with the initialization file `.fweb`, which is intended to set the user's default environment for all runs.) Style-file parameters that are intended to permanently override FWEB's defaults should be put into `.fweb` by using the '`+p`' option.

A mechanism is also provided to aid in installation-wide customization done when FWEB is compiled. This is explained in the separate documentation about installation and in the source file `custom.web`.

20.13.1 Customizing FWEAVE's index

`index.tex`	*(string)*	Name of the file into which the index is written. The character '`#`' is translated into the root name of the web file. **Default:** `"INDEX.tex"`.
`index.preamble`	*(string)*	TEX commands to start the index. **Default:** `"\\Winx"`.
`index.postamble`	*(string)*	TEX commands to end the index. **Default:** `"\\Wfin"`.
`index.collate`	*(string)*	Collating sequence for the index.
`group_skip`	*(string)*	TEX commands to insert between letter groups. (A letter group is all index entries that begin with the same character.) **Default:** `""`.
`lethead.prefix`	*(string)*	(Partial) TEX command to begin identifying letter at start of group. The letter starting the next group is inserted immediately following this string. **Default:** `""`.
`lethead.suffix`	*(string)*	(Partial) TEX command to insert after identifying letter. **Default:** `""`.
`lethead.flag`	*(integer)*	This controls the kind of letter that is inserted between `lethead_prefix` and `lethead_suffix`. If `lethead_flag` is 0, no letter is inserted. If `lethead_flag` > 0, an uppercase letter is inserted. If `lethead_flag` < 0, a lowercase letter is inserted. **Default:** `0`.
`item_0`	*(string)*	TEX command to begin an index entry. **Default:** `"\\:"`.
`delim_0`	*(string)*	String to insert after the identifier in an index entry. **Default:** `",␣"`.
`delim_n`	*(string)*	String to insert between two module numbers in an index entry. **Default:** `",␣"`.
`underline.prefix`	*(string)*	(Partial) TEX command to begin an underlined index entry. **Default:** `"\\["`.
`underline.suffix`	*(string)*	(Partial) TEX command to end an underlined index entry. **Default:** `"]"`.
`language.prefix`	*(string)*	(Partial) TEX command to begin a language reference in the index. **Default:** `"\\("`.
`language.suffix`	*(string)*	(Partial) TEX command to end a language reference. **Default:** `")"`.

20.13.2 Customizing FWEAVE's module list

`modules.tex`	*(string)*	Name of the file into which the module names are written. The character '`#`' is translated into the root name of the `web` file. **Default:** `"MODULES.tex"`.
`modules.preamble`	*(string)*	TEX commands to begin the list of modules. **Default:** `"\\Wmods"`.
`modules.postamble`	*(string)*	TEX commands to end the list of modules. **Default:** `""`.
`modules.info`	*(string)*	TEX macro name that formats the command line and related information. **Default:** `"\\Winfo"`.

20.13.3 Customizing FWEAVE's table of contents

`contents.tex`	*(string)*	Name of the file into which the table of contents is written. The character '`#`' is translated into the root name of the web file. **Default:** `"CONTENTS.tex"`.
`contents.preamble`	*(string)*	TEX string that begins printing the table of contents. **Default:** `"\n\\Wcon"`.
`contents.postamble`	*(string)*	TEX string that ends the table of contents. **Default:** `""`.

20.13.4 Customizing cross-reference subscripts

`mark_defined.generic_name`	*(boolean)*	Identifier explicitly marked by `@[`. **Default:** `1`.
`mark_defined.fcn_name`	*(boolean)*	Function name. **Default:** `0`.
`mark_defined.WEB_macro`	*(boolean)*	WEB macro. **Default:** `0`.
`mark_defined.outer_macro`	*(boolean)*	Outer macro. **Default:** `0`.
`mark_defined.exp_type`	*(boolean)*	Identifier explicitly marked by `@``. **Default:** `1`.
`mark_defined.typedef_name`	*(boolean)*	A **typedef**-like statement in C or C++. **Default:** `0`.

20.13.5 Overriding or completing definitions in `fwebmac.sty`

`format.reserved`	*(string)*	The macro to use to format reserved words such as **integer**. **Default:** `"\&"`.
`format.short_identifier`	*(string)*	As above, but for single-character identifiers. **Default:** `"|"`.
`format.identifier`	*(string)*	As above, but for ordinary identifiers. **Default:** `"\\"`.
`format.outer_macro`	*(string)*	As above, but for outer macros (defined with '`@d`'). **Default:** `"\\"`.
`format.WEB_macro`	*(string)*	As above, but for WEB macros (defined with '`@m`'). **Default:** `"\\"`.
`format.intrinsic`	*(string)*	As above, but for intrinsic functions such as ***sin***. **Default:** `"\@"`.
`format.keyword`	*(string)*	As above, but for FORTRAN keywords such as `BLOCKSIZE`. **Default:** `"\."`.
`format.typewriter`	*(string)*	The macro that generates typewriter type. **Default:** `"\."`.
`format.wildcard`	*(string)*	The macro for user-defined entries in the table of contents. **Default:** `"\9"`.
`indent.TeX`	*(string)*	Paragraph indentation for the TEX part. (Was formerly `parindent`.) **Default:** `"1em"`.
`indent.code`	*(string)*	Paragraph indentation for the code part. (Formerly was same as `indent.TeX`.) **Default:** `"1em"`.
`LaTeX.options`	*(string)*	When running under LATEX, the document is (effectively) begun by the command "`\documentstyle[`*options*`]{`*style*`}`." This string sets the *options* field. **Default:** `""`.
`LaTeX.style`	*(string)*	As above, but sets the *style* field. **Default:** `"article"`.

20.13.6 Miscellaneous customization commands for FWEAVE

`macros`	*(string)*	The default name of the macro package to be read in. (Overridden by the command-line option '`-w`'.) **Default:** `"fwebmac.sty"`.
`limbo`	*(string)*	TEX material to be printed at the beginning of the limbo section, just before the text from '`@l`' commands. **Default:** `""`.
`meta.TeX.begin`	*(string)*	TEX macros that initiate the verbatim environment for the '`@(`' command. **Default:** `"\\WBM\\Begintt\n"`. (TEX) or **Default:** `"\\WBM\\begin{verbatim}\n"`. (LATEX)
`meta.TeX.end`	*(string)*	TEX macros that terminate the verbatim environment for the '`@(`' command. **Default:** `"\\Endtt\\WEM"`. (TEX) or **Default:** `"\\end{verbatim}\\WEM"`. (LATEX).
`named_preamble`	*(string)*	TEX macros to be emitted immediately after the start of a named modules. **Default:** `""`.
`unnamed_preamble`	*(string)*	TEX macros to be emitted immediately after the start of an unnamed module. **Default:** `""`.

20.13.7 Customizations for FTANGLE

ASCII_fcn	*(string)*	In FORTRAN, the command @"..." is tangled to *string*('...') if *string* is non-empty. If it is empty, the parentheses are omitted. **Default:** "ASCIIstr".
cchar	*(character)*	Continuation character for FORTRAN code output by FTANGLE. This character must be printable, non-blank, and non-zero. **Default:** '&'.
cdir_start.C	*(string)*	Insert immediately after '@?' when the language is C. **Default:** "#pragma␣".
cdir_start.Cpp	*(string)*	As above, but for C++. **Default:** "#pragma␣".
cdir_start.K	*(string)*	As above, for MAKE. **Default:** "".
cdir_start.N	*(string)*	As above, for FORTRAN–77. **Default:** "C".
cdir_start.N90	*(string)*	As above, for FORTRAN–90. **Default:** "C".
cdir_start.R	*(string)*	As above, for RATFOR–77. **Default:** "C".
cdir_start.R90	*(string)*	As above, for RATFOR–90. **Default:** "C".
cdir_start.X	*(string)*	As above, for TEX. **Default:** "".
line_length.N	*(integer > 0)*	Line length for FORTRAN code output by FTANGLE. **Default:** 72.
meta.top.*l*	*(string)*	Text that precedes the body of material enclosed by '@('...'@)' (meta-comment). (Here $l \in$ {C,Cpp,N,N90,R,R90,X}.) **Default:** "".
meta.prefix.*l*	*(string)*	Each line of the meta-comment is begun by *string*. **Default:** "".
meta.bottom.*l*	*(string)*	Like meta.top, but follows the meta-comment. **Default:** "".
suffix.C	*(string)*	Extension for the C output file. **Default:** "c".
suffix.Cpp	*(string)*	As above, for C++. **Default:** "c++".
suffix.N	*(string)*	As above, for FORTRAN–77. **Default:** "f".
suffix.N90	*(string)*	As above, for FORTRAN–90. **Default:** "".
suffix.R	*(string)*	As above, for RATFOR–77. **Default:** "r".
suffix.R90	*(string)*	As above, for RATFOR–90. **Default:** "".
suffix.X	*(string)*	As above, for TEX. **Default:** "sty".

20.13.8 Miscellaneous customizations for both FTANGLE and FWEAVE

null_file	*(string)*	Name of the null file. (Default depends on the operating system.)
dot_constant.begin	*(character)*	Delimiter that replaces beginning period in FORTRAN "dot constants" such as '.EQ.'. **Default:** '.'.
dot_constant.end	*(character)*	As above, but replaces ending period. **Default:** '.'.

20.13.9 Automatic file-name completion

`ext.web`	*(string)*	Extensions (space-delimited if more than one) for the web file (first file name on command line). **Default:** `"web"`.
`ext.change`	*(string)*	Extensions (space-delimited) for the change file (second file name on command line). **Default:** `"ch"`.
`ext.hweb`	*(string)*	Extensions (space-delimited) for an include file (first file name on an `@i` line). **Default:** `"hweb"`.
`ext.hchange`	*(string)*	Extensions (space-delimited) for change files associated with include files (second file name on an `@i` line). **Default:** `"hch"`.

20.13.10 Colors

`color.mode`	*(integer)*	Selects one of several color settings. **Default:** `0`.
`color.ordinary`	*(string)*	Color of otherwise unspecified fields. **Default:** `"default"`.
`color.program`	*(string)*	Color of program name. **Default:** `"yellow"`.
`color.info`	*(string)*	Color of information messages. **Default:** `"green"`.
`color.warning`	*(string)*	Color of warning messages. **Default:** `"default"`.
`color.error`	*(string)*	Color of error messages. **Default:** `"red"`.
`color.fatal`	*(string)*	Color of fatal messages. **Default:** `"red"`.
`color.module_num`	*(string)*	Color of module numbers. **Default:** `"orange"`.
`color.line_num`	*(string)*	Color of line numbers. **Default:** `"orange"`.
`color.input_file`	*(string)*	Color of input file names. **Default:** `"yellow"`.
`color.include_file`	*(string)*	Color of include file names. **Default:** `"blue"`.
`color.output_file`	*(string)*	Color of output file names. **Default:** `"yellow"`.
`color.timing`	*(string)*	Color of the timing information. **Default:** `"default"`.
`color.default`	*(string)*	Escape sequences for the default color. **Default:** `"me"`.
`color.red`	*(string)*	As above, for red. **Default:** `"md mr"`.
`color.green`	*(string)*	As above, for green. **Default:** `"md"`.
`color.blue`	*(string)*	As above, for blue. **Default:** `"me"`.
`color.orange`	*(string)*	As above, for orange. **Default:** `"me"`.
`color.yellow`	*(string)*	As above, for yellow. **Default:** `"md"`.

20.13.11 Customizing FWEB's control codes

ascii_constant	*(string)*	Signify an **ASCII** constant. **Default:** "'"".
begin_C	*(string)*	Switch into C language. **Default:** "cC".
begin_FORTRAN	*(string)*	Switch into FORTRAN language. **Default:** "nN".
begin_meta	*(string)*	Start a meta comment. **Default:** "(".
begin_RATFOR	*(string)*	Switch into RATFOR language. **Default:** "rR".
begin_code	*(string)*	Start the unnamed module. **Default:** "aA".
big_line_break	*(string)*	A new line with some extra space. **Default:** "#".
compiler_directive	*(string)*	Signify a compiler directive. **Default:** "!".
defd_at	*(string)*	Explicitly mark as defined. **Default:** "[".
definition	*(string)*	Define an outer macro. **Default:** "dD".
end_meta	*(string)*	End a meta comment. **Default:** ")".
force_line	*(string)*	Force a new line. **Default:** "\\".
format	*(string)*	Format an identifier or module name. **Default:** "fF".
explicit_reserved	*(string)*	Format as reserved (integer-like) word. **Default:** "‘".
insert_bp	*(string)*	When breakpointing is turned on, insert an explicit breakpoint instruction here. **Default:** "}bB".
invisible_cmnt	*(string)*	Comment ignored by both processors. **Default:** "%".
join	*(string)*	Join two items together on output. **Default:** "&".
math_break	*(string)*	Insert a math break. **Default:** "|".
module_name	*(string)*	Begin a module name. **Default:** "<".
no_line_break	*(string)*	Cancel a line break here. **Default:** "+".
pseudo_colon	*(string)*	A pseudo-colon. **Default:** ":".
pseudo_expr	*(string)*	A pseudo-expression. **Default:** "e".
pseudo_semi	*(string)*	A pseudo-semicolon. **Default:** ";".
switch_math_flag	*(string)*	Toggle the math flag. **Default:** "$".
TeX_string	*(string)*	Insert a TEX string of commands. **Default:** "tT".
thin_space	*(string)*	Insert a thin space. **Default:** ",".
undefinition	*(string)*	Undefine an outer macro. **Default:** "uU".
underline	*(string)*	Underline the following entry in the index. **Default:** "_".
verbatim	*(string)*	Send control text verbatim to output. **Default:** "=".
WEB_definition	*(string)*	Define an inner or WEB macro. **Default:** "mM".
xref_roman	*(string)*	Typeset index entry in Roman type. **Default:** "^".
xref_typewriter	*(string)*	Typeset index entry in typewriter type. **Default:** ".".
xref_wildcard	*(string)*	Typeset index entry using the macro \9. **Default:** "9".

20.14 APPENDIX N: MEMORY ALLOCATION

The command-line option '`-y`' is used to change the default allocation for a dynamic memory array, as in '`-ym4000`'. To query the present allocations of variable *aa*, where *aa* is the abbreviation in the list below, just say "`-y`*aa*" with no numeric argument. To query everything, say "`-y`".

The option '`-s`' reports memory-usage statistics at the end of the run. The option '`-sm`*[n]*' reports allocations of n or more bytes as they occur. If n is omitted, $n = 10000$ is assumed.

Here is a brief discussion *(not completed yet!)* of the dynamic arrays and their abbreviations. (For more information, please study the code.)

buf_size (`"bs"`) — Size of the change buffer.
C_buf_size (`"cb"`) — Buffer size for single-character buffered output in C.
cmd_fmt_size (`"cf"`) — Buffer size for certain output messages in RATFOR.
cmd_msg_size (`"cg"`) — As above.
delta_dots (`"d"`) — Number of additional entries to reallocate for the *dots* array if necessary.
line_length (`"ll"`) — Line length for `FWEAVE`'s output.
longest_name (`"ln"`) — Module names or strings shouldn't be longer than this.
max_bytes (`"b"`) — Maximum number of bytes in identifiers, index entries, and module names.
max_dtexts (`"dx"`) — Maximum number of deferred replacement texts.
max_dtoks (`"dt"`) — Maximum number of tokens in `FTANGLE`'s deferred macro pool.
max_expr_chars (`"lx"`) — Maximum length of expressions for compound assignments.
max_lbls (`"lb"`) — Maximum nesting level in RATFOR.
max_modules (`"m"`) — Must be larger than the maximum number of modules.
max_names (`"n"`) — Maximum number of identifiers, strings, and module names.
max_refs (`"r"`) — Maximum number of cross-references.
max_scraps (`"s"`) — Maximum number of scraps during `FWEAVE`'s parsing.
max_texts (`"x"`) — Maximum number of replacement texts for `FTANGLE`.
max_toks (`"tt"`) — Maximum number of tokens in `FTANGLE`'s compressed code.
max_toks (`"tw"`) — Maximum number of tokens in current code text being parsed by `FWEAVE`.
mbuf_size (`"mb"`) — Size of the area into which macros are expanded. This must be large enough to hold all intermediate levels of expansion as well as the final result. Furthermore, in some complicated situations, especially in RATFOR, more than one macro buffer can be open at once.
num_files (`"nf"`) — Number of open files, especially for the `@o` command.
op_entries (`"op"`) — Size of the table that handles overloaded operators. A fixed table of length 128 is always used to handle operators such as '`=`'. The quantity *op_entries* must be greater than that amount by the number of new names that are explicitly overloaded.
sbuf_len (`"sb"`) — Length of input line buffer for style file.
stack_size (`"kt"`) — `FTANGLE`'s stack size.
stack_size (`"kw"`) — `FWEAVE`'s stack size.
X_buf_size (`"xb"`) — Size of TEX's output buffer.

Thus, for example, to set the maximum number of modules to be 4000, say "`-ym4000`".

20.15 APPENDIX O: CHARACTER SETS

FWEB works internally with the ASCII character set. Users of some IBM machines may need to be familiar with the EBCDIC character set as well.

20.15.1 The ASCII character set

Here is the ASCII character set, shown in octal, decimal, and hexadecimal. The escape sequences recognized by C and FWEB are also shown where appropriate. [This table is a minor modification of that given in the excellent book by the Mark Williams Company, *ANSI C: A Lexical Guide* (Prentice Hall, Englewood Cliffs, New Jersey, 1988), p. 66.]

000	0	0x00	NUL	⟨**ctrl**–@⟩	Null character
001	1	0x01	SOH	⟨**ctrl**–A⟩	Start of header
002	2	0x02	STX	⟨**ctrl**–B⟩	Start of text
003	3	0x03	ETX	⟨**ctrl**–C⟩	End of text
004	4	0x04	EOT	⟨**ctrl**–D⟩	End of transmission
005	5	0x05	ENQ	⟨**ctrl**–E⟩	Enquiry
006	6	0x06	ACK	⟨**ctrl**–F⟩	Positive acknowledgement
007	7	0x07	BEL	⟨**ctrl**–G⟩	Alert (“bell”) (’\a’)
010	8	0x08	BS	⟨**ctrl**–H⟩	Backspace (’\b’)
011	9	0x09	HT	⟨**ctrl**–I⟩	Horizontal tab (’\t’)
012	10	0x0A	LF	⟨**ctrl**–J⟩	Line feed (“newline”) (’\n’)
013	11	0x0B	VT	⟨**ctrl**–K⟩	Vertical tab (’\v’)
014	12	0x0C	FF	⟨**ctrl**–L⟩	Form feed (’\f’)
015	13	0x0D	CR	⟨**ctrl**–M⟩	Carriage return (’\r’)
016	14	0x0E	SO	⟨**ctrl**–N⟩	Shift out
017	15	0x0F	SI	⟨**ctrl**–O⟩	Shift in
020	16	0x10	DLE	⟨**ctrl**–P⟩	Data link escape
021	17	0x11	DC1	⟨**ctrl**–Q⟩	Device control 1 (XON)
022	18	0x12	DC2	⟨**ctrl**–R⟩	Device control 2 (tape on)
023	19	0x13	DC3	⟨**ctrl**–S⟩	Device control 3 (XOFF)
024	20	0x14	DC4	⟨**ctrl**–T⟩	Device control 4 (tape off)
025	21	0x15	NAK	⟨**ctrl**–U⟩	Negative acknowledgement
026	22	0x16	SYN	⟨**ctrl**–V⟩	Synchronize
027	23	0x17	ETB	⟨**ctrl**–W⟩	End of transmission block
030	24	0x18	CAN	⟨**ctrl**–X⟩	Cancel
031	25	0x19	EM	⟨**ctrl**–Y⟩	End of medium
032	26	0x1A	SUB	⟨**ctrl**–Z⟩	Substitute
033	27	0x1B	ESC	⟨**ctrl**–[⟩	Escape
034	28	0x1C	FS	⟨**ctrl**–\⟩	Form separator
035	29	0x1D	GS	⟨**ctrl**–]⟩	Group separator
036	30	0x1E	RS	⟨**ctrl**–^⟩	Record separator
037	31	0x1F	US	⟨**ctrl**–_⟩	Unit separator

040	32	0x20	␣	Space
041	33	0x21	!	Exclamation point
042	34	0x22	"	Quotation mark (’\"’)
043	35	0x23	#	Pound (sharp) sign
044	36	0x24	$	Dollar sign
045	37	0x25	%	Percent sign
046	38	0x26	&	Ampersand
047	39	0x27	’	Apostrophe (right quote) (’\’’)
050	40	0x28	(	Left parenthesis
051	41	0x29	)	Right parenthesis
052	42	0x2A	*	Asterisk
053	43	0x2B	+	Plus sign
054	44	0x2C	,	Comma
055	45	0x2D	-	Hyphen (minus sign)
056	46	0x2E	.	Period
057	47	0x2F	/	Virgule (slash)
060	48	0x30	0	
061	49	0x31	1	
062	50	0x32	2	
063	51	0x33	3	
064	52	0x34	4	
065	53	0x35	5	
066	54	0x36	6	
067	55	0x37	7	
070	56	0x38	8	
071	57	0x39	9	
072	58	0x3A	:	Colon
073	59	0x3B	;	Semicolon
074	60	0x3C	<	Less-than (left angle bracket)
075	61	0x3D	=	Equal sign
076	62	0x3E	>	Greater-than (right angle bracket)
077	63	0x3F	?	Question mark (’\?’)

0100	64	0x40	`@`	At sign
0101	65	0x41	`A`	
0102	66	0x42	`B`	
0103	67	0x43	`C`	
0104	68	0x44	`D`	
0105	69	0x45	`E`	
0106	70	0x46	`F`	
0107	71	0x47	`G`	
0110	72	0x48	`H`	
0111	73	0x49	`I`	
0112	74	0x4A	`J`	
0113	75	0x4B	`K`	
0114	76	0x4C	`L`	
0115	77	0x4D	`M`	
0116	78	0x4E	`N`	
0117	79	0x4F	`O`	
0120	80	0x50	`P`	
0121	81	0x51	`Q`	
0122	82	0x52	`R`	
0123	83	0x53	`S`	
0124	84	0x54	`T`	
0125	85	0x55	`U`	
0126	86	0x56	`V`	
0127	87	0x57	`W`	
0130	88	0x58	`X`	
0131	89	0x59	`Y`	
0132	90	0x5A	`Z`	
0133	91	0x5B	`[`	Left bracket
0134	92	0x5C	`\`	Backslash (`'\\'`)
0135	93	0x5D	`]`	Right bracket
0136	94	0x5E	`^`	Circumflex
0137	95	0x5F	`_`	Underscore

0140	96	0x60	`` ` ``	Grave (left quote)
0141	97	0x61	a	
0142	98	0x62	b	
0143	99	0x63	c	
0144	100	0x64	d	
0145	101	0x65	e	
0146	102	0x66	f	
0147	103	0x67	g	
0150	104	0x68	h	
0151	105	0x69	i	
0152	106	0x6A	j	
0153	107	0x6B	k	
0154	108	0x6C	l	
0155	109	0x6D	m	
0156	110	0x6E	n	
0157	111	0x6F	o	
0160	112	0x70	p	
0161	113	0x71	q	
0162	114	0x72	r	
0163	115	0x73	s	
0164	116	0x74	t	
0165	117	0x75	u	
0166	118	0x76	v	
0167	119	0x77	w	
0170	120	0x78	x	
0171	121	0x79	y	
0172	122	0x7A	z	
0173	123	0x7B	{	Left brace
0174	124	0x7C	|	Vertical bar
0175	125	0x7D	}	Right brace
0176	126	0x7E	~	Tilde
0177	127	0x7F	DEL	Delete

20.15.2 The EBCDIC character set

This will be completed someday.

20.16 APPENDIX P: INDEX

The page numbers in the index for this manual can appear in a variety of fonts. These have the following meaning:

Roman	— The keyword or phrase is mentioned here.
Roman	— Definition of concept or keyword.
boldfaced	— Reference to an entire **topic**.
italics	— An *example* is given.

— FWEB USER'S MANUAL —

Version 1.30
June 15, 1993

Page