The Spidery WEB System of Structured Documentation

This memo describes how to write programs in the WEB *language, using* WEB *systems generated by Spider. This class of* WEB *systems will be referred to (collectively) as "Spidery* WEB*." Most of the material is taken verbatim from Donald Knuth's original memo introducing* WEB*; new material (like this paragraph) appears in slant type. Knuth's original memo is available as Stanford University CS technical report number 980 (September 1983). Unlike Knuth's, this memo is not accompanied by the full* WEB *documentation for Spidery* WEAVE *and* TANGLE*, the programs that read* WEB *input and produce TEX output and code, respectively. Nor does it contain the* WEB *documentation for Spider, the program that generates the language-dependent parts of* WEAVE *and* TANGLE*. This memo should be accompanied by "A Spider User's Guide," which explains how to use Spider to build a Spidery* WEB *system.*

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 for programming. *For convenience, we'll give this programming language a name; let's call it the "X" programming language.* 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 *"X"* 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. *Spidery* WEB *is also an experimental system and the same warning applies. Before continuing, you may want to read some more introductory material on Spider and* WEB*. Here are some references worth looking at:*

- Donald E. Knuth, "Literate Programming," *Computer Journal* **27:2** (1984), 97–111.. The original introduction of WEB.
- Jon L. Bentley, "Programming Pearls," *Communications of the ACM* **29:5** (May 1986), 364–368 and **29:6** (June 1986), 471–483. Two columns on literate programming with WEB.
- Norman Ramsey, "Building a Language-Independent WEB," *Communications of the ACM*, September 1989. An overview of how Spider and WEB work.
- Wayne Sewell, "Weaving a program: Literate programming in WEB," Van Nostrand Reinhold, 1989.

In addition to these, there is a periodic column in the Communications of the ACM devoted to literate programming.

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 *"X"*. 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 *"X"*. 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.

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 a file `COB.`*X* that contains the "*X*" code of your `COB` program. The "*X*" compiler will convert `COB.`*X* 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.`*X*, because `COB.WEB` contains the definitive source code. Examples of the behavior of `WEAVE` and `TANGLE` are *not* appended to this manual. *The names* `WEAVE` *and* `TANGLE` *may vary, especially since there may be versions for multiple languages. My site supports, in addition to the original* `TANGLE`, `CEETANGLE`, `AWKTANGLE`, *and* `ADATANGLE`.

Besides providing a documentation tool, `WEB` enhances the "*X*" language by providing a 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. 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 "*X*"; 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 "*X*" 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*"!

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 my wife's mother, Wilda Ernestine Bates.

The work on Spidery `WEB` *would not have been possible without the help of Silvio Levy, who first ported* `WEB` *to C. The name Spider was chosen because (in nature at least) webs are built by spiders.*

The appendices to this report *do not* contain complete `WEB` programs for the *Spider*, `WEAVE`, and `TANGLE` processors. 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.

General rules. A `WEB` file is a long string of text that has been divided into individual lines. *Unlike original* `WEB`, *Spidery* `WEB` *respects the programmer's choice of line breaks. A newline in code is normally passed along through tangle and will appear in the resulting program. This is important in languages like awk where line breaks are significant. In documentation,* 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 line length of the input is limited and is rather small.*

Two kinds of material go into `WEB` files: TeX text and "*X*" text. A programmer writing in `WEB` should be thinking both of the documentation and of the "*X*" 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." "*X*" 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 "*X* mode" that is added to TeX's horizontal mode, vertical mode, and math mode.

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

1) A TeX part, containing explanatory material about what is going on in the module.
2) A definition part, containing macro definitions that serve as abbreviations for "*X*" constructions that would be less comprehensible if written out in full each time.
3) An "*X*" part, containing a piece of the program that `TANGLE` will produce. This "*X*" 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.

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 "*X*" code. Any of the parts may be empty.

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` and copied essentially verbatim by `WEAVE`, so its function is 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.

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 "*X*" 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 '`@>`'. When `WEAVE` outputs a module name, it replaces the '`@<`' and '`@>`' by angle brackets and inserts the module number in small type. Thus, when you read the output of `WEAVE` it is easy to locate any module that is referred to in another module.

Spidery `WEB` *supports a new kind of module called a "file module." File modules open with* `@(` *instead of* `@<`*, and the module name is taken to be the name of a file to which* `TANGLE` *will write the contents of the module. When* `WEAVE` *outputs a file module name, it replaces the '*`@(`*' and '*`@>`*' by parentheses and sets the entire name in* `typewriter` *font.*

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 an "X" part. The definition part begins with the first appearance of '@d' or '@f' in the module, and the "X" part begins with the first appearance of '@u', '@<', *or* '@('.The options '@<' *and* '@('.stand 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 "X" text that follows, so you say '⟨ module name ⟩ = "X" text'. Alternatively, if the "X" part begins with '@u' instead of a module name, the current module is said to be *unnamed.* Note that module names cannot appear in the definition part of a module, because the first '@<' *or* '@('.in a module signals the beginning of its "X" part. Any number of module names might appear in the "X" part, however, once it has started.

The general idea of TANGLE is to make an "X" program out of these modules in the following way: First all the "X" 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 *unless there are file modules* there will be no program.) Then all module names that appear in the initial text T_0 are replaced by the "X" parts of the corresponding modules, and this substitution process continues until no module names remain. Then all defined macros are replaced by their equivalents, according to certain rules that are explained later. All comments will have been removed from this "X" program except for 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. *Spidery* WEB *does not support the use of the meta-comments '*@{*' and '*@}*'. Their absence can be partially compensated for by clever use of the "verbatim" control sequence '*@=...@>*'.*

Incidentally, module numbers are automatically inserted as meta-comments into the "X" program, in order to help correlate the outputs of WEAVE and TANGLE (see Appendix C). *So are* #line *directives, the exact workings of which are obscure (but see the "Spider User's Guide" for more information).*

If the same name has been given to more than one module, the "X" text for that name is obtained by putting together all of the "X" parts in the corresponding modules. This feature is useful, for example, in a module named 'Global variables in the outer block', since one can then declare global variables in whatever modules those variables are introduced. 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 "X" 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 ⟩ ≡ "X" 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 "X" text that follows is being appended to the "X" text of another module.

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 be '\input *X*web'; this will cause TEX to read in the macros that define WEB's documentation conventions *for the "X" programming language.* 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 "X" 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.

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.

Spidery `WEB` *supports a hack that makes the module structure appear more hierarchical in the table of contents. If the title begins with the special character* `=`*, or one of the digits* `1` *through* `4`*, it will be formatted specially. This table shows the effect of the special titles:*

Marker	**Level**	**Page Eject?**	**Other**
`@*=`	*chapter*	*yes*	*Bold in toc*
`@*`	*section*	*yes*	
`@*1`	*subsection*	*yes*	
`@*2`	*subsection*	*no*	
`@*3`	*subsubsection*	*yes*	
`@*4`	*subsubsection*	*no*	

Caution: Do not use TeX control sequences in such titles, unless you know that the `webmac` 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.

Second caution: do not allow the titles to begin with a non-letter, unless the non-letter is the control sequence `\relax`*. The reason is that the first character of the titles is checked to see whether it is the special character* `=`*, or one of the digits* `1` *through* `4`*. A non-letter that is not one of these will only confuse the formatter.*

The TeX output produced by `WEAVE` for each module consists of the following: First comes the module number (e.g., '`\M123.`' at the beginning of module 123, except that '`\N`' appears in place of '`\M`' 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 "X" part, formatted so that there will be a little extra space between them if both are nonempty. The definition and "X" parts are obtained by inserting a bunch of funny looking TeX macros into the "X" program; these macros handle typographic details about fonts and proper math spacing, as well as line breaks and indentation.

When you are typing TeX text, you will probably want to make frequent reference to variables and other quantities in your "X" 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 "X" editing within TeX text, if you place '`|`' marks before and after the "X" material. For example, suppose *you are writing Pascal code and* you want to say something like this:

The characters are placed into *buffer*, which is a **packed array** $[1 \to n]$ **of** *char*.

The TeX text would look like this in your `WEB` file:

```
The characters are placed into |buffer|, which is a |packed array [1..n] of char|.
```

And `WEAVE` translates this into something you are glad you didn't have to type:

```
The characters are placed into \\{buffer},
which is a \&{packed} \&{array} $ [1\to\|n]$ \&{of} \\{char}.
```

Incidentally, the cross-reference index that WEAVE would make, in the presence of a comment like this, would include the current module number as one of the index entries for *buffer* and *char*, even though *buffer* and *char* might not appear in the "X" 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 **packed** and **array** and n and **of** 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.

Speaking of identifiers, the author of WEB thinks that *IdentifiersSeveralWordsLong* look terribly ugly when they mix uppercase and lowercase letters. He recommends that *identifiers_several_words_long* be written with underline characters to get a much better effect. The WEAVE processor will properly alphabetize identifiers that have embedded underlines when it makes the index. *Using Spidery* WEB*, you're out of luck if programming language "X" doesn't support underlines in identifiers.*

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

Macros. *In the current implementation of Spidery* WEB, *a* WEB *programmer can define two kinds of macros to make the programs shorter and more readable.*

'`@d` *identifier* = "X" *text' defines a simple macro, where the identifier will be replaced by the "X" text when* TANGLE *produces its output.*

'`@d` *identifier* (*formal parameters*) = "X" *text' defines a parametric macro, where the identifier will be replaced by the "X" text and where occurrences of the formal parameters in that "X" text will be replaced by arguments. For C programmers, this is similar to the C macro processor, but the* = *sign is used to avoid the need for bizarre lexical requirements. Whitespace can appear between a macro name and a list of formal paramters, and newlines in macros need not be escaped with backslashes. A macro definition is terminated by a following* `@d`, `@f`, `@u`, `@<`, *or* `@(`.

Spidery WEB *does not support a string pool file. Pleas for its restoration will be heard.*

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 '@'. *In Spidery* WEB *the "magic at sign" need not be '@'; the person who built your* WEB *(with Spider) may have defined it to be something else. By convention, people building Spidery* WEB*s use '@' as the magic at sign, except when it is deemed unsuitable for some reason, in which case cases they use '#'.*

Here is a complete list of the legal control codes. The letters L, T, X, M, C, and/or S following each code indicate whether or not that code is allowable in limbo, in TEX text, in "X" text, in module names, in comments, and/or in strings. 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.

`@@` $[C, L, M, X, S, T]$ A double @ denotes the single character '@'. This is the only control code that is legal in limbo, in comments, and in strings.

`@i` $[L, T]$ *The* `@i` *control code should always be followed by a file name and a newline. It causes the file to be included at the current point in the* WEB *source, just like the C* `#include` *facility.*

`@␣` $[\overline{L}, \overline{X}, \overline{T}]$ 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.

`@*` $[\overline{L}, \overline{X}, \overline{T}]$ 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. *As explained above,* *the first character of the title should be either a letter or the control sequence* `\relax`*, unless special formatting is desired. Titles beginning with* `=`*,* `1`*,* `2`*,* `3`*, or* `4` *are formatted specially.* As explained above, TEX control sequences should be avoided in such titles unless they are quite simple. When `WEAVE` and `TANGLE` read a `@*`, they print 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.

`@d` $[\overline{X}, \overline{T}]$ Macro definitions begin with `@d` (or `@D`), followed by the "X" text for one of the three kinds of macros, as explained earlier.

`@f` $[\overline{X}, \overline{T}]$ Format definitions begin with `@f` (or `@F`); they cause `WEAVE` to treat identifiers in a special way when they appear in "X" text. The general form of a format definition is '`@f` l r', followed by an optional comment enclosed in braces, where l and r are identifiers; `WEAVE` will subsequently treat identifier l as it currently treats r. This feature allows a `WEB` programmer to invent new reserved words and/or to unreserve some of "X"'s reserved identifiers. *It is offensive that the syntax is different from the* `2d` *syntax, but it is that way for historical reasons (to avoid invalidating existing code.* The definition part of each module consists of any number of macro definitions (beginning with `@d`) and format definitions (beginning with `@f`), intermixed in any order.

`@u` $[\overline{X}, \overline{T}]$ *The "X" part of an unnamed module begins with* `@u` *(or* `@U`*).* This causes `TANGLE` to append the following "X" code to the initial program text T_0 as explained above. The `WEAVE` processor does not cause a '`@u`' 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 `@u` in the appropriate places of the unnamed modules.

`@<` $[X, \overline{T}]$ 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 "X" 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 `@>`. Module names may not appear in "X" text that is enclosed in `|...|`, nor may they appear in the definition part of a module (since the appearance of a module name ends the definition part and begins the "X" part).

`@(` $[X, \overline{T}]$ *A module beginning with* `@(` *is treated like one beginning with* `@<`*, except that the module name is treated as the name of a file to which the module will be written. If such modules are used in other modules the results are undefined.*

`@'` $[X, T]$ This denotes an octal constant, to be formed from the succeeding digits. For example, if the `WEB` file contains '`@'100`', the `TANGLE` processor will treat this an equivalent to '`64`'; the constant will be formatted as "*'100*" in the TEX output produced via `WEAVE`. You should use octal notation only for positive constants; don't try to get, e.g., -1 by saying '`@'777777777777`'.

`@"` $[X, T]$ A hexadecimal constant; '`@"D0D0`' tangles to `53456` and weaves to '"`D0D0`'.

`@&` $[X]$ The `@&` operation causes whatever is on its left to be adjacent to whatever is on its right, in the "X" 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.

`@^` $[X, T]$ The "control text" that follows, up to the next '`@>`', will be entered into the index together with the identifiers of the "X" 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, like a string, 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!`'.)

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

`@:` $[X, T]$ 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.

`@t` [X] The "control text" that follows, up to the next '`@>`', will be put into a TeX `\hbox` and formatted along with the neighboring "X" 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 "X" and classical mathematics, as in '$size < 2^{15}$', by typing '`|size < @t$2^{15}$@>|`'. 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.

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

`@!` [X, 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. *Some Spidery* `WEAVE`*s will insert implicit '*`@!`*' tokens in places like function definitions or variable declarations. The person who built your Spidery* `WEB` *should have documented these places; if not, they can be discovered by examing the* `.spider` *file. The automatic definitions work well for some languages, but for the best possible index you should insert your own '*`@!`*' token before the defining instance of every identifier.*

`@?` [X, T] This cancels an implicit (or explicit) '`@!`', so that the next index entry will not be underlined. *This is not implemented in the current version of Spidery* `WEB`.

`@,` [X] 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.

`@/` [X] This control code causes a line break to occur within an "X" program formatted by `WEAVE`; it is ignored by `TANGLE`. Line breaks are chosen automatically by TeX according to a scheme that works 99% 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. *The line following such a line break is indented at the same level as the line broken.* 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.

`@-` [X] *This control code makes* `WEAVE` *continue the current statement on the next line; it is ignored by* `TANGLE`. *If is just like '*`@/`*', except that the following line is indented, to show that it is a continuation of the line broken.*

`@|` [X] 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.

`@#` [X] 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. You might use it, for example, between procedure definitions or between groups of macro definitions that are logically separate but within the same module.

`@+` [X] This control code cancels a line break that might otherwise be inserted by `WEAVE`, e.g., before the word '**else**', if you want to put a short if-then-else construction on a single line. It is ignored by `TANGLE`.

`@;` [X] Usually this control code is treated like a semicolon, for formatting purposes, except that it is invisible. *The person who built your* `WEB` *can redefine it, though, so check his documentation to make sure.*

You can use it, for example, after a module name when the "X" text represented by that module name ends with a semicolon.

The last *seven* control codes (namely '`@,`', '`@/`', '`@-`', '`@|`', '`@#`', '`@+`', and '`@;`') have no effect on the "X" program output by `TANGLE`; they merely help to improve the readability of the TeX-formatted "X" 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 "X"'s syntax. 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.

Because of the rules by which every module is broken into three parts, the control codes '`@d`', '`@f`', and '`@u`' are not allowed to occur once the "*X*" part of a module has begun.

Additional features and caveats.

3. The TeX file output by WEAVE is broken into lines having 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 "*X*" text in `|...|` that expands to more than eighty characters long. Such lines should not have '`%`' signs.

4. "*X*" 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 *prettyprinting* grammar. The grammar and the translation process are explained in the "Spider User's Guide." 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, but the general mechanism is somewhat complex because it must handle much more than "*X*" 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 "*X*" text should appear only after statements or clauses; i.e., *(if "X" were PASCAL)* after semicolons, after reserved words like **then** and **do**, or before reserved words like **end** and **else**. Otherwise WEAVE's parsing method may well get mixed up.

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

5. Comments and module names are not permitted in `|...|` text. After a '`|`' signals the change from TeX text to "*X*" text, the next '`|`' that is not part of a string or control text ends the "*X*" text.

6. *Spidery* WEB *does not handle nested comments.*

7. Reserved words of "*X*" must appear entirely in lowercase letters in the WEB file; otherwise their special nature will not be recognized by WEAVE. *This could be changed, but the author of Spidery* WEB *does not believe in case-insensitivity for programming languages.* You could, for example, have a macro named *END* and it would not be confused with **end**.

However, you may not want to capitalize macro names just to distinguish them from other identifiers. Here is a way to unreserve Pascal's reserved word '**type**' and to substitute another word '**mtype**' in the WEB file.

```
@d type(#) = mem[#].t
@d mtype = t @& y @& p @& e
@f mtype = type
@f type = true
```

In the output of TANGLE, the macro `mtype` now produces '`TYPE`' and the macro `type(x)` now produces '`MEM[X].T`'. In the output of WEAVE, these same inputs produce **mtype** and *type*(x), respectively.

8. The `@f` feature allows you to define one identifier to act like another, and these format definitions are carried out sequentially, as the example above indicates. 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.

9. You may want some `@f` formatting that doesn't correspond to any existing reserved word. In that case, WEAVE could be extended *by someone who built Spidery* WEB to include new "reserved words" in its vocabulary. *(The "Spider User's Guide" explains how.)*

10. Sometimes it is desirable to insert spacing into "*X*" 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. (This control sequence is used, for example, at the beginning of lines that contain labeled statements, so that the label will stick out a little at the left.)

11. `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`, which are allowed only in change files, 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.

This dual-input feature is useful when working with a master `WEB` file that has been received from elsewhere (e.g., `TANGLE.WEB` or `WEAVE.WEB` or `TEX.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.

There are several multiple-changefile tools in the TeX community; check back issues of TUGboat for more information.

Appendices.

Appendix G: How to use `WEB` macros. The macros in `webkernel` make it possible to produce a variety of formats without editing the output of `WEAVE`, and the purpose of this appendix is to explain some of the possibilities.

1. Three fonts have been declared in addition to the standard fonts of `PLAIN` format: You can say '`{\sc stuff}`' to get STUFF in small caps; and you can select the largish fonts `\titlefont` and `\ttitlefont` in the title of your document, where `\ttitlefont` is a typewriter style of type.

2. When you mention an identifier in TeX text, you normally call it '`|identifier|`'. But you can also say '`\\{identifier}`'. The output will look the same in both cases, but the second alternative doesn't put *identifier* into the index, since it bypasses `WEAVE`'s translation from "X" mode.

3. To get typewriter-like type, as when referring to '`WEB`', you can use the '`\.`' macro (e.g., '`\.{WEB}`'). In the argument to this macro you should insert an additional backslash before the symbols listed as 'special string characters' in the index to `WEAVE`, i.e., before backslashes and dollar signs and the like. A '`\␣`' here will result in the visible space symbol; to get an invisible space following a control sequence you can say '`{␣}`'.

4. 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.

5. 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.

6. The `\title` macro will appear at the top of each page in small caps. For example, Appendix D was produced after saying '`\def\title{WEAVE}`'. *(Appendix D was omitted from this document; see Don Knuth's original* `WEB` *technical report for all the appendices.)*

7. The first page usually is number 1; if you want some other starting page, just set `\pageno` to the desired number. For example, the initial limbo section for Appendix D included the command '`\pageno=16`'.

8. 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 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.

9. 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, when Appendix D was being generated, a file containing

```
\Z { Introduction}{1}{16}
\Z { The character set}{11}{19}
```

and similar lines was created. The `\topofcontents` macro could redefine `\Z` so that the information appears in another format.

10. 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.

Warning: life is a little different in Spidery `WEB`*. TeX hackers should go straight to the source.*

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.

11. 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 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 '`@:`' feature, declaring a macro that simply removes a sort key as follows:

```
\def\9#1{}
```

Now you can say, e.g., '`@:TeX}{\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".

12. The control sequence `\modno` is set to the number of the module being typeset.

13. 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.