An Introduction to the `WEB` Style of Literate Programming

by Bart Childs

Literate Programming

One of the greatest needs in computing is the reduction of the cost of maintenance of codes. Maintenance programmers spend at least half of their time trying to understand what code does [PaZv83] and maintenance is accepted to be 60% to 80% of a code's cost. In *The Mythical Man-Month, Essays on Software Engineering,* Brooks stated:

> **Self-Documenting Programs**
>
> A basic principle of data processing teaches the folly of trying to maintain independent files in synchronism. It is far better to combine them into one file with each record containing all the information both files held concerning a given key.
>
> Yet our practice in programming documentation violates our own teaching. We typically attempt to maintain a machine-readable form of a program and an independent set of human-readable documents, consisting of prose and flow charts.
>
> The results in fact confirm our teachings about the folly of separate files. Program documentation is notoriously poor, and its maintenance is worse. Changes made in the program do not promptly, accurately, and invariably appear in the paper.
>
> The solution, I think, is to merge the files, to incorporate the documentation in the source program. This is at once a powerful incentive toward proper maintenance, and an insurance that the documentation will always be handy to the program user. Such programs are called *self-documenting*...

A method of improving the readability of a code and merging the documentation with the high level language source was developed by Donald E. Knuth at Stanford during his second writing of TEX, the typesetting system. Knuth has termed this methodology *literate programming* and can be thought of as "instead of imagining that our task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do[Knuth84]." In other words, a literate program should be thought of as a work of literature rather than a set of instructions for a computer. A literate program should have several characteristics. These include:

- well-written documentation which is integrated with the code,
- well-presented code with indentation and extra vertical white space to emphasize logical divisions of code,
- optional typographical niceties like bold keywords, italic programmer supplied names, logical symbols,
- standard links to enable navigation of code and documentation including:
  - ⋆ table of contents,
  - ⋆ indices of variables, comments, user flags, procedures, messages, ...
  - ⋆ used-in and defined-in pointers for code fragments, macros, and subprograms,
- a source system which is sufficiently extensible to be tailored to the reader's desires or format(s) dictated by the application.

Many of these characteristics should be available in interactive and paper environments for the development and maintenance of literate programs. The interactive environment should also have some other features that are unique to the development process.

WEB

Knuth implemented *literate programming* through the WEB system of structured documentation. Lins [Lin89] sums the concept in this manner:

$$\textit{literate programming} = \textit{structured programming} + \textit{structured documentation}$$

A WEB document contains both source code and documentation. WEAVE produces the TEX file from the WEB source file. TEX can then be used to create the typeset document. To create code for compilation from a WEB source, TANGLE is used. Knuth's version of WEB was developed for Pascal. The remainer of this document will be based upon a FORTRAN WEB. Pages 7 and 8 are the resulting FORTRAN code and pages 9 through 16 are the formatted version of a *literate programming* example.

The combination of WEAVE and TEX produce a document which appears more like a work of literature than the standard source file listing. WEAVE enhances the typeset out in many ways:

1. The code will be like an annotated pseudo-code (both top-down and bottom-up code).
2. Reserved words are shown in **boldface** and variable names, ... are shown in *italics*.
3. WEAVE will automatically line up and indent all statements (*i.e.* **begin/end**, **if-then-else**, loop constructs) regardless of the format of the source.
4. Calls to named modules will contain the module number where it was defined. Named modules will contain the statement 'This code is used in section #' where # is the module number(s) that contain the call(s) to that specific named module.
5. Boolean and relational operators will be replaced with their corresponding mathematical symbols. Boolean operators such as `and` and `or` are replaced by $\wedge$ and $\vee$, relational operators such as <> and >= are replaced by $\neq$ and $\geq$...
6. It contains a table of contents which lists each title of each part and chapter, followed by the actual sections, and ending with an index and a list of sections names.
7. WEAVE automatically numbers modules sequentially in the order they appear in the WEB source file. Numbering the modules makes referencing code quicker and easier.
8. Index entries include variable names, functions, keywords used in documentation, and forced entries. Each entry will be referenced according to the module number in which it appeared. Underlined numbers indicate the module where the variable or function was defined.
9. WEAVE will automatically number the pages and produce a page header which will contain the program title and the chapter title.

WEAVE will output the TEX source file which will have the same name as the WEB source file but, with the extension of `.tex` instead of `.web`.

Literate programming encourages programmers to write in a logical order; however, compilers can not always handle code that is "logical" to humans. To make the code readable by the compiler, TANGLE will replace the module calls with the actual module definitions no matter where they appear in the program. "Since the compilation order of the code no longer dictates how the program is designed and presented the resulting program is much more comprehensible and thus will be more maintainable for the future." [Smith90] TANGLE will output the compiler compatible program source file, which will have the same name as the WEB file with a language dependent extension.

WEB programs are coded in modules (also called sections). A part (major module) begins a new train of thought or logical grouping for a program. Parts in the source code begin with an '`@*`' and cause a page break in the typeset output. Each part title is included in the table of contents. A chapter (an emphasized module) will be a subdivision of a part and will begin with '`@*1`' (or 2, 3, ...), but will not cause a page break. A chapter title will be indented in the table of contents. Modules begin with an '`@␣`' (␣- <space>) and will neither appear in the table of contents nor cause a page break.

All modules must contain at least one of three parts: documentation, definitions, and program code. Ideally, each part of a module should be no more than 25 lines (one screen for those who are not fortunate enough to be using a workstation). Modules should be subdivided until their functionality is easily comprehensible. At times this may violate the previous rule.

Documentation Portion

The documentation portion will be neatly typeset in paragraphs. Although no knowledge of TEX is required; the more TEX a user knows the more control a user will have over the output. Most TEX commands begin with a backslash(\), the remaining commands are comprised of infrequently used non-alphabetic characters. To print these characters the character must be preceded by a '\'.

TEX Command	`\$`	`\#`	`\%`	`\&`
Output	$	#	%	&

One advantage of using TEX, especially for scientific programming, is the ability to typeset mathematical formulas as they would appear in a math textbook (see module 1, page 10). TEX switches from text mode to math mode with the use of a '`$`' before and after the equation. Displayed equations are delimited by a pair of dollar signs (`$$`) before and after the equation.

TEX Command	`$x_k$`	`$x^k$`	`$\pi$`	`$\sum x_k $`	`$\sqrt 5 $`	`$\Delta $`
Output	x_k	x^k	π	$\sum x_k$	$\sqrt{5}$	Δ

For more mathematical symbols, see the TEXbook chapters 16-18.

In the documentation portion of a WEB, a user may want to refer to pieces from the code portion of the current module (*i.e.* variables, function names, reserved words, etc). By placing the desired piece of code inside vertical bars ('`|`'), the user is escaping into code mode. Each element within the '`| |`' will be formatted as though it were in the code portion and will appear in the index in the same format.

Definition Portion

Definitions are abbreviations for code to make the code more comprehensible. This will provide short cuts for the programmer, since the abbreviation can be substituted instead of rewriting the code in full each time it is needed. WEB definitions are seldom needed or used when the high level language is C. It is quite valuable with Pascal codes.

Code Portion

Actual program code will appear in one of two type of modules: unnamed and named. The WEB command to begin an unnamed module will be dependent upon the version of WEB that is being used. the code portion of an unnamed module begins with an ‘`@a`’. The names of ‘named modules’ are TeX strings beginning an ‘`@<`’ and followed by an ‘`@>=`’. To reference the code defined in a named module, put the exact name inside ‘`@<`’ and ‘`@>`’. This can occur in both named and unnamed modules. **Note**: Calls to named modules **cannot** be recursive.

TANGLE will take every reference to a module a substitute the code that corresponds to the module name. In addition, definition names will be replace by the actual code that they represent.

FWEB

FWEB is an extension of Knuth's WEB written by John Krommes at Princeton University. (There was an intervening CWEB written by Silvio Levy.) FWEB is different than most WEBs in that it is designed to handle more than one programming language. Krommes' FWEB incorporates Silvio Levy's CWEB and can also handle FORTRAN and Ratfor (a C-like modification of FORTRAN). The default language for FWEB is FORTRAN (surprised?), but it is easy to change to another language even within the same WEB file. The global language is defined before the first ‘`@*`’ (part): ‘`@n`’ for FORTRAN, ‘`@r`’ for Ratfor, and ‘`@c`’ for C. The global language will be in effect throughout the web file. We mentioned that, in Knuth's WEB, unnamed modules begin with ‘`@p`’, in Krommes' FWEB unnamed modules begin with ‘`@a`’. This tells FWEB that the code portion of an unnamed module is about to begin. The following is an excerpt from NewtonC.web and corresponds to module 7 in the WEAVEd output:

```
125  @* A Newton method program.  This program will use the
126  Newton method to solve the equation $\cos(x)=0$. In \Cee{} we
127  have to |#include| some files to get out standard input-output
128  and also the mathematics functions.  Later we must remember
129  to include the {\tt -lm} switch for loading the library.
130
131  @a
132  #include <stdio.h>
133  #include <math.h>
134  main(){@#
135    @<Declarations@>@#
136    @<Initialize Variables@>@#
137    @<Iterate on the answer@>@#
138    @<Print Results@>@#
139    }
```

The part title in this excerpt is on line 125. ‘`@*`’ and ‘`.␣`’ delimit the title. Titles should be descriptive yet fairly short as they will appear as part of the headings found on each page of that part. The following lines of text make-up the documentation portion. Line 126 illustrates the use of TeX's math mode. In math mode, TeX converts everything to algebraic symbols. Line 126 also has a user-defined TeX command, ‘`\Cee`.’ The formatted result is seen on page 11.

As stated earlier the '`@a`' begins the code portion of an unnamed module. Lines `135–138` contain names of modules whose sources will be included in the C code. Though modules can be defined in any order and any location in the `WEB` file, the C file will contain the expanded code in the same order as the modules are used. As `TANGLE` sees the name of a module it substitutes the name with the code that it represents. C source code (created by `TANGLE`) and typeset output (created by `WEAVE` and TeX) are included. **Notice,** *the* C *is not pretty*! In the original `WEB` Knuth intentionally made the output code "unfit for human consumption." The use of source language debuggers makes this unattractive but the output shown here is corrupted with many pointers to the `WEB` source.

The first few lines of the C file will contain comments inserted by `TANGLE` followed by "legal" C code. `TANGLE` will find the first '`@a`' and output its module number on a comment line (*e.g.* `C*6:*`). The next line is a comment showing the line number and file name of the `WEB` source file (*e.g.* `*line 131 "NewtonC.web"`). It will then copy the code as it appears in the `WEB` file. When it encounters a reference to a named module, it will find that module definition and output its module number on a comment line (`C*7:*`). On the next line it will output the line number of `WEB` source file and follow this with the actual code. When a module expansion is complete, `TANGLE` will output the module number from which it is exiting on a comment line (`C*:7*`). In this case, there is further expansion of the declarations from module 8, then from module 13... The process continues until all code is copied and/or modules expanded.

The final part of the `WEB` source file is the '`@* Index.`' This causes the index to begin on a new page. In addition, 'Index' will appear in the table of contents. To add entries other than those chosen by `WEAVE`, use '`@^name@>`' where name is the entry to add to the index. This is especially helpful for future maintenance issues such as system and compiler dependencies.

For `FWEB` users programming in C, `TANGLE` adds one more option. When a C compiler points out an error, the compiler will give you the line number in the '`.web`' file not the '`.c`' file. Using a debugger will also be easier with this feature. This feature is due to the **#define** option in C.

`FWEB` Processing

The process used to process `WEB`s on an RS-6000 is:

1. Create ASCII text file following the rules of `WEB`. Declare global language before first '`@*`' and have at least one unnamed module. The following apply for the attached program.
2. Create the listing by the following steps.
   a. "`weave NewtonC`" (NewtonC.web)
   b. "`tex NewtonC`" (NewtonC.tex)
   c. "`xdvi NewtonC`" (to preview the `NewtonC` document)
   d. "`dvips filname`" (to create a postscript file)
   e. "`lpr -Plw<room#> NewtonC.ps`" (to print typeset output on a postscript device)
3. Create and use the executable by the following steps.
   a. "`tangle NewtonC`" (NewtonC.web)
   b. "`xlc -o Newt NewtonC.c`" (compile and link)
   c. "`Newt`" (to execute file)

REFERENCES

[Lin89] C.A. Lins, "A First Look at Literate Programming," *Structured Programming*, vol. 10, no. 1, pp. 60-62.

[Knuth84] D.E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, May 1984.

[PaZv83] G. Parikh and N. Zvegintov, *Tutorial on Software Maintenance.* Silver Spring, MD: Computer Society Press, 1983.

[Smith90] L. Smith, "Measuring Complexity and Stability of WEB Programs," Masters Thesis, Oklahoma State University, 1990.

Extensive sources are available in the directory `/usr/local/lib/web` and it subdirectories.

The `web-mode` sources and documentation are also available for anonymous ftp from `ftp.cs.tamu.edu` **not operational as of Oct. 20, 1992**

Listing of An Edited `NewtonC.c`, Not a Pretty File

```
#if(0)
  FTANGLE v1.23, ANSI/UNIX on "Thursday, April 2, 1992 at 16:57."
  COMMAND LINE: "ftangle NewtonC"
  RUN TIME: "Thursday, December 3, 1992 at 12:53."
  WEB FILE:    "NewtonC.web"
  CHANGE FILE: (none)
#endif
/* 6: */
#line 131 "NewtonC.web"
#include <stdio.h>
#include <math.h>
main(){
/* 7: */
#line 148 "NewtonC.web"
double x_0,x,delta_x;
/* :7 *//* 8: */
#line 154 "NewtonC.web"
int k,limit;
/* :8 *//* 13: */
#line 218 "NewtonC.web"
double delta_x_max,delta_x_previous;
/* :13 */
#line 135 "NewtonC.web"
/* 9: */
#line 169 "NewtonC.web"
x_0= 1.2L;
delta_x= 0.000001L;
limit= 10;/* more than safe */
/* :9 *//* 14: */
#line 223 "NewtonC.web"
delta_x_previous= 0.0L;delta_x_max= 0.5L;
/* :14 */
#line 136 "NewtonC.web"
/* 10: */
#line 179 "NewtonC.web"
for(k= 1;delta_x>0.0L&&k<=limit;k++){
/* 11: */
#line 197 "NewtonC.web"
delta_x= -(cos(x_0))/(-sin(x_0));
printf("The Newton step is %g\n",delta_x);
/* :11 */
#line 181 "NewtonC.web"
/* 15: */
#line 230 "NewtonC.web"
if(delta_x>delta_x_max)delta_x= delta_x_max;
else if(delta_x<(-delta_x_max))delta_x= -delta_x_max;
/* :15 *//* 16: */
#line 244 "NewtonC.web"
```

```
if((delta_x*delta_x_previous)<0.0L)printf("Oscillating %d\n",k);
/* :16 */
#line 182 "NewtonC.web"
/* 12: */
#line 204 "NewtonC.web"
printf("The step is %g\n",delta_x);
x= delta_x+x_0;
x_0= x;
delta_x_previous= delta_x;
/* :12 */
#line 183 "NewtonC.web"
}
/* :10 */
#line 137 "NewtonC.web"
/* 17: */
#line 251 "NewtonC.web"
printf("The solution to cos(x)=0 is %g\n",x);
/* :17 */
#line 138 "NewtonC.web"
}
/* :6 */
```