Literate Programming, A Practioner's View

Bart Childs
Texas A&M University
Department of Computer Science
College Station, TX 77843-3112
Phone (409) 845-5470; FAX (409) 847-8578
Internet: bart@cs.tamu.edu

Abstract

I have been using the WEB style of Literate Programming since my first efforts to port TEX to the Data General AOS system. When I looked back at those efforts, the work in porting drivers that were not written in WEB and the writing of drivers in WEB (based upon DVITYPE, of course), the value of this method of programming became evident.

I have concentrated my research (and some teaching) efforts upon this style of programming. I will relate my insights and opinions of the following: some quantitative and qualitative measures of the value of WEB programming; a description of some tools that are part of an environment for writing and maintaining literate programs; literate programming environments that are alternatives to the WEB style; an annotated list of some literate programming systems; and I will conclude with my perception of the future of literate programming.

Introduction

Donald Knuth created the WEB system of literate programming when he wrote the TEX typesetting system a second time (see "The WEB system of structured documentation", 1983; "Literate programming", 1984; *TEX: The Program*, 1986; and METAFONT*: The Program*). The WEB system can be described as the merging of documentation, code, and presenting the listings in a typeset format with aids of table of contents, cross-referencing, and indices.

I have used Knuth's original WEB system and several descendants for a number of years. It is my opinion that the training necessary to learn how to program in a literate style is relatively small. I believe the benefits of literate programming make it worthwhile. The benefits are better and more maintainable code (it can be argued that this is not proven.)

In this paper I will also report on available literate programming systems, some of my successes and failures as a literate programmer, creation of some tools to aid the literate programmer, the community of literate programmers that I have been able to identify, alternatives to the WEB system, and suggested directions for use and research in literate programming.

A Definition

I use the following list of requirements to imply a definition of a literate program and the minimum set of tools which are needed to prepare, use, and study the resulting code.

- The high-level language code and the system documentation of the program come from the same set of source files.
- The documentation and high-level language code are complementary and should address the same elements of the algorithms being written.
- The literate program should have logical subdivisions. Knuth called these *modules* or *sections*.
- The system should be presented in an order based upon logical considerations rather than syntactic constraints.
- The documentation should include an examination of alternative solutions and should suggest future maintenance problems and extensions.
- The documentation should include a description of the problem and its solution. This should include all aids such as mathematics and graphics that enhance communication of the problem statement and the understanding of its challenge.

- Cross references, indices, and different fonts for text, high-level language keywords, variable names, and literals should be reasonably automatic and obvious in the source and the documentation.

These requirements have been adapted from (Knuth, 1992), and (VanWyk, 1989 and 1990). My adaptations of the list were affected by my experience as a `WEB` user — first in a maintenance mode, then as an author, and finally using `WEB` in undergraduate and graduate education environments. The last has involved the creation of some tools to enhance the use of literate programming in all environments.

Knuth posed this thought to introduce literate programming: "Instead of imagining that our main 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" (Knuth, 1984). His thesis was that it should be just as important to communicate with the other persons who read the program as it is to communicate with the computer which executes it (Knuth, 1992). David Ness said "it is the **most** important task."

Knuth's WEB System

When Don Knuth wrote TEX the second time, he gave thought to making it portable to many different systems. `WEB` was created as a superset of TEX and Pascal.

`WEB`'s design encourages writing programs in small chunks which Knuth called modules (he also used the term sections). Modules have three parts: *documentation, definitions,* and *code*. At least one of these three parts must be *non-null*.

The *documentation* portion is often a verbal description of the algorithm. It may be any textual information that aids the understanding of the problem. It is not uncommon for a `WEB` to have a number of 'documentation only' modules. These usually describe the problem independent of the chosen language for implementation. For example, a `WEB` for a subprogram that solves the linear equation, $Ax = b$, could have discussion of singularity, condition numbers, partial pivoting, the banded nature of the expected coefficient matrices, etc. It should be an unusual but not exceptional case when a module contains no documentation.

The definition part of a module was often used by Knuth to offset shortcomings of Pascal when used in *systems programming*. (Wirth created Pascal to be a language for pedagogy, not systems programming.) This is rarely the language of choice by today's systems programmers but sometimes it is a convenient way to represent certain ideas. (Knuth used Pascal because "it was everybody's second best language" at the time (Knuth, *TUGboat*, 7(2), 1986). It was before C was widely available.)

Some of My Successes and Failures

My introduction to the `WEB` system of literate programming was rather abrupt — I was porting TEX82. I had not written a Pascal program or a TEX document at the time. I had written a number of systems programs in PL/1.

Version 0.6 of the TEX system was made available to me in the fall semester of 1982. Distribution tapes contained about 300 files. Most of these had to do with the fonts in a binary form. Binary files were written in a format of twenty bytes per record. Each byte was converted to four ASCII characters; for example, '`␣255`' for the byte with all bits being one. Over a two or three week period I ported the `TANGLE` processor and began work on the `WEAVE` processor. I also had the 'report version' of volume B of the *C & T* series (Knuth, 1986).

The end of the semester caused me to stop the work and I did not resume until the spring with version 0.9 of TEX. I spent some time reviewing the changes in the *necessary* parts of the TEX system: `TANGLE`, `WEAVE`, and TEX.

The `WEB` sources were usually complemented with the change files for TOPS, VMS, and UNIX systems. These were valuable, but I recall the feeling that the "system dependencies" entries in the index were even more valuable. It helped to see what was changed for other systems, but often those changes were in terms of "system calls" whose documentation I did not have.

The *necessary* codes are approximately 2,700, 7,000, and 25,000 lines of Pascal each of the `INITeX` and TEX processors. I was able to port approximately 60,000 lines of Pascal code to a system on which only 2,700 lines had previously been ported. I did this in **three days** (probably 15 hours of work). I regret that I did not keep a diary or log errors like Don Knuth did, (Knuth, *TUGboat*, 7(2), 1986; and Knuth, chapters 10 and 11, 1992).

The experience of porting the system convinced me that the literate programming style had significant merit. Most of the programs I subsequently wrote for the TEX system were `WEB`s. The existence of `dvitype` as a model for drivers helped this decision. I created a family of drivers for the AOS system for QMS, HP, and Canon printers. These used a common source and adaptations for the different printers were accomplished by change files.

This further experience led to the logical conclusion that an "environment" could help significantly. We created several environments to automate the steps of creation of code and documentation for the AOS system. The emergence of UNIX and availability of workstations has relegated these early works to simply being a pleasant memory that in some sense could be called a failure.

Marcus Brown built part of an "Interactive Environment for Literate Programming" as part of his dissertation under my direction (Brown, 1988 and 1990). This was essentially the output side of an editor which also allowed the navigation of the source with views of the typeset output, a graphical tree structure of the module interconnections, and other functionalities. It did not include a real editor. The environment was tested by giving senior computer science students the tasks of identifying the changes to make a "big TEX" and changing **tangle** to make code that is more readable. (Knuth's original WEB created code that was to be "unfit for human consumption".) These students had been using WEB in processing system performance logs.

The positive results of this work were that the subjects performed well in identifying the necessary changes using the typeset listings of the codes and in the on-line form using the environment.

The environment was dependent upon SUN graphics and did not lend itself to incorporation with public domain editors. A later environment was based upon GNU Emacs.

Other WEB Systems

There have been a number of WEB and WEB-like systems developed. They can be divided into several categories.

- WEB systems that have the same set of tools that are adapted to a different high-level language. The high-level languages supported include: C (Guntermann and Schrod, 1986 and Levy, 1987), **FORTRAN** (Krommes, 1989), Modula-2 (Sewell, 1987), LISP dialects, and Reduce. In the references I also indicate the sites where the 'definitive' sources are available.
  There are some differences in functionality in many of these such as support of multiple change files (Guntermann) and several high level languages (Krommes).
- WEB systems that have been adapted to a different high-level language by pre-processors and/or post-processors. These include support for **FORTRAN**, and the first WEB for Reduce.
- WEB systems that have a different basis for their creation but generally follow the same WEB concepts. Ramsey's Spider enables reasonably easy creation of WEBs for several different languages (Ramsey, 1989). His example include WEBs for Ada, C, `awk`, and SPL. Gragert and Roelofs created the second WEB for Reduce with Spider (Gragert, 1991). They are now creating a WEB for Maple (Gragert, 1992).
- WEB systems using a different language or formatting system. Three will be mentioned. Thimbelby did his CWEB with the UNIX standards of C and *troff* (Thimbelby, 1986). The limitations of *troff* caused problems. I have had personal communication about another literate programming system that used a Macintosh WYSIWYG editor and the C language. All the documentation was done in German. David Ness created a CWEB-like system at *TV Guide*, which was not published or distributed.
- A NOWEB system that relaxed significant WEB requirements. Ramsey characterized his NOWEB as a "low-tech" literate programming system (Ramsey, 1991). It does not *indent* the source but passes it through.
- Jim Fox created c-~~web~~, which gives a nice listing of the source and some organization of the code. It can be argued that this is not a literate programming system because its indexing is minimal and the order of the code is dictated by C syntax. The c-~~web~~ package assumes that the C comments are written in TEX. The only other assumptions are that two C comments of a specific format appear near the start and end of the program, otherwise it is ordinary C with pretty output. The user simply TEX's the C source.

What is a Good WEB?

This is a question which still needs to be answered. It probably can't be answered today because there is not a large enough body of programs written in a literate style that are available for study. Later I will show one graph that appears to be a good indicator of characteristics of WEBs. I think several graphs of this type could be an effective indicator of quality.

The TEX system and the work reported in (Ramsey and Marceau, 1991) are the most significant examples from which we can begin to search for answers. Ramsey's work is based on a project in which there was not a single programmer (as in Knuth's work). More studies like this are needed.

Many of the characteristics I think of in using for evaluative purposes relate to the module. A module should be a 'sensible chunk.' In most cases, I interpret this to mean most modules should not be more than one screen of source, in spite of varying screen sizes of up to 40 lines. This is not an absolute rule, just a guideline. Some modules will be documentation only and may be several pages in length. In some cases, it is convenient to make each paragraph a module of its own. Further, there are modules that seem to be best presented when the documentation and code are **each** about a screen.

I can't yet answer the question as to what are the characteristics of a good web, but I think that studying with figures like Figure 1 and some statistics will begin to give us some 'normal characteristics.' These kinds of graphics make it easy to spot inconsistencies in documentation, modularity, and other causes of maintenance problems.

I do not doubt that there will be more than just one description of a good literate program. However, I think that most will have consistent "patterns" when analyzed in similar graphical manners using the "right statistics."

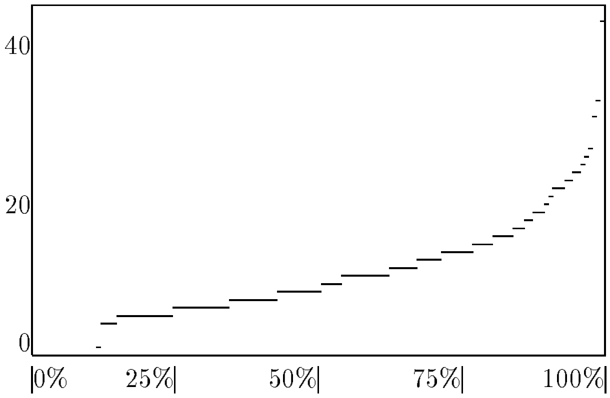

Figure 1. Percent modules *vs.* number of lines of code per module in `PS_Quasi.web`.

Mamoun Babiker wrote the tool to extract the data and create the TEX code for drawing the figure. Figures like this are better when they are done with a graphics package rather than relying on TEX's rules, for portability.

Some of the elements of a good essay will apply. Programming practices must change significantly for the spelling requirement to apply. Many codes and their documentation are full of acronyms and mnemonics. It can be argued that with today's systems and compilers accepting long variable names, there should be some change in this.

Tools for the Literate Programmer

I admit to believing in the value of literate programming even when I only had the usual editors and hardcopy (TEXed) output with the indices and cross-references. This was obviously **not** the preferred literate programming environment.

The editor is the most important "tool". It will be discussed briefly and most of the `web-mode` commands will be listed in a table. Other tools will be discussed without further introduction.

Editing environment. Mark Motl created the `web-mode` for GNU Emacs (Motl, 1990). It makes the editor sensitive to the rules of WEB, TEX, and some aspects of the high-level language. The table of contents, index, and index of modules that are helpful in studying WEBs are on-line. The user can select a variable while viewing the index and then with two keystrokes proceed to view each of the modules where the variable is referenced, in succession. All lists (module names, extra index entries like 'system dependencies', etc.) are kept on-line and the user only has to select from the list rather than retype them.

Details of `web-mode` are beyond the scope of this paper. A list of the functions that have been added to GNU Emacs to create a `web-mode` is shown in the appendix. We know it is used by a number of literate programmers on several types of UNIX systems and on MS-DOS machines and it certainly seems to be effective.

Counters. These are used for calculating the basic statistics and metrics that can be used to "measure" a web. We are using variants of these to build statistical summaries of WEBs, which we hope will lead to 'proven' statements describing the characteristics of good WEBs. The earlier figure was prepared by using one of these.

Table 1 was created by `wst` (a WEB statistics tool written by Mark Gaither) and is a count of the WEB commands in `tex.web`, while Table 2 is a count of the TEX commands. This data was extracted from version 3.14 of TEX which has 24,863 lines in its source and 523 of those lines have at least one WEB command. The complete Table 2 would be too long for this paper. However, the header is informative. I have left in only the eight most-used TEX control sequences. On the other hand, 69 TEX control sequences were used once and another 38 were used twice.

Table 1. WEB control sequences in `tex.web`.

Unique control sequences =	27
Occurrences of control sequences =	11609
Words in file =	127592
Unique control sequences/words =	0.0002
Control sequences/words =	0.0910

Control Seq.	Count
@	1325
@!	1655
@"	3
@#	28
@$	3
@&	5
@'	260
@*	55
@+	227
@,	8
@.	259
@/	685
@:	407
@;	90
@<	1774
@>	2868
@?	29
@@	16
@\	2
@^	259
@d	1216
@f	12
@p	178
@t	169
@{	5
@|	66
@}	5

Table 2. TeX control sequences in `tex.web`.

Unique control sequences =	202
Occurrences of control sequences =	6186
Words in file =	127592
Unique control sequences/words =	0.0016
Control sequences/words =	0.0485

Control Seq.	Count
\␣	381
\.	1728
\PASCAL	86
\TeX	477
\\	1431
\cr	125
\hang	166
\yskip	142

A recent scientific code of mine has 3,900 lines of WEB, 0.0072 unique control sequences per word, and 0.0431 control sequences per word — I think that this was expected. I consider this data to show an upper bound of the amount of TeX that might be included in a WEB. This is evidence that it is **not** necessary to be a TeXpert to use WEB.

Change file analyzer. Change files are analyzed in detail by this code (written by Mark Gaither). The UNIX `diff` utility is used to show exactly what changed in the change file for each module. Each module is flagged if it is only part of the lines of the module. The reason for this is that the GNU Emacs `web-mode` assumes that change files always contain complete modules. Also, since a module should be the minimum part of a code that can stand alone, it seems wrong to have only a part of it. If code is changed, documentation should also be changed.

A WEB structure viewer. `Web-view` (written by Kevin Borden) gives a high-level view of the structure of a WEB in a manner much like the browsing of the directory structure in a NeXT computer. This design was selected because the previous graphics representation of a WEB did not have sufficient screen to show enough of a module name to always be indicative of the purpose of the module. This tool is based upon X.

TeXbraces. This is a simple WEB (written by David Ness) that is used to flag those problem places where the dread braces get out of balance. I use this in the obvious fashion on both TeX and WEB sources. This functionality is present in `web-mode` and `tex-mode`.

Integration of RCS. Configuration management tools are a significant part of the toolbox on most code developers. RCS is a reasonably popular system that aids in keeping up with the changes from one version to the next. William Needels has recently finished a project on the integration of RCS and *imake* to automatically produce `Makefile`s for the generation of executables and documentation for systems created with WEBs. It shall be released shortly after packaging and another round of proofing.

The sources of all the tools that we support are available for anonymous `ftp` from: `ftp.cs.tamu.edu`. We are attempting to support all except the original environment of Marcus Brown. Another source for literate programming tools is `ftp.th-darmstadt.de`, the `pub/tex/src/webware` directory.

Community of Literate Programmers

Van Wyk indicated that only the creators of literate programming systems use them (Van Wyk, 1990). There is a significant international 'community of literate programmers.' I think the number of users is surprising since, to my knowledge, there has not been a conference or major portion of a conference dedicated to the exposition of the use of literate programming or the many unanswered questions on literate programming and environments.

It is difficult to accurately estimate the number of programmers using WEBs. I think the number is probably on the order of 1,000 or more because:

- Each semester several students contact me to show me some of their work or ask for help. These are not "my" students. They 'discover' WEB by scanning TEX directories.
- The distribution list for announcements about John Krommes' FWEB has several dozen addresses. I estimate that it does not include half the sites that really use it and most addresses probably represent several to many users.
- I know of three sites in Australia that are extensively using FWEB and which are not on the distribution list.
- A large multinational company is using CWEB as their methodology for code development.
- I have been politely chastised for not placing `web-mode` in the GNU or Archie archives. (It will be there soon.)
- There are at least two journals and four universities I know of listing literate programming as a viable/current research area. The subject literature contains about 100 papers, reports, and monographs from about 60 authors.
- I know of several professionals (such as David Ness) who are active literate programmers.
- Apparently a significant portion of the DANTE contributions to the TEX systems is being done in WEBs. I have seen the sources of BM2FONT and I know that Joachim Schrod and associates continue to use their CWEB.
- A discussion list for literate programming `LitProg@SHSU.edu` was started in July 1992 and generated many subscribers and more than 300k-bytes of messages in one month.

I intend to formalize the above list and seek permission to distribute the names and addresses of those I know. I believe that literate programming will be aided greatly by the publication of Don Knuth's new book, *Literate Programming* (Knuth, 1992).

Conclusions and Recommendations

I have no reservations in recommending the use of Knuth's original WEB, Levy's CWEB, or Krommes' FWEB. Our primary WEB system is Krommes' FWEB because it supports **Fortran**, **Fortran**90, **Ratfor**, C, and C++. We also find it to be the best-documented of the WEBs. It was created by extending Levy's CWEB.

I believe that the use of `web-mode` has made literate programming easier to introduce to my students. A similar tool (but not as complete) should soon be available for VMS systems based upon LSEDIT.

Ramsey's Spider system is a good tool for creating new WEBs and his NOWEB is certainly worthy of study.

Future Work

The questions related to determining the quality of a literate program are still not answered. I believe that we are beginning to have tools and examples that will help answer these questions or at least give us some general ideas. We need a significant body of WEBs written by teams and a range of programmers for study.

I believe that we need literate programming systems whose output can be tailored to personal tastes. For example, a Pascal code could have `<=` instead of $\leq$, braces instead of **begin-end**, and physical things like page size, etc.

Generic WEBs should be available for other styles of languages such as VAX DCL files, Unix scripts, hand-held calculator code, etc. Norman Ramsey's `noweb` may be a good vehicle for this since it does not reformat the code. One of the most underused features of FWEB is that it allows the creation of literate style files for TEX macro writing. This is an obvious improvement over the `.doc` to `.sty` contribution in Leslie Lamport's original LATEX.

Much of today's computing is no longer considered to be a single-language environment. Scientific computing is often a mix of high-level languages. FWEB allows the mixture of **Fortran**, C, etc. The inclusion of MATLAB, GNUPLOT, etc., should be included in a literate form.

A comment

In the interest of brevity I have omitted many references that should be a part of the complete paper. I point to Nelson Beebe's bibliography on literate programming which is available by anonymous `ftp` from `science.utah.edu` for a complete list of many relevant citations.

Acknowledgements

TUG's anonymous reviewer made **many** comments that contributed to this paper. The thoroughness of that review is appreciated.

David Ness and I have had many conversations about literate programming and his vision has certainly helped my views and understanding about literate programming and several other aspects of the computing professions.

References

Brown, Marcus E. "An Interactive Environment for Literate Programming". Ph.D. thesis, Texas A&M University, College Station, TX, August 1988.

Brown, Marcus E. and Bart Childs. "An interactive environment for literate programming". *Journal of Structured Programming*, 11(1), pages 11–25, 1990.

Fox, Jim. "Webless literate programming". *TUGboat*, 11(4), pages 511–513, November 1990. `u.washington.edu`

Gragert, Peter, and Marcel Roelofs. *Reduce* `WEB` *version 3.4.* `utmfu0.math.utwente.nl`

Gragert, Peter. Personal communication.

Guntermann, Klaus, and Joachim Schrod. "`WEB` adapted to C". *TUGboat*, 7(3), pages 134–137, October 1986. *This* `WEB` *is no longer supported. They recommend the Levy/Knuth* `CWEB`. `schrod@iti.informatik.th-darmstadt.de`

Knuth, Donald E. "The `WEB` system of structured documentation". Stanford Computer Science Report CS980, Stanford University, Stanford, CA, September, 1983. `labrea.stanford.edu`

Knuth, Donald E. "Literate programming". *The Computer Journal*, 27(2), pages 97–111, May 1984. Also appears as chapter 4 in "*Literate Programming*".

Knuth, Donald E. *TEX: The Program*, volume B of *Computers & Typesetting*. Reading, MA: Addison-Wesley, 1986. ISBN 0-201-13437-3.

Knuth, Donald E. METAFONT: *The Program*, volume D of *Computers & Typesetting*. Reading, MA: Addison-Wesley, 1986. ISBN 0-201-13438-1.

Knuth, Donald E. "Remarks to celebrate the publication of *Computers & Typesetting*" *TUGboat*, 7(2), pages 95–98, June 1986.

Knuth, Donald E. *Literate Programming*. Center for the Study of Language and Information, Stanford University, CA: Distributed by Univ. of Chicago Press, ISBN 0-937073-80-6, 1992.

Krommes, John A. *The* `FWEB` *System*. Princeton University. 1989. `lyman.pppl.gov`

Levy, Silvio. "`WEB` adapted to C, another approach". *TUGboat*, 8(1), pages 12–13, April 1987. `princeton.edu` and `labrea.stanford.edu`

Motl, Mark B. "A Literate Programming Environment Based on an Extensible Editor". Ph.D. thesis, Texas A&M University, College Station, TX, December, 1990. `csseq.cs.tamu.edu`

Ramsey, Norman. "Weaving a language-independent `WEB`". *Communications of the ACM*, 32(9), pages 1051–1055, September 1989. `princeton.edu`

Ramsey, Norman , and Carla Marceau. "Literate programming on a team project". *Software — Practice & Experience*, 21(7), pages 677–683, July 1991.

Ramsey, Norman. "Literate programming tools need not be complex". Submitted for publication, 1991.

Sewell, E. Wayne. "How to `MANGLE` your software: the `WEB` system for Modula-2". *TUGboat*, 8(2), pages 118–122, July 1987.

Sewell, E. Wayne. *Weaving a Program: Literate Programming in* `WEB`. New York, NY: Van Nostrand Reinhold, 1989. ISBN 0-442-31946-0.

Thimbleby, Harold. "Experiences of 'literate programming' using `cweb` (a variant of Knuth's `WEB`)". *The Computer Journal*, 29(3), pages 201–211, June 1986.

Van Wyk, Christopher J. "Literate Programming: Moderator's Introduction". *Communications of the ACM*, 32(6), page 740, June 1989.

Van Wyk, Christopher J. "Literate programming— an assessment". *Communications of the ACM*, 33(3), pages 361 and 365, March 1990.

Appendix

Listing of `WEB-MODE` Commands by Functionality

Functionality	Command	Key Binding
Movement Among Buffers (Files)	`web-goto-buffer-by-name`	`C-c b n`
	`web-goto-buffer-change-file`	`C-c b c`
	`web-goto-buffer-include-file`	`C-c b i`
	`web-goto-buffer-web-file`	`C-c b w`
Movement Among Modules	`web-goto-module`	`C-c g m`
	`web-next-module`	`C-c n m`
	`web-previous-module`	`C-c p m`
Interactive Access to and Movement Among Sections	`web-goto-section`	`C-c g s`
	`web-next-section`	`C-c n s`
	`web-previous-section`	`C-c p s`
	`web-view-section-names-list`	`C-c v s`
Interactive Access to Index	`web-next-index`	`C-c n i`
	`web-previous-index`	`C-c p i`
	`web-view-index`	`C-c v i`
Interactive Access to Modules	`web-next-define`	`C-c n d`
	`web-next-use`	`C-c n u`
	`web-previous-define`	`C-c p d`
	`web-previous-use`	`C-c p u`
	`web-view-module-names-list`	`C-c v m`
Change File Editing and Movement	`web-edit-module`	`C-c e m`
	`web-goto-change-corresponding-to-module`	`C-c g c`
	`web-next-change`	`C-c n c`
	`web-previous-change`	`C-c p c`
`Web` Structure Information	`web-delimiter-match-check`	`C-c d m`
	`web-determine-characteristics`	`C-c d c`
	`web-view-changed-modules-list`	`C-c v c`
	`web-what-change`	`C-c w c`
	`web-what-module`	`C-c w m`
	`web-what-section`	`C-c w s`
Miscellaneous	`web-insert-index-entry`	`C-c i i`
	`web-mode-save-buffers-kill-emacs`	`C-x C-c`
	`web-rename-module`	`C-c r m`