

# Literate Programming

Andreas Klein

March 11, 2009

## Contents

<b>1</b>	<b>Introduction to Literate Programming</b>	<b>1</b>
<b>2</b>	<b>Pweb Desgin goals</b>	<b>2</b>
<b>3</b>	<b>Pweb Manual</b>	<b>2</b>
3.1	Structure of a WEB-Document . . . . .	2
3.2	Text sections . . . . .	3
3.3	Code sections and Modules . . . . .	3
3.4	Marcos . . . . .	4
3.5	Special variable names . . . . .	5
3.6	Include files . . . . .	5
3.7	Conditional Compilation . . . . .	6
3.8	Compatibility Features . . . . .	6
3.9	Editing Pweb documents . . . . .	6
3.10	Extending Pweb . . . . .	7

## 1 Introduction to Literate Programming

Most People working in mathematics know D. E. Knuth name]Knuth, Donald Ervin  $\text{\TeX}$  typesetting system. Less known is that in companion with  $\text{\TeX}$  Knuth developed a new programming style called literate programming (see [2]). The idea is that a program should read like an article.

A literate programming environment also called web-system consist of two programs called `tangle` and `weave`.

The `tangle` program extracts the source code of the web-file and generates an input file for the compiler (interpreter) of our programming language. It must resort the code from the human readable form into the machine readable form.

The `weave` program generates the input for a text processor. It must apply syntax highlighting to our code and generate an index, etc.

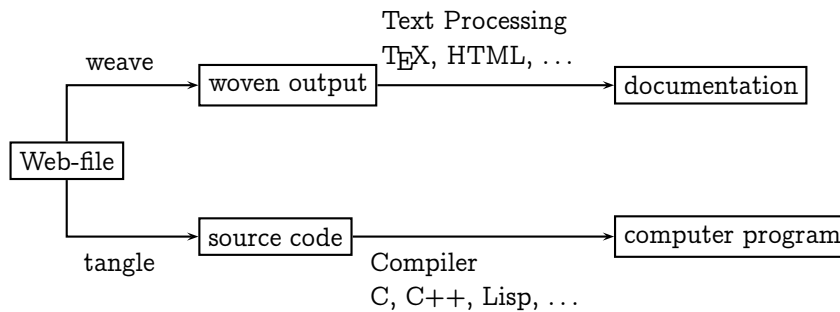


Figure 1: The literate programming environment

In principle you can use any text processing tool and any computer language as basis for a literate programming environment. In practice the weave and tangle program need some information about the destination language, so a specific literate programming environment support only few languages.

## 2 Pweb Desgin goals

When I started this project, I found that no literate programming satisfies exactly my needs. So I began with the development of a new literate programming tool. I called pweb, since its main part is a bunch of Perl scripts.

I designed pweb with the following goals in mind:

- It should be easy to add support for additional programming language. To guarantee this pweb should only use regular expressions and not a complete parser. I was willing to scarify advanced formating functions, like auto-detection of variables, for this goal.
- Not every language have such a good preprocessor like LISP. Therefore pweb should provide its own preprocessor. I decided to use the macro processor M4 for that purpose.
- It should be possible to have (in documentation) variable names like  $\alpha$ ,  $x'$ ,  $\hat{x}$ .

## 3 Pweb Manual

### 3.1 Structure of a WEB-Document

All web commands start with a @ symbol. The first line of a pweb-document must be of the form @<programing language>2<formating language>. For example a C++ Program that is documented in L<sup>A</sup>T<sub>E</sub>X starts with @cpp2latex.

The next lines up to the first `@*` command form the limbo section. The contents of the limbo section are copied directly into the header of the documentation (the part between `\documentclass{article}` and `\begin{document}` if you use  $\LaTeX$  or the part between `<head>` and `</head>` if you use HTML).

If you use latex as formatting language the limbo section must define the title and the author.

The remaining part of the web-document is split in text sections, code sections, named modules and macros. A text section must precede each other type.

## 3.2 Text sections

First line of a text section has the form

```
@*n <title of the section>.
```

where  $n \in \mathbb{N}$ . This defines a section of depth  $n$  with the given title. `pweave` transform it to the  $\LaTeX$  commands `\section`, `\subsection` and so on or the HTML tags `<H1>...</H1>`. It depends on the chosen formatting language how many substructures exist.

`@*` is a short form for the highest level `@*0`.

Inside a text section you can use all commands from the chosen formatting language (like `\emph{...}` in  $\LaTeX$  or `<em>...</em>` in HTML). Of course once you start using these commands you cannot change the formatting language.

## 3.3 Code sections and Modules

A code section starts either with `@a` or `@c` and ends at the next text section. Both commands are equivalent and the only reason for having two commands is the smooth transition from another literate programming environment (`cweb` uses `@c` and `fweb` uses `@a`). I recommend to use only `@c` in new `pweb` projects.

The effect of a code section is that `ptangle` copies that code directly in to the source code of the program.

In addition to code sections `fweb` knows modules. A module starts with the line `@<Module name@>=` and ends at the next text section.

You can insert a module in any code section or any other module by `@<Module name@>`. A typical use of a module is initialisation code like:

```
@* Initialise variables.
Explain what has to be done
@<init@>=
    set some variables ...
@* The main Program.
Explain the program.
```

```

@c
int main() {
    @<init@>

    Do something ...
}

```

The presence of modules changes the programming style. It is no longer necessary to use functions for structuring the program. The only define is you mean a function in all other case a module does a better job.

In addition you can split the code in smaller parts than it is normally possible in a programming language. D. E. Knuth [1] recommends that a single section contain no more than 12 lines of code. If you every find your self writing lager code section you should consider define a new module to structure the code.

### 3.4 Marcos

Macros are useful technique for doing computation at compile time, but the support of macros is very different in the programming languages. For example Common Lisp has a perfect macro support, where macros can be written in almost the same syntax as the rest of the programming language. C has limited macro support (no recursion in macros) and the macro language differs completely from the programming language. Java has no macros at all.

pweb offers an interface to the M4 macro processor, that can used in addition to the macros provided by the chosen programming language.

A simple macro has the form

```
@define MACRO-NAME SUBSTITUTION
```

and the effect is that in the, whole program MACRO-NAME is replaced by SUBSTITUTION. This corresponds to the C-Marco

```
#define MACRO-NAME SUBSTITUTION
```

More complex macros must be define in one section.

The macro section must start with @m and ends at the begin of the next text section. Inside a macro section you can use the whole syntax of the M4 macro language. Especially you have conditions and recursion. Here is a small example of how to define a macro for the factorial function.

```

@m
define('FACTORIAL', 'ifelse($1,1,1,
    eval($1*'FACTORIAL'(eval($1-1)'))')')

```

A complete description of M4 can be found in [3].

A side effect of using macros is that a single line of the fweb document expands to multiple lines in the programming language. This tweaks up the line numbering and makes it difficult to interpret a later error message which refers to the wrong line number. To help you fweb contains a special command `@l` which resets the line numbering.

You should long macros always like this:

```
@c
some commands
A_BIG_LONG_MACRO @l
some more commands
```

The `@l` command only works with programming languages that support redirection of error messages like C and C++ do with `#line` macro.

### 3.5 Special variable names

Most programming languages require that a variable name must start with a letter and allow only letters, digits and the underscore sign (`_`) afterwards. But in mathematics we often use variables like  $\alpha$ ,  $x'$  or  $\hat{x}$ . Fweb pay attention to this by providing special commands for such variables.

With the command `@alpha` you define a variable (or to be more precise a code fragment) named `alpha` in the source code and printed  $\alpha$  in the documentation. All Greek letters are supported by this facility.

With `x@'` you define a variable  $x'$ . It becomes `x_prime` or `xprime` in the programming language. Similarly you can define variables  $\hat{x}$ ,  $\tilde{x}$  by `x@hat` and `x@tilde`, respectively.

Combinations like `@beta@'` for  $\beta'$  are possible.

### 3.6 Include files

You can use `@include <filename>` to include other pweb documents into your document. The include mechanism is similar to the one used by C and C++.

- If the include-file has a preamble it will be ignored.
- Each include-file must start with a text-section and it can contain only complete sections.
- The `@include` command must stand alone in a line.

### 3.7 Conditional Compilation

In addition to the conditional compilation features that may be provided by the underlying programming languages pweb provides an extra mechanism.

Code between `@tangle off` and `@tangle on` are ignored by `ptangle`. The commands `@tangle off` and `@tangle on` must stand alone in a line and are *not* nestable. Typically one use them to give a short example code in the documentation.

```
@* How to use the new functions.
Some explanation ...
@tangle off
@c
some lines of example code ...
@tangle on
```

Similar `pweave` ignores the code between `@weave off` and `@weave on`. One can use these commands to hide some definitions in the documentation. Since the purpose of literate programming is to provide a full documentation, one should use `@weave off` and `@weave on` rarely. A exception is that it can be a good idea, to use code like

```
@weave off
@include somefile.web
@weave on
```

and run `pweave direct somefile.web`. This gives to separate articles one for the main program and one for the include-file.

### 3.8 Compatibility Features

To improve the comparability with other literate programming systems `pweb` ignores the following commands:

- `@;` used by `fweb` as invisible semicolon.
- `@\` used by `fweb` as forced line break.
- `@~` used by `fweb` to inhibit a line break.

You should not use any these commands in a new `pweb` project.

### 3.9 Editing Pweb documents

You can use any text editor to create you `pweb` documents. But the correct syntax highlighting can be problem. If you use Emacs for editing you can use the

mmm-mode from M. A. Shulman [4] to patch together the syntax highlighting for  $\LaTeX$ , C and M4. Install the MMM Mode and add the following code to your .emacs.

```
(mmm-add-group
  'web
  '(web1
    :submode c-mode
    :face mmm-code-submode-face
    :delimiter-mode nil
    :front "@<[^>]*@>="
    :back "@[* ]"
  )
  (web2
    :submode c-mode
    :face mmm-code-submode-face
    :delimiter-mode nil
    :front "@[ac]"
    :back "@[* ]" )
  (web3
    :submode m4-mode
    :face mmm-code-submode-face
    :delimiter-mode nil
    :front "@[m]"
    :back "@[* ]" )
  ))

(setq mmm-global-mode 'maybe)
(mmm-add-mode-ext-class 'latex-mode "\\\\.web\\\\" 'web)
(add-to-list 'auto-mode-alist '(("\\\\.web\\\\" . latex-mode))
```

### 3.10 Extending Pweb

The formatting of pweb is driven by a bunch of perl-scripts. For each programming language lang and each formatting language text you need the following files.

In principle you can add any combination of programming language and formatting language.

<code>begin_lang2text.pl</code>	Prints the text at the begin of the formatted document (like <code>\documentclass{article}</code> if you use $\LaTeX$ ).
<code>middle_lang2text.pl</code>	Prints what ever come after the limbo section (if you use HTML it would be most likely <code>&lt;/head&gt;&lt;body&gt;</code> ).
<code>end_lang2text.pl</code>	Prints the text that closes the formatted document (like <code>\end{document}</code> if you use $\LaTeX$ ).
<code>formater_lang2text</code>	Converts the quell code of the programming language in printable code. A typical entry of <code>formater_cpp2tex</code> is <pre>s/(?&lt;![a-zA-Z0-9_])\b(asm auto bool  ... while)\b/\{\rm \bf \$1\}/g;</pre> This regular expression prints the keywords in boldface.
<code>formater_m4lang2text</code>	Like <code>formater_lang2text</code> but treats the macro sections.
<code>codestart_lang2text</code>	Contains things that have to be printed at the start of a code section
<code>codeend_lang2tex</code>	Same as before but for the end of the code section.
<code>codestart_m4lang2text</code>	Contains things that have to be printed at the start of a macro section
<code>codeend_m4lang2text</code>	End of a macro section

Table 1: Files needed by pweb

## References

- [1] D. E. Knuth. Literate programming. *The Computer Journal*, 27:97–111, 1985. Reprintet with correction in [2].
- [2] Donald E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992. Stanford, California.
- [3] R. Seindal, F. Pinard, G. V. Vaughan, and E. Blake. *GNU M4, version 1.4.11*, 2008. Available online: <http://www.gnu.org/software/m4/manual/index.html>.
- [4] Michael Abraham Shulman. *MMM Mode for Emacs*, version 0.4.8 edition, 2004. <http://mmm-mode.sourceforge.net/>.