Literate Programming, Why?

Bart Childs
Texas A&M University

Abstract

Knuth's WEB system for literate programming has slowly built a significant following. Systems now exist for most common high level languages.

I will give an overview of literate programming and a biased view of its status including:

1. an annotated bibliography of available WEB systems,
2. elementary software metrics that may be used in evaluating codes,
3. an indication of the using (practicing?) communities, and
4. expected changes/evolutions in these systems.

The 'Literate Programming' column of the CACM was canceled because it seemed every literate programmer had created their own system. Both sides of this issue will be addressed.

Finally, we will discuss the use of a literate programming environment in a first programming/problem solving/CS-1, and CS-2 type courses. This is being done on an experimental basis in honors sections of our courses this semester.

1 Introduction

Literate programming is a style in which the design of the code reflects that the human reader is as important as the machine reader. The human reader is often associated with the expensive process of maintenance and the machine reader is the compiler/interpreter. Literate programming is a process which should lead to more carefully constructed programs with better, relevant 'systems' documentation.

We acknowledge that a program which produces *correct* results may be quite difficult (to nearly impossible) for a human reader to understand. It is also possible to produce programs that appear correct to the human that the machine will not accept. We claim that these extremes are part of the causes of the high cost of maintenance. This cost is reason enough for our professions to seek a dramatically different methodology in programming.

We **define** literate programming to be a methodology and process for simultaneous construction of a code and its documentation. The code and documentation will coexist in the same source(s). The author must keep as a goal the requirement of human readability ...

A second requirement in this **definition** is that the author take care to modularize the code in appropriate *cognitive chunks*; be descriptive in naming the cognitive chunks; and select variable names to contribute to clear understanding of the function of the code. One screen ...
One page ...
They should not be so short as to be trivial. *Pseudo-code* descriptions should be carefully constructed ... not just verbose. Variable names should be descriptive, Knuth envisioned "literate programmers with thesaurus in hand selecting the best names for variables."

Finally, our **definition** requires that the process produce aids to the human reader that are similar to the best technical books. The minimum set includes: a table of contents; presentation of both the documentation and code at a level of graphic excellence consistent with reasonably available current practice; indices of variable names (similar to compiler cross references) and author supplied supplements; cross references of the *cognitive chunks*; and diagrams or other graphics that will aid understanding of the code.

We believe that verbosity is preferable to omission of documentation. Good documentation and code will be concise, however it is better to err on the side of slight excess in documentation rather than too little.

Current programming methodology is characterized by:

- code and documentation in separate files and rarely synchronized,
- variable names that are mnemonics and acronyms, not words,
- documentation that is seldom created by the programmer, and
- documentation that has a lower priority than the program.

Knuth created the style of literate programming we are following and stated "it will lead to the construction of better programs and that those programs will actually be created in less time than comparable programs created in the usual manner [Knuth84]."

These advantages have not been proven by carefully designed tests, carried out repeatedly. However, we are confident that fair tests would show this to be true. This is based upon circumstantial evidence we have observed. The necessary testing would be expensive because it would have to be based upon several large projects. The resulting software may have to be monitored for years to be conclusive.

The maintenance of codes is frequently quoted as being from 60% to 80% of the cost of significant codes during their lifetimes. Bad grammar and spelling; use of acronyms, mnemonics, and creation of 'new words' from intentional misspellings...

Literate programming is **not a stand alone concept**...
`make`, CASE tools, configuration management, ...
It is a disciplined way of doing what we should be doing ...
We consider literate programming to be a methodology that aids the programmer (team) in producing the 'systems documentation' of codes simultaneous with the code. This integrated documentation is necessary for effective maintenance. Further, we believe the **design** concepts that Knuth selected encourage the use of these concepts without being intrusive.

The 'Literate Programming' column was canceled in the CACM because ...
TheLitProg discussion list leads me to believe that there are several thousand people trying to do it worldwide. Messages often have these themes:

1. just one more feature,
2. I would adopt it if it behaved like the editor on my Macintosh (or ...),
3. I would use it but those I work with will not ...,
4. there is too much to learn, and
5. I would use it if it used WYSIWYG system X instead of TEX.

The first theme validates the CACM rationale. Why do people write these systems?

The second and third themes are indications of the problems in our profession. Thorough, effective communication between teams and members thereof is important. We suspect that the most successful teams have such extensive training but in tracking recent graduates, we know that many teams do not.

The first, fourth, and fifth should also be considered together. We need simple literate programming systems. Statistical studies of existing literate programs show that many features of the existing systems are not used...

We believe that quality software and documentation can be achieved only through purposeful efforts.

Lin posed the question, "since programmers are notorious for not documenting programs, why might they want to write literate ones?" We feel that there are three reasons:

1. the shortage of programmers has disappeared and it is time for a more professional approach;
2. the profitability of the employer should make the programmer's environment better; and
3. the lessons learned in manufacturing by the American automobile industry are significant. We cannot afford to continue *creating software to be repaired.*

Reading maketh a full man; conference a ready man; and writing an exact man. – Francis Bacon

What is written without effort is in general read without pleasure.
– Samuel Johnson

The reason this letter is long is that I did not have time to write a short one.
– Blaise Pascal

2 Literate Programming Systems, Fundamentals

Knuth's WEB is apparently the most commonly used style of literate programming. It can be characterized as a formalized and structured pseudo-code system. It strongly encourages the writing of the code in small parts through the use of a pseudo-code like labeling strategy.

These small parts or chunks may vary depending upon the problem, programming language, and programmer. These are generally called sections. Sections may have three parts: documentation, definitions, and code. The definitions part may be macro definitions or instructions about local formatting conventions. Definition parts of sections are more common in Pascal WEBs than in other languages.

The figure on the next page is from a study by Babiker, Childs, and Fujihara. This figure is a distribution of the percentages of sections in `tex.web` with the indicated number of lines of documentation and code.

In most cases sections should be approximately one screen full. This a guideline and is not intended to be a rule. If Knuth's WEB's are taken to be defining examples, it is obvious that there are a number of sections that are more than one "screen" or significantly less.

The WEB style of literate programming can easily be used with both top-down and bottom-up programming. The sources of TEX and METAFONT are examples of this.

Inside every big program there is a small one struggling to get out.

C. A. R. Hoare

We list these characteristics of a WEB style literate program and the minimum set of tools which are needed to prepare, use, and study the resulting code. Some of these were explicit in the definition given earlier.

1. The high-level language code and the documentation (needed for maintenance functions or writing a user's guide) will be in the same (set of) source file(s).
2. The documentation and high-level language code are complementary and should address the same elements of the algorithms being written.
3. The literate program should have logical subdivisions. We call these *sections*. The current forms of WEBs that we use also allow a higher level module which we call chapters. (Another level is obvious because codes often are based upon the linking of separately compiled parts.)
4. The system should be presented in an order based upon logical considerations rather than syntactic constraints.
5. The documentation should include an examination of alternative solutions and should suggest future maintenance problems and extensions.
6. 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.
7. 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). This list has been affected by our experiences as WEB users: 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.

2.1 The WEB Style of Literate Programming

$$\begin{tikzcd}
 & \texttt{tangle} \arrow[r] & \texttt{foo.hll} \arrow[r] & \text{compiler/linker} \arrow[r] & \text{executable} \\
\text{foo.web} \arrow[ur] \arrow[dr] & & & & \\
 & \texttt{weave} \arrow[r] & \texttt{foo.tex} \arrow[r] & \text{\TeX/dvi\_?} \arrow[r] & \text{document}
\end{tikzcd}$$

Figure 1. The WEB process. The source of the code and documentation are in the source file `foo.web`. The `tangle`, compile, and link process creates an executable. The `weave`, TEX, and `dvi` conversion process creates a document.

There have been a number of WEB and WEB-like systems developed. We will list a simple classification structure for these systems and identify some of the systems in each classification.

- WEB systems that are like the original by Knuth. These systems will have at least the two processors to extract code and documentation, use similar command structures, macro files, and produce similar outputs.
  - Knuth's original WEB
  - Levy and Knuth's CWEB
  - Krommes' FWEB **is multi-lingual!**
- Ramsey created Spider which allows the user to create a WEB for the language of choice. Ramsey's examples include: Ada, awk, C, and SPL.
  - Gragert and Roelofs Reduce
  - Gragert and ? Maple
- Ramsey also created a NOWEB system, a "low-tech" literate programming system (Ramsey, 1991).

3 Is Program X a Literate Program?

The concept of creation of a program with relevant and closely coupled documentation is straightforward. However, it seems to be easy to lose sight of the goal. We feel that literate programs will generally be done only as the result of a concentrated effort. *Literate programs are not created accidentally.* Some obvious questions are:

- Is the level of documentation appropriate?
- Does the documentation relate to the code at hand?
- Are the code fragments appropriately concise?
- Is the index appropriately supplemented with user entries?
- Do the section names agree in tenor with its code?
- Are the variable names appropriate?

We will use this digression to illustrate the point that the literacy of a program is not accidental. Codes done with DoD 2167 are required to have several levels of documents associated with them. In a minimal sense these include: requirements, specifications, design, and code.

It could be argued that the documentation portions of the sections of a literate program could simply be hypertext links to appropriate paragraphs of these different sources. We feel there is no merit to such a proposal. Good documents can be constructed by teams but they require significant editing to ensure continuity, consistent language, and . . . ??? The use of hypertext facilities to see background information would be powerful, but the documentation should generally be written explicitly to complement the code in most cases.

Additional figures from Babiker, Fujihara, and Childs. . .
There are some measures that should not be presented graphically, but simply in tabular form. These include: percent of modules with user supplied index entries, ratio of number of unique user supplied index entries to number of sections, mean (and standard deviation) of number of words per 'section name,' and percent of words in variable names that are not in a dictionary.

4 Companion Tools

The `tangle` and `weave` tools are necessary for the `WEB` style of literate programming. They enable the characteristic of the code source and documentation being in a common file. However, we have found a number of other tools to be of significant value. These tools span the range from editors to aiding checks on syntax, similar to the unix tool 'lint.'

As we stated earlier, we consider literate programming to be a disciplined means of integrating the tasks that are usually ascribed to as good software engineering practices. Most (or all?) of the tools we suggest exist in many other environments. These include:

Literate programming environments are not central to this paper. We include mention of them only to the extent that they should:

- be context sensitive to the structure of the literate program;
- provide access to and use of the index of variables, ...;
- provide access to and use of the pseudo-code names of sections;
- enable navigation through a literate program source based upon the previous items;
- enable navigation through a literate program source using chapter, section, ... mechanisms; and
- be compatible with existing configuration management systems or provide reasonable alternatives.

The editor functions listed above are available in a GNU Emacs based `web-mode`. A complete programming environment would include at least the following coordinated windows (or views) of a program:

- the editor's view of the source of the literate program,
- typeset output, the *listing*,
- debugger views of the code, ..., and
- some form of a view of the pseudo-code structure of the program.

We would expect that a *button* would be available to synchronize the windows when desired. We feel that continuous synchronization would be wasteful and distracting.

- a syntax sensitive editor – The primary features are not the understanding of the detailed syntax of the high level language(s). The most important concept is to be able to view the WEB at a high level (pseudocode constructs, outline editing, ...); navigation via chapters, sections, included files, section names, variable names, and user flags; and elimination of repetitive typing of long section names.
- counters of WEB (and TEX) constructs – This can indicate a great lack of user supplied index entries and needless propagation of TEX macros.
- syntax checkers at a low level – C and TEX both rely heavily on the use of matched braces for grouping. We have found it is often better to leave the editor and check for balanced delimiters off-line.
- graphical and other non-textual views of code structure – Out first effort was a graphical view and the limited screen real estate led us to consider a list view (much like the NeXT presentation of directory structures.
- `makefile` creators – These tools will scan WEBs and put one more level of dependencies in the `makefile`. The output high level language is often dependent upon a web of WEBs and hWEBs.
- configuration management – Systems such as SCCS and RCS should be an integral part of development of quality codes in literate programming as in traditional methodologies.

We have also worked with some tools that are application specific. Examples such as converting the TEX description of a mathematical statement of a differential equation into the C or FORTRAN version of a Fröbenius recurrence relation are powerful but too specific for this paper.

5 Future Changes

Tailorable seems nice but is that defeating the purpose.

The CACM cancellation was because everybody seemed to write their own LP system. Heck, nobody really makes full use of the existing system except for the writers. BC often thinks he has made pretty full use of the system and then finds that he has not used 60 to 70% of the FWEB commands.

Bart's placement of braces in C. Bart's preference for **begin end** over braces. Are such views worthwhile?

I think that I read that somebody could not stand reading WEB because if formatted the Pascal `<>` as $\neq$ ... When I write it in pseudo-code I darned sure don't write the former!

White space is a great aid to reading. However, I find that LaTEX's use is sometimes bothersome, FWEB to some extent too.

Should the editor be really syntax directed? If you are coding in FORTRAN and you have to have an open statement, how about getting this

```
      open ( [UNIT = ]
     o  ,    FILE =
     o  ,    STATUS = 'NEW' 'OLD' 'UNKNOWN' 'SCRATCH'
     o  ,    IOSTAT =
     o  ,    ACCESS = 'SEQUENTIAL' 'DIRECT'
     o  ,    FORM =  'FORMATTED' 'UNFORMATTED'
     o  ,    RECL =
     o  ,    BLANK = 'NULL' 'ZERO'
     o    )
```

and letting the user edit it to the desired form. The environment of learning how to handle every possible completion could be horrible.

If you want to make a user interface more difficult to use, add functionality to it.

Edsger Dijkstra

6 Conclusions

There was a question in the title of this talk, the title was **Literate Programming, Why?**

I think the reason to do literate programming is that we are creating a mess with the current processes. Computer programs are an important part of today's world and if they are worth doing, they are worth doing correctly. I believe that literate programming is the easiest simple step we can take toward creating economical, reliable, and maintainable programs.

There is are three simple requirements for creating literate programmers:

training, training, and training.

If Aggies can learn it, surely anybody can?

7 References

`WEB` for Pascal

`CWEB` for C and C++

`FWEB` for Fortran 77, Fortran 90, C, C++, Ratfor, and TeX

`Spider` for the creation of WEB systems

`labrea.stanford.edu` is the first source for `WEB` and `CWEB`.
`lyman.pppl.gov` is the first source for `FWEB` and `Spider`.

`ftp.cs.tamu.edu` is the first source for `web-mode.el`.

Most are available from `shsu.edu`, `ftp.th-darmstadt.de`, and a number of other servers.