Thirty years of literate programming and more?

Bart Childs

Abstract

Don Knuth created Literate Programming about thirty years ago. It could be called a methodology, discipline, paradigm, ... Bentley's "Programming Pearls" article about Knuth's book, *TEX: The Program*, caused a huge stir in the computing professions. Soon there was announcement of a Literate Programming section for the *CACM*. Several "Literate Programming systems" quickly appeared. This was followed by a few years of mild interest, cancelling the Literate Programming section in the *CACM*, and an apparent lack of public interest in the subject.

Really, what is literate programming?
What is the state of literate programming?

1 Introduction

It is commonly accepted in software engineering circles that 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 and maintenance is accepted to be 60% to 80% of a code's cost. In *The Mythical Man-Month: Essays on Software Engineering,* Frederick 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. ...
>
> The results in fact confirm our teachings about the folly of separate files. Program documentation is notoriously poor, and its maintenance is worse...
>
> The solution ... 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* ...

2 Definition of literate programming

Literate programming is a methodology/process/system for creation of codes in the form of a work of literature as well as executable programs.

The *characteristics of literate programs*, based on Knuth's `WEB` [2] and `CWEB` [3] (with Silvio Levy):

1. The section is the basic unit in the source of a literate program which is analogous to paragraphs in usual textual documents. The section's source should generally be about a single screen.
2. Sections can contain documentation, definitions (macros), and/or code.
3. The order of presentation of sections should maximize the readability of the literate program. There are elements of the structure of the code parts of sections to ensure the correct placement of the code in the resulting program source.
4. Documentation elements of a section should be presented in a format consistent with book quality documents.
5. Code parts and code fragments in documentation parts of sections should be presented in a format consistent with book quality documents. This imposes a requirement of the system parsing the computer language(s) used.

These characteristics should be in a literate programming system for programming in most high level languages. There are examples where a restricted form of literate programming is quite helpful — often the last item in the above list is omitted. These systems still meet Brooks' call for *self-documenting* programs.

My initial experience with literate programs was porting the TEX system to several different systems. I found the formatting of the code to be a great help, especially font selection for keywords, variables, and literals as well as consideration of grouping, loops, etc. Obviously, this requires parsing the code.

2.1 Knuth's Pascal `WEB` and descendants

There were a number of features in the original `WEB` that were needed to make up for problems doing systems programming in Pascal. Levy did not include these in `CWEB` because of the nature of the C language. This literate programming system has endured for nearly three decades. There is no apparent need to change it because of the evolving nature of Pascal. Pascal is not a sufficiently prominent language for systems programming and `web2c` has enabled TEX and its components and friends to be widely ported. Knuth's original `WEB` was done at an early time — relative to most of today's understanding of systems and programming languages — and many features were due to Pascal limitations. The selection of Pascal was done before C would have been a reasonable choice — by only a few years. I strongly recommend reading the original documentation and the documents produced by processing the original literate

programs in the suite of tools to support the TEX system.

There are at least two systems still in use that are quite faithful to the philosophy that Knuth elucidated in his original Pascal-based WEB system and are consistent with the definition and the list of characteristics given: CWEB and FWEB. Each of these support more than one language.

2.2 CWEB — Levy and Knuth

Levy's original CWEB was an adaption of WEB to the use of C. Several features of WEB that were needed for Pascal were removed. Knuth joined Levy in the support and evolution of CWEB and others contributed the addition of support of C++ and Java.

2.3 FWEB — Krommes

John Krommes' FWEB is based on CWEB. FWEB supports C, C++, Fortran (77) and Ratfor. Krommes' research dictated the need for this multilingual nature of FWEB because his research was based on both long running Fortran programs and programs to interpret the data that were done in C++ or C.

Fortran had many vagaries that exceeded those of Pascal, including its ancient card orientation. The acronym was changed to a noun in the '90s. Fortran is central on many parallel systems and each seems to have a unique system of compiler directives that are often required to be in-line with the code. Krommes handled these with admirable foresight. The Fortran standards committee has been active in trying to bring that community into the twenty-first century. For example, semicolons are now allowed to end statements.

Krommes included multiple output and input files as well as the option of being language independent that enabled the logical next step of including scripts as part of the literate program. This language-independent mode is called verbatim.

All in all, I would have liked a much smaller version of FWEB. That sounds like a common whine about TEX: "It is too big!"

We found that a small part of TEX and a web can be taught to beginning students (see section 4). It simply requires some work.

2.4 WEB-like systems

It is my opinion that the formatted code that the above literate programming systems give for their high level languages is of great benefit. Others obviously do not share my enthusiasm.

Several systems that have been called literate programming by their creators are language independent and therefore do not meet the characteristic in item 5, section 4. This feature of *not parsing and formatting* the programming language allows the use of many different languages. This simplicity and flexibility is desirable but I believe the benefits of the formatting are crucial.

Creators of two of these systems, Williams (FunnelWEB [8]) and Ramsey (NoWEB [6]) obviously have a different opinion and focus on the benefits of being able to order sections for expository reasons rather than compiler requirements, and include effective documentation as making this form of literate programming worthwhile.

I recall a reply by Williams in the literate programming discussion group in response to a user complaining about the tangled output not looking like the user wanted: "Crikey, will they ever learn? If the web is well written you will not want to look at that version of the code." Well, something like that.

2.5 docstrip and doc.sty — LaTeX tools

Frank Mittelbach created the doc.sty package to combine the TEX code and documentation for LATEX. He then created docstrip to complete a literate programming system for LATEX [4]. This might be the most used literate programming system on a regular basis because it is the common format for LATEX distributions. It should also be noted that there have been several contributors to the evolution of these tools, as is typical of the TEX community.

The advantage of having the documentation and code in one file has been discussed earlier. Since docstrip is written in TEX, which is an interpreter, minimizing comments in the output code was important to execution speeds.

The code part is not parsed when the document is processed, thus the system can be used for code other than (LA)TEX. I have found references to the use of docstrip with other coding systems, notably statistical packages. Perhaps a future article will explore docstrip in more detail.

2.6 Literate programming-like usage

Nelson Beebe created a system he described as "like literate programming" to document the many scripts (each a code fragment) for his book *Shell Scripting* [7] (with Arnold Robbins).

Like the previous subsection, a future paper is needed for a survey of many such uses of the ideas of literate programming.

3 web-mode — An Emacs-based tool

Mark Motl finished his dissertation under my direction by developing and testing a tool to adapt Emacs

to literate programming for WEB and CWEB [5].

The selection of Emacs was a bit like Knuth's selection of Pascal for the second writing of TEX. Emacs was/is a large, stable system and relatively platform independent. The emergence of workstations with a tightly coupled graphics screen was also a great benefit to the Emacs philosophy. Much of the early development of web-mode was done on shared resource systems and the final work was done on workstations.

Finishing touches were added by several students and some by me. Some characteristics of web-mode:

- Emacs is cross platform. Most of my use in the last few years has been on Macs, PCs, Sun, and Linux workstations.
- It is open in the same sense as TEX and Emacs.
- The user specifies he/she is using WEB, CWEB, or FWEB.
- If the web is a new file, the appropriate header files are inserted with customized user-specific information.
- For existing webs, navigation information is developed unless files (like .aux in LATEX and similar WEB files) are newer than the web source.
- As the user enters source, the source is parsed to ensure that section elements are complete. For example, the meta-ness of code section names have proper balance and the trailing = sign, if appropriate.
- Knuth included a feature to allow the user not to type the entire code section name, but use an ellipsis for completion. In web-mode the Emacs completion feature makes this unnecessary.
- Navigation of a web can be done by chapter/section name/number, by sections referencing variables, etc. These actions can be invoked by function keys or pull down menus.
- The user can view the web and change file by source, or preview of the DVI or PDF output.
- Execution of the TANGLE, WEAVE, TEX, LATEX, can be invoked by function key or pull-down menus.
- Outline editing of the source is especially useful. The first line of sections (and chapters) and the lines defining code sections are displayed. It gives a new meaning to scrolling through source.

There is much more but that detail is not needed here.

The distribution of web-mode will be available soon from my home page after I perform some consistency checks on current Windows and Apple systems.

4 My CS/1 experience

This section is adapted from our paper at the 16th TUG meeting in St. Petersburg [1].

We embarked on a project to teach the first computer science course using literate programming while covering all the topics covered in the usual sections. We differed from the environment used by the other sections by using Emacs and the literate programming environment web-mode and GNU Pascal as opposed to Turbo Pascal.

Our CS/1 course was entitled "Programming I" and although the catalog did not specify Pascal, it was understood that all sections of the course would use the same language and that problem solving would be central.

An inherent part of these CS/1 courses is to develop the student's skills in *problem solving*. Indeed, in many course outlines, that is part of the title and the main emphasis in the description of the course contents. A problem solving methodology is often stated in CS/1 courses which generally has steps like:

1. State the problem completely!
2. Develop all necessary assumptions.
3. Develop an algorithm and test data set(s).
4. Code the problem.
5. Analyze the results (and iterate?).

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 think that the first sentence in this paragraph should be particularly relevant to students because the human reader (the one who assigns grades) is obviously the most important reader.

The features of literate programming that gave us the confidence to expect positive results are:

1. Top-down and bottom-up programming since it is structured pseudo-code.
2. Programming in small sections where most sections of code and documentation (section in this use is similar to a paragraph in prose) are approximately a screen or less of source.
3. Typeset documentation (after all, Knuth was rewriting TEX).
4. Pretty-printed code where the keywords are in bold, user supplied names in italics, etc.
5. Extensive reading aids are automatically generated including a table of contents and index.

The readability is improved by the programmer's liberal use of the tools furnished.

Each item in the following list could be added to the corresponding item in the previous list. We think there is merit to this split.

1. These topics are usual in CS/1 books but they generally lack the integration to make them really effective for the student.
2. Divide and conquer is also espoused but the larger examples given in many books forsake the principle.
3. It may be argued that this is 'feeding pearls to the swine' but we like the cognitive emphasis that comes from logical substitution of words for key-words, etc.
4. The fact that `weave` breaks lines based on its parsing is another cognitive reinforcement.
5. Encouraging/requiring students to review their programs as documents makes them *think* about readability.

I am still firmly convinced that literate programming is the way to set budding programmers and systems developers on the right track to writing beautiful and excellent programs. There was a section entitled *Problems with 'Problem Solving'* in the TUG 16 paper which gave many of the arguments for this that we gleaned from the literature.

"Are You Crazy!?" This was frequently shouted at us because, as *everybody in the world knows:*

- Emacs is impossible to learn and use,
- TEX is impossible to learn,
- the addition of `WEB` makes for too many steps to learn, and
- there is a reason for all those Aggie jokes.

and *therefore* our project was doomed!

Sometimes items from that list were suggested gently instead of being yelled or blurted out while the correspondent was writhing on the floor. We exercised a little judgement and did it on the smallest sections of the course, namely the honors sections.

There were several important items that we considered in the design of the course and how we executed the course.

Testing We intended the course to be more *problem-oriented* rather than *program-oriented*. The tests included a pre-test that was no part of the other sections. All tests were slanted away from Pascal details.

Emacs and `web-mode` We felt that in spite of this being a new editor to nearly all, the `web-mode` literate programing tool was our only choice. We modified the Emacs reference card and gave the students a five page memo based on Knuth's `WEB` introduction.

Knuth's `WEB` We were restrained to the use of Pascal and therefore Knuth's original `WEB` was appropriate. Some of the necessary minutiae was easily omitted by use of `web-mode`.

How the course was taught The focus of the semester was on problem solving. Pascal syntax was brought along as a means of presenting and then implementing a solution. The lecturer presented the week's information and handled questions on a daily basis. The TA handled the labs. Total enrollment in the class was about 40 and about half in each lab section.

Do all labs twice The labs in our courses are usually twice a week rather than one long period per week. We took advantage of that and each lab was done twice. The first time was used to have the documentation parts of sections being somewhat complete and the code parts sketchy. `WEB` used in this manner can be a documented pseudo-code system. This draft was marked quickly and returned for the more complete version to be finished before moving on to the next lab.

4.1 Results — informal summary

Three pages of detailed results were presented in the TUG 16 paper. I will present a high level view of those I think that are most important and relevant.

The initial design would have led to failure if we had analyzed the results promptly as planned. Irrelevant personal problems delayed that analysis for more than a year. The one result from the immediate analysis was

> The pre-test showed that the non-majors did not have the problem solving skills of the majors. The change was steady and by the second regular test the non-majors were superior.

Most of the students had completed more computer science courses by the time the analysis was started in earnest. The evaluation process was modified to include tracking those students who took these additional courses. Also, data was extracted for students taking the same class the previous year, this was taught by a more experienced teacher.

The additional courses were a CS/2 course which was dominated by learning the C language and then a data structures class. The performance in the CS/2 courses were not significantly different for those with and without the literate programming exposure.

The literate programming exposure apparently made a significant difference in the data structures

courses. With hindsight, that seems logical because a data structures course is much more of a problem solving experience and as implied above CS/2 was too much a memorization of C syntax. The CS/2 and data structures courses mentioned were not taught by those involved in this study.

We feel the background of the students was not atypical of many CS/1 type courses. The majority of the class were majoring in computer science, but a significant number were using the course as a minor elective, a basis for deciding if they want CS as a major, or other reasons. There was not an unusual change of majors for the students in the study.

4.2 Student comments and evaluation

Upon nearing completion of the CS/1 course, the students were asked to submit a paper reflecting their feelings and attitudes towards the `WEB` programming methodology. This was to be written as a typical one-page technical note. Some expected comments were made in the evaluation process at the end of the CS/1 course taught using literate programming.

- TEX is not easy to learn,
- learning `WEB` was OK.
- Emacs was difficult (the replacement of function keys by pull-down menus was not complete in `web-mode` at the time).

4.3 Conclusions about teaching

We taught an honors section of a CS/1 course in a different manner than usual, namely using literate programming. The students used an editor, a formatting system, and a coding style that was new to all. The students' performance in subsequent courses was not hurt and may have been helped with the different methodology. The results of using the program development methodology in the CS/1 course indicate that the methodology is successful in teaching novice programmers good problem solving skills.

These are the results of the experiment:

- The students showed an increase in their problem solving skills.
- Those students unfamiliar with the Pascal programming language, or any other programming language, were more successful then those familiar with Pascal at using the literate programming paradigm to capture and document their problem solving process.
- The students were able to learn the `WEB` rules, the `web-mode` environment, GNU Emacs, and TEX rules, as well as the Pascal syntax and constructs.
- Those students exposed to the program development methodology utilizing the literate programming paradigm were as successful in the subsequent CS/2 course as those not exposed to the methodology.
- Those students exposed to literate programming were significantly more successful in the data structures course than those not exposed to the methodology.
- The subject program development methodology may lead to an improved software development process; however, more tests should be conducted.

5 Tools

This list is used to describe some of the tools I know of that exist to aid in literate programming. Some have not been widely published, much to my shame.

CWEB There are several tools referenced on Knuth's home page; see his `CWEB` area.

Leo There are several references to this "outline" editor. I have not had time to seriously look at this yet.

`web-mode` I have corresponded with many users over the years but did not realize that `web-mode` was invisible to the usual literature reviews. This will be corrected after a detailed review and making sure of Emacs updates. The only problems I have encountered are AUCTEX's implementation of `description` environments and conscription of some function keys.

TAMU We did several tools and need to organize and publish them as a set. A tool that was frequently used when applied to a `WEB` or a TEX source would give a statistical analysis of commands (I used more "features" than the students).

6 Examples of literate programs

The long term success of literate programming may depend on the number and quality of published literate programs. A partial listing of literate programs that are available openly is offered here as a start.

WEB The sources yield `TANGLE` and `WEAVE` at a minimum.

CWEB As above. The manual was last updated eight years ago and is now out of print.

TEX and Metafont, the programs The printed books are available from Addison-Wesley or you can exercise your printer and paper budget.

Stanford GraphBase Knuth presents 31 `WEB`s in his platform for combinatorial computing.

Don Knuth's home page Nearly innumerable interesting `CWEB`s for a wide range of topics.

CACM The few that were contributed to the Literate Programming column.

BC While preparing this I encountered a reference to a code of mine, `PS_Quasi`, which related to experimental, theoretical numerical solutions of ordinary differential equations. The link failed but I need to reestablish that.

I also have a few dozen that were done by our team that should be termed tools for analyzing literate programs. These need to be cleaned up, cataloged and published (on the web).

I have emphasized literate programs based on the three systems that meet the definition I stated earlier. The intended functions of those systems center on programming in specific high level languages.

7 The state of literate programming

This is a difficult topic to treat with authority and a straight face. Most of the systems have had only minor changes, if any, in the last twenty years. However, stability is frequently a good thing. It is common knowledge that adding functions to an interface will make it more difficult to use.

Identification of and counting the number of users of literate programming and "literate programming like" systems may be impossible. Some visibility includes:

original style Knuth's home page and books point to many excellent examples. I am remiss in not making a number of examples and tools available, but I will.

literateprogramming.com Some elements point to example literate programs, but only a few. It includes a fair number of references to many items that someone has called literate programming. This includes a note entitled "POD is not literate programming." Most of the entries are from the last century, but that is not so long ago.

literate programming *like* Robbins and Beebe's *Shell Scripting* book. This perhaps could have been done using `docstrip`. How many similar projects are there that could be helped by a good survey?

literate programming — other There is a community of users of the packages FunnelWEB and NoWEB. I particularly like Ramsey's philosophy of piping small tools, following the Unix philosophy. Application of these systems (and `docstrip`) is also available with the statistical system R (see Uwe Ziegenhagen's article in these proceedings, pp. 189–192).

8 Conclusions

I believe that this style of program development is a great contribution to the goal of creating excellent and maintainable programs, if it is used diligently. I have often wondered how many of the errors that Knuth has rewarded us for would have even been found if the program had been in the style of Unix "pretty printing." In spite of this, it is referenced too little.

We have observed first year students are already like the professionals: "No, I do not want to learn anything new if I already have some knowledge in the area." We think it should appear early and repeatedly in the curriculum. The design process that is called for in most software engineering treatises is a natural fit for literate programming, in my opinion.

References

[1] Bart Childs, Deborah Dunn, and William Lively. Teaching CS/1 courses in a literate manner. *TUGboat*, 16(3):300–309, September 1995.

[2] Donald E. Knuth. The `WEB` system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, September 1983.

[3] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison-Wesley, Reading, MA, USA, 1993.

[4] Frank Mittelbach, Denys Duchier, Johannes Braams, Marcin Woliński, and Mark Wooding. The `docstrip` program. Technical report, Universität Mainz, Mainz, Germany, 2005.

[5] Mark Bentley Motl. *A Literate Programming Environment Based on an Extensible Editor*. PhD thesis, Texas A&M University, College Station, TX, December 1990.

[6] Norman Ramsey. Weaving a language-independent `WEB`. *Communications of the ACM*, 32(9):1051–1055, September 1989.

[7] Arnold Robbins and Nelson H. F. Beebe. *Classic Shell Scripting*. O'Reilly, Sebastopol, CA, 2005.

[8] Ross N. Williams. Funnelweb User's Manual. `http://www.ross.net/funnelweb/`, May 1992. V1.0 for FunnelWeb V3.0.

◇ Bart Childs
Texas A&M University
College Station, TeXas 77843-3112
USA
`bart (at) tamu dot edu`
`http://faculty.cse.tamu.edu/bart/`