tutorial

Literate Programming

Michael A. Guravage
NLR
Anthony Fokkerweg 2
1059 CM Amsterdam NL
guravage@nlr.nl

abstract

This article is a short introduction to the theory and practice of a programming style known as Literate Programming; a style that changes the focus of writing programs away from telling a computer what to do and toward explaining to a person what it is we are telling the computer to do. Literate Programming overcomes the limitations inherent in presenting traditionally structured program text. Using a balanced mix of informal and formal methods, literate programs are presented in a way suited for human understanding. Processing a literate program source results in both a nicely typeset document describing the parts of the program in an order that elucidates their design, and source code in an order in which it will compile.

keywords

Literate Programming, Structured Programming, WEB

Introduction

Writing good programs is hard; so is writing good program documentation.

Structured programming in the 1970's and Object Oriented Technologies in the 1980's are methods that help us capture and organize the design and implementation of complex software. However, though our design decisions are embodied in the code we write, they are often obscured by the code itself. Few people enjoy navigating through pages of curly-braces, control-structures, and function calls in unfamiliar code. Literate Programming provides a way to expose and elucidate a program's design by presenting its parts in an order and at a level of abstraction that places a premium on understanding.

The tasks of writing a program and writing program documentation are often seen as separate and sequential. There are several undesirable consequences of this separation. First is that their writing does not inform each other. If the program is written before its documentation, the program's relationship to its documentation is merely coincidental. Second is that separate code and documentation tend to diverge over time. As a consequence, the code and documentation do not cooperate as well as they could toward either maintenance or reuse. Literate Programming tries to address these issues by integrating, or if you prefer - blurring the distinction between, code and documentation in such a way that code and documentation contribute to and complement each other.

The remainder of this article will be arranged as follows: we begin by discussing the motives for and ideas behind literate programming. Next, we identify the properties that characterize literate programs. The process of transforming literate programs into running code and typeset documents is explained. We compare language dependent and language independent literate programming tools and enumerate the benefits and liabilities of each. Lastly, the costs of using literate programming are estimated.

Motivation for Literate Programming

Literate Programming originated in the early 1980's as part of Knuth's work on TEX and digital typography. By this time, the concept of *structured programming* was already well established. Structured programming advocates decomposing a problem into a hierarchy of smaller problems according to some subordinating principle. The level of abstraction proceeds from generalities to specifics until each task at the bottom of the hierarchy admits a solution. One useful criteria for subdividing tasks into smaller ones is that it should be possible at each level of abstraction to give an informal description of its subtasks.

Marc van Leeuwen (van Leeuwen, 1990) observed that, "Although the composition of a structured program should reflect the design decisions that led to its construction, the traditional way of presenting such programs, e.g. listings of code containing comments, lacks the appropriate facilities for communicating this information effectively to the readers of the program, seriously limiting the readability, especially to people other than the programmer of the code. Yet readability is of vital importance, because it is only by careful reading that we can verify that the design of a program is sound and well-implemented, and to understand where and how changes can be made when such a need arises."

Knuth expressed what many have long recognized, that the order in which a traditional program is written is a concession to the computer. A Euclidean proof begins with first principles, and builds a consistent hierarchy of definitions, postulates, and theorems in service of a proof. Likewise, a computer program is written as a hierarchy

of definitions, function prototypes, data structures, and instructions whose order satisfies the demands of a computer. While the Euclidean style of proof is unparalled for its completeness, its style is less applicable when the goal is to convey a general appreciation for a subject to the uninitiated. Though a computer program implementing an algorithm is sufficient to instruct a computer how to perform the algorithm, a listing of the program may be, and often is, insufficient to explain the workings of the algorithm to a person. Vice versa, an informal description of an algorithm can reveal to a person how the algorithm works, the same description is useless to a computer. It is apparent that there is one way a program must be ordered in order to be parsed and compiled by a computer, but there may be many different ways to arrange the program to explain its workings to a person.

"Literate Programming is a natural sequel to structured programming (van Leeuwen, 1990)", and overcomes the shortcomings of the latter by combining informal natural language, a formal programming language, and a flexible order of elaboration in such a way that places a premium on exposition and understanding. By tightly coupling program code and program documentation, a literate programmer strives to write programs that are comprehensible because their concepts have been introduced in an order and at a level of detail that is suited for human understanding (Knuth, 1992a).

What's in a name?

The origin of the term *literate programming* has both a serious and a light side. On a serious side, Knuth found that the more he concentrated on developing a programming style that concentrated on clarity of exposition, the more he treated his programs as works of literature, the better his programs became. Their designs were improved, their implementations had less bugs than their traditional counterparts; and they were readable. Like an author writing a story or a researcher writing a technical article, the programmer adopts the common goal of tell his audience just what they need to know, just when they need to know it.

On the lighter side, Knuth wrote, "I must confess that there may also be a bit of malice in my choice of a title. During the 1970s I was coerced like everyone else into adopting the ideas of structured programming, because I couldn't bear to be found guilty of writing unstructured programs. Now I have a chance to get even. By coining the phrase literate programming, I am imposing a moral commitment on everyone who hears the term; surely nobody wants to admit to writing an illiterate program (Knuth, 1992a)."

Knuth also coined the term WEB to describe a literate program. He thought that a complex piece of software is best regarded as a web that has been delicately pieced together from simple materials (Knuth, 1992a).

To complete the literate programming nomenclature, TANGLE is the name of the processes that rearranges a literate program in an order that is easier for a computer to understand; while WEAVE is the name of the process that rearranges a literate program into an order that is easier for a person to understand. Throughout this article, the names WEB, TANGLE, and WEAVE will refere to these processes; without reference to specific literate programming tools with the same names.

Properties of Literate Programs

A program has to exhibit three properties before it can be called a literate program. The first property is that a single literate program source should, when processed, produces both a runnable program and a nicely typeset document. The work of *tangling* the code and *weaving* the document will be explained in detail in the following sections. The second property is that a literate program must exhibit a flexible order of presentation. As has already been mentioned, the order of presentation to a person is independent of the order of compilation by a machine. The third and last property is that a literate program, and the tools that process it, should facilitate the automatic generation of cross-references, indices, and a table of contents.

Anatomy of Literate Programs

Knuth (Knuth, 1992a) describes a literate program as, "A traditional computer program that has been cut up into pieces and rearranged into an order that is easier for a person to understand. A traditional computer program is a literate program that has been rearranged in an order that is easier for a computer to understand. Literate and traditional programs are essentially the same kind of things, but their parts are arranged differently. You should be able to understand the traditional program better in its literate form, if its author has chosen a good order of presentation."

A literate program consists of numbered *sections* or *chunks*. A section can be either named or unnamed and corresponds to a paragraph in written language. Within a section, a single idea is developed. A section is divided into two parts: an informal specification in a natural language, and a formal specification in a programming language. The informal half contains a written description of the idea which is the focus of the section. The formal half contains the code implementing that idea.

To earn the title of literate programmer, one needs to add a handful new control sequences to his repertoire of programming tools. Most begin with an '`@`' and control such things as sectioning, cross-references, and layout. Figure

2 shows a portion of literate C source code extracted from the word count example program that comes with the CWEB literate programming package. The section begins with an '`@`' followed by several lines of commentary. Next is the section name which appears between angle brackets: '`@< Name of section @>=`.' The section name delimits the informal commentary from the formal code - 'also called the *replacement text*.'

In this example the code is dominated by a '`do ... while`' loop; which contains several references to other sections whose names each appear between '`@<`' and '`@>`' pairs. Notice that, at this level of abstraction, the program text for the inner levels are suppressed, making the outline of the outer level more clear.

Processing a WEB

A literate programmer writes code that serves as the source for two different system routines. Figure 1 shows the two paths a literate program can take. One path is called *weaving* the web; the result of applying WEAVE and TEX is a nicely typeset document. The other path is called *tangling* the web; the result of applying TANGLE is program code rearranged in an order ready for compilation.

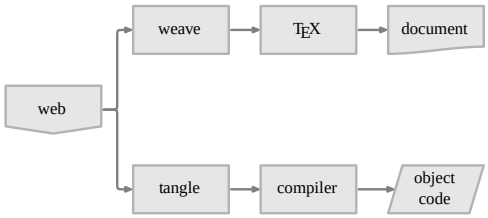


Figure 1 Processing a WEB

WEAVE

Figure 3 shows the result of passing the previous piece of literate code through WEAVE and then typesetting it with TEX. To enhance readability, reserved words, like '**do** and **while**', are set in boldface type, and identifiers, like '*argc*', are set in italics.

Note that WEAVE has resolved the section numbers, references, and cross-references. The footnote at the end of the section complements the section number by showing in which other sections the current section is referenced.

TANGLE

TANGLE removes all the commentary from a WEB and rearranges the code into an order in which it will be compiled - see Figure 4. The reordering porceeds as follows: first, all unnamed sections are collected in relative order; then references to named sections are replaced with their replacement text. This continues until all section references have been replaced.

Like object code, the tangled source code can be considered incidental. However, to help you navigate through the code if you venture to read it, TANGLE adds comments that show which literate section the code originated. TANGLE also can add '`#line`' directives to allow compilers and debuggers to map lines in the tangled code back to lines in the original literate source.

A comparison of the tangled code to either the original literate source or the woven result should leave no one in doubt how TANGLE got its name.

Tools for Literate Programming

Literate Programming tools can be divided into two broad groups: those that are language dependent and those that are language independent. Members of the first group include WEB - Knuth's original literate programming environment for Pascal, CWEB - an adaption of WEB to C by Silvio Levy, and CWEBX - an implementation of CWEB by Marc van Leeuwen.

Language dependent tools tend to be large monolithic programs. This makes them easier to port but difficult to modify or extend. Knowing the syntax of the language for which they are written, language dependent tools provide native support for pretty-printing. They also recognize language specific types, which facilitates the automatic indexing of function and variable names.

An example of a language independent literate programming tool is NOWEB written by Norman Ramsey(Ramsey, 1993). The motivation behind NOWEB was that its author thought WEB and its derivatives were too complex and inflexible. The result was a literate programming tool that is simple, extensible, and independent of the target programming language. Restricting itself to writing named chunks of code in any order, with interleaved documentation, NOWEB provides much of the functionality of WEB at a fraction of the complexity.

Unlike monolithic language dependent tools, NOWEB adopts a pipelined architecture. Like the UNIX notion of pipes where the output of one program is the input to another, the transformation from literate source to typeset document, or program code, is achieved via a succession of distinct steps. Besides NOWEB's predefined pipeline, filters can be added to both NOWEAVE and NOTANGLE allowing one to easily change their behavior or add new features. Filters add such features as pretty-printing, indices, and cross-references. Currently NOWEB produces TEX, LATEX, HTML, and TFOFF.

```
@ Now we scan the remaining arguments and try to open a file, if
possible.  The file is processed and its statistics are given.
We use a |do|~\dots~|while| loop because we should read from the
standard input if no file name is given.
@<Process...@>=
argc--;
do@+{
  @<If a file is given, try to open |*(++argv)|; |continue| if unsuccessful@>;
  @<Initialize pointers and counters@>;
  @<Scan file@>;
  @<Write statistics for file@>;
  @<Close file@>;
  @<Update grand totals@>; /* even if there is only one file */
}@+while (--argc>0);
```

Figure 2 A portion of literate source code

Counting the Cost

At the lowest level, where we consider machine time, the additional time needed to process a literate program is negligible. With all the mechanics burried in a makefile, invoking TANGLE takes just slightly longer than the time it takes to compile the resulting code. Likewise, invoking WEAVE does not take long to produce a file, but typesetting the result with TEX does take longer by comparison. Still, typesetting even a moderate size document goes rather quickly.

On a higher level there is no consensus on how long it takes to write a literate program in comparison to writing a traditional program. Knuth says that the time he needs to write and debug a literate program is no longer than that for a traditional program. He contends that any extra time he spends writing commentary is recovered because the result needs less debugging. The best one can say is that the cost of creating a high-quality, well documented literate program is the same order of magnitude as writing and documenting an equivalent traditional program.

Literate Programming scales well and can be applied to everything from one-off disposable examples to complicated programming projects. The quality of the end result is a function of how much time and effort you are willing to expend to achieve it.

Style

What style of writing is best suited to literate programming? Since literate programming grants the programmer the same freedom of expression as any author, there is no definitive style. But there are a few useful guidelines to follow. Choose section names that are long enough to capture the meaning of the code in that section. Another tip can be found in the word count program (Figure 3) where Knuth strengthens the section names by begining each with an imperative verb. Search for a balance between formal and informal exposition so you can convey the essence of a section clearly, without distracting the reader with unnecessary detail.

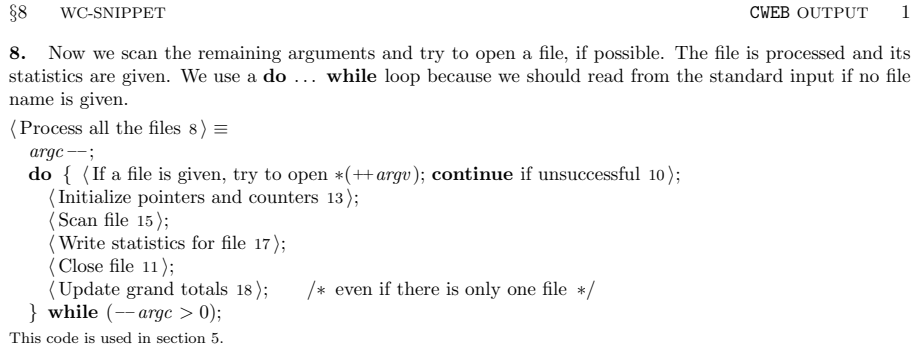
For general remarks on writing styles, you will find some excellent advice in George Orwell's 1946 essay with the foreboding title: "Politics and the English Language(Orwell, 1946)." He encourages the scrupulous writer, in every sentence that he writes, to ask the following questions, "What am I trying to say? What words will express it? What image or idiom will make it clearer? Is this image fresh enough to have an effect? Could I put it more shortly?"

Another indispensable guide to writing is Strunk and White's classic "Elements of Style(Strunk and White, 1979)" where Strunk advises us to be, "direct, simple, brief, vigorous, and lucid."

Summary

The main points to remember when thinking about literate programming are:

- Literate programs use a balanced mix of formal and informal methods, and a flexible order of elaboration, to present information in an order and at a level of detail suited for human understanding.
- Literate Programming does not enforce any one programming paradigm.
- A single literate program produces both a runnable program and a nicely typeset document.
- Literate Programming tools can be divided into two broad groups: those that are language dependent and those that are language independent.
- The cost of writing a literate program is usually equal to or greater than that of writing a traditional one,



§8 WC-SNIPPET CWEB OUTPUT 1

8. Now we scan the remaining arguments and try to open a file, if possible. The file is processed and its statistics are given. We use a **do** ... **while** loop because we should read from the standard input if no file name is given.

⟨ Process all the files 8 ⟩ ≡
  argc−−;
  do { ⟨ If a file is given, try to open ∗(++*argv*); **continue** if unsuccessful 10 ⟩;
    ⟨ Initialize pointers and counters 13 ⟩;
    ⟨ Scan file 15 ⟩;
    ⟨ Write statistics for file 17 ⟩;
    ⟨ Close file 11 ⟩;
    ⟨ Update grand totals 18 ⟩; /∗ even if there is only one file ∗/
  } **while** (−−*argc* > 0);
This code is used in section 5.

Figure 3 The result of applying WEAVE

```
#line 106 "wc.w"
argc--;
do{/*10:*/
#line 127 "wc.w"
if(file_count>0&&(fd=open(*(++argv),READ_ONLY))<0){
fprintf(stderr,"%s: cannot open file %s\n",prog_name,*argv);
status|=cannot_open_file;
file_count--;
continue;
}/*:10*/
#line 108 "wc.w"
;/*13:*/
#line 154 "wc.w"
ptr=buf_end=buffer;line_count=word_count=char_count=0;in_word=0;/*:13*/
#line 109 "wc.w"
;/*15:*/
```

Figure 4 The result of applying TANGLE

but time is often regained since literate programs take less time to debug.

Resources

If you are interested in learning more about literate programming, you should begin with Knuth's book, "Literate Programming(Knuth, 1992a)." Manuals for CWEB and CWEBX can be found in their respective distributions. In addition, the manual for CWEB is published as a book entitled: "The CWEB System of Structured Documentation(Knuth and Levy, 1994)."

Volumes A and B of Knuth's series on Computers & Typesetting describe TEX and METAFONT; volumes C and D contain their literate Pascal sources(Knuth, 1993a, 1992b). A fine example of programming with CWEB is "The Stanford GraphBase(Knuth, 1993b)", Knuth's book on combinatorial data structures and programs.

There are several online resources: the literate programming frequently asked questions can be found at: `shelob.ce.ttu.edu/daves/lpfaq/faq.html`. The ftp site for the literate programming archive is: `ftp.th-darmstadt.de`, and the literate programming newsgroup's name is: `comp.programming.literate`.

A Trustworthy Opinion

Knuth's own opinion about literate programming is expressed in this little WEB which appears on the cover of his book of the same name.

```
⟨ Emphatic declarations 1 ⟩;
  examples: array [vast] of small .. large; beauty; real;
⟨ True confessions 10 ⟩;
  for readers(human) do write(webs);
  while programming = art do
    begin incr(pleasure); decr(bugs); incr(portability);
    incr(maintainability); incr(quality); incr(salary);
    end   { happily ever after }
```

This code is used in theory and practice.

References

van Leeuwen, M. A. A. (1990). *Literate Programming in C*. Manual for CWEBx.

Knuth, D. E. (1992a). *Literate Programming*. CLSI. Lecture Notes Number 27.

Ramsey, N. (1993). Literate-programming tools can be simple and extensible. .

Orwell, G. (1946). Politics and the english language. essay.

Strunk, W. and White, E. B. (1979). *The Elements of Style*. Macmillian, 3rd edition.

Knuth, D. E. and Levy, S. (1994). *The* CWEB *System of Structured Documentation*. version 3.0.

Knuth, D. E. (1993a). *TEX the program*, volume B of *Computers & Typeseting*. Addison Wesley.

Knuth, D. E. (1992b). METAFONT *the program*, volume D of *Computers & Typeseting*. Addison Wesley.

Knuth, D. E. (1993b). *The Stanford GraphBase*. ACM Press.