Literate Programming

by Mike Gradman

Introduction

Computer programming has often seen its share of code that is hard to understand and read due to poor documentation and style. Unfortunately, this not so straightforward code is found in most programs. Frequently, programmers write code that only they can understand, not considering others who may work with the code or read through it. Even if the programmer tries to clearly explain what he is doing through writing comments in the code or uses good style, others may not be able to make sense out of the code. Nasty bugs might pop up that would be hard to fix or a developer might need to know what is going on inside a particular piece of code. He might have to nag the original programmer to clarify points about the program if he is even reachable at all. Also, in the software development process, either management, the designers, or programmers have to keep up with many different kinds of documents, or artifacts throughout the lifecycle. For example, something in the requirements for the application might either be misunderstood or totally overlooked in the code as the developers have to refer to a separate document to see what needs to be implemented. Similar inconsistencies can occur due to the myriad of documents that are produced in the software life cycle. Most of these problems occur due to a lack of proper understanding of the application being developed or used or the code written for that application. A paradigm called "literate programming" helps developers, teachers, and students in the process of solving various problems and understanding their solutions, especially if they have to build on or otherwise comprehend other people's work.

Programming Languages: Conventional Attempts at Understanding Programs

Before Programming Languages

In the early years of the Computer Age, programmers had to use machine language to instruct the computers what they wanted them to do, which involved commands such as moving values into registers, manipulating memory, or handling control flow in a complicated manner. At that time, there was no essence of entities or other abstractions that could give the layperson or even expert programmers a true understanding of the problem at hand.

FORTRAN

Then, in the 1950's, the first attempts to develop high-level programming languages were made with the aim to allow the programmer to think more in terms of what the program needed to do and how to do it. One of these first languages was FORTRAN, with abstractions such as data types and control structures in the form of simple English language (such as the DO-loop and the IF-THEN statement) which represented common idioms that the programmer could grasp [Mac87][1]. For example, if

[1] Using a documentation style used in many IEEE journals … take first three letters of last name of first author listed concatenated with the year of publication … article by Edward Jones and Robert Smith from 1998 would be listed as [Jon98]. References list uses this shorthand followed by the citation in MLA format.

the programmer wanted to double the value of each element of a ten-element array A, he would write:

```
      DO 100 I=1, 10
        A(I) = A(I) * 2
100 CONTINUE
```

Also, the IF-statement could clearly express limited forms of control such as taking the absolute value of a variable named VALUE:

```
      IF (VALUE.LT.0)
         VALUE = -VALUE
```

Structures like these allowed the developer to state what he wanted to do in a clear and concise fashion by 1950's standards.

Also, the developer could break up his work and design pieces of the application at a time by using subroutines [Mac87] and either try to understand each small part of the program and how it fits in the whole (bottom-up design) or get the general idea of what the program needs to do and then handle more specific matters as he develops the application (top-down design). However, the control structures in FORTRAN were primitive and produced code that was hard to follow (thanks to GOTO's) along with and difficult to comprehend [Mac87].

COBOL

Another attempt in the late 1950's was COBOL. This language's main goal was to produce code that resembled the English language in a natural-seeming form. COBOL was meant to be a self-documenting language, code that made it clear on its own what was being done in the program and thus easy to understand. However, the syntax was very verbose which made it hard to program in the first place, especially due to certain divisions of the code.

An example of the verbosity of COBOL's "natural language" is a program that reads customer records and computes average sales, that is sales per call [Bel89]. Several pieces stand out. First, about ten lines of code spell out how the input file is formatted. The file itself is a simple chart that has the customer information and total sales and number of sales calls. More heinous though is the output file, which is another simple chart. This chart summarizes the customer information and has the average sales per call listed. It looks something like:

CUSTOMER LIST

CUSTOMER	CUST. NAME	ADDRESS	CITY	STATE	ZIP	AVG. SALES/CALL
00001	John Doe	1 Here St.	Houston	TX	77333	0931.11
00002	Mary Suarez	2 2nd Ave.	New York	NY	01931	4943.32

However, even this easy output chart takes over thirty lines of code to describe. Here is just a small snippet from the code that describes the format of the output chart:

```
01 DETAIL-LINE.
        05 FILLER                 PIC X(2)      VALUE SPACES.
        05 CUSTOMER-NUMBER   PIC 9(5)      VALUE ZEROS.
        05 FILLER                 PIC X(3)      VALUESPACES.
        05 CUSTOMER-NAME      PIC X(30)     VALUE ZEROS.
        [some more code … snipped for space]
        05 CUSTOMER-SALES     PIC 9999.99   VALUE ZEROS.
        05 FILLER                 PIC X(11)     VALUE SPACES.
```

[Bel89].

This code describes the formatting for each row of the table and must be specified correctly in order for the chart to properly appear. Also from this same example, the code to populate a line of the table looks like:

```
FORMAT-DETAIL-LINE-MODULE
        MOVE I-CUSTOMER-NUMBER TO CUSTOMER-NUMBER.
        MOVE I-CUSTOMER-NAME TO CUSTOMER-NAME.
        [same for rest of fields of table]

        DIVIDE I-TOTAL-SALES BY I-SALES-CALLS
                GIVING AVERAGE-SALES.
        MOVE AVERAGE-SALES TO CUSTOMER-SALES.

        MOVE DETAIL-LINE TO 0-PRINTER-RECORD.
        MOVE 1 TO LINE-SPACING.
        PREFORM PRINTING-MODULE.
```

[Bel89].

Again, the syntax is very verbose and must be written exactly for the program to work. In other languages, such as C or Pascal, this routine would be much clearer by the use of mathematical notation and assignment statements (In C, the division would be done by a statement like “average_sales = i_total_sales / i_sales_calls”).

From the customer sales example, one can see that COBOL’s “natural language” was really hard to understand and use instead of the program being self-documenting and easy to comprehend. Thus, even though the ideals established for COBOL were in the direction of making code easier to understand, the realization of that goal fell well short of its expectations.

Procedural Programming

Over the next twenty to thirty years, other languages appeared which made code easier to understand and work with. Algol provided more intuitive control structures such as the FOR and WHILE loops and the IF-ELSE statement, eliminating the need for the GOTO’s which would have made code harder to follow. Pascal and Ada added

facilities to declare abstract data types (ADT's) [Mac87], adding yet another way to make applications easier to design and understand. For example, in Pascal a programmer can declare new data types using the "type" keyword:

```
{Example taken from [Mac87] and modified with comments}


type
        { a person type }
        person =
                record
                   name: string;
                   age: 16..100;  {age can be from 16 to 100}
                   salary: 10000..100000;   {salary in dollars}
                   sex: (male, female);  {an employee is either male or female}
                   birthdate: date;  {date is also an ADT}
                   hiredate: date;
                end;

      string = packed array [1..30] of char;

      { a date type }
      date = record
            mon: month;  {month is an ADT}
            day: 1..31;
            year: 1900..2000;
           end;

     {months of the year}
      month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
```

The programmer can declare variables of these types and give them values they understand like this:

```
var
   newhire: person;  {newhire is a person}

begin
   {new person is a female of age 25 and was hired on June 1, 2000}
   {which is easy to see from the code below}
   newhire.age = 25;
   newhire.sex = female;
   newhire.hiredate.mon = Jun;
   newhire.hiredate.day = 1;
   newhire.hiredate.day = 2000;
end.
```

These languages are classified as “procedural”, which encourages the use of procedures and functions to implement common tasks which must be done repeatedly in the code. Procedural programming also encourages both top-down and bottom-up design. Like with COBOL, the aim was clear, self-documenting code. However, with more complex problems, the ability to comprehend the code declined sharply. Numerous procedure or function calls could make the code resemble FORTRAN to an extent as control would jump all over the place:

```
{this is “pseudocode”, code that doesn’t necessary follow the syntax of any }
{programming language, but is used to convey ideas to the reader            }

program jumps;
{A, B, C, and D are procedures}
procedure A;
begin
    if (some condition is true) then
            C;  {procedure C is called}
    else if (something else is true) then
            B;
    else if (yet another thing is true) then
            D;
    else
            writeln(“We’re done!”);
end;

procedure B;
begin
     if (proposition is true) then
             D;
    else
             C;
end;


procedure C;
begin
    if (case is true) then
            set values such no conditions in procedure A will be satisfied;
            A;
   else
            D;
end;
```

```
procedure D;
begin
        if (some situation did occur) then
            do some processing;
            B;   {things may have changed … so let's call B again}
end;
{ main program }
begin
        A;
end.
```

This program is hard to follow as the reader must constantly jump from one procedure's code to another in order to try to understand it. Following sequentially in a piece of program code is usually clearer than hopping from place to place. Complex code might actually contain situations like this and could hinder the person's understanding of any part of the application which tries to use this code or he otherwise must work with it.

Object-Oriented Programming

Another attempt which is still in progress for understandable code is the object-oriented paradigm. Object-oriented programming gives a more concrete way of thinking about entities and their relationships. Unlike the procedural model where objects are just passive data, objects act on each other through methods. Object-oriented programming explicitly provides more powerful facilities for understanding such as polymorphism, inheritance, and abstract classes [Arn96]. Adherents to the object-oriented paradigm may also see other relationships between classes such as aggregation which also aid in the design and programming process. It is the power of these facilities that attempts not just to integrate software development process into a single object model that evolves with the project. But again, just like with any of these other paradigms, complex projects are naturally hard to understand and document, especially those with many classes or complex inheritance hierarchies.

Style Conventions

With all of the approaches described so far came the idea of style conventions. These rules guide the programmer into writing code where others who read the code should be able to understand it more. For example, commenting complex pieces of code gives developers and maintainers a good clue to what the actual code does and how it does it. Also, giving variables meaningful names tells a person what a piece of data represents and helps him or her make sense out of the calculations that are done and appropriately coined functions that do what their name implies. Other conventions such as indenting and spacing also aid the understanding process by laying out the code in visual chunks [Dow93]. Unfortunately, many programmers just want to get their application to work, not giving a care about whether anyone will be able to understand the code at all. Others might just think making their code readable is not worth the extra effort. Even those that follow these rules may follow them poorly or at best be inconsistent at their efforts. Clearly, another approach is needed in conjunction with the

conventional approaches discussed to make code clearer so that it may be understood by programmers, maintainers, and even users.

Literate Programming 101

Literate programming solves many of the problems that occur by using conventional means without taking away from the ideals of the ordinary approaches. But first, a crash course in literate programming will give the necessary background to understand the advantages of this paradigm.

Definition and Rationale of Literate Programming

Literate programming is a paradigm of solving problems that combines documentation with program code in such a way that it is easy to read and understand **by human beings**, often with the aid of automated tools. Don Knuth, famous for his work in the field, sees programming as a work of art and by looking at programs as books, developers will better document their code and make it easy for themselves and others to understand. This approach encourages the programmer to use good practice and clear explanations to get ideas across to the reader [Knu92]. The paradigm provides several major features:

- **Integrated Feel for Design Strategies:** Ability to program effectively in either top-down or bottom-up fashion through the use of structured pseudocode [Chi95].
- **Divide and Conquer:** Code in small pieces where most snippets of code (including both actual program source and documentation) are less than a page long [Chi95].
- **Cognitive Reinforcement of Concepts:** Typeset documentation in a pretty-printed format where the constructs of the programming language are clearly displayed in good style [Chi95].
- **Think about Readability:** A table of contents and index are generated along with other reading aids [Chi95]. The typeset document is then effectively a hardcopy hypertext with pointers to related sections.
- **Design Alternatives:** The opportunity to discuss alternative solutions and make suggestions regarding maintenance and possible extensions [Chi92].
- **Problem Description and Solution:** Include all visual aids and mathematics to enhance communication of the problem and its understanding [Chi92].
- **Augment Programming Language:** Provide enhanced features for the underlying programming language.
- **Environments:** Provide a near ideal setting for literate programming through user-friendly environments. Examples include Emacs web-mode [Mot90] and Osterbye's Smalltalk browser [Ost95].

WEB Process for Literate Programming

Most of these features become evident by looking at how the literate programming process works. A popular model of literate programming is Don Knuth's WEB [Knu92]. Pieces of code are self-contained nodes that have relationships to each other and can be navigated through a web of such nodes. These nodes can have either documentation notes, macro definitions, and/or code. Using a special syntax, the

programmer creates such sections and links them together in a WEB file, an example for the "Hello, World" program, shown as Figure 1 in the Appendix. The WEB syntax is rather simple. Commands can be used to create sections of code or documentation such as the ones listed in Figure 2 of the Appendix. Code sections can have natural language descriptions (or tags, denoted by pointy brackets such as "< tag >" within the typeset documentation outputted for readers of the program code) of a part of the program, which refer to a section named uniquely with that tag, which explains in more detail that piece of code, also through documentation, macros, or program source. The sections of code may appear in any order in the WEB file. The programmer can also specify typesetting directives to guide the way the human readable documentation will appear. The documentation includes the code in a form called "pretty-print", a style that reflects the syntactic structure of the language with keywords highlighted, identifiers italicized, and spacing and indentation to clearly mark data, control, and other program structures. An example of the pretty-printed documentation in Figure 3 of the Appendix. This kind of typeset document is programmed using an underlying formatting language, such as TeX in the case of WEB. Two applications process the WEB file, one to prepare the code for the underlying programming language's compiler (Tangle, example output shown as Figure 4 in Appendix) and the other to generate the pretty-printed typeset document (Weave) [Knu92]. Many literate programming tools are based on the WEB model. Anyone who uses the WEB in any way is a "reader" of that WEB, whether he is a programmer, client, manager, student or user of the application that the WEB implements and documents.

Advantages of Literate Programming

Literate programming has many aspects that really shine. The paradigm has many general strengths attributed mostly because Weave produces a form of the code that is more readable and understandable for humans. Also, software engineers can find literate programming to be useful in design and communication with other parties and each other. Furthermore, students can learn essential programming and design skills by using this paradigm by orienting themselves to solve the problem, rather than orienting themselves on just learning how to code. Literate programming helps readers understand applications and how they are implemented in these ways.

General Strengths

The Typeset Document

Most of the general advantages of literate programming come from the Woven document (an example is shown in Figure 3 of the Appendix), in that the code in its typeset form is simple to understand. Each section of code is clearly explained in the documentation that accompanies it. The reader can digest small pieces of source and through the help of the various reading aids navigate the WEB to get a better understanding of the application. Pretty-printing in the form of indentation of code, boldfacing of keywords in the program, allowing the inclusion of captions, and other enhancements provided by the text formatter used also assists in the reader's scanning of the document by cognitively reinforcing the distinct concepts in the code.

Use of Natural Language

Another major strength of the WEB model is the extensive use of natural language. People tend to understand information best that is presented to them in plain English. The natural language description of pieces of code gives a clear overview of what the section does, how it is implemented in the accompanying code, or hints or information to the user of the program. Literate programming encourages the use of structured pseudocode as natural language through the use of tags within the code to describe what the piece of source does; recall that each tag is a pointer to another section that elaborate on the action specified in the tag, establishing a concrete relationship between two sections of code [Knu92]. A clear example of this aspect of literate programming involves programming design languages (PDL's). PDL's are essentially pseudocode that express clear ideas about what is being done using natural language [Bro90]. The example below shows how such pseudocode translates easily into an equivalent code snippet in the literate programming paradigm:

Pseudocode for Binary Search (fast algorithm to find an element in a sorted list):

```
DO WHILE first is less than last and not found
        find the middle element
        IF the middle element is the largest
                THEN BEGIN
                        found the item, return it
        ELSE
                IF target is larger than the middle item
                        THEN reset the lower limit
                        ELSE reset the upper limit
                ENDIF
        ENDIF
ENDDO
```

[Bro90].

Literate Equivalent for Binary Search:

```
begin
< Initialize binary search variables >
while < Not done and not found >
        do begin
                < Find the middle component >
                if < Middle component is target >
                        then < Found the item >
                        else
                                if < Target is larger than middle >
                                        then < Reset lower bound >
                                        else < Reset upper bound >
        end
end
```

[Bro90].

As you can see, the translation from pseudocode to literate programming code is fairly straightforward. Thus, literate programs can be treated as a kind of pseudocode that is written in easy to understand natural language. Effectively, literate programming gives a simple way of programming and documenting mostly through the use of natural language.

Flexible for Presentation

Also, WEB files can be used to generate documentation for various audiences or purposes. All a presenter needs to do is first Weave a template typeset document from the WEB file. Then, he just edits this template as appropriate for his situation:

- A professor at a University might use the Woven template to put together a journal article that has all of the pertinent code, explanations, and graphics he wishes to present.
- A software project team could maintain a truly **living** design document throughout the software life cycle, making necessary changes throughout the process. The project group might use multiple copies of the template as a new version is created. User documentation can come from one copy of the template and project artifacts such as requirements, code, or test data can derive from another.
- A computer science teacher can explain key concepts through using literate programs as handouts. The students would see these teaching aids as like lecture nodes, appearing as code with explanations accompanying each piece of program that the teacher thinks is relevant or sections that explain general concepts and contain no code at all.

Literate Programming Can Be Fun

Don Knuth states one other rather important general advantage of literate programming: it can be fun [Knu92]! An implementor gets to build a WEB, which is like a work of art. He sees the program develop right before his eyes. Literate programming is a rare breed that can both be fun and useful!

Software Engineering Advantages

Besides its general advantages, software developers find literate programming to be very useful. Literate programming supports and encourages incremental design without forcing the developers to resort to a particular methodology. Also, the software team can clearly communicate clearly with various readers of the literate program. Software engineers can take full advantage of the plethora of language-independent tools available online, thus freeing the team to use whatever programming language is best for the project and whatever text formatter will make the documentation as clear and understandable as possible.

Design Process

The software engineer might develop a program by creating a WEB file with a problem statement, add requirements at a later time, code at a later point, and maybe later testing information as is done incrementally in the lawn service example in [Dun95]. Also within the coding process, literate programming supports both top-down and

bottom-up design through the allowing any ordering of sections of code in the WEB file [Knu92]. Both of these styles of design have their uses as in the conventional approaches to programming as they help ease the design process into smaller pieces, ultimately resulting in the realization of the entire application. Literate programming also encourages work upstream in the software life cycle. Developers can start off with a WEB consisting of a problem statement and its constituent parts. They can do the requirements and high level analysis in there and then let the project grow in this WEB right from project inception. This process would result in a better understanding of what the application and its problem domain. As the software life cycle continues, a better understanding of the project means a smoother ride downstream in the form of fewer feature changes, more predictable testing, and simpler debugging work.

Communication with Others

The use of literate programming gives the parties involved in the software process a better understanding of the application and its innards. The literate programming paradigm encourages communication through clear and understandable documentation accompanying the code which may contain natural language pseudocode in the form of tags referring to other sections. Two main groups of readers can benefit from the WEB for the project: those internal to the project and the clients. Internally, developers can explain what the code is supposed to do using literate programming so others who work with the code should be able to understand it:

- **Other programmers:** They need to see what particular pieces of code do, what variables, types, or functions they need to use, or get a general feel for the project if they are unfamiliar with the code.
- **Testers:** Given clear-cut cases and key variables by the literate program, these workers can develop more effective test data and try to figure out more ways to break the code. Thus, literate programming will help the testers uncover more hard-to-find bugs in the code.
- **Maintenance Technicians:** These are those people that may have to look at the code years down the road. They will more likely never have looked at the system before, especially if they are maintaining a variety of applications. Also, the system may have been handed to them by the developers and thus have to get a thorough idea of how this unfamiliar system works and is built. The clear explanations and code in the WEB will tell the maintenance technicians what they need to know about the software.

Literate programming also benefits the customers of the software project. The developers can draft the documentation using the Woven output from the WEB file. They can add documentation-only sections that contain instructions, explanations, and tutorials that include text, charts, and graphics. The customer can then ask questions or make clarifications during the development process based off of this clear documentation from the developer. Changes resulting either from customer feedback or negative results from usability testing may cause changes to need to be made to the documentation which can be directly done in the WEB file. Literate programming reinforces and encourages the need for communication between customer and developer through the Woven documentation.

Language Independence

In the early days of literate programming, tools usually assumed a certain underlying programming language and a specific text formatting language, such as Knuth's WEB [Knu92]. This limitation severely limited any software project that wanted to use literate programming as it restricted their choice of programming language to one that might not be the most appropriate for their application or leave the documentation in an undesirable format. However, as literate programming evolved, so did the tools, and researchers realized the need for language-independent tools. Soon came some tools that supported a group of languages such as FWEB (could handle C/C++, Java, and Fortran, but only supported TeX typesetting) and Noweb (only could typeset in HTML, Tex, or LaTeX, but was programming language independent) [Tho97]. But now there are tools out there that are truly language independent such as VAMP [Van92], AOPS [Shu93], FunnelWeb [Lee94], and SPIDER [Ram95]. The use of a language-independent tool now can free up software projects from any restrictions caused by literate programming while allowing them to fully reap the paradigm's benefits.

Literate Programming in the Classroom

CS/1 Experiment

In the conventional introductory computer science course, a student learns how to program, but not how to actually solve programming problems. Most of the problems experienced by novices are a result of not understanding how to put the pieces together for particular programs [Chi95]. At the introductory level, students are only interested in getting code to work, not how to think out the problem they are trying to solve or effective ways of documenting their programs. For them, the finished product is all that matters [Chi95]. Studies show that an effective teacher presents concepts on how to solve problems [Chi95]. Students need to be shown that the important lesson to be learned in programming is design. Bart Childs performed an experiment with beginning students, exposing them to literate programming and thus emphasizing the problem solving skills rather than the program itself, though they were responsible for learning the syntax and constructs of Pascal [Chi95]. The students were taught iterative design techniques and were required to turn in labs twice to make sure they were understanding the problem solving process and then able to convert it into code [Chi95] such as in the lawn service example from [Dun95].

The results from this experiment confirmed that literate programming aids beginning students in learning how to solve problems and ultimately to program better. Most importantly, students' problem solving skills increased. Those who were unfamiliar with Pascal performed better than those were already familiar with the language because they were able to "[use] the literate programming paradigm to capture and document their problem solving process" [Chi95]. Also, students were able to learn the three languages necessary to program in WEB, WEB itself, Pascal, and LaTeX [Chi95]. The students exposed to literate programming performed significantly better in their subsequent data structures course than those who didn't get taught using the literate programming paradigm [Chi95]. Childs also hypothesized that this paradigm might improve the software process [Chi95]. Overall, the students were able to understand the key concepts of problem solving and documenting their code through the design process and using them for their own benefit.

Problem-Solving vs. Program-Oriented Approach

Literate programming emphasizes a problem-solving approach from the start, rather than a program-oriented one [Chi95]. The real emphasis on developing software is understanding the problem and designing a solution for it, not the coding itself. Starting with a top-down approach to get a high-level understanding of the problem and then using bottom-up to fill in the details where specific details may be needed to build bigger pieces is how this thought process works in many cases. Emphasizing the importance of the design process early on in a programmer's education would take him a long way. The student would spend more time thinking about the problem and how to solve it, then implement it, as is actually done in software projects. Also, by getting a core understanding of the design process early on, the student would learn and master the key concepts to be a good programmer much sooner than on who does not have experience with literate programming. A beginning student who only learns how to program, but not how to solve problems will gradually pick up the problem-solving approach, but probably much later than the student who learned how to solve problems using literate programming. The software development process stresses analyzing and understanding the problem and how to progressively come up with a solution in the form of a well-documented program. This new breed of programmers would view the whole software life cycle as a single unit and the living entity, the application being developed, grows through the WEB and the people that develop it.

Furthermore, experience with the literate programming can help people approach problems in whatever their fields may be. Literate programming encourages the developer to get an understanding of the problem, get deeper into the design, and then iteratively code the application and document his path to that solution for both him and readers of the WEB to understand. With each problem a person faces in his field, he has to be able to first understand what needs to be done. Next, he has to develop a blueprint for how to solve the problem, similar to the design phase of the software engineering process. Finally, he has to implement his solution in a way that people **themselves** will understand how to use and maintain where applicable, including appropriate documentation where necessary. A course emphasizing literate programming would thus also be a great course to teach problem solving techniques that anyone can use later in their field of study.

Disadvantages of Literate Programming

Just as with any other paradigm, literate programming does have its shortcomings. For small programs (one or two pages worth of code) or programs that may be used once and then thrown away, using literate programming may be overkill. Having to write a WEB file complete with textual descriptions for the documentation and really short sections for pieces of code might be too much of a hassle and take longer to get right than just writing the application in a conventional manner. However, even for small applications, the programmer and other readers may have to understand and work with the code at some point in the future. Thus, the work put into the documentation and chunking of code into sections will pay off. Another problem with literate programming is that the developer has to learn three languages in order to use the paradigm: the underlying programming language, and the underlying typesetting language, and the

WEB syntax that ties them together. But with even beginning students in general having no trouble learning the three languages needed [Chi95], experienced programmers should be able to pick up the literate programming paradigm easily. Also, WEB ties together the features of the two languages and blends them together into "a combination of languages [that] proves to be much more than either single language by itself. WEB does not make the other languages obsolete; on the contrary it enhances them" [Knu92]. One other disadvantage of literate programming is merely a fallacy: "it is difficult to find software that is truly independent of programming language and typesetting language and that is also easy to understand". This statement is just a misconception as there are language independent tools out there. Most literate programmers are well-versed enough anyway and should be able to learn these tools, especially those like Spider that allow the user to customize how the WEB file is Woven and Tangled based on either underlying language or other special situations [Ram95]. The major hurdle for literate programming right now is that the paradigm is not a well-known concept. Fewer papers have been written than there should be on the subject and most interest currently still seems to be restricted to the academic community. This lack of publicity may be occurring because few organizations currently use literate programming and/or it just does not generate the hype that object-oriented programming or the World Wide Web do. However, given time and word of mouth, enough people can learn for themselves how powerful and fun literate programming is.

Conclusion – The Future of Literate Programming?

With some more exposure to literate programming, developers, programmers, students, and application users can take advantage of this powerful and intuitive paradigm that continues to evolve.

Modifications to Existing Paradigm

Where might literate programming head? David Cordes and Marcus Brown propose to modify the existing paradigm to make literate programming a more practical methodology. One of their proposals is to provide a multilevel table of contents to allow the programmer to layout his structure of sections more naturally and clearly instead of the current scheme which numbers all sections as top-level [Cor91]. The reader of the WEB can also see this hierarchical display of the sections and thus understand more about the code. Another of their proposals is to add a GUI interface to help the user select WEB commands more easily (new users would not even have to learn the commands!) and traversing relationships between entities in the WEB [Cor91]. A debugger would also help developers in writing and maintaining the code [Cor91]. An enhanced index would give the WEB reader clear information about the use of variables in the code [Cor91]. Besides these enhancements, Cordes and Brown also proposed some restrictions to the current paradigm to make code easier to understand and write such as limiting the structure of the code within a section and reducing the number of WEB commands [Cor91]. These features would help literate programming on its way to becoming more of a practical paradigm and as a result may become more popular.

True Natural Language Systems

Literate programming would hit its peak when natural language processing (NLP) makes some major breakthroughs, both in understanding (NLU) and generation (NLG). Then, programmers and readers of the typeset documentation will be able to clearly understand the code and what it should do. All would be in plain English and the code itself would truly be self-documenting though documentation for most sections would still probably be recommended.

Natural Language Understanding

On the NLU front, a system might be feasible within a matter of years that takes natural language in the form of high level instructions (maybe just a description of what the program does would be enough in the far future), parses it, and generates code in a conventional language that may be compiled and executed as well as the typeset documentation. Recent progress has been made on the LogiMOO and NALIGE projects. LogiMOO is a virtual world where the users may move around and perform actions using a form of restricted natural language that is then translated into a dialect of Prolog to execute [Tar99]. NALIGE is a natural language interface to the underlying operating system. At the prompt, the user enters a natural language command that which NALIGE translates into code that the operating system executes [Man94]. These projects attempts are the first steps to realizing a true natural programming language to enhance the literate programming paradigm.

Natural Language Generation

Work is proceeding on an intelligent system called "dynamic hypertext" that uses NLG to produce online documents. The system bases the pages it builds on schema it gathers from user responses and tendencies [Dal98]. Literate programming can take advantage of this approach in a similar system using a technique that may or may not resemble WEB. The literate program file would contain source code and a baseline documentation with some information about what the expected user might be like. The source code is Tangled as is done presently. For the "Weaving" process, the baseline documentation and the expected user information then get sent to an NLG system which constructs and stores initial schema that is used to build the initial documentation in an online hypertext format. Then as the user navigates each page, the NLG system updates its schema based on his responses and tendencies that it picks up. Finally, the NLG system uses these schema to build the next page. The NLG system's goal would be to display the literate program in a clear fashion that is geared towards the characteristics of the user and aid him in the understanding and problem solving process. Until natural language work reaches this point however, literate programming should serve well as-is to describe the work being done in the program in a natural and clear way for all readers.

Closing Statement

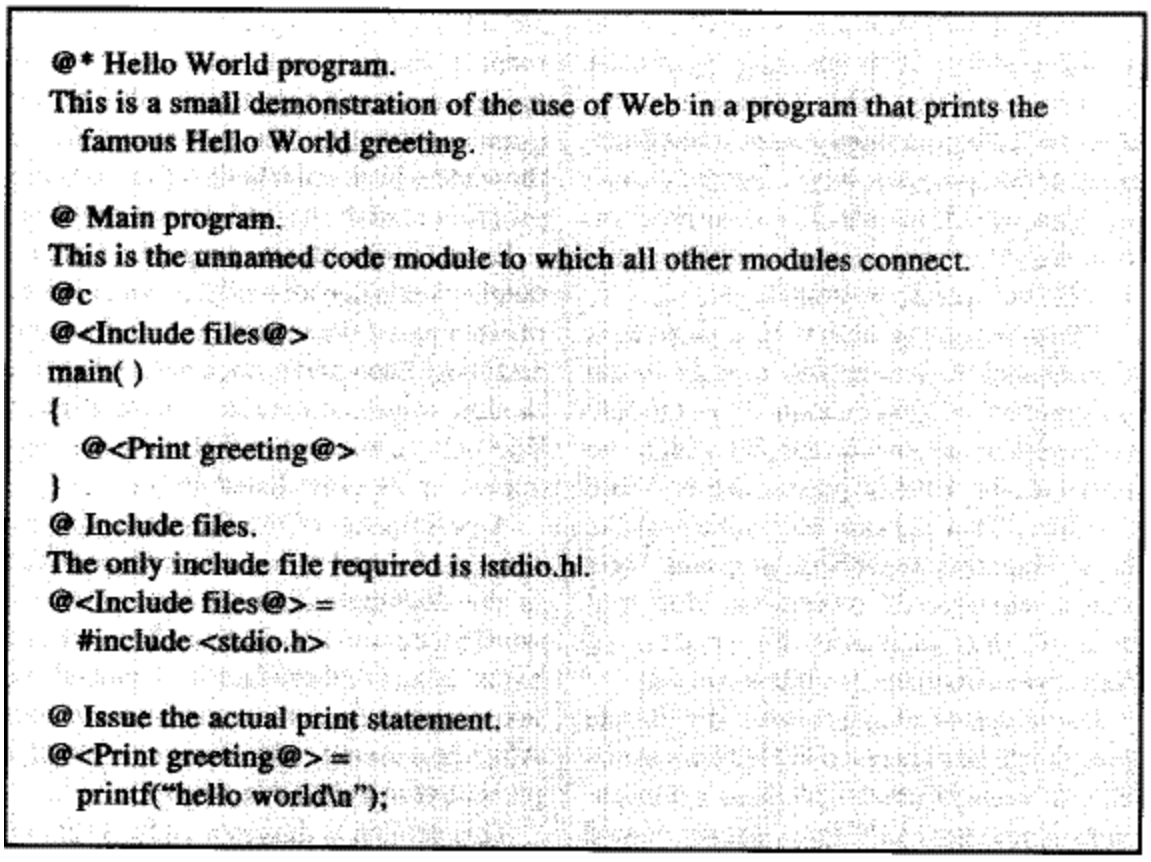
How ever the paradigm of literate programming evolves, its goal will remain the same: give everyone a version of the code that is well-documented, laid out, and presented, so they may work with it and most importantly, **understand** it!

Appendix

"Raw" Source Code for "Hello, World" Program

```
#include <stdio.h>
main()
{
  printf("hello world\n");
}
```

Figure 1: WEB file text for "Hello, World"



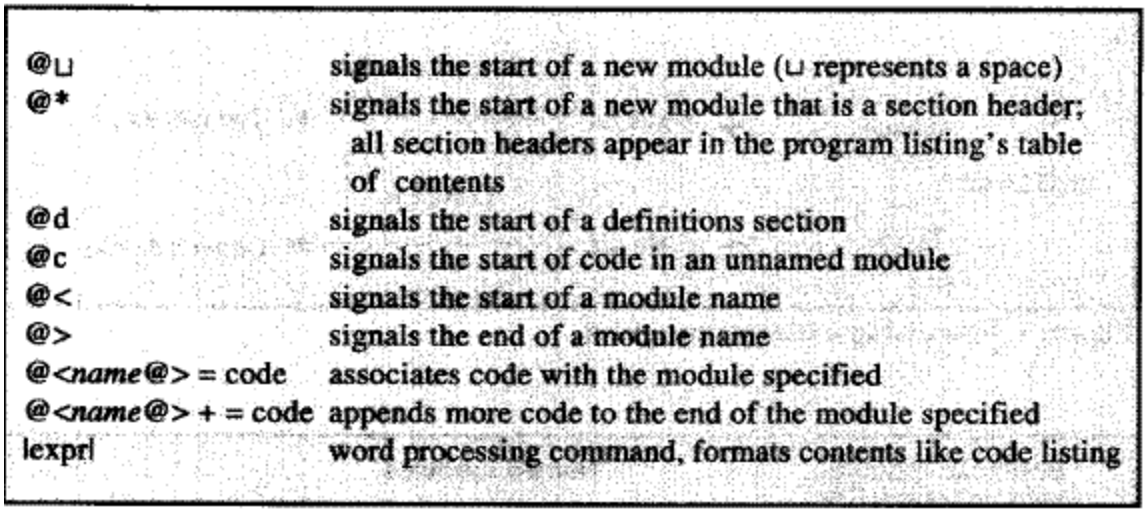
```
@* Hello World program.
This is a small demonstration of the use of Web in a program that prints the
  famous Hello World greeting.

@ Main program.
This is the unnamed code module to which all other modules connect.
@c
@<Include files@>
main( )
{
  @<Print greeting@>
}
@ Include files.
The only include file required is |stdio.h|.
@<Include files@> =
  #include <stdio.h>

@ Issue the actual print statement.
@<Print greeting@> =
  printf("hello world\n");
```

Source: Cordes, David, and Marcus Brown. "The Literate-Programming Paradigm." Computer 24.6 (1991): 54.

Figure 2: Basic WEB Command Set



@␣	signals the start of a new module (␣ represents a space)
@*	signals the start of a new module that is a section header; all section headers appear in the program listing's table of contents
@d	signals the start of a definitions section
@c	signals the start of code in an unnamed module
@<	signals the start of a module name
@>	signals the end of a module name
@<*name*@> = code	associates code with the module specified
@<*name*@> + = code	appends more code to the end of the module specified
|expr|	word processing command, formats contents like code listing

Source: Cordes, David, and Marcus Brown. "The Literate-Programming Paradigm." Computer 24.6 (1991): 54.

Figure 3: Woven document for "Hello, World"

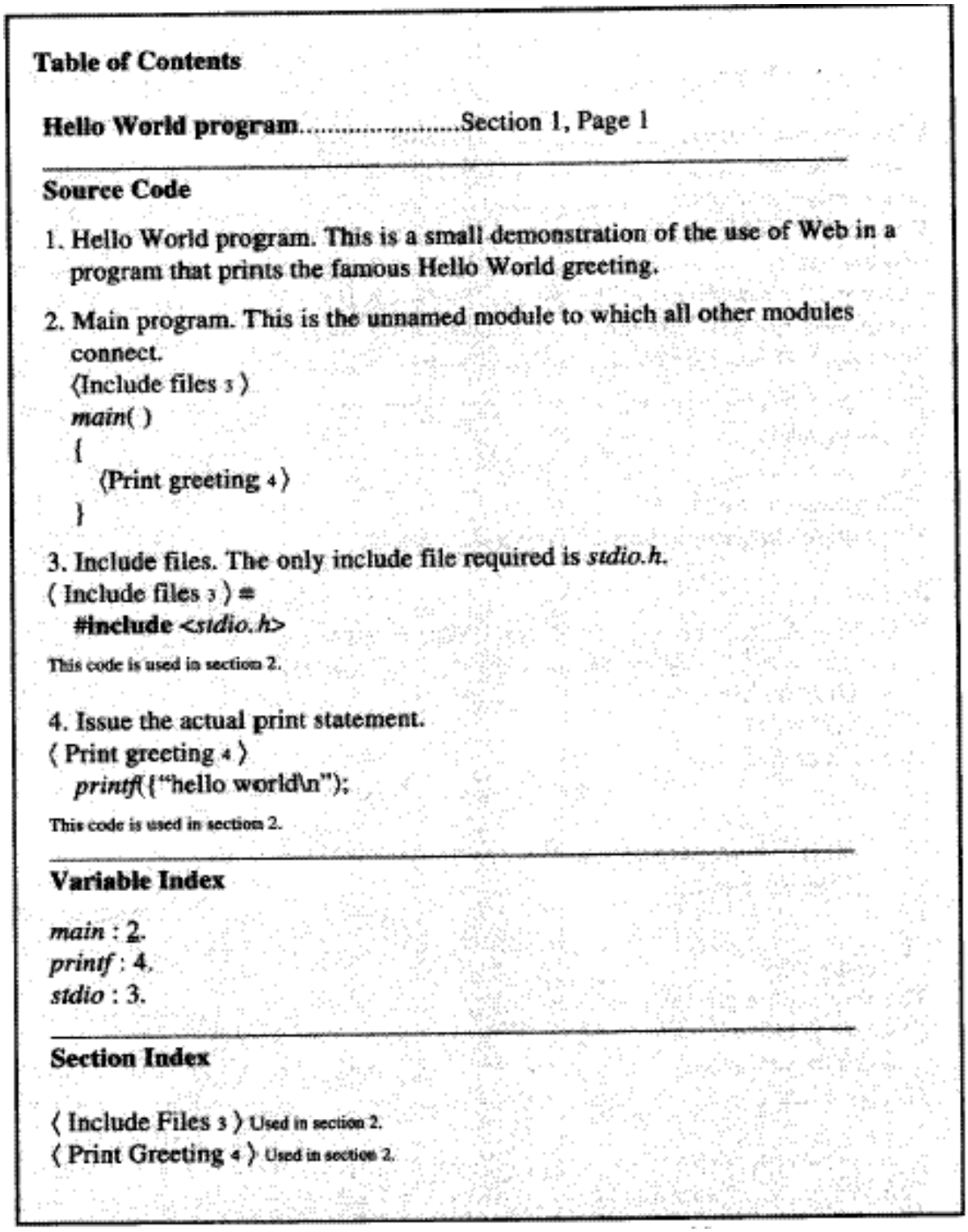


Table of Contents

Hello World program.....Section 1, Page 1

Source Code

1. Hello World program. This is a small demonstration of the use of Web in a program that prints the famous Hello World greeting.

2. Main program. This is the unnamed module to which all other modules connect.

```
⟨Include files 3⟩
main( )
{
  ⟨Print greeting 4⟩
}
```

3. Include files. The only include file required is *stdio.h*.

```
⟨ Include files 3 ⟩ ≡
  #include <stdio.h>
```

This code is used in section 2.

4. Issue the actual print statement.

```
⟨ Print greeting 4 ⟩
  printf("hello world\n");
```

This code is used in section 2.

Variable Index

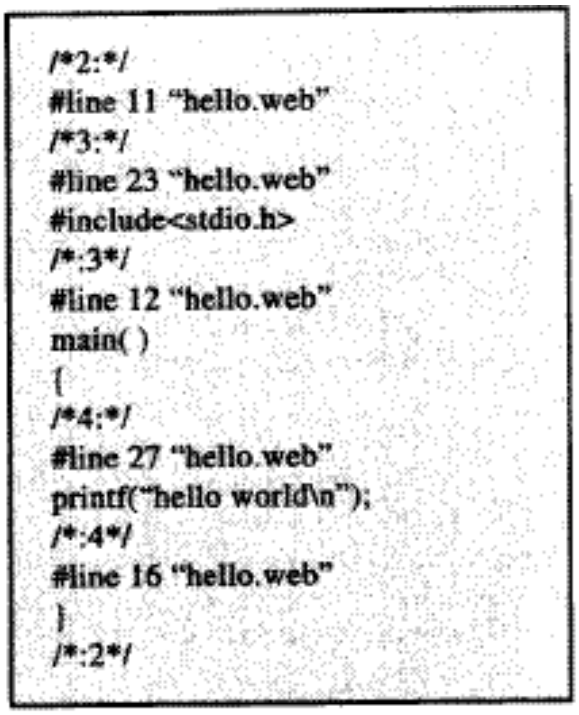
main : 2.
printf : 4.
stdio : 3.

Section Index

⟨ Include Files 3 ⟩ Used in section 2.
⟨ Print Greeting 4 ⟩ Used in section 2.

Source: Cordes, David, and Marcus Brown. "The Literate-Programming Paradigm." Computer 24.6 (1991): 55.

Figure 4: C Program Generated from Tangle for "Hello, World"



```
/*2:*/
#line 11 "hello.web"
/*3:*/
#line 23 "hello.web"
#include<stdio.h>
/*:3*/
#line 12 "hello.web"
main( )
{
/*4:*/
#line 27 "hello.web"
printf("hello world\n");
/*:4*/
#line 16 "hello.web"
}
/*:2*/
```

Source: Cordes, David, and Marcus Brown. "The Literate-Programming Paradigm." Computer 24.6 (1991): 55.

References

[Arn96] Arnold, Ken, and James Gosling. The Java Programming Language. Reading, MA: Addison-Wesley, 1996.

[Bel89] Belcher, Linda K. The COBOL Handbook: A Modular Approach. Belmont, CA: Wadsworth, 1989.

[Bro90] Brown, Marcus, and David Cordes. "A Literate Programming Design Language." Proceedings, CompEuro (1990): 548-549.

[Chi92] Childs, Bart. "Literate Programming, A Practitioner's View." TUGboat 13.3 (1992): 261-268.

[Chi95] Childs, Bart, Deborah Dunn, and William Lively. "Teaching CS/1 Courses in a Literate Manner." Proceedings. TUGboat 16.3 (1995): 300-309.

[Cor91] Cordes, David, and Marcus Brown. "The Literate-Programming Paradigm." Computer 24.6 (1991): 52-61.

[Dal98] Dale, Robert, et. al. "Integrating Natural Language Generation and Hypertext to Produce Dynamic Documents." Interacting with Computers 11 (1998): 109-135.

[Dow93] Dowd, Kevin. High Performance Computing. Sebastopol, CA: O'Reilly & Associates, 1993.

[Dun95] Dunn, Deborah Lynn. "Literate Programming as a Mechanism for Improving Problem Solving Skills." Diss. Texas A&M University, 1995.

[Knu92] Knuth, Donald E. Literate Programming. CSLI, 1992.

[Lee94] Lee, Christopher. "Literate Programming – Propaganda and Tools." Web page. http://www.cs.cmu.edu/~vaschelp/Programming/Literate/literate.html. October 17, 1994.

[Mac87] MacLennan, Bruce J. Principles of Programming Languages. 2nd ed. New York: Oxford UP, 1987.

[Man94] Manaris, Bill Z., Robert Glanville, and Timothy E. Gillis. "Developing Natural Language Interfaces through NALIGE." Proceedings. Sixth International Conference on Tools with Artificial Intelligence (1994): 260-266.

[Mot90] Motl, Mark Bentley. "A Literate Programming Environment Based on an Extensible Editor." Texas A&M University, 1990.

[Ost95] Osterbye, Kasper. "Literate Smalltalk Programming Using Hypertext." IEEE Transactions on Software Engineering 21.2 (1995): 138-145.

[Ram95] Ramsey, Norman. "Weaving a Language Independent WEB." 1995 version of paper originally published in Communications of the Association for Computing Machinery 32 (1989): 1051-1055.

[Shu93] Shum, Stephen, and Curtis Cook. "AOPS: An Abstraction-Oriented Programming System for Literate Programming." Software Engineering Journal 8.3 (1993): 113-120.

[Tar99] Tarau, Paul. "LogiMOO: An Extensible Multi-User Virtual World with Natural Language Control." The Journal of Logic Programming 38 (1999): 331-353.

[Tho97] Thompson, David B.. "The Literate Programming FAQ." Newsgroup posting: comp.programming.literate. August 15, 1997.

[Van92] Van Ammers, Eric W., and Mark R. Kramer. "VAMP: A Tool for Literate Programming Independent of Programming Language and Formatter." Proceedings. CompEuro (1992): 371-376.