The Literate-Programming Paradigm

David Cordes and Marcus Brown
University of Alabama

Literate programming presents a tool for constructing readable, understandable, and maintainable code. We review this paradigm, analyze its strengths and weaknesses, and present some enhancements.

The ability to comprehend a program written by other individuals is becoming increasingly important in software development. Given that the general cost of program maintenance may reach 60 percent of the total software costs associated with a certain product,[1] a real demand exists for systems that can present the program text in a readable, understandable format. Comprehension of another individual's program should be made as simple as possible. Therefore, methods that promote program comprehension should be encouraged.

Donald Knuth introduced the concept of literate programming as a method to improve the quality of computer software.[2] One of the chief goals of literate programming is to present a technique for coding software systems that promotes readability and comprehension. Realizing that these programs must be read and maintained by others, we draw an analogy to the writing of a book or an essay. Since the programmer is essentially providing a work of literature to be read by others, this technique for program coding has become known as literate programming.

In this article, we examine this paradigm in detail, first reviewing the current literate-programming paradigm with two sample literate programs, next presenting a critique of literate programming as it is currently used, and then exploring methods for enhancing the process. We propose a number of new facilities and present restrictions on current literate-programming practices.

The literate paradigm

Computer scientists have always targeted computer programs for specific audiences, employing compilers that will accept specific code and convert it to sets of instructions that can be carried out on hardware. However, these programs have been written by humans, tested and debugged by humans, and maintained by humans. If humans thought and acted like computer hardware, this scenario would pose no problems; programmers and hardware, operating in a similar manner, could take a programming language designed for either of them and function equally well.

However, that is not the case. The organizational and structuring techniques that express actions to a piece of hardware in a programming language are not necessarily the same techniques used when expressing these ideas to a group of programmers. We find that today's programming languages target one of these two groups, the hardware, at the expense of the other, the programmers.

This is not to say that current programming languages are inappropriate for the programming community. Structured programming constructs, high-level lan-

0018-9162/91/0600-0052$01.00 © 1991 IEEE

guages, and proper indentation style can significantly improve the readability and comprehension of a program. However, the fact remains that programmers primarily construct given programs for compilation and give little regard to organization and management of the thought processes that represent the logic of the programs. They generally keep such information in auxiliary documents, if it is generated at all.

The literate paradigm recognizes that different audiences — other programmers and a compiler — will receive this program. To address these audiences, the program must be processed in two different paths, as Figure 1 shows. For the human audience, the source code is processed by Weave, producing a document for typesetting. For the computer, the source is processed by Tangle, producing a program in the underlying source code (for example, Pascal, C, or Fortran) suitable for the compiler in a format that is basically unreadable by a human.

A typical literate source program is normally referred to as a web. The file name that contains the program usually includes .web as its suffix. The use of the term web illustrates the complex structures within the underlying program. This web can either be woven into a document or tangled into a machine-readable object file.

Figure 1 illustrates one very important point regarding the literate paradigm. When you program in a literate environment, you write in multiple languages at the same time. First, you enter statements in a conventional programming language; these are the statements that will be compiled. Second, you generate a description of the program using a word processing language. This text will explain your actions to others who read the program. Finally, a set of commands is required to link the programming language statements and word processing statements into a coherent document.

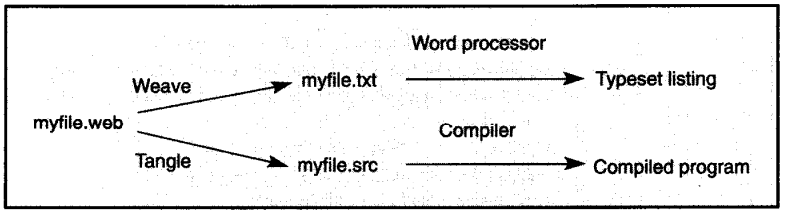

Figure 1. Processing a literate program.

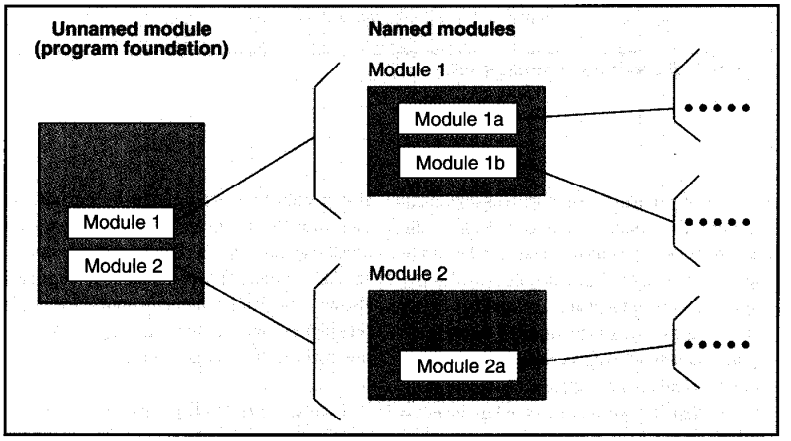

Figure 2. Program composition via modules.

The Web literate-programming language. Knuth developed the Web system as a working literate-programming language. The system has been used for small and large programs.[3] For example, the Tex typesetting system is written in the Web programming language. In each case, the primary goal has been to write the program for reader comprehension. Originally created for the Pascal language, Web has since been adapted to C,[4] Fortran,[5] Ada, and other programming languages.[6]

As with any other literate language, the Web source file consists of statements in three distinct languages: a programming language, a word processing language, and literate (in this case, Web) commands. The literate command set is used to organize and manage the code and documentation associated with a literate program.

The literate program is composed of a set of modules, each generally identified by a unique name. This name is entirely optional, although supplying most modules in a literate program with a name is common practice. A module in a literate program is designed to convey one thought or idea within the overall program logic.

While assigning names to a module is recommended, at least one module in the program doesn't have a name. This module functions as an anchor around which the rest of the modules are positioned, as Figure 2 shows. This permits abstraction of the basic thoughts regarding program logic into a sequence of lower level ideas and actions, thus capturing the process of stepwise refinement for the algorithm.

Using these modules allows the literate program to be expressed in a way not available to conventional structured programming languages. As Figure 2 shows, named modules are pulled into the program as demanded by the root (unnamed) module. However, their place in the actual text of the Web program is completely arbitrary. The program author can place these modules as deemed appropriate.

This ordering is expected to convey the activities of the program in an organized manner. Freed from constraints regarding management and organization during program construction, the author can present these ideas in an order believed to maximize program readability and comprehension.

Examining a literate program in greater detail reveals that all modules contain three distinct parts: documentation, definitions, and code. Each part is optional, and a given module might contain any or all of them. The only restriction is that the parts must appear in the order listed above.

A description of the functions and services performed by a given module is kept in the documentation section, written in word processor-like fashion. The author is expected to use these facilities, embedding text-processing commands into the text. After processing with Weave, the text is presented in a readable format.

This section is designed to be written in parallel with the code contained in the

@␣	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
lexpr	word processing command, formats contents like code listing

Figure 3. Basic Web command set.

module. By identifying a location for documentation to be placed in the module, the designers of the literate paradigm hoped to increase the use of documentation by the application programmers. Instead of being appended as extra information to the program, documentation is thus viewed as an integral part of the program. Its omission, rather than its inclusion, is what is most noticeable.

The definitions section supports macro definitions that can be either simple macros that define user-oriented mnemonics for standard programming language conventions or single-parameter macros.

With one exception, the code section simply contains programming language statements in the language being used to generate the program. The code may include (at any point) the name of another module within the program. As Figure 2 shows, the code from any named module referenced within the current module is included in the current module's code.

Two sample Web programs. Next, we develop two sample programs using the Web programming language. The first program is a reproduction of the famous Hello World program seen in textbooks. The second program computes the first 25 Fibonacci numbers.

We will develop these programs using the literate-programming language CWeb.[4]

```
@* 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.hl.
@<Include files@> =
  #include <stdio.h>

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

Figure 4. Hello World program text.

Web files generated using CWeb rely on Tex for word processing and the C programming language to generate the actual code. For each of these programs, we present the initial Web source code, the woven typeset document, and the tangled programming language code. The Web language is the literate language used. Figure 3 contains a listing of the Web commands used to compose these programs and should serve as a reference for reading the original source code.

The first program, Hello World, is a trivial piece of code to implement. The example is presented solely to introduce a number of the concepts inherent to literate programming. Figure 4 shows the actual CWeb code for this program.

When a literate program is woven, a table of contents is automatically generated for the program. Any module in the literate program identified as a section header is listed in the table of contents by its number; the page where it's found is also listed in the table of contents.

Identification of modules that function as section headers depends on the program author. The author must identify the modules that logically define the structure of the developed program by introducing the module with a @*, as opposed to the standard @ ␣ (␣ represents a blank). The string following @* represents the section header and ends with a period. Except for this introduction, no differences in appearance or content exist between a module that functions as a section header and any other module in the program. The section headers simply identify the logical sections of the program.

The program index is also generated during literate-program weaving. The index captures all variable names, function names, and procedure names in one concise index. It indicates both the declaration and use of these elements throughout the program.

The declaration of a variable or function causes the associated entry to be underlined. All references to the variable are simply listed in the index. Additional entries, generated as the author desires, can be added. Finally, a second index is generated consisting of the module names used within the program.

Running Weave on the Hello World program generates the output shown in Figure 5. A horizontal line separates each page of the actual output in the figure.

While the woven output from Tex is a readable document, the tangled C code generated from the Hello World program is

not. Figure 6 illustrates the output generated from a tangled CWeb program. As you can see by examination, this output is not designed for human comprehension. However, a standard C compiler can function very well with this input. The comments that exist in the code refer to the number of the module responsible for generating those lines.

Next, we present the second example, a program designed to generate the first 25 Fibonacci numbers in its woven form (using Tex) and without any other introduction. In the belief that this format proves readable without further explanation, we suggest scanning through the program, as shown in Figure 7, and analyzing the readability of this document. Figure 8 shows the actual CWeb file that generated the document.

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.

Figure 5. Document generated from Weave for Hello World.

Critique of literate programming

To date, the programming community hasn't widely accepted the use of literate-programming languages for a number of reasons. The majority of the arguments revolve around the lack of integration of the literate paradigm into the conventional software development life cycle and the current environment supporting literate programming.

The literate paradigm and the software life cycle. Literate programming provides a technique for program development that represents partial covering of the overall software development life cycle. Figure 9 indicates the actual part of the life cycle covered by the literate paradigm. When reviewing the life-cycle processes, it becomes apparent that literate programming is essentially a technique to be used for system implementation. The current literate paradigm concentrates on the development and documentation of the actual system code. In addition, the literate paradigm claims benefits in the life cycle's final stage, that of maintenance.[2]

In itself, the literate paradigm does not offer a complete mechanism for software development. Such development must be provided with the proposed system's specification and high-level design. This input is normally expressed using conventional specification and design strategies, such as structured design or object-based techniques.

This is the data that provides the functionality and design necessary to guide the construction of a literate program. However, most of the literate examples to date[7] have minimized the use of formalized specification and design strategies as a necessary prerequisite for literate programming.

The projects illustrated do not properly emphasize the importance of either the system specification or system design. The literate paradigm does not explicitly provide for the development of either functional system specifications or high-level design plans.

Additionally, the literate paradigm provides no guidelines for establishing a set of testing criteria. The ability to test the implemented software product is of paramount importance to the developmental process; yet, testing remains an ill-defined, user-oriented function within the literate

```
/*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*/
```

Figure 6. C program generated from Tangle for Hello World.

Table of Contents

Source Code

1. Fibonacci numbers.
Program to compute the first 25 Fibonacci numbers.

2. Main program.
This is a skeletal program body around which the program is constructed. We define the constant *MAX* as 25 in the definitions section.

```
#define MAX 25
  ⟨ Include files 3 ⟩
  main ( )
  {
    ⟨ Main variable declarations ⟩
    ⟨ Generate Fibonacci ⟩
  }
```

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

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

This code is used in section 2.

4. Local (main) variables.
Here are our local variables for the routine *main*. Right now we know we need a counter to count the first 25 Fibonacci numbers.

```
⟨ Main variable declarations ⟩ ≡
  int count;
```

See also section 5.

This code is used in section 2.

5. Generation of Fibonacci numbers.
This requires an initial priming with two values. The first two values are zero and one, respectively. After this, the next value is computed by $fib_n = fib_{n-1} + fib_{n-2}$. So, we need variables to keep the last two values and the computed value.

```
⟨ Main variable declarations ⟩ + ≡
  int fib1 = 0, fib2 = 1, newfib;
```

6. Generation of Fibonacci numbers.
The count of numbers generated is started at two. We will print a title and the first two numbers, and then loop until we have generated 23 additional numbers.

```
⟨ Generate Fibonacci ⟩ ≡
  ⟨ Print titles and first two ⟩
  count = 2;
  ⟨ Loop to Print Remainder ⟩
```

This code is used in section 2.

7. Printing titles.
The program will double space before the title, print a heading, and then output the first two Fibonacci numbers.

```
⟨ Print titles and first two ⟩ ≡
  printf ("Fibonacci numbers");
  printf ("%d", fib1);
  printf ("%d", fib2);
```

This code is used in section 6.

8. Generate the remainder of the list.
The loop will run until *count* has a value greater than 25. For each pass through the loop, a new value is generated and then the previous two values are reset for the next pass of the loop.

```
⟨ Loop to print remainder ⟩ ≡
  While (count} < MAX) {
    ⟨ Compute new Fibonacci ⟩
    printf ("%d", newfib); ⟩
    ⟨ Reset last two Fibonacci values ⟩
    count = count + 1;
```

This code is used in section 6.

9. Computing the next Fibonacci.
The formula, as given above, is *fibn* = *fibn* − 1 + *fibn* − 2.

```
⟨ Compute new Fibonacci ⟩ ≡
  newfib = fib1 + fib2;
```

This code is used in section 8.

10. Reset the last two values.

```
⟨ Reset last two Fibonacci values ⟩ ≡
  fib1 = fib2;
  fib2 = newfib;
```

This code is used in section 8.

Variable Index

Section Index

Figure 7. The woven output for the Fibonacci program.

```
@* Fibonacci numbers.
Program to compute the first 25 Fibonacci numbers.

@* Main program.
This is a skeletal program body around which the program is
  constructed. We define the constant |MAX| as 25 in the
  definitions section.
@d
MAX 25
@c
@<Include files@>
main( )
{
  @<Main variable declarations@>
  @<Generate Fibonacci@>
}
@ The only required include file is |stdio.h|.
@<Include files@> =
  #include <stdio.h>

@* Local (main) variables.
Right now, we know we need a counter to count the first 25
  numbers.
@<Main variable declarations@> =
  int count;

@ Generation of Fibonacci numbers requires an initial
  priming with two values. The first two values are zero and
  one, respectively. After this, the next value is computed
  by $fib_n = fib_{n - 1} + fib_{n - 2}$.
So we need variables for the last two values and the
  computed value.
@<Main variable declarations@> + =
  int fib1 = 0, fib2 = 1, newfib;

@* Generation of Fibonacci numbers.
The count of numbers generated starts at two. We print a
  title and the first two numbers, and then loop until we
have generated 23 additional numbers.
@<Generate Fibonacci@> =
  @<Print titles and first two@>
  count = 2;
  @<Loop to print remainder@>

@* Printing titles.
Double space before the title, print a heading, and then
  output the first two Fibonacci numbers.
@<Print titles and first two@> =
  printf("\n\nFibonacci numbers\n");
  printf("\n\t%d\n", fib1);
  printf("\t%d\n", fib2);

@* Generate the remainder of the list.
The loop will run until |count| has a value greater than 25.
  For each pass through the loop, a new value is generated
  and then the previous two values are updated for the next
  pass of the loop.
@<Loop to print remainder@> =
  while ( count < MAX ) {
    @<Compute new Fibonacci@>
    printf("\t\%d\n", newfib);
    @<Update last two Fibonacci values@>
    count = count + 1;
}

@* Computing the next Fibonacci.
The formula for computing the next Fibonacci number is
  $fib_n = fib_{n - 1} + fib_{n - 2}$.
@<Compute new Fibonacci@> =
  newfib = fib1 + fib2;

@* Reset the last two values.
@<Reset last two Fibonacci values@> =
  fib1 = fib2;
  fib2 = newfib;
```

Figure 8. Fibonacci program.

world. Due to differences in both program organization and presentation, a different approach is often required when debugging literate programs.

While the current literate paradigm hasn't been integrated into the software development life cycle, recent research has shown the ability to merge literate programming with the design process.[8] This research has demonstrated that literate programs provide a strong bridge between low-level system design and system implementation.[9] In addition, the enhancements set forth in the "Enhancing the paradigm" section permit further integration with the existing software life cycle.

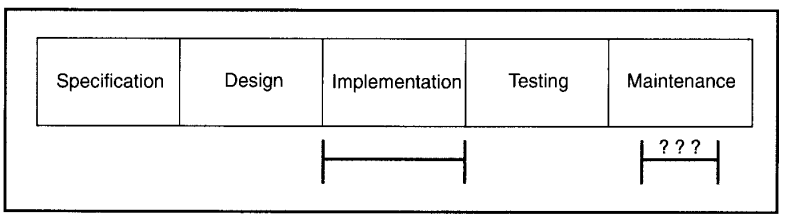

Figure 9. Life-cycle coverage offered by the literate paradigm.

The elitism of literate programming. The literate paradigm has also suffered from its own claims. As suggested by its founder, literate-programming languages are not suitable for everybody. Knuth argued that a literate-programming-language user must "be comfortable dealing with several languages simultaneously."[2]

1.	Fibonacci numbers
2.	Main program
2.1	Main variable declarations
2.2	Generate Fibonacci
2.2.1	Print titles and first two
2.2.2	Loop to print remainder
2.2.2.1	Compute new Fibonacci
2.2.2.2	Reset last two Fibonacci values

Figure 10. Single level vs. multilevel table of contents.

Specifically, the user must not only know the underlying implementation language, but also a word processing language, a literate-programming language, and the possible interactions of all three. This implies that errors from a variety of languages may now appear within a single program. Knuth hoped that computer scientists (as opposed to programmers) will have a greater desire to communicate the essentials of a program to others and, therefore, be more disposed to literate programming.

While not intended to impede the development of a literate-programming community, this argument has essentially restricted the concept of literate programming to the academic community. Although it has been adopted for use in some organizations, this tends to be the exception rather than the rule in a nonacademic environment.

The benefits of literate programming. Despite the drawbacks listed above, the literate paradigm provides exactly the features and tools demanded in large-scale software engineering projects. In fact, we believe that the concepts inherent to the literate paradigm should be used in all large-scale software development projects.

To effectively manage the information associated with program development, the program author should have all necessary data regarding the problem at hand. This includes domain knowledge, relevant programming techniques, and prior programming language expertise. Lacking any of this will impede the development of quality software products.

The literate paradigm permits the program author to organize and manage the logic of the program being developed as appropriate. The structure of the developed program can thus mirror the author's mental image of the program and its associated processes. The author can group and organize the program's concepts and characteristics in a manner designed to promote readability and understandability.

Literate programming demands the generation of system documentation in parallel with system composition. The reason a software engineer can't provide this documentation isn't from a lack of text-processing skills (most software engineers are well versed in both programming and text processing languages), but because of the author's inability to properly organize and document his or her thoughts.

However, the argument remains regarding whether or not proper code documentation can be achieved by merely adding a specific section of each module to hold this documentation. As mentioned previously, any section of the module can be omitted by the program author.

Is it possible to generate literate programs devoid of documentation? Unfortunately, the answer to this query is yes. Most advocates of literate programming claim that they feel a moral obligation to document literate code and do not experience a similar feeling when programming in a conventional language. Nevertheless, no empirical study exists that demonstrates documentation does indeed improve in a literate environment.

Brook's Law[10] shows that the limiting factor in large software projects has stemmed more from communication limitations between programmers than from the difficulty of the programming itself. The literate paradigm's encouragement of programs written to be understood by others should ease these communication limitations.

In most professions, the individuals who inspire the most confidence are those who can not only perform their assigned duties, but who can also explain what has happened in a clear, concise manner.

Perhaps the best analogy for the claim that the software engineer must be able to properly organize and manage the concepts associated with a given software project is as follows: If you are taking a class on ancient civilization and are absent with illness one day, you will need to see someone else's notes. You would prefer to see a set of notes in which the organization and content of the lecture are clearly expressed.

The same holds true for software development. You want a piece of code that tells you what it does and presents information in a simple, organized manner. The software author should not only possess the ability to generate code but provide a structure and organization to that code.

Enhancing the paradigm

Given the strengths and weaknesses of current literate programming, a number of modifications must be made to enhance its potential use as a practical software coding strategy. The next two sections specifically deal with a set of features we propose should be added and several features that should be removed.

Additional items demanded by the paradigm. We propose adding four items to the literate paradigm to improve the usability and presentation of literate programs. They are a multilevel table of contents, a graphical user interface, program debugging tools, and an enhanced index.

These four items are designed to assist in the development, debugging, and maintenance of literate programs. We developed the graphical interface and debugging tools, and are currently developing the multilevel table of contents and enhanced index. The graphical interface supports an X Windows-based programming environment. The debugging tools operate within an Emacs-style editing environment.

Multilevel table of contents. We propose a substantial restructuring of the table of contents. Currently, all modules that function as section headers (designated by the programmer) are automatically included in the program table of contents, as illustrated in Figures 5 and 7. Web automatically generates the numbering and page numbers for these modules. However, the table of contents format is a single-level structure, resembling the chapters of a novel. The literate paradigm would benefit through the use of a multilevel table of contents, as is used in textbooks. Figure 10 illustrates the benefits of such a multilevel table of contents, using the Fibonacci program introduced previously.

A multilevel table of contents provides a more accurate reflection of the author's desired structure and organization of information within the program. Current literate-programming languages have struggled with the identification of section headers, since all headers appear to be equal entities within the table of contents. By defining a logical hierarchy for these section headers, a more readable and un-

derstandable program overview is presented within the table of contents.

Graphical user interface. Writing a literate program requires using the set of Web commands that govern the scope of modules, the relationship between modules, indices, typesetting commands, and a variety of other matters. By presenting an interactive graphical interface to program construction, a number of Web commands can be performed automatically through the use of windowing and mouse selection.

Marcus Brown developed an interactive environment for literate programming and tested a prototype. Programmers using this type of window-based environment could work with Web code without having to know the Web commands. Initial studies using a prototype of the environment show a strong preference for the graphical interface over other current interfaces ($p<0.01$).[11]

As Knuth argues, the combination of programming language commands, typesetting commands, and Web commands produces a complex environment in which the programmer is required to work. The graphical interface clearly separates the Web commands from the programming and typesetting commands, and visually delineates modules and parts of modules. The use of a different input device for the Web commands (for example, the mouse) permits the programmer to physically as well as mentally separate these functions from those of coding and typesetting.

Program debugging tools. A set of graphical tools for assisting in program development and debugging should be available to the literate-programming community. These tools not only serve the author in writing the program, but they also assist in debugging and maintaining literate programs.

Two tools have been developed to assist the programmer in understanding the structure of the overall program. The first tool, called Cnest, illustrates the location and nesting level of the current module within the overall scope of the actual program code. The second tool, called Cscope, displays the overall program and the modules that constitute the program. Both tools provide information regarding the physical structure of the program under construction.

Cnest, illustrating the nesting level and scope of the current module, is primarily designed for the program author. The module in question is placed in its proper perspective relative to the overall program.

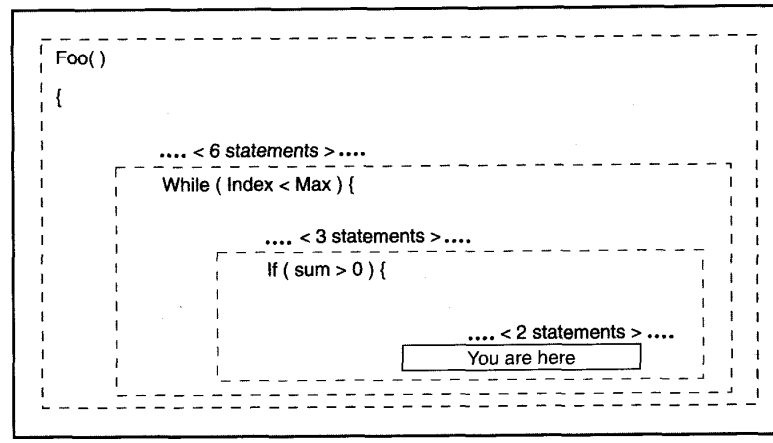

Figure 11. The scope of a sample module.

MakeString( )
  defined in module 13
  contains no other modules
InOrderTraverse( )
  defined in module 34
  contains no other modules
PreOrderTraverse( )
  defined in module 35
  contains no other modules
PostOrderTraverse( )
  defined in module 36
  contains no other modules
main( )
  defined in module 2
  contains modules 3, 4, 5, 6, 7, 15

Figure 12. The structure of a literate program (Tree-traversal program).

Identifiers and Author-Generated Entries

Identifiers (constants, variables, procedures, functions)
  sections in which it appears
  sections defining it
  only definitions for single-letter identifiers
Author-generated entries
  system dependencies
  key program topics
  other author notes

Module Index

Modules
  module name and section number
  list of modules referencing it

Figure 13. Items currently found in a program index.

Figure 11 gives a sample of the output from this tool.

Cscope enables readers versed in conventional programming languages to see a literate program in a more familiar setting. This view of the overall structure of the underlying program is useful during debugging and maintenance. Figure 12 identifies the general outline of the complete program. The association of modules with functions is needed to perform literate-program runtime debugging. Once the function in question is isolated, the modules that compose that function can be analyzed.

Enhanced index. The index to a literate program provides the author and reader with a standard set of information regarding the specific location of items within the program. As illustrated in Figure 13, Knuth identified a set of entities that belong with a program index. The current index format places all variable declarations and references, as well as the author's hand-coded entries, into the major index and provides a separate index for all the named modules within the program.

In its current format, the index presents each unique variable name once. A single

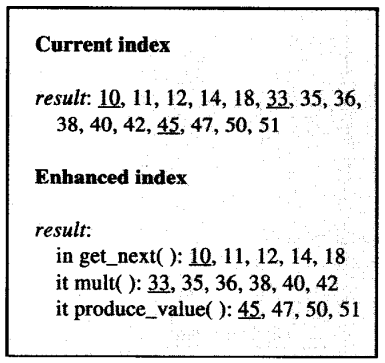

Figure 14. Current Web index vs. enhanced index.

variable name used in different locations for different purposes shows up as only one entry in the index. This entry is listed as being defined in a number of different modules. (The convention of underlining a module number indicates that the identifier is defined in that module.)

Multiple instances of the same variable name must be discovered manually. By enhancing the current index format, it is possible to capture the individual variables within the index, listing the common variable name and noting the enclosing procedure or function as shown in Figure 14. This information is obtained by examining the source code generated within the literate program.

Additionally, the ability to provide Weave with runtime options regarding the format of the presented index would improve the interface for the program reader. Allowing an individual to select any or all of the program identifiers and author-generated entries and place them into separate appendixes would also benefit the reader. For example, the person concerned with porting the program might want to place all system dependencies and user-defined index entries in a separate index. The maintenance programmer might want to separate program variables, program functions, and procedures, as well as constant declarations.

Restrictions imposed on the current implementation. In addition to the recommendations we made regarding new features for literate programming, we believe the use of a number of existing features should be curtailed; specifically, restrictions on the structure of literate programs and the size of the literate command set.

Restrictions on the structure of literate programs. The first set of restrictions regarding literate programming limits the physical structure of an individual module. Each module within the program should function as a logical single entity, as a normal block-structured language would demand. Each module containing a "begin" for some segment of code must contain a corresponding "end" statement. Naturally, the contained code may use submodules to abstract out the details of the processing. However, each function, procedure, or block opened within a given module should always be closed within the same module.

The restriction regarding begin-end constructs not only helps in defining module functionality but also assists the reader in tracking variable definitions and scope. Variables declared in a module that spans the next several modules can find their declaration hidden from the maintenance programmer. The scope of individual variables is more apparent in single-module functions and procedures.

Related to this issue is the demand that all functions and procedures within the program be contained in separate modules. The inclusion of multiple functions within a single module can obscure or convolute the purpose of the individual functions. By separating each into its own module, the interface to any single function is more apparent.

Reduction of the literate command set. The commands that allow the programmer to fine-tune the typesetting of a literate program should be sharply reduced. For example, CWeb currently defines three different commands (@^, @., and @:) that allow the programmer to enter a string in the program index. These three vary only in how the string is typeset. Instead, use a single command and, if necessary, add the typesetting instructions to the string itself.

Literate typesetting commands permit the programmer to override the standard formatting algorithms used within the Weave program. The programmer is thus capable of producing code that conforms to individual stylistic considerations and not a universal standard.

In conjunction with this elimination of Web typesetting commands, a template-based documentation generation routine must be provided. Programmers or organizations should be able to provide a template for all program documents generated in the literate environment. Any code generated in this environment would conform to the standards set in this template. Companywide standards for documentation style could be implemented with little overhead to individual programmers.

If the Web programmer has the use of a literate-programming environment, as described in Brown and Childs,[11] programming in Web can be done without explicitly using any Web commands. However, many programmers do not have access to such a tool, and the Web command set must be used. This set of commands should be as small as possible.

The literate paradigm offers a platform for code generation that promotes readability and understandability. As it exists today, this platform is ill-suited for adoption by most software developers. The lack of a proper interface into the software development life cycle, the lack of an environment that minimizes the overhead associated with the literate paradigm, and the awkwardness of current literate languages discourage the adoption of literate techniques. Furthermore, to offer a practical, usable methodology for software development, the literate paradigm must encompass the entire life cycle, not just the implementation phase.

However, these obstacles are not insurmountable. As we outlined, techniques exist to eliminate or minimize a number of potential problems with today's literate-programming languages. As more tools become available for literate programming, the paradigm will become appealing to a wider audience. Perhaps the second generation of literate-programming languages will gain the acceptance that those in the first generation sorely missed. ■

Acknowledgments

We thank the reviewers for their insightful comments and improvements.

References

1. R.S. Pressman, *Software Eng.: A Practitioner's Approach*, McGraw-Hill, New York, N.Y., 1987.

2. D.E. Knuth, "Literate Programming," *Computer J.*, Vol. 27, No. 2, May 1984, pp. 97-111.

3. D.E. Knuth, *Computers and Typesetting*, Addison-Wesley, Reading, Mass., 1986.

4. S. Levy, "Web Adapted to C — Another Approach," *TUGboat*, Vol. 8, No. 1, Apr. 1987, pp. 12-14.

5. A. Avenarius and S. Oppermann, "FWeb: A Literate-Programming System for Fortran8x," *SIGPlan Notices*, Vol. 25, No. 1, Jan. 1990, pp. 52-58.

6. N. Ramsey, "Weaving a Language-Independent Web," *Comm. ACM*, Vol. 32, No. 9, Sept. 1989, pp. 1,051-1,055.

7. J. Bentley, "Programming Pearls: Literate Programming," *Comm. ACM*, Vol. 29, No. 5, May 1986, pp. 364-369, and "Programming Pearls: A Literate Program, *Comm. ACM*, Vol. 29, No. 6, June 1986, pp. 471-483.

8. M. Brown and D. Cordes, "A Literate Programming Design Language," *Proc. Comp Euro 90, IEEE Int'l Conf. Computer Systems and Software Eng.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2041, 1990, pp. 548-549.

9. M. Brown and D. Cordes, "Literate Programming Applied to Conventional Software Design," *J. Structured Programming*, Vol. 11, No. 2, 1990, pp. 85-98.

10. F.P. Brooks, *The Mythical Man-Month: Essays in Software Eng.*, Addison-Wesley, Reading, Mass., 1975.

11. M. Brown and B. Childs, "An Interactive Environment for Literate Programming," *J. Structured Programming*, Vol. 11, No. 1, 1990, pp. 11-25.

David Cordes is an assistant professor of computer science at the University of Alabama. His research interests include requirements gathering, specification languages and methodologies, high-level design, and literate programming.

Cordes received his PhD in computer science from Louisiana State University, his MS in computer science from Purdue University, and his BS from the University of Arkansas. He is a member of the IEEE Computer Society and the ACM.

Marcus Brown is an assistant professor of computer science at the University of Alabama. His research interests center on the human-computer interface, including literate programming, hypertext and hypermedia, computer-aided instruction, and the psychology of programming.

Brown received his BA and MDiv from Abilene Christian University and his PhD from Texas A&M University. He is a member of the ACM and the ACM SIGCHI.

Readers may contact David Cordes at the Department of Computer Science, University of Alabama, Tuscaloosa, AL 35487-0290, Internet dcordes@ualvm.ua.edu, Bitnet dcordes@ualvm.