Literate Programming

Donald E. Knuth
Computer Science Department, Stanford University, Stanford, CA 94305, USA

The author and his associates have been experimenting for the past several years with a programming language and documentation system called WEB. This paper presents WEB by example, and discusses why the new system appears to be an improvement over previous ones.

A. INTRODUCTION

The past ten years have witnessed substantial improvements in programming methodology. This advance, carried out under the banner of "structured programming," has led to programs that are more reliable and easier to comprehend; yet the results are not entirely satisfactory. My purpose in the present paper is to propose another motto that may be appropriate for the next decade, as we attempt to make further progress in the state of the art. I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be *works of literature*. Hence, my title: "Literate Programming."

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.

The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that nicely reïnforce each other.

I dare to suggest that such advances in documentation are possible because of the experiences I've had during the past several years while working intensively on software development. By making use of several ideas that have existed for a long time, and by applying them systematically in a slightly new way, I've stumbled across a method of composing programs that excites me very much. In fact, my enthusiasm is so great that I must warn the reader to discount much of what I shall say as the ravings of a fanatic who thinks he has just seen a great light.

Programming is a very personal activity, so I can't be certain that what has worked for me will work for everybody. Yet the impact of this new approach on my own style has been profound, and my excitement has continued unabated for more than two years. I enjoy the new methodology so much that it is hard for me to refrain from going back to every program that I've ever written and recasting it in "literate" form. I find myself unable to resist working on programming tasks that I would ordinarily have assigned to student research assistants; and why? Because it seems to me that at last I'm able to write programs as they should be written. My programs are not only explained better than ever before; they also are better programs, because the new methodology encourages me to do a better job. For these reasons I am compelled to write this paper, in hopes that my experiences will prove to be relevant to others.

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 everybody 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 writing an *illiterate* program.

B. THE WEB SYSTEM

I hope, however, to demonstrate in this paper that the title is not merely wordplay. The ideas of literate programming have been embodied in a language and a suite of computer programs that have been developed at Stanford University during the past few years as part of my research on algorithms and on digital typography. This language and its associated programs have come to be known as the WEB system. My goal in what follows is to describe the philosophy that underlies WEB, to present examples of programs in the WEB language, and to discuss what may be the future implications of this work.

I chose the name WEB partly because it was one of the few three-letter words of English that hadn't already been applied to computers. But as time went on, I've become extremely pleased with the name, because I think that a complex piece of software is, indeed, best regarded as a *web* that has been delicately pieced together from simple materials. We understand a complicated system by understanding its simple parts, and by understanding the simple relations between those parts and their immediate neighbors. If we express a program as a web of ideas, we can emphasize its structural properties in a natural and satisfying way.

WEB itself is chiefly a combination of two other languages: (1) a document formatting language and (2) a programming language. My prototype WEB system uses TEX as the document formatting language and PASCAL as the programming language, but the same principles would apply equally well if other languages

CCC–0010–4620/84/0027–0097 $07.50

were substituted. Instead of TEX, one could use a language like Scribe or Troff; instead of PASCAL, one could use ADA, ALGOL, LISP, COBOL, FORTRAN, APL, C, etc., or even assembly language. The main point is that WEB is inherently bilingual, and that such a combination of languages proves to be much more powerful than either single language by itself. WEB does not make the other languages obsolete; on the contrary, it enhances them.

I naturally chose TEX to be the document formatting language, in the first WEB system, because TEX is my own creation;[1] I wanted to acquire a lot of experience in harnessing TEX to a variety of different tasks. I chose PASCAL as the programming language because it has received such widespread support from educational institutions all over the world; it is not my favorite language for system programming, but it has become a "second language" for so many programmers that it provides an exceptionally effective medium of communication. Furthermore WEB itself has a macro-processing ability that makes PASCAL's limitations largely irrelevant.

Document formatting languages are newcomers to the computing scene, but their use is spreading rapidly. Therefore I'm confident that we will be able to expect each member of the next generation of programmers to be familiar with a document language as well as a programming language, as part of their basic education. Once a person knows both of the underlying languages, there's no trick at all to learning WEB, because the WEB user's manual is fewer than ten pages long.

A WEB user writes a program that serves as the source language for two different system routines. (See Figure 1.) One line of processing is called *weaving* the web; it produces a document that describes the program clearly and that facilitates program maintenance. The other line of processing is called *tangling* the web; it produces a machine-executable program. The program and its documentation are both generated from the same source, so they are consistent with each other.

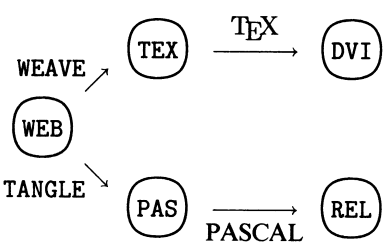

Figure 1. Dual usage of a WEB file.

Let's look at this process in slightly more detail. Suppose you have written a WEB program and put it into a computer text file called COB.WEB (say). To generate hardcopy documentation for your program, you can run the WEAVE processor; this is a system program that takes the file COB.WEB as input and produces another file COB.TEX as output. Then you run the TEX processor, which takes COB.TEX as input and produces COB.DVI as output. The latter file, COB.DVI, is a "device-independent" binary description of how to typeset the documentation, so you can get printed output by applying an appropriate system routine to this binary file.

You can also follow the other branch of Figure 1, by running the TANGLE processor; this is a system program that takes the file COB.WEB as input and produces a new file COB.PAS as output. Then you run the PASCAL compiler, which converts COB.PAS to a binary file COB.REL (say). Finally, you can run your program by loading and executing COB.REL. The process of "compile, load, and go" has been slightly lengthened to "tangle, compile, load, and go."

C. A COMPLETE EXAMPLE

Now it's time for me to stop presenting general platitudes and to move on to something tangible. Let us look at a real program that has been written in WEB. The numbered paragraphs that follow are the actual output of a WEB file that has been "woven" into a document; a computer has also generated the indexes that appear at the program's end. If my claims for the advantages of literate programming have any merit, you should be able to understand the following description more easily than you could have understood the same program when presented in a more conventional way. However, I am trying here to explain the format of WEB documentation at the same time as I am discussing the details of a nontrivial algorithm, so the description below is slightly longer than it would be if it were written for people who already have been introduced to WEB.

Here, then, is the computer-generated output:

1. Printing primes: An example of WEB. The following program is essentially the same as Edsger Dijkstra's "first example of step-wise program composition," found on pages 26–39 of his *Notes on Structured Programming*,[2] but it has been translated into the WEB language.

[[Double brackets will be used in what follows to enclose comments relating to WEB itself, because the chief purpose of this program is to introduce the reader to the WEB style of documentation. WEB programs are always broken into small sections, each of which has a serial number; the present section is number 1.]]

Dijkstra's program prints a table of the first thousand prime numbers. We shall begin as he did, by reducing the entire program to its top-level description. [[Every section in a WEB program begins with optional *commentary* about that section, and ends with optional *program text* for the section. For example, you are now reading part of the commentary in §1, and the program text for §1 immediately follows the present paragraph. Program texts are specifications of PASCAL programs; they either use PASCAL language directly, or they use angle brackets to represent PASCAL code that appears in other sections. For example, the angle-bracket notation '⟨Program to print ... numbers 2⟩' is WEB's way of

saying the following: "The PASCAL text to be inserted here is called 'Program to print ... numbers', and you can find out all about it by looking at section 2." One of the main characteristics of WEB is that different parts of the program are usually abbreviated, by giving them such an informal top-level description.]]

⟨ Program to print the first thousand prime numbers 2 ⟩

2. This program has no input, because we want to keep it rather simple. The result of the program will be to produce a list of the first thousand prime numbers, and this list will appear on the *output* file.

Since there is no input, we declare the value $m = 1000$ as a compile-time constant. The program itself is capable of generating the first m prime numbers for any positive m, as long as the computer's finite limitations are not exceeded.

[[The program text below specifies the "expanded meaning" of '⟨ Program to print ... numbers 2 ⟩'; notice that it involves the top-level descriptions of three other sections. When those top-level descriptions are replaced by their expanded meanings, a syntactically correct PASCAL program will be obtained.]]

```
⟨ Program to print the first thousand prime
      numbers 2 ⟩ ≡
program print_primes(output);
  const m = 1000;
    ⟨ Other constants of the program 5 ⟩
  var ⟨ Variables of the program 4 ⟩
    begin ⟨ Print the first m prime numbers 3 ⟩;
    end.
```

This code is used in section 1.

---

3. Plan of the program. We shall proceed to fill out the rest of the program by making whatever decisions seem easiest at each step; the idea will be to strive for simplicity first and efficiency later, in order to see where this leads us. The final program may not be optimum, but we want it to be reliable, well motivated, and reasonably fast.

Let us decide at this point to maintain a table that includes all of the prime numbers that will be generated, and to separate the generation problem from the printing problem.

[[The WEB description you are reading once again follows a pattern that will soon be familiar: A typical section begins with comments and ends with program text. The comments motivate and explain noteworthy features of the program text.]]

```
⟨ Print the first m prime numbers 3 ⟩ ≡
  ⟨ Fill table p with the first m prime numbers 11 ⟩;
  ⟨ Print table p 8 ⟩
```

This code is used in section 2.

4. How should table p be represented? Two possibilities suggest themselves: We could construct a sufficiently large array of boolean values in which the kth entry is *true* if and only if the number k is prime; or we could build an array of integers in which the kth entry is the kth prime number. Let us choose the latter alternative, by introducing an integer array called $p[1 .. m]$.

In the documentation below, the notation '$p[k]$' will refer to the kth element of array p, while 'p_k' will refer to the kth prime number. If the program is correct, $p[k]$ will either be equal to p_k or it will not yet have been assigned any value.

[[Incidentally, our program will eventually make use of several more variables as we refine the data structures. All of the sections where variables are declared will be called '⟨ Variables of the program 4 ⟩'; the number '4' in this name refers to the present section, which is the first section to specify the expanded meaning of '⟨ Variables of the program ⟩'. The note 'See also ...' refers to all of the other sections that have the same top-level description. The expanded meaning of '⟨ Variables of the program 4 ⟩' consists of all the program texts for this name, not just the text found in §4.]]

```
⟨ Variables of the program 4 ⟩ ≡
p: array [1 .. m] of integer;  { the first m prime
      numbers, in increasing order }
```

See also sections 7, 12, 15, 17, 23, and 24.

This code is used in section 2.

---

5. The output phase. Let's work on the second part of the program first. It's not as interesting as the problem of computing prime numbers; but the job of printing must be done sooner or later, and we might as well do it sooner, since it will be good to have it done. [[And it is easier to learn WEB when reading a program that has comparatively few distracting complications.]]

Since p is simply an array of integers, there is little difficulty in printing the output, except that we need to decide upon a suitable output format. Let us print the table on separate pages, with rr rows and cc columns per page, where every column is ww character positions wide. In this case we shall choose $rr = 50$, $cc = 4$, and $ww = 10$, so that the first 1000 primes will appear on five pages. The program will not assume that m is an exact multiple of $rr \cdot cc$.

```
⟨ Other constants of the program 5 ⟩ ≡
  rr = 50;  { this many rows will be on each page in
      the output }
  cc = 4;  { this many columns will be on each page
      in the output }
  ww = 10;  { this many character positions will be
      used in each column }
```

See also section 19.

This code is used in section 2.

6. In order to keep this program reasonably free of notations that are uniquely PASCALesque, [[and in order to illustrate more of the facilities of WEB,]] a few macro definitions for low-level output instructions are introduced here. All of the output-oriented commands in the remainder of the program will be stated in terms of five simple primitives called *print_string*, *print_integer*, *print_entry*, *new_line*, and *new_page*.

[[Sections of a WEB program are allowed to contain *macro definitions* between the opening comments and the closing program text. The general format for each section is actually tripartite: commentary, then definitions, then program. Any of the three parts may be absent; for example, the present section contains no program text.]]

[[Simple macros simply substitute a bit of PASCAL code for an identifier. Parametric macros are similar,

but they also substitute an argument wherever '#' occurs in the macro definition. The first three macro definitions here are parametric; the other two are simple.]]

```
define print_string(#) ≡ write(#)
        { put a given string into the output file }
define print_integer(#) ≡ write(# : 1)
        { put a given integer into the output file,
          in decimal notation, using only as many
          digit positions as necessary }
define print_entry(#) ≡ write(# : ww)  { like
        print_integer, but ww character positions
        are filled, inserting blanks at the left }
define new_line ≡ write_ln  { advance to a new line
        in the output file }
define new_page ≡ page  { advance to a new page
        in the output file }
```

7. Several variables are needed to govern the output process. When we begin to print a new page, the variable *page_number* will be the ordinal number of that page, and *page_offset* will be such that $p[page\_offset]$ is the first prime to be printed. Similarly, $p[row\_offset]$ will be the first prime in a given row.

[[Notice the notation '+≡' below; this indicates that the present section has the same name as a previous section, so the program text will be appended to some text that was previously specified.]]

```
⟨ Variables of the program 4 ⟩ +≡
page_number: integer;  { one more than the number
      of pages printed so far }
page_offset: integer;  { index into p for the first entry
      on the current page }
row_offset: integer;  { index into p for the first entry
      in the current row }
c: 0 .. cc;  { runs through the columns in a row }
```

8. Now that appropriate auxiliary variables have been introduced, the process of outputting table p almost writes itself.

```
⟨ Print table p 8 ⟩ ≡
  begin page_number ← 1; page_offset ← 1;
  while page_offset ≤ m do
    begin ⟨ Output a page of answers 9 ⟩;
    page_number ← page_number + 1;
    page_offset ← page_offset + rr * cc;
    end;
  end
```

This code is used in section 3.

9. A simple heading is printed at the top of each page.

```
⟨ Output a page of answers 9 ⟩ ≡
  begin print_string('␣First␣');
  print_integer(m);
  print_string('␣Prime␣Numbers␣---␣Page␣');
  print_integer(page_number); new_line; new_line;
      { there's a blank line after the heading }
  for row_offset ← page_offset to page_offset + rr − 1
        do ⟨ Output a line of answers 10 ⟩;
  new_page;
  end
```

This code is used in section 8.

10. The first row will contain

$$p[1],\ p[1+rr],\ p[1+2*rr],\ \ldots;$$

a similar pattern holds for each value of the *row_offset*.

```
⟨ Output a line of answers 10 ⟩ ≡
  begin for c ← 0 to cc − 1 do
    if row_offset + c * rr ≤ m then
      print_entry(p[row_offset + c * rr]);
  new_line;
  end
```

This code is used in section 9.

---

11. Generating the primes. The remaining task is to fill table p with the correct numbers. Let us do this by generating its entries one at a time: Assuming that we have computed all primes that are j or less, we will advance j to the next suitable value, and continue doing this until the table is completely full.

The program includes a provision to initialize the variables in certain data structures that will be introduced later.

```
⟨ Fill table p with the first m prime numbers 11 ⟩ ≡
  ⟨ Initialize the data structures 16 ⟩;
  while k < m do
    begin ⟨ Increase j until it is the next prime
        number 14 ⟩;
    k ← k + 1; p[k] ← j;
    end
```

This code is used in section 3.

12. We need to declare the two variables j and k that were just introduced.

```
⟨ Variables of the program 4 ⟩ +≡
j: integer;  { all primes ≤ j are in table p }
k: 0 .. m;  { this many primes are in table p }
```

13. So far we haven't needed to confront the issue of what a prime number is. But everything else has been taken care of, so we must delve into a bit of number theory now.

By definition, a number is called prime if it is an integer greater than 1 that is not evenly divisible by any smaller prime number. Stating this another way, the integer $j > 1$ is not prime if and only if there exists a prime number $p_n < j$ such that j is a multiple of p_n.

Therefore the section of the program that is called '⟨ Increase j until it is the next prime number ⟩' could be coded very simply: '**repeat** $j \leftarrow j+1$; ⟨ Give to *j_prime* the meaning: j is a prime number ⟩; **until** *j_prime*'. And to compute the boolean value *j_prime*, the following would suffice: '*j_prime* ← *true*; **for** $n \leftarrow 1$ **to** k **do** ⟨ If $p[n]$ divides j, set *j_prime* ← *false* ⟩'.

14. However, it is possible to obtain a much more efficient algorithm by using more facts of number theory. In the first place, we can speed things up a bit by recognizing that $p_1 = 2$ and that all subsequent primes are odd; therefore we can let j run through odd values only. Our program now takes the following form:

```
⟨ Increase j until it is the next prime number 14 ⟩ ≡
  repeat j ← j + 2;
    ⟨ Update variables that depend on j 20 ⟩;
    ⟨ Give to j_prime the meaning: j is a prime
        number 22 ⟩;
  until j_prime
```

This code is used in section 11.

15. The **repeat** loop in the previous section introduces a boolean variable *j_prime*, so that it will not be necessary to resort to a **goto** statement. (We are following Dijkstra,[2] not Knuth.[3])

⟨ Variables of the program 4 ⟩ +≡
j_prime: *boolean*; { is j a prime number? }

16. In order to make the odd-even trick work, we must of course initialize the variables j, k, and $p[1]$ as follows.

⟨ Initialize the data structures 16 ⟩ ≡
$j \leftarrow 1$; $k \leftarrow 1$; $p[1] \leftarrow 2$;

See also section 18.

This code is used in section 11.

17. Now we can apply more number theory in order to obtain further economies. If j is not prime, its smallest prime factor p_n will be $\sqrt{j}$ or less. Thus if we know a number *ord* such that

$$p[ord]^2 > j,$$

and if j is odd, we need only test for divisors in the set $\{p[2], \ldots, p[ord-1]\}$. This is much faster than testing divisibility by $\{p[2], \ldots, p[k]\}$, since *ord* tends to be much smaller than k. (Indeed, when k is large, the celebrated "prime number theorem" implies that the value of *ord* will be approximately $2\sqrt{k/\ln k}$.)

Let us therefore introduce *ord* into the data structure. A moment's thought makes it clear that *ord* changes in a simple way when j increases, and that another variable *square* facilitates the updating process.

⟨ Variables of the program 4 ⟩ +≡
ord: 2 .. *ord_max*;
{ the smallest index ≥ 2 such that $p_{ord}^2 > j$ }
square: *integer*; { $square = p_{ord}^2$ }

18. ⟨ Initialize the data structures 16 ⟩ +≡
ord ← 2; *square* ← 9;

19. The value of *ord* will never get larger than a certain value *ord_max*, which must be chosen sufficiently large. It turns out that *ord* never exceeds 30 when $m = 1000$.

⟨ Other constants of the program 5 ⟩ +≡
ord_max = 30; { $p_{ord\_max}^2$ must exceed p_m }

20. When j has been increased by 2, we must increase *ord* by unity when $j = p_{ord}^2$, i.e., when $j = square$.

⟨ Update variables that depend on j 20 ⟩ ≡
if $j = square$ **then**
begin *ord* ← *ord* + 1;
⟨ Update variables that depend on *ord* 21 ⟩;
end

This code is used in section 14.

21. At this point in the program, *ord* has just been increased by unity, and we want to set $square := p_{ord}^2$. A surprisingly subtle point arises here: How do we know that p_{ord} has already been computed, i.e., that $ord \leq k$? If there were a gap in the sequence of prime numbers, such that $p_{k+1} > p_k^2$ for some k, then this part of the program would refer to the yet-uncomputed value $p[k+1]$ unless some special test were made.

Fortunately, there are no such gaps. But no simple proof of this fact is known. For example, Euclid's famous demonstration that there are infinitely many prime numbers is strong enough to prove only that $p_{k+1} \leq p_1 \ldots p_k + 1$. Advanced books on number theory come to our rescue by showing that much more is true; for example, "Bertrand's postulate" states that $p_{k+1} < 2p_k$ for all k.

⟨ Update variables that depend on *ord* 21 ⟩ ≡
square ← $p[ord] * p[ord]$; { at this point $ord \leq k$ }

See also section 25.

This code is used in section 20.

---

22. The inner loop. Our remaining task is to determine whether or not a given integer j is prime. The general outline of this part of the program is quite simple, using the value of *ord* as described above.

⟨ Give to *j_prime* the meaning: j is a prime number 22 ⟩ ≡
$n \leftarrow 2$; *j_prime* ← *true*;
while $(n < ord) \wedge$ *j_prime* **do**
begin ⟨ If $p[n]$ is a factor of j, set *j_prime* ← *false* 26 ⟩;
$n \leftarrow n + 1$;
end

This code is used in section 14.

23. ⟨ Variables of the program 4 ⟩ +≡
n: 2 .. *ord_max*;
{ runs from 2 to *ord* when testing divisibility }

24. Let's suppose that division is very slow or nonexistent on our machine. We want to detect nonprime odd numbers, which are odd multiples of the set of primes $\{p_2, \ldots, p_{ord}\}$.

Since *ord_max* is small, it is reasonable to maintain an auxiliary table of the smallest odd multiples that haven't already been used to show that some j is nonprime. In other words, our goal is to "knock out" all of the odd multiples of each p_n in the set $\{p_2, \ldots, p_{ord}\}$, and one way to do this is to introduce an auxiliary table that serves as a control structure for a set of knock-out procedures that are being simulated in parallel. (The so-called "sieve of Eratosthenes" generates primes by a similar method, but it knocks out the multiples of each prime serially.)

The auxiliary table suggested by these considerations is a *mult* array that satisfies the following invariant condition: For $2 \leq n < ord$, $mult[n]$ is an odd multiple of p_n such that $mult[n] < j + 2p_n$.

⟨ Variables of the program 4 ⟩ +≡
mult: **array** [2 .. *ord_max*] **of** *integer*;
{ runs through multiples of primes }

25. When *ord* has been increased, we need to initialize a new element of the *mult* array. At this point $j = p[ord-1]^2$, so there is no need for an elaborate computation.

⟨ Update variables that depend on *ord* 21 ⟩ +≡
$mult[ord-1] \leftarrow j$;

26. The remaining task is straightforward, given the data structures already prepared. Let us recapitulate the current situation: The goal is to test whether or not j is divisible by p_n, without actually performing a division. We know that j is odd, and that $mult[n]$ is an odd multiple of p_n such that $mult[n] < j + 2p_n$. If $mult[n] < j$, we can increase $mult[n]$ by $2p_n$ and

the same conditions will hold. On the other hand if $mult[n] \geq j$, the conditions imply that j is divisible by p_n if and only if $j = mult[n]$.

```
⟨ If p[n] is a factor of j, set j_prime ← false 26 ⟩ ≡
  while mult[n] < j do
    mult[n] ← mult[n] + p[n] + p[n];
  if mult[n] = j then j_prime ← false
```

This code is used in section 22.

27. Index. Every identifier used in this program is shown here together with a list of the section numbers where that identifier appears. The section number is underlined if the identifier was defined in that section. However, one-letter identifiers are indexed only at their point of definition, since such identifiers tend to appear almost everywhere. [[An index like this is prepared automatically by the WEB software, and it is appended to the final section of the program. However, underlining of section numbers is not automatic; the user is supposed to mark identifiers at their point of definition in the WEB source file.]]

This index also refers to some of the places where key elements of the program are treated. For example, the entries for 'Output format' and 'Page headings' indicate where details of the output format are discussed. Several other topics that appear in the documentation (e.g., 'Bertrand's postulate') have also been indexed. [[Special instructions within a WEB source file can be used to insert essentially anything into the index.]]

D. HOW THE EXAMPLE WAS SPECIFIED

Everything reproduced above, from the table of contents preceding the program to the indexes of identifiers and section names at the end, was generated by applying the program WEAVE to a source file PRIMES.WEB written in the WEB language. Let us now look at that file PRIMES.WEB, in order to get an idea of what a WEB user actually types.

There's no need to show very much of PRIMES.WEB, however, because that file is reflected quite faithfully by the formatted output. Figure 2 contains enough of the WEB source to indicate the general flavor; a reader who is familiar with the rudiments of TEX will be able to reconstruct all of PRIMES.WEB by looking only at the formatted output and Figure 2.

Figure 2a starts with TEX commands (not shown in full) that make it convenient to typeset double brackets [[...]] and to give special typographic treatment to names like 'WEB' and 'PASCAL'. A WEB user generally begins by declaring such special aspects of the document format; for example, if nonstandard fonts of type are needed, they are usually stated first. It may also be necessary to specify the correct hyphenation of non-English words that appear in the document.

Then comes '@*', which starts the program proper. WEB uses the symbol '@' as an escape character for special instructions to the WEAVE and TANGLE processors. Everything between such special commands is either expressed in TEX language or in PASCAL language, depending on the context.

```
\font\ninerm=cmr9
\let\mc=\ninerm % medium caps
\def\WEB{{\tt WEB}}
\def\PASCAL{{\mc PASCAL}}
\def\[{\ifhmode\ \fi$[\![$}
\def\]{$]\!]$\ }
    ⋮
\hyphenation{Dijk-stra}

@* Printing primes: An example of \WEB.
The following program is essentially the same
as Edsger Dijkstra's @^Dijkstra, Edsger@>
``first example of step-wise program
composition,'' found on pages 26--39
of his {\sl Notes on Structured
Programming},$^\Dijk$  but it has been
translated into the \WEB\ language. @.WEB@>

\[Double brackets will be used in what
follows to enclose comments relating to \WEB\
    ⋮
an informal top-level description.\]

@p @<Program to print the first thousand
prime numbers@>
```

Figure 2a. The beginning of PRIMES.WEB.

```
@ This program has no input, because we want
to keep it rather simple.  The result of the
program will be to produce a list of the
first thousand prime numbers, and this list
will appear on the |output| file.

Since there is no input, we declare the value
|m=1000| as a compile-time constant. The
program itself is capable of generating the
first |m| prime numbers for any positive |m|,
as long as the computer's finite limitations
are not exceeded.

\[The program text below specifies the
``expanded meaning'' of `\X2:Program to print
$\ldots$ numbers\X'; notice that it involves
the top-level descriptions of three other
sections. When those top-level descriptions
are replaced by their expanded meanings, a
syntactically correct \PASCAL\ program will
be obtained.\]

@<Program to print...@>=
program print_primes(output);
const @!m=1000;
@<Other constants of the program@>@;
var @<Variables of the program@>@;
begin @<Print the first |m| prime numbers@>;
end.
```

Figure 2b. The WEB code that generated §2.

```
@ In order to keep this program reasonably
free of notations that are uniquely
\PASCAL esque, \[and in order to illustrate
    ⋮
The first three macro definitions here are
parametric; the other two are simple.\]

@d print_string(#)==write(#)
 {put a given string into the |output| file}
@d print_integer(#)==write(#:1)
 {put a given integer into the |output|
  file, in decimal notation, using only as
  many digit positions as necessary}
@d print_entry(#)==write(#:ww)
 {like |print_integer|, but
  |ww| character positions are filled,
  inserting blanks at the left}
@d new_line==write_ln
 {advance to a new line in the |output| file}
@d new_page==page
 {advance to a new page in the |output| file}
```

Figure 2c. The WEB code that generated §6.

Each section of the program begins either with '@ ' (i.e., at-sign and space) or '@*' (i.e., at-sign and asterisk); WEB supplies the section numbers automatically. The latter case, '@*', denotes a *major section* of the program, for which a special title is given. This title will appear in boldface type, and it will also appear in the table of contents, and as a running headline on all pages of the woven documentation until another major section begins. Each major section starts at the top of a page. (Such page beginnings have been indicated by horizontal lines in our example, because WEB's normal output format has been adapted to the format of this journal. The output of WEAVE usually has a lot more white space, and the individual lines of text are usually quite a bit wider.)

The lines that follow in Figure 2a show a few more WEB instructions: '@^' marks the beginning of an index entry to be set in roman type; '@>' marks the end of an argument to a WEB command; '@.' marks the beginning of an index entry to be set in typewriter type; '@p' marks the beginning of the PASCAL program; and '@<' marks the beginning of a top-level description, i.e., of a section name in the WEB program.

Figure 2b immediately follows Figure 2a in the WEB file. This material is what generated §2 of the documentation, and it illustrates the bilingual nature of WEB: The commentary at the beginning of each section is typed in TEX language, and the program text at the end is typed in PASCAL language.

Language-switching between TEX and PASCAL is occasionally desirable. For example, when you refer to technical details about the program, you usually want to describe them in PASCAL, hence you want WEAVE to format them with the typographic conventions it uses for PASCAL programs. Conversely, when you put comments in a PASCAL program, you want the text of those comments to be formatted by TEX in the normal way. WEB files use vertical bars to introduce PASCAL formatting in the midst of TEX formatting; for example, Figure 2b says 'the |output| file' in order to typeset 'the *output* file'.

The program text in Figure 2b begins with '@<' instead of with the '@p' command used in Figure 2a, because the program text in §2 is the expansion of a specific top-level description. Notice that the top-level description has been abbreviated to '@<Program

to print...@>'. Since the names of sections tend to be rather long, it is a nuisance to type them in full each time; WEB allows you to type '...' after you have given enough text to identify the remainder uniquely.

The '@!' operation in the program text of Figure 2b governs the underlining of index entries. The '@;' specifies an invisible symbol that has the effect of a semicolon in PASCAL syntax. Commands such as these are comparatively unimportant, but they are available for polishing up the final documentation when you want to maintain fine control.

Figure 2c shows key portions of the WEB text that generated §6. Notice that the command '@d' introduces a macro definition. All features of WEB that appear in our example program are illustrated in Figures 2a, 2b, and 2c; the remainder of PRIMES.WEB simply uses the same conventions again and again. In fact, most of the WEB file is much simpler than the examples shown here; Figure 2 has illustrated only the difficult parts.

E. THE TANGLED OUTPUT

Figure 3 shows the PASCAL program PRIMES.PAS that results when TANGLE is applied to PRIMES.WEB. This program is not intended for human consumption—it's only supposed to be readable by a PASCAL compiler—so TANGLE does not go to great pains to produce a beautiful format. Notice that underlines have been removed from the identifier names, and that all of the letters have been converted to uppercase (except in strings); TANGLE tries to produce a format that will be acceptable to a standard PASCAL compiler.

TANGLE removes all of the commentary in the WEB file, but it inserts new comments of its own. If for some reason you need to correlate the tangled PASCAL code with the woven documentation, you can find the program text for, say, §8 by looking between the comments '{8:}' and '{:8}'.

A comparison of Figure 3 to Figure 2 should make it clear why the TANGLE processor has acquired its name.

F. THE WOVEN OUTPUT

I mentioned earlier that WEAVE is a program that converts a file like PRIMES.WEB into a file PRIMES.TEX that is a syntactically correct source file for TEX. Figure 4 gives a sampling of PRIMES.TEX, which is even more unreadable than PRIMES.PAS. The instructions that cause TEX to produce formatted PASCAL programs, with appropriate typefaces and indentation, etc., are somewhat complex because they are supposed to give decent results regardless of the page size.

There is no need to discuss Figure 4 further in the present paper, because the details of "pretty printing" are not relevant to my main theme. I have shown this much of PRIMES.TEX only to make the point that it is nice to have a program like WEAVE to do all the formatting; computer programs are not easy to typeset.

```
{1:}{2:}PROGRAM PRINTPRIMES(OUTPUT);
CONST M=1000;{5:}RR=50;CC=4;WW=10;{:5}{19:}
ORDMAX=30;{:19}VAR{4:}
P:ARRAY[1..M]OF INTEGER;{:4}{7:}
PAGENUMBER:INTEGER;PAGEOFFSET:INTEGER;
ROWOFFSET:INTEGER;C:0..CC;{:7}{12:}J:INTEGER;
K:0..M;{:12}{15:}JPRIME:BOOLEAN;{:15}{17:}
ORD:2..ORDMAX;SQUARE:INTEGER;{:17}{23:}
N:2..ORDMAX;{:23}{24:}
MULT:ARRAY[2..ORDMAX]OF INTEGER;{:24}
BEGIN{3:}{11:}{16:}J:=1;K:=1;P[1]:=2;{:16}
{18:}ORD:=2;SQUARE:=9;{:18};
WHILE K<M DO BEGIN{14:}REPEAT J:=J+2;{20:}
IF J=SQUARE THEN BEGIN ORD:=ORD+1;{21:}
SQUARE:=P[ORD]*P[ORD];{:21}{25:}
MULT[ORD-1]:=J;{:25};END{:20};{22:}N:=2;
JPRIME:=TRUE;
WHILE(N<ORD)AND JPRIME DO BEGIN{26:}
WHILE MULT[N]<J DO MULT[N]:=MULT[N]+P[N]+P[N]
;IF MULT[N]=J THEN JPRIME:=FALSE{:26};N:=N+1;
END{:22};UNTIL JPRIME{:14};K:=K+1;P[K]:=J;
END{:11};{8:}BEGIN PAGENUMBER:=1;
PAGEOFFSET:=1;
WHILE PAGEOFFSET<=M DO BEGIN{9:}
BEGIN WRITE('The First ');WRITE(M:1);
WRITE(' Prime Numbers --- Page ');
WRITE(PAGENUMBER:1);WRITELN;WRITELN;
FOR ROWOFFSET:=PAGEOFFSET TO PAGEOFFSET+RR-1
DO{10:}
BEGIN FOR C:=0 TO CC-1 DO IF ROWOFFSET+C*RR<=
M THEN WRITE(P[ROWOFFSET+C*RR]:WW);WRITELN;
END{:10};PAGE;END{:9};
PAGENUMBER:=PAGENUMBER+1;
PAGEOFFSET:=PAGEOFFSET+RR*CC;END;END{:8}{:3};
END.{:2}{:1}
```

Figure 3. PASCAL program generated from the WEB file.

G. ADDITIONAL BELLS AND WHISTLES

A system like WEB can be successful only if it is capable of handling large programs as well as small ones, and only if it is complete enough to take care of all the practical requirements that arise when many different kinds of programs are considered. A small example like PRIMES.WEB is a satisfactory vehicle for illustrating the general ideas, but it cannot be convincing as a demonstration of WEB's ability to produce quality software in the "real world." My original design of WEB in September, 1981, was followed by a year of extensive experiments, so that by the time Version 1 was released in September, 1982, I could be fairly confident that the language was reasonably complete. Since then only one or two small extensions have proved to be necessary; and although numerous enhancements can easily be imagined, I believe that a useful stopping point for a working system called WEB83 has now been reached.

A full description of WEB83 appears in a Stanford report,[4] which also contains the complete WEB programs for WEAVE and TANGLE. The full language contains only a few features that do not show up in the PRIMES example considered above:

```
\input webmac
\font\ninerm=amr9
    :
syntactically correct \PASCAL\ program will
be obtained.\]

\Y\P$\4\X2:Program to print the first
thousand prime numbers\X\S$\6
\4\&{program}\1\  \37$\\{print\_primes}(%
\\{output})$;\6
\4\&{const} \37$\|m=1000$;\5
\X5:Other constants of the program\X\6
\4\&{var} \37\X4:Variables of the program\X\6
\&{begin} \37\X3:Print the first \|m prime
numbers\X;\6
\&{end}.\par
\U section~1.\fi
    :
The first three macro definitions here are
parametric; the other two are simple.\]

\Y\P\D \37$\\{print\_string}(\#)\S\\{write}(%
\#)$\C{put a given string into the %
\\{output} file}\par
    :
\inx
\:{Bertrand, Joseph, postulate}, 21.
\:\\{boolean}, 15.
    :
\:\.{WEB}, 1.
\:\\{write}, 6.
\:\\{write\_ln}, 6.
\:\\{ww}, \[5], 6.
\fin
    :
\:\X4, 7, 12, 15, 17, 23, 24:Variables of
the program\X
\U section~2.
\con
```

Figure 4. TEX program generated from the WEB file.

1) There are facilities to override WEAVE's automatic formatting of PASCAL programs. For example, it is possible to force a statement to begin on a new line, or to force several statements to appear on the same line, or to suggest a desirable breakpoint in the middle of a long expression. In unusual cases, WEAVE must parse program fragments that are not syntactically complete—for example, there may be a **begin** without a matching **end**—so a WEB user must be given a chance to control the results. Furthermore there is a facility for changing WEAVE's formatting rules by declaring that a certain identifier should be treated as a certain PASCAL reserved word, or by declaring that a certain reserved word should be treated as an ordinary identifier.

2) There is a way to force TANGLE to omit a space between two adjacent pieces of text, so that a name like '$x3$' can be manufactured from 'x' and '3'. Similarly, there is a way to pass an arbitrary sequence of characters through TANGLE so that the same sequence will appear "verbatim" in the PASCAL file; and there is a way to force beginning-of-line in that file. The latter extensions have proved to be necessary to deal with various nonstandard conventions of different PASCAL compilers. When a comment in braces is sent to the PASCAL file, TANGLE is careful not to introduce further braces inside the comment.

3) There are facilities for octal and hexadecimal constants in WEB thees. TANGLE converts such constants to decimal form; WEAVE gives them an appropriate typographic treatment.

4) There is a facility for dealing with alphabetic constants. When a program contains a double-quoted character like "A", TANGLE converts this to an integer between 0 and 127 that equals the corresponding ASCII code (in this case 65). The use of ASCII code facilitates the construction of software that is readily portable from one machine to another, independent of the actual character set in use.

5) Furthermore, if a double-quoted constant is a string of several characters, like "cat", TANGLE converts it into a unique integer that is 128 or more. A special *string pool file* is written, containing all of the strings that have been specially encoded in this way. I have used this general mechanism only in large programs, but experience has shown that it makes quite a nice substitute for the string-processing capabilities that PASCAL lacks. (Incidentally, I noticed after several months that a program needs to have some indication that the string-pool file it is reading contains the same strings that TANGLE generated when the program itself was tangled. Therefore a "check sum" is included in the string pool file; each program is able to refer to its own check sum and to compare it with the value in the file. This check-sum extension was one of the last features to be added to WEB.)

6) The PRIMES example illustrates macros with parameters and macros without parameters. WEB also allows "numeric" macros, which are small integer constants; TANGLE is capable of doing simple arithmetic on such constants. This feature of WEB was introduced specifically to overcome PASCAL's unfortunate inability to do compile-time arithmetic. For example, it is impossible to have a PASCAL array whose bounds are '$0 .. n-1$', or to write '$20+3$:' as the label of one of the cases in '**case** $x+y$'; WEB's numeric macros make it possible for TANGLE to preprocess such constants.

H. OCCAM'S RAZOR

I would also like to mention several things that were intentionally left out of WEB, since I have tried to keep the language as simple as I could.

There are no "conditional macros," nor does TANGLE evaluate Boolean expressions that might influence the output. I found that everything I needed could be done satisfactorily by commenting out the optional code.

For example, a system program is often designed to gather statistics about its own operation, but such statistics-gathering is pointless unless someone is actually going to use the results. In order to make

the instrumentation code optional, I include the word '**stat**' just before any special code for statistics, and '**tats**' just after such code; and I tell WEAVE to regard **stat** and **tats** as if they were **begin** and **end**. But **stat** and **tats** are actually simple macros. When I do want to gather the statistics, I define **stat** and **tats** to be null; but in a production version of the software, I make **stat** expand to '@{' and **tats** expand to '@}', where @{ and @} are special braces that TANGLE does not remove. Thus the optional code appears as a harmless comment in the PASCAL program.

WEB's macros are allowed to have at most one parameter. Again, I did this in the interests of simplicity, because I noticed that most applications of multiple parameters could in fact be reduced to the one-parameter case. For example, suppose that you want to define something like

```
mac(#1,#2) == m[#1*r+#2]
```

which WEB doesn't permit. You can get essentially the same result with two one-parameter macros

```
mac_tail(#) == #]
mac(#) == m[#*r+mac_tail
```

since, e.g., 'mac(a)(b)' will expand into 'm[a*r+b]'.

Here is another example that indicates some of the surprising generality of one-parameter macros: Consider the two definitions

```
define two_cases(#)==case j of
              1:#(1); 2:#(2); end
define reset_file(#)==reset(file@#)
```

where '@&' in the second definition is the concatenation operation that pastes two texts together. You can now say

```
two_cases(reset_file)
```

and the resulting PASCAL output will be

```
case j of
1:reset(file1);
2:reset(file2);
end
```

In other words, the name of one macro can usefully be a parameter to another macro. This particular trick makes it possible to live with PASCAL compilers that do not allow arrays of files.

I. PORTABILITY

One of the goals of my TEX research has been to produce portable software, and the WEB system has been extremely helpful in this respect. Although my own work is done on a DEC-10 computer with Stanford's one-of-a-kind operating system, the software developed with WEB has already been transported successfully to a wide variety of computers made by other manufacturers (including IBM, Control Data, XEROX, Hewlett-Packard), and to a variety of different operating systems for those machines. To my knowledge, no other software of such complexity has ever been transported to so many different machines. It seems likely that TEX will soon be operating on all but the smallest of the world's computer systems.

To my surprise, the main bottleneck to portability of the TEXware has been the lack of suitable PASCAL compilers, because PASCAL has often been implemented without system programming in mind. Anybody who has a decent PASCAL compiler can install WEB (and all programs written in WEB) without great difficulty, essentially as follows:

1) Start with the three files WEAVE.WEB, TANGLE.WEB, and TANGLE.PAS. (The programs have not been copyrighted, so these files are not difficult to obtain.)
2) Run TANGLE.PAS through your PASCAL compiler to get a working TANGLE program.
3) Check your TANGLE by applying it to TANGLE.WEB; your output file should match TANGLE.PAS.
4) Apply your TANGLE to the file WEAVE.WEB, obtaining WEAVE.PAS; then apply PASCAL to WEAVE.PAS and you'll have a working WEAVE system.
5) The same process applies to any software written in WEB, notably to TEX itself. (However, you need fonts and suitable output equipment in order to make proper use of TEX; that may be another bottleneck.) Once you have TEX working, you can apply WEAVE and TEX to your WEB files, thereby getting program documents as illustrated above.

Notice that a TANGLE.PAS file is needed in order to get this "bootstrapping" process started. If you have just WEAVE.WEB and TANGLE.WEB, you can't tangle them until you have TANGLE.

However, anybody who has looked seriously into the question of software portability will realize that my comments in the preceding paragraphs have been oversimplified. I have glossed over some serious problems that arise: Character sets are different; file naming conventions are different; special conventions are needed to interact with a user's terminal; data is packed differently on different machines; floating-point arithmetic is always nonstandard and sometimes nonexistent; users want "friendly" interaction with existing programs for editing and spooling; etc., etc. Furthermore, many of the world's PASCAL compilers are incredibly bizarre. Therefore it is quite naïve to believe that a single program TANGLE.PAS could actually work on very many different machines, or even that one single source file TANGLE.WEB could be adequate; some system-dependent changes are inevitable.

The WEB system caters to system-dependent changes in a simple but surprisingly effective way that I neglected to mention when I listed its other features. Both TANGLE and WEAVE are designed to work with *two* input files, not just one: In addition to a WEB source file like TEX.WEB, there is also a "change file" TEX.CH that contains whatever changes are needed to customize TEX for a particular system. (Similarly, the source files WEAVE.WEB and TANGLE.WEB are accompanied by WEAVE.CH and TANGLE.CH.)

Here's how change files work: Each change has the form "replace $x_1 \ldots x_m$ by $y_1 \ldots y_n$," for some $m \geq 1$ and $n \geq 0$; here x_i and y_j represent lines in the change file. The WEAVE and TANGLE programs read data from the WEB input file until finding a line that matches x_1;

this line, and the $m-1$ following lines, are replaced by $y_1 \ldots y_n$. An error message is given if the m lines replaced did not match $x_1 \ldots x_m$ perfectly.

For example, the program PRIMES.WEB invokes a *page* procedure to begin a new page; but *page* was not present in Wirth's original PASCAL and it is defined rather vaguely in the PASCAL standard. Therefore a system-dependent change may be needed here. A change file PRIMES.CH could be made by copying the line

```
@d new_page==page
```

from Figure 2c and specifying one or more appropriate replacement lines.

The program TANGLE itself contains about 190 sections, and a typical installation will have to change about 15 of these. If you want to transport TANGLE to a new environment, you therefore need to create a suitable file TANGLE.CH that modifies 15 or so parts of TANGLE.WEB. (Examples of TANGLE.CH are provided to all people who receive TANGLE.WEB, so that each implementor has a model of what to do.) You need to insert your changes by hand into TANGLE.PAS, until you have a TANGLE program that works sufficiently well to support further bootstrapping. But you never actually change the master file TANGLE.WEB.

This approach has two important advantages. First, the same master file TANGLE.WEB is used by everybody, and it contains the basic logic of TANGLE that really defines the essence of tangling. The system-dependent changes do not affect any of the subtle parts of TANGLE's control structures or data structures. Second, when the official TANGLE has been upgraded to a newer version, a brand new TANGLE.WEB will almost always work with the old TANGLE.CH, since changes are rarely made to the system-dependent parts. In other words, this dual-input-file scheme works when the WEB file is constant and the CH file is modified, and it also works when the CH file is constant but the WEB file is modified.

Change files were added to WEB about three months after the system was initially designed, based on our initial experiences with people who had volunteered to participate in portability experiments. We realized about a year later that WEAVE could be modified so that only the changed parts of a program would (optionally) be printed; thus, it's now possible to document the changes by listing only the sections that are actually affected by the CH file that WEAVE has processed. We also generalized the original format of CH files, which permitted only changes that extended to the end of a section. These two important ideas were among the final enhancements incorporated into WEB83.

J. PROGRAMS AS WEBS

When I first began to work with the ideas that eventually became the WEB system, I thought that I would be designing a language for "top-down" programming, where a top-level description is given first and successively refined. On the other hand I knew that I often created major parts of programs in a "bottom-up" fashion, starting with the definitions of basic procedures and data structures and gradually building more and more powerful subroutines. I had the feeling that top-down and bottom-up were opposing methodologies: one more suitable for program exposition and the other more suitable for program creation.

But after gaining experience with WEB, I have come to realize that there is no need to choose once and for all between top-down and bottom-up, because a program is best thought of as a web instead of a tree. A hierarchical structure is present, but the most important thing about a program is its structural relationships. A complex piece of software consists of simple parts and simple relations between those parts; the programmer's task is to state those parts and those relationships, in whatever order is best for human comprehension—not in some rigidly determined order like top-down or bottom-up.

When I'm writing a longish program like TANGLE.WEB or WEAVE.WEB or TEX.WEB, I invariably have strong feelings about what part of the whole should be tackled next. For example, I'll come to a point where I need to define a major data structure and its conventions, before I'll feel happy about going further. My experiences have led me to believe that a person reading a program is, likewise, ready to comprehend it by learning its various parts in approximately the order in which it was written. The PRIMES.WEB example illustrates this principle on a small scale; the decisions that Dijkstra made as he composed the original program[2] appear in the WEB documentation in the same order.

Top-down programming gives you a strong idea of where you are going, but it forces you to keep a lot of plans in your head; suspense builds up because nothing is really nailed down until the end. Bottom-up programming has the advantage that you continually wield a more and more powerful pencil, as more and more subroutines have been constructed; but it forces you to postpone the overall program organization until the last minute, so you might flounder aimlessly.

When I tear up the first draft of a program and start over, my second draft usually considers things in almost the same order as the first one did. Sometimes the "correct" order is top-down, sometimes it is bottom-up, and sometimes it's a mixture; but always it's an order that makes sense on expository grounds.

Thus the WEB language allows a person to express programs in a "stream of consciousness" order. TANGLE is able to scramble everything up into the arrangement that a PASCAL compiler demands. This feature of WEB is perhaps its greatest asset; it makes a WEB-written program much more readable than the same program written purely in PASCAL, even if the latter program is well commented. And the fact that there's no need to be hung up on the question of top-down versus bottom-up—since a programmer can now view a large program as a web, to be explored in a psychologically correct order—is perhaps the greatest lesson I have learned from my recent experiences.

Another surprising thing that I learned while using WEB was that traditional programming languages had been causing me to write inferior programs, although I hadn't realized what I was doing. My original idea was that WEB would be merely a tool for documentation, but I actually found that my WEB programs were better than the programs I had been writing in other languages. How could this be?

Well, imagine that you are writing a small subroutine that updates part of a data structure, and suppose that the updating takes only one or two lines of code. In practical programs, there's often something that can go wrong, if the user's input is incorrect, so the subroutine has to check that the input is correct before doing the update. Thus, the subroutine has the general form

```
procedure update;
begin if ⟨input data is invalid⟩ then
  ⟨Issue an error message and try to recover⟩;
⟨Update the data structure⟩;
end.
```

A subtle phenomenon occurs in traditional programming languages: While writing the program for '⟨Issue an error message and try to recover⟩', a programmer subconsciously tries to get by with the fewest possible lines of code, since the program for '⟨Update the data structure⟩' is quite short. If an extensive error recovery is actually programmed, the subroutine will appear to have error-message printing as its main purpose. But the programmer knows that the error is really an exceptional case that arises only rarely; therefore a lengthy error recovery doesn't look right, and most programmers will minimize it (without realizing that they are doing so) in order to make the subroutine's appearance match its intended behavior. On the other hand when the same task is programmed with WEB, the purpose of *update* can be shown quite clearly, and the possibility of error recovery can be reduced to a mere mention when *update* is defined. When another section entitled '⟨Issue an error message and try to recover⟩' is subsequently written, the whole point of that section is to do the best error recovery, and it becomes quite natural to write a better program.

This fact—that WEB allows you to let each part of the program have its appropriate size, without distorting the readability of other parts—means that good programmers find their WEB programs better than their PASCAL programs, even though their PASCAL programs once seemed to be the work of an expert.

K. STYLISTIC ISSUES

I found that my style of using WEB evolved quite a bit during the first year. The general format, in which each section beings with commentary and ends with a formal program fragment, is extremely versatile; you have the freedom to say anything you want, and this freedom entails the responsibility of deciding what to say. I imagine that different programmers will converge to quite different styles, but I would like to note down some of the things that have seemed to work best for me.

Consider first the question of macros versus section names. A named section, like '⟨Issue an error message and try to recover⟩', is essentially the same as a parameterless macro; WEB provides both. I prefer to use parameterless macros for "small" things that can be embodied in a word or two, but named sections for longer portions of the program that merit a fuller description.

I usually start the name of a section with an imperative verb, but I give a declarative commentary at the beginning of a section. Thus, PRIMES.WEB says '**8.** Now that appropriate ... ⟨Print table p 8⟩ ≡ ... '; I wouldn't do the opposite and say '**8.** Print the table. ⟨Code for printing 8⟩ ≡ ...'.

The name of a section (enclosed in angle brackets) should be long enough to encapsulate the essential characteristics of the code in that section, but it should not be too verbose. I found very early that it would be a mistake to include all of the assumptions about local and global variables in the name of each section, even though such information would strictly be necessary to isolate that section as an independent module. The trick is to find a balance between formal and informal exposition so that a reader can grasp what is happening without being overwhelmed with detail.[5]

Another lesson I learned early in the game was that the name of a section should explicitly mention any nonstandard control structures, even though its data structures can often be left implied. Furthermore, if the control flow is properly explained, you can avoid the usual errors associated with **goto** statements; such statements can safely be introduced in a restrained but natural manner.

For example, §14 of the prime-printing example could be reprogrammed as follows, using '**loop**' as a macro abbreviation for '**while** *true* **do**':

```
⟨Increase j until it is the next prime number 14⟩ ≡
  loop begin j ← j + 2;
    ⟨Update variables that depend on j 20⟩;
    ⟨If j is prime, goto found 22⟩;
    end;
found:
```

With this change, §22 could become

```
⟨If j is prime, goto found 22⟩ ≡
  n ← 2;
  while n < ord do
    begin ⟨If p[n] is a factor of j, goto not_found 26⟩;
    n ← n + 1;
    end;
  goto found;
not_found:
```

if §26 changes in the obvious way. The resulting program will be more efficient on most machines; and I believe that it is actually easier to read and to write, in spite of the fact that two **goto** statements appear, because the labels have been used with appropriate interpretations of their abstract significance.

Of course, PASCAL makes it difficult to use **goto** statements, because Wirth decided that labels should be numeric, and that they should be declared in advance. If I were to introduce the **goto** statements as suggested, I would have to define numeric macros *found* and *not_found*, and I would have to insert '**label** *found*, *not_found*' into the program at the right place. Such extra work is a bit of a nuisance, but it can be done in WEB without spoiling the exposition.

PASCAL has a few other misfeatures that prove to be inconvenient with respect to WEB exposition. The worst of these is the inability to declare local variables in the midst of a program or procedure. For example, a programmer often finds it most natural to define an

integer variable when a **for** loop is introduced, but the rules of PASCAL insist that such a variable be declared rather far away from that **for** loop. My WEB programs overcome this problem by having sections like '⟨ Local variables for *xyzzy* ⟩' whenever there's a rather lengthy procedure '*xyzzy*' whose local variables should not be declared all at once. But when a procedure is short, say only half a dozen sections long, there's usually no harm in declaring its local variables in PASCAL style, because the entire text of the procedure will tend to appear on one or two adjacent pages of the WEB documentation.

Another slightly awkward aspect of PASCAL is its treatment of semicolons. If you look closely at the prime-number example, you'll see that I had to be a bit careful about where I put semicolons; sometimes they occur at the end of the expanded text of a section, but usually they don't. With a little self discipline, a person can learn to do this quite satisfactorily, but it is a nuisance until you get used to it.

L. ECONOMIC ISSUES

What does it cost to use WEB? Let's look first at the lowest level, where computer costs are considered, because it is easy to make quantitative statements at this level. The running time to TANGLE a WEB file is approximately the same as the time needed to compile the resulting PASCAL program; hence the extra preprocessing does not cost much. Similarly, WEAVE doesn't take long to produce a file for TEX. However, TEX needs a comparatively large amount of time to typeset the final document. For example, if we assume that each page requires four seconds, it will take four minutes to produce a 60-page document. The running time for WEAVE-plus-TEX is quite reasonable when you consider that your program is effectively being converted into a fairly substantial booklet; but the costs are sufficiently large to discourage remaking and reprinting such a booklet more than once or twice a day. When a new program is being developed, it is therefore customary to work with hardcopy documentation that is slightly obsolete, and to read the WEB source file itself when up-to-date information is required; the source file is sufficiently easy to read for such purposes.

The costs of WEB are more difficult to estimate at higher levels, but I have found to my surprise that the total time of writing and debugging a WEB program is no greater than the total time of writing and debugging an ALGOL or PASCAL program, even though my WEB programs are much better, and even though I am putting substantially more documentation into the programs. Therefore I have lately been using WEB for all of my programming, even for one-off jobs that I write "for my eyes only" just to explore occasional problems. The extra time I spend in preparing additional commentary is regained because the debugging time is reduced.

In retrospect, the fact that a "literate" program takes much less time to debug is not surprising, because the WEB language encourages a discipline that I was previously unwilling to impose on myself. I had known for a long time that the programs I construct for publication in a book, or the programs that I construct in front of a class, have tended to be comparatively free of errors, because I am forced to clarify my thoughts as I do the programming. By contrast, when writing for myself alone, I have often taken shortcuts that proved later to be dreadful mistakes. It's harder for me to fool myself in such ways when I'm writing a WEB program, because I'm in "expository mode" (analogous to classroom lecturing) whenever a WEB is being spun. Ergo, less debugging time.

Now that I am writing all my programs in WEB, an unforeseen problem has, however, arisen: I suddenly have a collection of programs that seem quite beautiful in my own eyes, and I have a compelling urge to publish all of them so that everybody can admire these works of art. A nice little 10-page program can easily be written and debugged in an afternoon and evening; if I keep accumulating such gems, I'll soon run out of storage space, and my office will be encrusted with webs of my own making. There is no telling what will happen if lots of other people catch WEB fever and start foisting their creations on each other. I can already envision the appearance of a new journal, to be entitled *Webs*, for the publication of literate programs; I imagine that it will have a large backlog and a large group of dedicated editors and referees.

M. RELATED WORK

Nothing about WEB is really new; I have simply combined a bunch of ideas that have been in the air for a long time. I would like to summarize in the next few paragraphs the things that had the greatest influence on my thinking as I put those pieces together.

George Forsythe wrote in 1966 that "A useful algorithm is a substantial contribution to knowledge. Its publication constitutes an important piece of scholarship."[6] His comments have always inspired me to strive for excellence in programming, and they have played a major rôle in shaping my present view that it is worthwhile to consider *every* program as a work of literature.

The design of WEB was influenced primarily by the pioneering work of Pierre-Arnoul de Marneffe,[7,8] whose research on what he called "Holon Programming" has not received the attention it deserves. His work was, in turn, inspired by Arthur Koestler's excellent treatise on the structure of complex systems and organisms;[9] thus we have another connection between programming and literature. A somewhat similar system was independently created by Edwin Towster.[10]

I owe a great debt to Edsger Dijkstra, Tony Hoare, Ole-Johan Dahl, and Niklaus Wirth for opening my eyes to the importance of abstraction in the reading and writing of programs, and to Peter Naur for stressing the importance of a balance between formal and informal methods of exposition.

Tony Hoare provided a special impetus for WEB when he suggested in 1978 that I should publish my program for TEX. Since very few large-scale software systems were available in the literature, he had been trying to promote the publication of well-written programs. Hoare's suggestion was actually rather terrifying to

me, and I'm sure he knew that he was posing quite a challenge. As a professor of computer science, I was quite comfortable publishing papers about toy problems that could be polished up nicely and presented in an elegant manner; but I had no idea how to take a piece of real software, with all the compromises necessary to make it useful to a large class of people on a wide variety of systems, and to open it up to public scrutiny. How could a supposedly respectable academic, like me, reveal the way he actually writes large programs? And could a large program be made intelligible? My previous attempts along these lines[11] were by now hopelessly out of date. I decided that this would be a good time to try out de Marneffe's ideas; furthermore, the TEX system itself provided me with new tools for printing and format control, so I suspected that it would be possible to obtain state-of-the-art documentation by making proper use of typography.

It is interesting to reread some of the comments that Tony made ten years ago in his keynote address to the first ACM symposium on Principles of Programming Languages:[12]

> Documentation must be regarded as an integral part of the process of design and coding. A good programming language will encourage and assist the programmer to write clear, self-documenting code, and even perhaps to develop and display a pleasant style of writing.

He foresaw many future trends, but not the impending improvements in typesetting quality:

> It is of course possible for a compiler or service program to expand the abbreviations, fill in the defaults, and make explicit the assumptions. But in practice, experience shows that it is very unlikely that the output of a computer will ever be more readable than its input, except in such trivial but important aspects as improved indentation.

Typographic formatting of computer programs has a long tradition, originating with ALGOL and its immediate precursors. I'm not sure who made the first experiments, but I believe that the lion's share of the credit for developing excellent programming-language typography belongs to two people: Peter Naur, who edited the ALGOL 60 report[13] and gave special care to its presentation; and Myrtle Kellington, who served for many years as executive editor of ACM publications and set the standards that have been adopted by other journals. The computing profession owes much to these people, who made published programs so much more readable than they would otherwise have been; the magnitude of their contribution can only be appreciated by people who submit computer programs to journals like *Acta Arithmetica* whose editors are unfamiliar with computer science. Bill McKeeman called attention to formatting issues when he published Algorithm 268, "ALGOL 60 reference language editor," in 1965.[14] There has been a flowering of such algorithms in recent years; the papers by Oppen[15] and by Rose and Welsh[16] are particularly noteworthy.

I began to design WEB in the spring of 1979, when I constructed a prototype system that was called DOC. Luis Trabb Pardo helped me to develop a suitable style of exposition at that time; then Ignacio Zabala Salelles gave a DOC a thorough test when he prepared a full implementation of TEX in PASCAL. Zabala's implementation was successfully transported to many different computers,[17–20] and this experience was of immense value to me when I cast WEB into its present form in 1981. Since then many significant improvements have been suggested by my colleague David R. Fuchs, and I have also benefited from the experiences of many volunteers who were willing to be guinea pigs for pre-released versions of TEX. It's impossible for me to name everyone who has helped, but I would like to give special thanks to Arthur Samuel, Howard Trickey, Joe Weening, and Pierre MacKay for important contributions. I'm fortunate indeed to share a working environment with such stimulating people.

When I originally designed the WEB system, I spent about six weeks preparing the files TANGLE.WEB and WEAVE.WEB, during which time I was continually changing the language and trying different styles of exposition. (The programs were neither long nor complicated, but this was rather intensive work, so I didn't get much else done during those six weeks. The first two weeks were actually spent drafting the first ten per cent of what is now TEX.WEB.) Then I spent about six tedious hours with a text editor, hand-simulating the behavior of TANGLE on TANGLE.WEB, so that I had a program TANGLE.PAS that was ripe for debugging. At first I had to correct errors both in TANGLE.WEB and TANGLE.PAS, but soon TANGLE was working well enough that I needed only TANGLE.WEB as a source file. Then WEAVE.WEB could be tangled and debugged too. The total time to create "Version 0" of the WEB system, including the language design and the time to debug the programs and write a brief manual for users, was about eight weeks; then enhancements were added at the rate of about one per month for the next 18 months. As a result of this experience I think it's reasonable to state that a WEB-like system can be created from scratch in a fairly short time, for some other pair of languages besides TEX and PASCAL, by an expert system programmer who is conversant with both languages. Indeed, I spoke about WEB on a recent visit to London and one of the people in the audience decided to test this hypothesis; shortly afterwards I received an elegant report from Harold Thimbleby, who had just constructed an excellent system called Cweb, based on Troff/Nroff and C instead of TEX and PASCAL.[21]

N. RETROSPECT AND PROSPECTS

Enthusiastic reports about new computer languages, by the authors of those languages, are commonplace. Hence I'm well aware of the fact that my own experiences cannot be extrapolated too far. I also realize that, whenever I have encountered a problem with WEB, I've simply changed the system; other users of WEB cannot operate under the same ground rules.

However, I believe that I have stumbled on a way of programming that produces better programs that are more portable and more easily understood and maintained; furthermore, the system seems to work with large programs as well as with small ones. I'm

pleased that my work on typography, which began as an application of computers to another field, has come full circle and become an application of typography to the heart of computer science; I like to think of WEB as a neat "spinoff" of my research on TEX. However, all of my experiences with this system have been highly colored by my own tastes, and only time will tell if a large number of other people will find WEB to be equally attractive and useful.

I made a conscious decision not to design a language that would be suitable for everybody. My goal was to provide a tool for system programmers, not for high school students or for hobbyists. I don't have anything against high school students and hobbyists, but I don't believe every computer language should attempt to offer all things to all people. A user of WEB needs to be good enough at computer science that he or she is comfortable dealing with several languages simultaneously. Since WEB combines TEX and PASCAL with a few rules of its own, WEB programs can contain WEB syntax errors, TEX syntax errors, PASCAL syntax errors, and algorithmic errors; in practice, all four types of errors occur, and a bit of sophistication is needed to sort out which is which. Computer scientists tend to be better at such things than other people. I have found that WEB programs can be debugged rapidly in spite of the profusion of languages, but I'm sure that many other intelligent people will find such a task comparatively difficult.

In other words, WEB seems to be specifically for the peculiar breed of people who are called computer scientists. And I'm pretty sure that there are also a lot of computer scientists who will not enjoy using WEB; some of us are glad that traditional programming languages have comparatively primitive capabilities for inserted comments, because such difficulties provide a good excuse for not documenting programs well. Thus, WEB may be only for the subset of computer scientists who like to write and to explain what they are doing. My hope is that the ability to make explanations more natural will cause more programmers to discover the joys of literate programming, because I believe it's quite a pleasure to combine verbal and mathematical skills; but perhaps I'm hoping for too much. The fact that at least one paper has been written that is a syntactically correct ALGOL 68 program[22] encourages me to persevere in my hopes for the future. Perhaps we will even one day find Pulitzer prizes awarded to computer programs.

And what about the future of WEB? If the next year or so of trial use shows that a lot of other people besides myself become "hooked" on this method of programming, there will be many ways to incorporate the WEB philosophy into a really effective programming environment. For example, it will be worthwhile to produce a unified system that does both tangling and compiling, instead of using separate programs as in Figure 1; and it will also be worthwhile to carry the unification one step further, so that run-time debugging as well as syntactic debugging can be done entirely in terms of the WEB source language. Furthermore, a WEB-like system could be designed to incorporate additional modularization, so that it would be easier to compile different parts of a program independently. The new generation of graphic workstations makes it desirable to display selected program sections on demand, by using TEX only on the sections that are of current interest, instead of producing hardcopy for an entire document. And so on; a considerable amount of additional research and development will be appropriate if the idea of literate programming really catches on.

Acknowledgements

The preparation of this paper was supported in part by the National Science Foundation under grants IST-8201926 and MCS-8300984, and by the System Development Foundation. 'TEX' is a trademark of the American Mathematical Society.

REFERENCES

1. D. E. Knuth, *The TEXbook*. Addison-Wesley, Reading, Mass., U.S.A. (1983).
2. O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. Academic Press, London and New York (1972).
3. D. E. Knuth, Structured programming with **go to** statements. *Computing Surveys* **6**, 261–301 (1974).
4. D. E. Knuth, *The WEB System of Structured Documentation*. Stanford Computer Science Report CS980 (1983).
5. P. Naur, Formalization in program development. *BIT* **22**, 437–453 (1982).
6. G. E. Forsythe, Algorithms for scientific computation. *Communications of the ACM* **9**, 255–256 (1966).
7. P. A. de Marneffe, *Holon Programming*. Univ. de Liege, Service D'Informatique (December, 1973).
8. P. A. de Marneffe and D. Ribbens, Holon Programming, in A. Günther et al. (eds.), *International Computing Symposium 1973*, Amsterdam, North-Holland (1974).
9. A. Koestler, *The Ghost in the Machine*. New York, Macmillan (1968).
10. E. Towster, A convention for explicit declaration of environments and top-down refinement of data. *IEEE Transactions on Software Engineering* **SE-5**, 374–386 (1979).
11. D. E. Knuth, Computer-drawn flow charts. *Communications of the ACM* **6**, 555–563 (1963).
12. C. A. R. Hoare, *Hints on Programming Language Design*. Stanford Computer Science Report CS403 (1973).
13. P. Naur (ed.) et al., Report on the algorithmic language ALGOL 60. *Communications of the ACM* **3**, 299–314.
14. W. M. McKeeman, Algorithm 268. *Communications of the ACM* **8**, 667–668 (1965).
15. D. Oppen, Prettyprinting. *ACM Transactions on Programming Languages and Systems* **2**, 465–483 (1980).
16. G. A. Rose and J. Welsh, Formatted programming languages. *Software—Practice & Experience* **11**, 651–669 (1981).
17. I. Zabala and L. Trabb Pardo, The status of the PASCAL implementation of TEX. *TUGboat* **1**, 16–17 (1980).
18. I. Zabala, TEX-PASCAL and PASCAL compilers. *TUGboat* **2** (1), 11–12 (1981).
19. I. Zabala, Some feedback from PTEX installations. *TUGboat* **2** (2), 16–19 (1981).
20. I. A. Zabala, How portable is PASCAL? Draft of paper in preparation (1982).
21. H. Thimbleby, *Cweb*. Preprint, University of York (August 1983).
22. C. H. Lindsey, ALGOL 68 with fewer tears. *The Computer Journal* **15**, 176–188 (1972).

Received September 1983