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 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 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 section sections 7, 12, 15, 17, 23, and 24.

This code is used in section 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 section 19.

This code is used in section 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 '$+\equiv$' 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 section 3.

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

```
⟨ Output a page of answers 9 ⟩ ≡
  begin print_string('The␣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 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 $c \leftarrow 0$ to $cc - 1$ do
    if $row\_offset + c * rr \le m$ then
      $print\_entry(p[row\_offset + c * rr]);$
  $new\_line;$
  end
```

This code is used in section 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⟩;
    $k \leftarrow k + 1$; $p[k] \leftarrow j$;
    end
```

This code is used in section 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 $\le j$ are in table $p$}
$k$: $0 \mathrel{..} m$;   {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* $\leftarrow$ *true*; **for** $n \leftarrow 1$ **to** k **do** ⟨If $p[n]$ divides j, set *j_prime* $\leftarrow$ *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** $j \leftarrow j + 2$;
    ⟨ 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 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 section 18.

This code is used in section 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 ⟩ $+\equiv$

```
  $ord \leftarrow 2$; $square \leftarrow 9$;
```

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 ⟩ $+\equiv$

```
  $ord\_max = 30$;  { $p^2_{ord\_max}$ must exceed $p_m$ }
```

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

⟨ Update variables that depend on j 20 ⟩ $\equiv$

```
  if $j = square$ then
    begin $ord \leftarrow ord + 1$;
    ⟨ Update variables that depend on $ord$ 21 ⟩;
    end
```

This code is used in section section 14.

21. At this point in the program, *ord* has just been increased by unity, and we want to set $square := p^2_{ord}$. A surprisingly subtle point arises here: How do we know that p_{ord} has already been computed, i.e., that $ord \le k$? If there were a gap in the sequence of prime numbers, such that $p_{k+1} > p^2_k$ 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 $p_{k+1} <= p_1 \ldots p_k + 1$. 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 ⟩ $\equiv$

```
  $square \leftarrow p[ord] * p[ord]$;  { at this point $ord \le k$ }
```

See also section section 25.

This code is used in section 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 ⟩ ≡
  $n \leftarrow 2$; $j\_prime \leftarrow true$;
  while $(n < ord) \wedge j\_prime$ do
    begin ⟨ If $p[n]$ is a factor of $j$, set
          $j\_prime \leftarrow false$ 26 ⟩;
    $n \leftarrow n + 1$;
    end
```

This code is used in section 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 $2 \le n < ord$, $mult[n]$ 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 $\mathit{mult}[n]$ is an odd multiple of p_n such that $\mathit{mult}[n] < j + 2p_n$. If $\mathit{mult}[n] < j$, we can increase $\mathit{mult}[n]$ by $2p_n$ and the same conditions will hold. On the other hand if $\mathit{mult}[n] \geq j$, the conditions imply that j is divisible by p_n if and only if $j = \mathit{mult}[n]$.

```
⟨ If $p[n]$ is a factor of $j$, set $j\_prime \leftarrow \mathit{false}$ 26 ⟩ ≡
  **while** $\mathit{mult}[n] < j$ **do**
    $\mathit{mult}[n] \leftarrow \mathit{mult}[n] + p[n] + p[n]$;
  **if** $\mathit{mult}[n] = j$ **then** $j\_prime \leftarrow \mathit{false}$
```

This code is used in section 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.⟧

$\langle$ Fill table p with the first m prime numbers 11 $\rangle$ Used in section section 3.
$\langle$ Give to *j_prime* the meaning: j is a prime number 22 $\rangle$ Used in section section 14.
$\langle$ If $p[n]$ is a factor of j, set *j_prime* $\leftarrow$ *false* 26 $\rangle$ Used in section section 22.
$\langle$ Increase j until it is the next prime number 14 $\rangle$ Used in section section 11.
$\langle$ Initialize the data structures 16, 18 $\rangle$ Used in section section 11.
$\langle$ Other constants of the program 5, 19 $\rangle$ Used in section section 2.
$\langle$ Output a line of answers 10 $\rangle$ Used in section section 9.
$\langle$ Output a page of answers 9 $\rangle$ Used in section section 8.
$\langle$ Print table p 8 $\rangle$ Used in section section 3.
$\langle$ Print the first m prime numbers 3 $\rangle$ Used in section section 2.
$\langle$ Program to print the first thousand prime numbers 2 $\rangle$ Used in section section 1.
$\langle$ Update variables that depend on j 20 $\rangle$ Used in section section 14.
$\langle$ Update variables that depend on *ord* 21, 25 $\rangle$ Used in section section 20.
$\langle$ Variables of the program 4, 7, 12, 15, 17, 23, 24 $\rangle$ Used in section section 2.