Avoiding Recursion

Brian Harvey
University of California, Berkeley

It is traditional to understand a computer program as a *sequence* of instructions. The computer carries out these instructions one after another. Various *control structures* allow the programmer to change the order in which these instructions are carried out; there are conditional structures that allow a choice between subsequences, and looping structures that allow the repetition of a sequence.

In contrast to this idea of sequential programming there is a more recent model called *functional programming* [Backus, 1978; Allen, 1984; Abelson, 1985]. In this approach, a program is viewed as a collection of mathematical functions, which can be applied to arguments and composed with other functions to create more complicated functions. The emphasis is on the input-output behavior of a function, rather than on the sequence of events by which the computer determines the output value. Programming languages such as LISP, Logo, and APL emphasize the functional model, while BASIC, Pascal, and C emphasize the sequential view. (This division is not absolute; any program can be written in any language, more or less. But each language encourages a certain natural style of programming.)

In this paper I begin by arguing that functional programming is not only more powerful technically but also more sensible pedagogically, especially when computer programming is to be used as a vehicle for teaching mathematical ideas. Why, then, is secondary school programming done almost exclusively in BASIC and Pascal? Why not Logo? Although there are several answers to this question, I want to focus attention on one technical reason: Teachers are put off by Logo's heavy reliance on recursion as the principal control mechanism. I shall discuss the relationship between functional programming and recursion, show how the two can be separated, and suggest specific approaches to sample mathematical programming problems.

Recursion is itself an interesting mathematical idea, because of its connection with proof by induction. I don't mean to suggest that mathematics teachers should wish to avoid recursion in every situation. Indeed, later we'll consider examples (e.g., computing the determinant of a matrix) for which the use of recursion is naturally connected with the task. Logo is, of course, an excellent vehicle for the study of recursion when that is desired. But not every topic is inherently recursive, and the value of functional programming will be enhanced if the issue of recursion can be separated from that of functions.

Sequential and Functional Programming

Consider the problem of adding two matrices:

$$\begin{pmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{pmatrix} + \begin{pmatrix} 87 & 0 & 14 \\ -6 & 28 & 9 \end{pmatrix}$$

The result is a matrix with the same shape as the two addends. Each element of the result is formed by adding the corresponding elements of the addends; in sequential programming this leads to six assignment instructions, which can be arranged in two nested loops to reflect the two-dimensional structure of the matrix. For now let's ignore the matter of reading these particular numbers into the computer, and suppose we have two arrays `A` and `B` containing the addends. A BASIC program to add these matrices might be written as follows.

```
10 DIMENSION A(2,3), B(2,3), C(2,3)
20 FOR I = 1 TO 2
30   FOR J = 1 TO 3
40     C(I,J) = A(I,J) + B(I,J)
50   NEXT J
60 NEXT I
70 END
```

This is a simple enough program. It has two main weaknesses. One has to do with the use of arrays to represent the matrices. The problem is that the particular dimensions, two rows by three columns, must be built into the program, both in the array declarations and in the control structures (the `FOR` statements) that control the sequence of events. The second weakness is that this program expects to find its data in fixed places, the arrays `A` and `B`, and leaves its result in a third predetermined location `C`. This is fine if we only want to compute $A + B$, but what if we want to solve a slightly different problem, like $A + 2B$? What if we want to check that addition of matrices is truly commutative, by adding first $A + B$ and then $B + A$? Or, having computed C as the sum of A and B, what if we want to use that result as part of a larger computation, like $C + B$? In each case we have to rewrite the program.

A better interactive system for students to use in exploring matrix arithmetic would supply functions whose domain and range are matrices. We should be able to enter instructions like

```
PRINT ADD(A,B)
PRINT ADD(A, ADD(B,B))
LET C = ADD(A,B)
PRINT ADD(C,B)
IF ADD(A,B) = ADD(B,A) THEN PRINT "IT'S COMMUTATIVE."
```

(In this fanciful example I am using the syntax of BASIC, but asking for computations that are beyond the powers of most versions of BASIC. Many versions do allow this sort of functional notation, but only for certain functions of single numbers, not array-valued functions.)

Although functions are one of the central ideas of mathematics, they are not limited to the narrow school version of mathematics as "stuff about numbers." Consider the functions

plural(box) = boxes
French(book) = livre
capital(England) = London
AtomicWeight(silver) = 107.87

I shall resist the temptation to explore possible pedagogic uses of functional programming in the areas suggested by these simple functions. A serious effort to translate English into French, for example, would need much more sophistication than the word-for-word translation function illustrated above. The point is that no matter what a computation is about, the ability to define, invoke, and compose functions is a natural and convenient tool.

Computer scientists are excited about functional programming for reasons unconnected with pedagogy. To a computer scientist, what distinguishes this methodology from traditional approaches is that a function has no "side effects." That is, when a programmer asks for the value of $f(7)$ the value returned is always the same. This computation does not cause a permanent change in the computer's memory that might affect later computations. (For example, there are no assignments to global variables.) This is important for two reasons. First, it is easier to carry out formal reasoning about a program without side effects. One branch of computer science is concerned with program *verification*: the attempt to provide formal proof of the correctness of a program. Functional programs lend themselves to this attempt more readily than traditional sequential programs that are heavily dependent on long-term variable assignment. Second, if the computation of a function has no side effects, then it makes no difference which of two desired computations is done first; each is guaranteed to be independent of the other. Recent developments in computer hardware include *parallel* processors, in which the two computations might actually take place at the same time. Functional programs can easily take advantage of this parallelism; traditional programs require detailed analysis before any subtask can be split off for parallel computation.

For our pedagogic purposes, however, the important points about functional programming are different. A student who wanted to use the BASIC instructions shown above to add two matrices in a larger program would have to remember that the result is in the array `C` and write the rest of the program accordingly. Two such addition problems with different matrices would require two copies of the instructions or reuse of the same arrays for different purposes. A function that can be applied to any two matrices, and whose return value can be used as part of a larger expression instead of being tied to a specific result array, allows the student to think about the mathematics instead of tripping over the programming language.

First-Class Data Aggregates

Consider now the problem of entering the actual matrix values into the program. In most programming languages, the facilities for dealing with individual numbers (scalars) are much more flexible than the facilities for arrays. It is as if numbers are "real things" while collections of numbers are less real. For example, in BASIC or Pascal a scalar constant can be directly assigned to a variable (`X = 3`) or can be part of a larger expression (`X = Y+3`), but you can't say

```
A = 2,3,4,5,6,7
```

to assign a value to an entire array at once. Instead, getting this value into the computer is a sequence of events in itself:

```
10 DIMENSION A(2,3)
20 FOR I = 1 TO 2
30   FOR J = 1 TO 3
40     READ A(I,J)
50   NEXT J
60 NEXT I
70 DATA 2,3,4,5,6,7
80 END
```

Logo's list processing makes matrix manipulation much more convenient. For one thing, the size of a list need not be declared in advance; we can write matrix functions that work on matrices of any size or shape. Also, a list is a "first-class" entity: Like a number, it can be used as part of an expression, as an input to or output from a function, as the value assigned to a variable, or as something to be read or printed.

We'll represent a matrix as a list of lists. Each of the sublists will be one row of the matrix. We can solve the addition problem posed above by assigning matrix values to variables:

```
MAKE "A [[2 3 4] [5 6 7]]
```

```
MAKE "B [[87 0 14] [-6 28 9]]
SHOW ADD :A :B
```

or we can use the matrix values directly as inputs to the addition function:

```
SHOW ADD [[2 3 4] [5 6 7]]  [[87 0 14] [-6 28 9]]
```

The brackets that delimit the sublists make the two-by-three shape of the matrix immediately clear to a human reader as well as to the computer program.

Adding Matrices in Logo

I have indicated some of the reasons why Logo is a good language in which to add matrices, but I have not yet gotten around to writing the actual program. I shall now do so, in a traditional Logo style. The two-dimensional nature of a matrix is reflected in a two-procedure program structure. The top-level procedure goes through the two input matrices row by row, adding pairs of rows to produce a row of the result. The subprocedure that adds a row has the same general structure; it goes through two vectors (that's what a row of a matrix is) element by element, adding pairs of numbers.

```
TO MATRIX.ADD :M1 :M2
 IF EMPTYP :M1 [OUTPUT []]
 OUTPUT FPUT (VECTOR.ADD FIRST :M1 FIRST :M2)
             (MATRIX.ADD BUTFIRST :M1 BUTFIRST :M2)
END

TO VECTOR.ADD :V1 :V2
 IF EMPTYP :V1 [OUTPUT []]
 OUTPUT FPUT (SUM FIRST :V1 FIRST :V2)
             (VECTOR.ADD BUTFIRST :V1 BUTFIRST :V2)
END
```

I've called the top-level procedure `MATRIX.ADD` rather than just `ADD` to make sure there is no confusion about which is which. The procedure `SUM` that's used by `VECTOR.ADD` is Logo's built-in addition function for scalars.

```
? SHOW MATRIX.ADD [[2 3 4] [5 6 7]]  [[87 0 14] [-6 28 9]]
[[89 3 18] [-1 34 16]]
```

These procedures are *recursive*. That is, each of them includes an invocation of the same function applied to different inputs. Defining a function in terms of itself is a familiar trick to mathematicians:

$$n! = \begin{cases} 1 & \text{if } n = 0; \\ n \times (n-1)! & \text{if } n > 0. \end{cases}$$

But mathematicians also know that it's hard to get students to understand these inductive definitions at first.

There are many reasons why Logo is not widely used in teaching mathematical topics such as matrix arithmetic. BASIC is popular because it often comes free with personal computers. Pascal is popular, in the United States, because of the Advanced Placement Exam in Computer Science. Logo, because of its widespread use in elementary schools, has an undeserved but nearly universal reputation as a language suitable *only* for very young students doing trivial problems. Many people believe that Logo can only do graphics, like a paint program. But even people who see past these misunderstandings often consider Logo list processing difficult because of the frequent use of recursion.

Why is a recursive definition needed here? Later I shall suggest that the use of recursion is *not* truly necessary; by providing a larger functional programming "toolkit" we can express the desired computation without writing self-referential procedures. Still, there is a connection between the functional approach and the use of recursion. To simplify the discussion, let's consider the subprocedure `VECTOR.ADD`. The sequential view of programming would express the job of this procedure as a sequence of events, one for each element of the result vector:

```
LET C(1) = A(1) + B(1)
LET C(2) = A(2) + B(2)
LET C(3) = A(3) + B(3)
```

In this illustration I am supposing that we are adding vectors of length three. Sequential programming languages generally provide a looping mechanism so that a string of similar events can be encoded without explicit repetition:

```
FOR I=1 TO 3
LET C(I) = A(I) + B(I)
NEXT I
```

In the functional programming model, we are trying to get away from this idea of a sequence of events. For one thing, we have no predetermined place, like the array `C` in the sequential version, to put the result. Still, it's possible to express the output value in terms of composition of built-in functions. We must use the addition function `SUM`, but also functions that select particular elements from a vector or combine elements into a new vector:

```
TO VECTOR.ADD :V1 :V2
 OUTPUT (LIST (SUM ITEM 1 :V1 ITEM 1 :V2)
              (SUM ITEM 2 :V1 ITEM 2 :V2)
              (SUM ITEM 3 :V1 ITEM 3 :V2))
END
```

Again, in this illustration I am assuming three-element vectors. I've used the built-in function `ITEM` to select one element from a vector and the function `LIST` to combine several elements into one vector. (I've formatted this procedure to indicate the groupings clearly but in most microcomputer versions of Logo the `OUTPUT` instruction must be entered as a single, long line.)

The difficulty comes in generalizing this definition to work with vectors of any length. Formal mathematical notation has no trouble with

$$(a_1, a_2, a_3) + (b_1, b_2, b_3) = (a_1 + b_1, a_2 + b_2, a_3 + b_3)$$

but when we want to generalize we must resort to ellipses:

$$(a_1, \ldots, a_n) + (b_1, \ldots, b_n) = (a_1 + b_1, \ldots, a_n + b_n)$$

This notation is okay on paper but not quite what we want for a computer program. The mathematician's solution is an inductive definition: Imagine that we know how to add n-element vectors, and use that to define addition of vectors with $n + 1$ elements. To do this we need a function "adjoin" that takes a number and a vector as inputs, giving as its output a slightly longer vector including the new number. Then we can say

$$\mathrm{adjoin}(a_0, \mathbf{a}) + \mathrm{adjoin}(b_0, \mathbf{b}) = \mathrm{adjoin}(a_0 + b_0, \mathbf{a} + \mathbf{b})$$

The Logo equivalent of adjoin is called `FPUT`, for "first put." It takes two inputs. The first can be anything, but the second must be a list. The output is a new list consisting of the old list with the new thing put in front. Taking apart a vector is accomplished with the functions `FIRST`, to extract the first element of a list, and `BUTFIRST`, whose output is a list equal to its input with the first element removed.

The recursive definition of `VECTOR.ADD`, then, expresses the sum of its input vectors in terms of the sum of two smaller vectors, namely the `BUTFIRST`s of the inputs, with one extra number adjoined, namely the ordinary numeric sum of the `FIRST` elements of each vector. Like any inductive definition, it needs a base case. This one says that the sum of two zero-length (empty) vectors is the empty vector. The analysis of `MATRIX.ADD` is similar, except that a matrix is a list of vectors rather than a list of numbers.

Why Is Recursion Hard?

There are several aspects to the difficulty that students have in coming to understand recursive programming style. One of these is also found in learning to accept inductive proofs: the sense of unfairness in any self-referential definition. "You're assuming what you're supposed to be proving!" But in some ways the difficulty is greater in the case of programming.

An inductive proof can be presented "ground up." We show explicitly that the theorem is true for $n = 1$. Then, by the inductive step, we can see that it must be true for $n = 2$. Then, by induction again, it must also be true for $n = 3$. The advantage of this perspective is that each step is taken from a firm footing, with nothing hanging in the air. That is, by the time we consider the case $n = 3$ we are completely satisfied that the theorem has been demonstrated for $n = 2$.

By contrast, in the programming context the more usual situation is that we have to work backwards. For example, we are presented with a pair of three-element vectors to add. That is our starting point. We then say, "we could do this if we knew how to add two-element vectors." The initial problem must be suspended while we consider this subproblem. By the time we get down to the base case there are three such suspended problems. This state of affairs is a challenge not only to our faith but to our memory. It is very easy to lose the thread of contexts in which subproblems arose. The difference between the proof and the program arises because in the proof we are not specifically concerned with any concrete example; the goal is a general truth, and it makes as much sense to start small as not. But we wrote the program because we wanted to solve a particular problem, and most often not a trivial base-case problem.

For teaching purposes, it is possible to introduce recursion from the ground up. We can introduce a general problem, then decide to start by solving the simplest possible case. This is the method I prefer in my own teaching. The advantage is that it avoids the sense of hanging in midair, but there are two costs. First, it means that the use of recursion can't be motivated by a problem that really arises in class; such problems won't be so simple. Second, although the students don't feel strained, they do feel foolish during the beginning stages of the explanation. The recursive style seems like an overly complicated approach to such simple problems. Then the more complicated cases can feel like rabbits suddenly pulled from a hat.

Another problem in learning recursion is faced by students who have previous experience in the sequential programming style. They are strongly tempted to interpret the recursive invocation as a loop, an instruction to "do it again." That interpretation can sometimes work for recursive Logo commands (programs with side-effect instructions), but is almost never appropriate for recursive operations (functions that return values, like `VECTOR.ADD`). The actual sequence of events in the computer is not what these students would predict, in which the first invocation computes the first element, then the second takes over. It would be more nearly accurate to say that the sequence happens backwards; the first invocation can't do its job until the second is complete, and so on. So it is the *last* invocation, the one for the base case, that really happens first!

A problem related to the "do it again" misunderstanding concerns the nature of local variables. In a loop, such as the `FOR–NEXT` examples seen earlier, there is one single loop variable whose value keeps changing. In a recursive procedure, the relevant variable names (the procedure inputs, like `V1` and `V2`) do not correspond to a single box into which changing values are placed. Rather, each invocation of the procedure creates a new set of variables. Several such variables can exist at the same time. Students have to work out how the program decides which variable to use when such a name appears in an instruction.

Functional Programming Without Recursion

One language specifically designed for matrix manipulation is APL. To add two matrices in APL, we can simply use the built-in `+` operation. Addition in APL is defined to work for scalars, vectors, matrices, or higher-dimensional arrays. The APL programmer avoids both looping and recursion. (Of course there is some kind of repetition going on behind the scenes, if the APL program is running on a conventional computer that can only perform one arithmetic operation at a time. But this repetition is invisible to the programmer, just as every language makes certain details invisible.)

APL encourages a functional programming style. The addition operator is a function; it can be composed with other functions to build new, more complex functions. Because the primitive functions all work on arrays as well as scalars, many problems can be solved using "one-liners," programs with no visible control structures at all. There is no looping construct in APL. When a control mechanism is needed, the APL programmer can choose between two extremes, Logo-like recursion and an unstructured "goto" operation. But many mathematical problems can be solved without raising the issue of control, which is advantageous for a math teacher.

Unfortunately, APL is not widely available. The largest obstacle is that it uses a notation very close to that of ordinary mathematics, full of Greek letters and other strange symbols. These are not part of the standard computer character set, and special hardware is required to display them. (Indeed, there aren't enough keys even on the APL keyboard. Some symbols are formed by overprinting two other symbols. This approach was designed for hardcopy computer terminals, which backspace and overprint naturally. It isn't suitable for most current display devices, which allow only one character in any screen position.)

Can we use, in Logo, the APL idea of operating on a vector or matrix "all at once," rather than explicitly manipulating each element individually? Tools for this programming metaphor have long been part of LISP, Logo's parent language. It is an easy matter to extend Logo (by writing procedures in Logo and loading them for students) with such tools.

Avoiding Recursion in Logo

Let me begin by defining a few simple arithmetic procedures.

```
TO SQUARE :X
 OUTPUT :X * :X
END

TO NEXT :X
 OUTPUT :X + 1
END

TO DOUBLE :X
 OUTPUT :X * 2
END
```

Now consider these interactions with Logo:

```
? SHOW MAP "SQUARE [2 3 4 5]
[4 9 16 25]
? SHOW MAP "NEXT [7 8 9]
[8 9 10]
? SHOW MAP "DOUBLE [5 3 20 6 1]
[10 6 40 12 2]
```

The procedures `SQUARE` and so on are defined to operate on scalars. The general tool called `MAP` allows these functions to work on vectors, by arranging to apply a function to each element of the vector. The output from `MAP` is another vector with the same number of elements, but with values computed by the function being mapped.

Logo is an extensible language. This means that a user-defined procedure is invoked in exactly the same way as a primitive procedure. `MAP` happens not to be a Logo primitive, but if it is included in a startup file it can be presented to students just as if it were a primitive. Its purpose is immediately clear, and should present no conceptual problems to students. The same idea can be extended to functions of two inputs:

```
? SHOW MAP.2 "SUM [1 2 3] [40 50 60]
[41 52 63]
? SHOW MAP.2 "PRODUCT [5 6] [7 8]
[35 48]
? SHOW MAP.2 "EQUALP [1 4 6 7] [30 4 5 9]
[FALSE TRUE FALSE FALSE]
```

`MAP.2` requires three inputs, a function and two lists. The two lists must have the same length, and the output will have that length also.

We are now in a position to return to the problem of matrix addition with which we began:

```
TO MATRIX.ADD :M1 :M2
 OUTPUT MAP.2 "VECTOR.ADD :M1 :M2
END
```

```
TO VECTOR.ADD :V1 :V2
 OUTPUT MAP.2 "SUM :V1 :V2
END
```

That's it! I find this version simpler than the original BASIC program with nested `FOR` loops. There are no auxiliary variables `I` and `J`; there is no need to build the size or shape of the matrices into the procedures. Once the general idea of `MAP` is understood, its application to this problem is straightforward.

The `MAP` and `MAP.2` procedures themselves are, naturally, a little more complicated. They are defined recursively, using some of the more advanced capabilities of Logo. If students (or teachers!) want to understand the inner workings of these tools, they must face the challenge of recursion. I think this is fine; people who are interested in programming must, eventually, understand recursion. The point is that that understanding no longer has to come first. The mathematics can be studied and the programming issues can be postponed. (The Logo definitions of all the tools presented in this paper are collected in an appendix.)

Templates

One problem with the mapping tools as I've presented them above is that we must define a named scalar function in order to use that name as an input to `MAP`. For example, when I first introduced `MAP`, I had to detour through definitions of `SQUARE`, `NEXT`, and `DOUBLE` in order to have something to map over the vectors in the examples. It's not so bad to have names for these functions, since the functions are useful in themselves and the names are sensible. But sometimes we need an *ad hoc* function just for one particular mapping operation. The problem will become clearer if we consider an example.

We would like to be able to multiply a matrix by a scalar. That is, we want to be able to say

```
? SHOW MATRIX.SCALE 3 [[1 2 3] [4 5 6]]
[[3 6 9] [12 15 18]]
```

The function `MATRIX.SCALE` (I'm saving the name `MULTIPLY` for multiplication of a matrix by a matrix) takes two inputs, a number and a matrix. Its output is a matrix with the same shape as the input, but with each element multiplied by the number.

It may be tempting to write `MATRIX.SCALE` as follows:

```
TO MATRIX.SCALE :NUM :MAT                    ; incorrect!
 OUTPUT MAP.2 "VECTOR.SCALE :NUM :MAT
END

TO VECTOR.SCALE :NUM :VEC
 OUTPUT MAP.2 "PRODUCT :NUM :VEC
END
```

This looks like a straightforward application of the same ideas we used in `MATRIX.ADD`, but the parallel doesn't work. The problem is that `MAP.2` requires two *lists* of equal length so that it can inductively step through them, applying the function input to pairs of elements. Here we are trying to use a single number `:NUM` repeatedly.

There is really only one matrix input in this problem, not two, and so we need a solution using `MAP`, not `MAP.2`. It would be easy enough to do this for any specific scale factor:

```
TO TIMES3 :X
 OUTPUT 3 * :X
END

TO MATRIX.SCALE3 :MAT
 OUTPUT MAP "VECTOR.SCALE3 :MAT
END

TO VECTOR.SCALE3 :VEC
 OUTPUT MAP "TIMES3 :VEC
END
```

But it's not obvious how to generalize this into the desired `MATRIX.SCALE` function. We'd have to be able to invent the functions like `TIMES3` "on the fly."

The problem we are experiencing here turns out to be similar to the problem that encouraged us to switch from BASIC to Logo as the medium for matrix manipulation. In BASIC, we couldn't compute the sum of two matrices without putting the result in a specific named array `C`. An array couldn't exist as a real object on its own, without a name. Logo lists, by contrast, are first-class objects that can be entered directly as arguments to functions like `MATRIX.ADD` without the intermediate step of assigning them as the value of a named variable. What we need now is *first-class functions*. We want to be able to say "the function 3 times x" as an input to `MAP`.

First-class functions are one of the central ideas of LISP, the language from which Logo was developed. Logo itself does not include LISP's mechanism for this purpose, but it does include other mechanisms that allow us to implement first-class functions within Logo. We can put a Logo expression in a list, like

```
[3 * :X]
```

and then we can ask Logo to `RUN` the list. That is, the function `RUN` takes an expression list as its input, and gives the value of the expression as its output.

It may seem that this is all we need. We could write

```
OUTPUT MAP [3 * :X] :VEC
```

instead of

```
OUTPUT MAP "TIMES3 :VEC
```

and just design the `MAP` procedure so that it runs the expression list for each element of the vector. This is almost the right thing, but there is a mathematical confusion involved. An expression is not the same thing as a function. In mathematical notation, we don't say $f = 3x$; we say $f(x) = 3x$. In this particular example the difference may not seem important; x is the only variable around, and so it's obvious what is meant. But consider the difference between $f(a, b) = 2a+b$ and $g(b, a) = 2a+b$. These two functions are not equivalent, even though they are defined by the same expression $2a+b$.

A similar naming problem arises in the `SCALE` problem. If we just want to scale by 3, then we could say that there's only one variable around. But we are going to want to write something like

```
TO VECTOR.SCALE :NUM :VEC
 OUTPUT MAP [:NUM * :X] :VEC
END
```

In this situation it's not obvious how MAP should know that the variable NUM has a particular value assigned to it externally, while the variable X is the one into which it should plug the elements of the vector. The traditional LISP solution is to represent a function as a *lambda expression* that contains the names of the input variables as well as the defining expression. So in LISP we might say

```
(DEFINE F (LAMBDA (A B) (+ (* 2 A) B)))
(DEFINE G (LAMBDA (B A) (+ (* 2 A) B)))
```

(LISP uses prefix form (* 2 A) rather than infix form (2 * A) to represent arithmetic operations. But the important point for our purposes is to notice the difference between F and G, namely the list of parameter names following the word LAMBDA.)

It would be possible to invent this precise mechanism for Logo. Then we could say

```
TO VECTOR.SCALE :NUM :VEC
 OUTPUT MAP [LAMBDA [X] [3 * :X]] :VEC
END
```

The trouble is that this notation seems needlessly obtrusive. If we present this to students, their attention will be focused on the meaning of the notation, rather than on matrix arithmetic. I wanted to find a better alternative. A quick-and-dirty solution would be to say that the parameter to every function must be called X; for two-input functions we could reserve X and Y. That would work, but it seems ugly to me. It would prevent the use of those names for other purposes. (For example, what if I'd wanted to call the scale factor X instead of NUM?)

I took my inspiration from an earlier problem in mathematics education that seems closely related. Some of the early New Math experimenters wanted to use algebraic ideas with elementary school students, presenting very simple equations like

$$x + 3 = 7$$

They found that the underlying idea of "what plus three is seven?" wasn't too difficult, but that the x notation was a problem for some students. Their solution was to present the equation in the form

$$\boxed{} + 3 = 7$$

Pedagogically, the box seems to suggest "this is a slot to be filled" without raising the difficulties about names and values that students find with x. Unfortunately there is no box character in the ASCII sequence, but I wanted to come as close as possible to this ideal of a graphically obvious slot indication. My solution was to use a question mark:

```
? SHOW MAP [3 * ?] [2 3 4]
[6 9 12]
? SHOW MAP [? * ?] [2 3 4 5]
[4 9 16 25]
? SHOW MAP [IFELSE (? < 0) [-?] [?]] [2 -3 12 0 -14]
[2 3 12 0 14]
```

An expression list with a question mark slot to be filled is called an expression *template*.

We can now solve the SCALE problem:

```
TO MATRIX.SCALE :NUM :MAT
 OUTPUT MAP [VECTOR.SCALE :NUM ?] :MAT
END
```

```
TO VECTOR.SCALE :NUM :VEC
 OUTPUT MAP [:NUM * ?] :VEC
END
```

We have redesigned MAP to accept templates instead of named functions. What about MAP.2? We need a way to express templates with two slots. My solution is to name the slots ?1 and ?2.

```
? SHOW MAP.2 [?2 - ?1] [1 2 3] [15 20 30]
[14 18 27]
```

This doesn't seem quite so compelling as the unadorned question mark, just as New Math equations with boxes and triangles seem more cluttered and less obvious than the ones with only boxes. Still, the notation clearly indicates which slot is which.

By the way, are you tired of having to write MATRIX and VECTOR versions of everything, when you really only care about the matrix version? You can invent two-level mapping tools that will allow direct definition of the matrix functions.

```
TO MMAP :FN :MAT              ; matrix map
 OUTPUT MAP [MAP :FN ?] :MAT
END

TO MMAP.2 :FN :M1 :M2
 OUTPUT MAP.2 [MAP.2 :FN ?1 ?2] :M1 :M2
END

TO ADD :M1 :M2
 OUTPUT MMAP.2 [?1 + ?2] :M1 :M2
END

TO SCALE :NUM :MAT
 OUTPUT MMAP [:NUM * ?] :MAT
END
```

The definitions of ADD and SCALE now embody the mathematical ideas quite directly, with very little programming language noise. You may find the definitions of MMAP and MMAP.2 a little tricky, but the structure of nested MAPs is exactly analogous to the nested FORs in the sequential version of matrix programming.

More Functional Programming Tools

I would like to define an exponentiation function. That is, I want to take a given base to a given power, using repeated multiplication. (The power must, of course, be a nonnegative integer.) This is an example of composition of functions, in which the function [? * :BASE] is invoked :EXP times, with the result from one invocation filling the slot for the next invocation. (The first invocation is done with 1 in the slot.)

```
TO POWER :BASE :EXP
 OUTPUT CASCADE :EXP [? * :BASE] 1
END
```

```
? SHOW POWER 2 5
32
```

CASCADE is a tool for repeated composition of a function with itself. It takes three inputs. First is a number saying how many times we want to invoke the function. Second comes a template indicating what the function is. Third is the initial value used to fill the slot in the first invocation. As another example of its use, here is Newton's method to approximate the square root of a number by repeated composition of a function:

```
TO NSQRT :X
 OUTPUT CASCADE 10 [(? + (:X/?))/2] 1
END
```

Newton's approximation function takes the average of the previous guess and the quotient of x and the guess. In this procedure we start with an initial guess of 1 and we apply the approximation function 10 times. This constant number of invocations is too simple; fewer invocations would do for small values of x, while more are needed to get a good approximation for large values. (For example, try NSQRT of a million.) To fix this problem, CASCADE accepts as its first input either a number or a *predicate template*, that is, an expression template whose value is TRUE or FALSE. The repeated composition continues until the value of the predicate template is TRUE.

```
TO NSQRT :X
 OUTPUT CASCADE [LESSP ABS (:X - ?*?) 0.00000001] [(? + (:X/?))/2] 1
END

TO ABS :X
 OUTPUT IFELSE :X<0 [-:X] [:X]
END
```

It is sometimes convenient for the function being cascaded to know how many times it's been invoked so far. For example, the factorial function is like the power function except that each new multiplication is by a larger number, instead of always by the same :BASE. Therefore, CASCADE templates can include another special symbol, #, indicating the number of invocations.

```
TO FACTORIAL :N
 OUTPUT CASCADE :N [? * #] 1
END
```

The template is first invoked with 1 in the question mark slot (because the third input to CASCADE is 1) and also 1 in the number sign slot (because this is the first invocation). The result is 1. Next time, ? is 1 because that's the result of the first invocation, but # is 2. The third time, ? is 2 and # is 3.

The function being cascaded is not limited to a scalar domain or range. We can use list processing operations to generate the vector of all the integers from 1 to n:

```
TO FROM1TO :N
 OUTPUT CASCADE :N [LPUT # ?] []
END

? SHOW FROM1TO 5
[1 2 3 4 5]
```

Here the initial value is an empty vector, and each invocation of the template appends the next integer in sequence. (LPUT means "last put"; it's like FPUT except that the new element is adjoined on the right instead of on the left.)

Just as there is a MAP.2 for two-input functions, we sometimes need CASCADE.2 for two-input functions. The parallel is not exact, however. In this context we need not only two initial inputs but also two functions, one to specify the next value of ?1 and one to give the next value of ?2. CASCADE.2 therefore takes five inputs:

```
CASCADE.2 end.test template.1 initial.1 template.2 initial.2
```

For example, here is how to compute the nth Fibonacci number:

```
TO FIB :N
 OUTPUT CASCADE.2 :N-1 [?1 + ?2] 1 [?1] 0
END
```

The final output from CASCADE.2 is the final value of ?1.

Because I'm presenting so many examples so briefly, I'm afraid the overall point is in danger of being lost, so let me take a moment to review where we are. I started by presenting the idea of functional programming, in which the focus is on the input-output behavior of a procedure rather than on the exact sequence of events through which the computer produces the desired output. I suggested that functional programming is much better suited to mathematics teaching than traditional, sequential programming. If this is true, teachers should prefer a functional language like Logo rather than a sequential one like BASIC or Pascal. One stumbling block has been the need to teach recursion in order to program even the simplest mathematical ideas, but this obstacle can be avoided by providing tools that are the functional analog to the looping construct in sequential programming.

So far I have introduced two such tools, MAP and CASCADE, in one-input and two-input versions. With this small toolkit I have been able to write procedures that are not self-referential for a variety of mathematical functions. In my experience, students quickly understand MAP with hardly any explanation, and can use it correctly themselves. CASCADE is a little harder, perhaps just because it has more inputs and the procedures are therefore harder to read. But the effort to understand the metaphor behind CASCADE (composing a function with itself) pays off in more expressive power than students of traditional languages get from similar effort put into FOR, WHILE, UNTIL, and so on. I now want to introduce two more functional programming tools.

We want to take the dot product of two vectors. To do this, we must multiply pairs of corresponding elements, and then add up all the products. The first part of this is clearly a job for MAP.2, but we need a tool for the second part.

```
? SHOW REDUCE "SUM [2 3 4]
9
? SHOW REDUCE "PRODUCT [2 3 4]
24

TO DOTPROD :V1 :V2
 OUTPUT REDUCE "SUM (MAP.2 "PRODUCT :V1 :V2)
END
```

Because the function I want to map over the two vectors happens to be included as a Logo primitive, I've just given MAP.2 its name, as I did in the earliest examples, instead of a template with question marks. We'll write these tools so that either form is allowed. REDUCE, the new tool in this example, takes a two-input associative function and applies it to all the elements of the indicated list. (We insist that the function be associative to avoid questions about the precise grouping of the elements of the list.)

It may be time for a reminder that functional programming tools can be applied to non-numeric computation as well:

```
TO ACRONYM :NAME
 OUTPUT REDUCE "WORD MAP "FIRST :NAME
END

? SHOW ACRONYM [AMERICAN CIVIL LIBERTIES UNION]
ACLU

TO PIGLATIN :WORD
 OUTPUT WORD (CASCADE [MEMBERP FIRST ? "AEIOU]
                      [WORD BUTFIRST ? FIRST ?]
                      :WORD)
             "AY
END

? SHOW PIGLATIN "SPAGHETTI
AGHETTISPAY
```

The last tool is used to select a subset of the members of a list, based on a predicate template:

```
? SHOW FILTER [EQUALP (REMAINDER ? 2) 0] [1 2 3 4 5]
[2 4]
? SHOW FILTER [EQUALP (SQRT ?) (INT SQRT ?)] [1 2 3 4 5]
[1 4]
```

Suppose we want to know if a number n is perfect. We need to find all the factors of n, then add them up:

```
TO PERFECT? :N
 OUTPUT EQUALP :N
              REDUCE "SUM
                     FILTER [EQUALP (REMAINDER :N ?) 0]
                            FROM1TO :N-1
END

? SHOW PERFECT? 6
TRUE
? SHOW PERFECT? 7
FALSE
```

In this procedure, FROM1TO gives us a list of all the numbers that might be factors of n. FILTER selects the ones that actually are factors, by checking the remainder of dividing n by each candidate. REDUCE adds all the factors, and EQUALP checks whether the sum is equal to n.

You may be wondering if there will be `REDUCE.2` and `FILTER.2` procedures. It turns out that these would not be meaningful. Functions can have more than one input, but they are only allowed one output. If we tried to select elements from two lists in parallel, we would end up with two sublists. Which would we output?

Some Problems Really Are Recursive

Suppose we want to find the determinant of a square matrix. The well-known algorithm requires us to select a single row or column of the matrix, then for each element of that row or column we must compute *the determinant of* a submatrix formed by removing the row and column containing that element. (Then we multiply by plus or minus the element, and then we add up all the results, but those are the easy parts.) The words in italics are a self-referential part of the definition of the determinant. The use of recursion to solve this problem is not an accidental result of missing features in the programming language. It's a naturally recursive problem.

It will turn out to be easiest if we choose the leftmost column of the matrix as the one to treat specially. If we chose the top row, which would be more traditional, then for each element of that row we must extract a submatrix in which its column is removed. If we start with a column, then for each of its elements the submatrix is found by removing a row. The latter operation is easier, since we store matrices as lists of rows. The left column of a matrix is found with the expression

```
MAP "FIRST :MATRIX
```

which selects the first element of each row. Similarly, the expression

```
MAP "BUTFIRST :MATRIX
```

returns a (non-square) matrix with the first column eliminated. This latter matrix will be used as the basis from which rows will be eliminated to form each submatrix.

```
TO DET :MATRIX
 IF EMPTYP BUTFIRST :MATRIX [OUTPUT FIRST FIRST :MATRIX]   ; base case, 1x1
 LOCAL "RIGHTPART
 MAKE "RIGHTPART MAP "BUTFIRST :MATRIX                     ; all but 1st col
 OUTPUT REDUCE "SUM
              MAP [(PRODUCT ? (SIGN #) (DET ALLBUT # :RIGHTPART))]
                  MAP "FIRST :MATRIX
END

TO ALLBUT :N :MAT                                          ; all but nth row
 OUTPUT FILTER [NOT EQUALP # :N] :MAT
END

TO SIGN :N
 OUTPUT IFELSE (EQUALP REMAINDER :N 2 0) [-1] [1]
END
```

Even though recursion is necessary in this problem, I've continued to use the non-recursive functional tools wherever possible. To make this work, I had to extend the `#` notation in templates, which I originally intended only for `CASCADE`, to work in `MAP` and `FILTER` also.

Since DET is defined recursively, it must have a base case. In this problem, the base case is that the determinant of a matrix with one row and one column is the single number in the matrix. We say FIRST FIRST because the number is the first element of the first row. For other cases, the program is more complicated. First we create a local variable RIGHTPART and assign to it the non-square matrix with the first column of the original matrix removed. The reason for this step is to avoid repetitive computation of the same matrix for every row, which would result if we just said

```
 OUTPUT REDUCE "SUM
              MAP [(PRODUCT ? (SIGN #) (DET ALLBUT # MAP "BUTFIRST :MATRIX))]
                  MAP "FIRST :MATRIX
```

without using the extra variable. Now we can pick apart the long OUTPUT instruction to see how it embodies the definition. We know from the use of REDUCE that we are going to compute a list full of numbers and then add up the numbers. What is the list? Well, it's computed by

```
MAP [...] MAP "FIRST :MATRIX
```

which means "compute some function of each element of the first column of the matrix." So far, so good. What function do we compute for each element of the first column? It's a product of three factors: the element itself, either positive or negative 1 depending on which element it is, and the determinant of a submatrix. That submatrix is computed by selecting all but a particular row from the matrix :RIGHTPART.

This is a complicated problem, and the complexity of the procedure reflects that. Still, the use of functional programming has allowed us to solve the problem without introducing auxiliary index variables. The procedure exactly reflects the structure of the definition of determinant.

Since we can't avoid recursion in the definition of DET, it might be worthwhile to compare this version with the way it would traditionally be written in Logo, relying even more heavily on recursion:

```
TO DET :MATRIX
 IF EMPTYP BUTFIRST :MATRIX [OUTPUT FIRST FIRST :MATRIX]
 OUTPUT DET1 (FIRSTS :MATRIX) (BUTFIRSTS :MATRIX) 1 1
END

TO DET1 :COL :REST :N :SIGN
 IF EMPTYP :COL [OUTPUT 0]
 OUTPUT SUM (PRODUCT (FIRST :COL) :SIGN (DET ALLBUT :N :REST))
           (DET1 (BUTFIRST :COL) :REST :N+1 -:SIGN)
END

TO FIRSTS :LIST
 IF EMPTYP :LIST [OUTPUT []]
 OUTPUT FPUT (FIRST FIRST :LIST) (FIRSTS BUTFIRST :LIST)
END

TO BUTFIRSTS :LIST
 IF EMPTYP :LIST [OUTPUT []]
 OUTPUT FPUT (BUTFIRST FIRST :LIST) (BUTFIRSTS BUTFIRST :LIST)
END
```

```
TO ALLBUT :N :LIST
 IF :N=1 [OUTPUT BUTFIRST :LIST]
 OUTPUT FPUT (FIRST :LIST) (ALLBUT :N-1 BUTFIRST :LIST)
END
```

To someone familiar with recursion, the procedures `DET` and `DET1` in this version may seem less intimidating than the earlier version. It would be possible to compromise, using `MAP` and `FILTER` to define `FIRSTS`, `BUTFIRSTS`, and `ALLBUT` as in the first version, but using `DET` and `DET1` from the second version.

On the other hand, to a reader who is not intimidated by the first version, it makes the structure of `DET` more plainly apparent. In the second version, the fact that the determinant is a sum of products is buried inside `DET1`, and you have to notice that one of the inputs to `SUM` is a recursive call in order to see that several products will be added. The first version says

```
OUTPUT REDUCE "SUM MAP [(PRODUCT ...
```

right up front.

Computing the determinant is a hard problem, and the program will be complex no matter how it's written. Here is an easier problem that is still most convenient in recursive form: compute the transpose of a matrix. That is, interchange the rows and the columns. The trick is to see that the first row of the output will be the first column of the input, while the remaining rows of the output will be the *transpose of* the remaining columns.

```
TO TRANSPOSE :MAT
 IF EMPTYP FIRST :MAT [OUTPUT []]
 OUTPUT FPUT (MAP "FIRST :MAT) (TRANSPOSE MAP "BUTFIRST :MAT)
END

? SHOW TRANSPOSE [[1 2 3] [4 5 6]]
[[1 4] [2 5] [3 6]]
```

As a third example in which recursion is helpful, let's return to the question of `MATRIX.ADD` and `VECTOR.ADD`. My first implementation defines one in terms of the other. Later, by inventing a special tool for two-dimensional mapping, I was able to define the matrix version directly. But now suppose that I want to be able to add vectors, matrices, and even multi-dimensional arrays. I don't want to have to define a separate `ADD` procedure for every possible dimension. Instead, I want one `ADD` procedure that will work on scalars, vectors, matrices, or any other array.

```
TO ADD :A :B
 IF NUMBERP :A [OUTPUT :A + :B]
 OUTPUT MAP.2 "ADD :A :B
END
```

This procedure extends the pattern of vector and matrix addition to any number of dimensions by mapping *itself* over its inputs!

Perhaps it's time for another review of our journey so far. In principle, there is no need for any control mechanism other than recursion in a programming language. Any problem can be solved using recursive procedures, without explicit looping mechanisms. However, the idea of recursion is hard for beginning students, so in order to make functional programming practical, it's helpful to provide tools that are analogous to looping constructs but consistent with the functional style. I have shown four such tools: MAP, CASCADE, REDUCE, and FILTER. These tools are adequate for many problems, but not for all problems. If a problem is fundamentally self-referential, then the student must understand recursive programming to solve it.

Continuing Development

Until the writing of this paper I had never written a determinant procedure. This is the first example I've found in which MAP and FILTER templates need the # mechanism to let the function know the position of its argument in the original input list. The procedure definitions at the end of the paper, therefore, are somewhat different from earlier published versions. Because these tools are language extensions written in Logo itself, I can make this change without having to rewrite the Logo interpreter. Other teachers can also invent their own tools, so that their students have exactly the necessary tools for whatever projects are relevant to their topic. For example, for a computer-based linguistics course [Goldenberg, 1987] Paul Goldenberg invented a tool to make systematic changes in the spelling of a word:

```
? SHOW REPLACE [M O] [P A] "MOMMY
PAPPY
```

The first input is a list of letters to look for in the word, and the second input is a same-length list of replacements.

Recently I have found myself, in certain situations, wanting something more like the LISP lambda notation, with explicit names for the slots in the template. For example, here is a program to multiply two matrices:

```
TO MULTIPLY :A :B
 LOCAL "TB
 MAKE "TB TRANSPOSE :B
 OUTPUT MAP [MULROW ? :TB] :A
END

TO MULROW :ROWA :TB
 OUTPUT MAP [DOTPROD :ROWA ?] :TB
END
```

Each element of the product is the dot product of a row of :A and a column of :B. (The program uses TRANSPOSE because we store matrices by rows, and a row of the transpose is a column of the original matrix.) The subprocedure MULROW computes a single row of the answer; the entire answer is found by mapping MULROW over the rows of the first input matrix.

I don't like having to have the auxiliary procedure MULROW. I'd rather use two nested MAPs, as in the MMAP tool. But if I try that, I end up with the instruction

```
OUTPUT MAP [MAP [DOTPROD ? ?] :TB] :A
```

One of those question marks refers to the slot in the inner template, while the other refers to the slot in the outer one. This can't work. (Nor would it help to use ?1 and ?2. We aren't using MAP.2 to map over two lists in parallel; we're using the single-slot MAP twice.) To disambiguate the two slots, I have to be able to give each one a name explicitly. The notation doesn't actually require the word LAMBDA, which LISP uses for historical reasons; if the first thing in a template is a list in square brackets, I want to take that as a list of names of slots. Then I can say

```
OUTPUT MAP [[ROWA] MAP [[COLB] DOTPROD :ROWA :COLB] :TB] :A
```

I have written tools that accept this format, but I haven't yet tried teaching it to anyone.

Another annoyance I'd like to eliminate is the need for separate procedures MAP and MAP.2. What if I want to map a three-input function over three lists of inputs? I'd like a single MAP that can take any number of inputs, just as certain Logo primitives do, inside parentheses. This change, unfortunately, does require modifications to the Logo interpreter. One published version of Logo (Object Logo, for the Macintosh, developed by Coral Software and now owned by Apple) allows user-defined procedures with variable numbers of inputs. I hope the idea is extended to other dialects.

Teaching Experience

When I published the first volume of *Computer Science Logo Style* [Harvey, 1985], I introduced mapping tools near the end of the book, *after* the reader was familiar with recursion, because my emphasis was on the implementation of the tools. I wanted to make the point that Logo is an extensible language by showing how to extend it. I had considered introducing the tools earlier, without showing their definitions, to allow readers to write interesting functional programs in the early chapters. What dissuaded me, at that time, was that I was trying to make the book usable without a teacher by readers who might have any of half a dozen versions of Logo at home. Each dialect is slightly different from the others. I felt that I couldn't use tool procedures unless I provided the definitions on diskette, and that seemed like too much trouble.

As it turned out, the biggest complaint I heard from people who taught courses based on that book was about the difficulty their students had in the chapter on recursive operations. Their confusion was not only about recursion, but about the more fundamental issue of writing operations at all. They were uncomfortable with OUTPUT and with list manipulation. The reason for this, I decided, was that until that chapter they had had very little practice writing operations at all. Without recursion, the number of interesting functions you can compute is small. By presenting tools like MAP earlier, it would be possible to present more and better examples of functional programming in the early chapters.

In the fall semester of 1988 I taught a Logo programming course to a small group of Berkeley undergraduates using this approach. I have not revised the book, but in the early weeks of the course I supplemented it with handouts and class discussion showing the mapping tools. (The tool procedures I used were the versions in [Harvey, 1987], slightly different from the ones in this paper.) My experience was that the students were, as expected, easily able to apply functional programming ideas to problems that could be solved without recursion. Their early efforts could focus on what it means for a procedure to output a value, without being confused by the additional issues of self-reference. On the other hand, I had hoped that with this basis they would find recursion itself easy when we got to it; in that respect I was disappointed. Recursion was still a big hurdle. By the end of the semester almost all of the students could understand recursive procedures that I presented, but not all could reliably write their own recursive procedures.

One mistake that I made was to try to motivate the use of recursion by starting with quite complicated examples that couldn't be done using the iterative tools the students already knew. I was overreacting to the anticipated question, "why don't we just use MAP?" I would now begin with simpler examples and announce by fiat that the iterative tools may not be used this week. I would explain that they are going to learn how things like MAP can be written in Logo, and to understand that, they have to pretend that MAP isn't available. (By the way, I have of course done something similar in this paper, in the section that explains why recursion is still sometimes necessary. But I trust that the reader is not new to Logo and has already seen simpler examples of recursion.)

Many Logo teachers have noticed that students find recursive operations more difficult than the command-heavy style of programming used in turtle graphics applications. Some have responded by restricting their Logo teaching to graphics. Others, wanting to preserve the ability to explore natural language issues, have introduced word processing commands into Logo so that English text can be processed in the easier command-heavy style. (See, for example, [Tempel, 1986] for a description of this approach in LogoWriter.) I am unwilling to withhold from students the powerful idea of functional programming; the work described here is an attempt to preserve the functional style while allowing students to take smaller steps than the enormous intellectual leap that seems to be required in moving from commands directly into recursive operations.

Appendix: The Implementation

Resist the temptation to use shorter variable names in these procedures. It's important that these names be different from any variable names used in other parts of the program that invokes these tools.

```
TO MAP :MAP.TEMPLATE :TEMPLATE.LIST
 OUTPUT MAP1 PREPARE.TEMPLATE :MAP.TEMPLATE :TEMPLATE.LIST 1
END

TO MAP1 :MAP.TEMPLATE :TEMPLATE.LIST :TEMPLATE.NUMBER
 IF EMPTYP :TEMPLATE.LIST [OUTPUT []]
 LOCAL "TEMPLATE.VAR
 MAKE "TEMPLATE.VAR FIRST :TEMPLATE.LIST
 OUTPUT FPUT (RUN :MAP.TEMPLATE)
             (MAP1 :MAP.TEMPLATE (BF :TEMPLATE.LIST) 1+:TEMPLATE.NUMBER)
END

TO PREPARE.TEMPLATE :TEMPLATE
 IF LISTP :TEMPLATE [OUTPUT :TEMPLATE]
 OUTPUT SENTENCE :TEMPLATE "?
END

TO ?
 OUTPUT :TEMPLATE.VAR
END

TO #
 OUTPUT :TEMPLATE.NUMBER
END
```

```
TO MAP.2 :MAP.TEMPLATE :TEMPLATE.LIST1 :TEMPLATE.LIST2
 OUTPUT MAP.21 PREPARE.TEMPLATE.2 :MAP.TEMPLATE
              :TEMPLATE.LIST1 :TEMPLATE.LIST2 1
END

TO MAP.21 :MAP.TEMPLATE :TEMPLATE.LIST1 :TEMPLATE.LIST2 :TEMPLATE.NUMBER
 IF EMPTYP :TEMPLATE.LIST1 [OUTPUT []]
 LOCAL [TEMPLATE.VAR1 TEMPLATE.VAR2]
 MAKE "TEMPLATE.VAR1 FIRST :TEMPLATE.LIST1
 MAKE "TEMPLATE.VAR2 FIRST :TEMPLATE.LIST2
 OUTPUT FPUT (RUN :MAP.TEMPLATE)
            (MAP.21 :MAP.TEMPLATE (BF :TEMPLATE.LIST1)
                    (BF :TEMPLATE.LIST2) 1+:TEMPLATE.NUMBER)
END

TO PREPARE.TEMPLATE.2 :TEMPLATE
 IF LISTP :TEMPLATE [OUTPUT :TEMPLATE]
 OUTPUT SENTENCE :TEMPLATE [?1 ?2]
END

TO ?1
 OUTPUT :TEMPLATE.VAR1
END

TO ?2
 OUTPUT :TEMPLATE.VAR2
END

TO CASCADE :CASCADE.LIMIT :CASCADE.TEMPLATE :TEMPLATE.VAR
 OUTPUT CASCADE1 PREPARE.LIMIT :CASCADE.LIMIT
                PREPARE.TEMPLATE :CASCADE.TEMPLATE
                :TEMPLATE.VAR 1
END

TO CASCADE1 :CASCADE.LIMIT :CASCADE.TEMPLATE :TEMPLATE.VAR :TEMPLATE.NUMBER
 IF (RUN :CASCADE.LIMIT) [OUTPUT :TEMPLATE.VAR]
 OUTPUT CASCADE1 :CASCADE.LIMIT :CASCADE.TEMPLATE
                (RUN :CASCADE.TEMPLATE) 1+:TEMPLATE.NUMBER
END

TO PREPARE.LIMIT :LIMIT
 IF NUMBERP :LIMIT [OUTPUT SENTENCE [GREATERP :TEMPLATE.NUMBER] :LIMIT]
 OUTPUT PREPARE.TEMPLATE :LIMIT
END
```

```
TO CASCADE.2 :CASCADE.LIMIT :CASCADE.TEMPLATE1 :TEMPLATE.VAR1
                            :CASCADE.TEMPLATE2 :TEMPLATE.VAR2
 OUTPUT CASCADE.21 PREPARE.LIMIT.2 :CASCADE.LIMIT
                   PREPARE.TEMPLATE.2 :CASCADE.TEMPLATE1 :TEMPLATE.VAR1
                   PREPARE.TEMPLATE.2 :CASCADE.TEMPLATE2 :TEMPLATE.VAR2 1
END

TO CASCADE.21 :CASCADE.LIMIT :CASCADE.TEMPLATE1 :TEMPLATE.VAR1
                             :CASCADE.TEMPLATE2 :TEMPLATE.VAR2 :TEMPLATE.NUMBER
 IF (RUN :CASCADE.LIMIT) [OUTPUT :TEMPLATE.VAR1]
 OUTPUT CASCADE.21 :CASCADE.LIMIT
                   :CASCADE.TEMPLATE1 (RUN :CASCADE.TEMPLATE1)
                   :CASCADE.TEMPLATE2 (RUN :CASCADE.TEMPLATE2)
                   1+:TEMPLATE.NUMBER
END

TO PREPARE.LIMIT.2 :LIMIT
 IF NUMBERP :LIMIT [OUTPUT SENTENCE [GREATERP :TEMPLATE.NUMBER] :LIMIT]
 OUTPUT PREPARE.TEMPLATE.2 :LIMIT
END

TO REDUCE :REDUCE.TEMPLATE :TEMPLATE.LIST
 OUTPUT REDUCE1 PREPARE.TEMPLATE.2 :REDUCE.TEMPLATE :TEMPLATE.LIST
END

TO REDUCE1 :REDUCE.TEMPLATE :TEMPLATE.LIST
 IF EMPTYP BUTFIRST :TEMPLATE.LIST [OUTPUT FIRST :TEMPLATE.LIST]
 LOCAL [TEMPLATE.VAR1 TEMPLATE.VAR2]
 MAKE "TEMPLATE.VAR1 FIRST :TEMPLATE.LIST
 MAKE "TEMPLATE.VAR2 REDUCE1 :REDUCE.TEMPLATE BUTFIRST :TEMPLATE.LIST
 OUTPUT RUN :REDUCE.TEMPLATE
END

TO FILTER :FILTER.TEMPLATE :TEMPLATE.LIST
 OUTPUT FILTER1 PREPARE.TEMPLATE :FILTER.TEMPLATE :TEMPLATE.LIST 1
END

TO FILTER1 :FILTER.TEMPLATE :TEMPLATE.LIST :TEMPLATE.NUMBER
 IF EMPTYP :TEMPLATE.LIST [OUTPUT []]
 LOCAL "TEMPLATE.VAR
 MAKE "TEMPLATE.VAR FIRST :TEMPLATE.LIST
 IF (RUN :FILTER.TEMPLATE)
    [OUTPUT FPUT :TEMPLATE.VAR
                 FILTER1 :FILTER.TEMPLATE BF :TEMPLATE.LIST
                         1+:TEMPLATE.NUMBER]
 OUTPUT FILTER1 :FILTER.TEMPLATE BF :TEMPLATE.LIST 1+:TEMPLATE.NUMBER
END
```

References

Abelson H. and Sussman G. 1985: *Structure and interpretation of computer programs.* Cambridge: MIT Press.

Allen J., Davis R., and Johnson J. 1984: *Thinking about [TLC] Logo.* New York: Holt, Rinehart and Winston.

Backus J. 1978: 'Can programming be liberated from the von Neumann style?' *Communications of the ACM*, 21, 613–641.

Goldenberg E. P. and Feurzeig W. 1987: *Exploring language with Logo.* Cambridge: MIT Press.

Harvey B. *Computer science Logo style.* Cambridge: MIT Press.
1985: Volume 1: *Intermediate programming.*
1986: Volume 2: *Projects, styles, and techniques.*
1987: Volume 3: *Advanced topics.*

Tempel M. and Michaud N. 1986: 'Beyond Turtle Graphics?' *Logo 86 Proceedings*, Massachusetts Institute of Technology.