10 Iteration, Control Structures, Extensibility

In this chapter we're taking a tour "behind the scenes" of Berkeley Logo. Many of the built-in Logo procedures that we've been using all along are not, strictly speaking, primitive; they're written in Logo itself. When you invoke a procedure, if the Logo interpreter does not already know a procedure by that name, it automatically looks in a *library* of predefined procedures. For example, in Chapter 6 I used an operation called `gensym` that outputs a new, unique word each time it's invoked. If you start up a fresh copy of Logo you can try these experiments:

```
? po "gensym
I don't know how  to gensym
? show gensym
g1
? po "gensym
to gensym
if not namep "gensym.number [make "gensym.number 0]
make "gensym.number :gensym.number + 1
output word "g :gensym.number
end
```

The first interaction shows that `gensym` is not really a Logo primitive; the error message indicates that there is no such procedure. Then I invoked `gensym`, which made Berkeley Logo read its definition automatically from the library. Finally, once Logo has read the definition, I can print it out.

In particular, most of the tools we've used to carry out a computation repeatedly are not true Logo primitives: `for` for numeric iteration, `foreach` for list iteration, and `while` for predicate-based iteration are all library procedures. (The word *iteration* just means "repetition.") The only iteration mechanisms that are truly primitive in Logo are `repeat` and recursion.

Computers are good at doing things over and over again. They don't get bored or tired. That's why, in the real world, people use computers for things like sending out payroll checks and telephone bills. The first Logo instruction I showed you, in the first volume, was

```
repeat 50 [setcursor list random 75 random 20 type "Hi]
```

When you were first introduced to turtle graphics, you probably used an instruction like

```
repeat 360 [forward 1 right 1]
```

to draw a circle.

Recursion as Iteration

The trouble with `repeat` is that it always does *exactly* the same thing repeatedly. In a real application, like those payroll checks, you want the computer to do *almost* the same thing each time but with a different person's name on each check. The usual way to program an almost-`repeat` in Logo is to use a recursive procedure, like this:

```
to polyspi :side :angle :number
if :number=0 [stop]
forward :side
right :angle
polyspi :side+1 :angle :number-1
end
```

This is a well-known procedure to draw a spiral. What makes it different from

```
repeat :number [forward :side right :angle]
```

is that the first input in the recursive invocation is `:side+1` instead of just `:side`. We've used a similar technique for almost-repetition in procedures like this one:

```
to multiply :letters :number
if equalp :number 0 [stop]
print :letters
multiply (word :letters first :letters) :number-1
end
```

Since recursion can express any repetitive computation, why bother inventing other iteration tools? The answer is that they can make programs easier to read. Recursion is such a versatile mechanism that the intention of any particular use of recursion may be hard to see. Which of the following is easier to read?

```
to fivesay.with.repeat :text
repeat 5 [print :text]
end
```

or

```
to fivesay.with.recursion :text
fivesay1 5 :text
end

to fivesay1 :times :text
if :times=0 [stop]
print :text
fivesay1 :times-1 :text
end
```

The version using `repeat` makes it obvious at a glance what the program wants to do; the version using recursion takes some thought. It can be useful to invent mechanisms that are intermediate in flexibility between `repeat` and recursion.

As a simple example, suppose that Logo did not include `repeat` as a primitive command. Here's how we could implement it using recursion:

```
to rep :count :instr
if :count=0 [stop]
run :instr
rep :count-1 :instr
end
```

(I've used the name `rep` instead of `repeat` to avoid conflict with the primitive version.) The use of `run` to carry out the given instructions is at the core of the techniques we'll use throughout this chapter.

Numeric Iteration

`Polyspi` is an example of an iteration in which the value of a numeric variable changes in a uniform way. The instruction

```
polyspi 50 60 4
```

is equivalent to the series of instructions

```
forward 50 right 60
forward 51 right 60
forward 52 right 60
forward 53 right 60
```

As you know, we can represent the same instructions this way:

```
for [side 50 53] [forward :side right 60]
```

The `for` command takes two inputs, very much like `repeat`. The second input, like `repeat`'s second input, is a list of Logo instructions. The first input to `for` is different, though. It is a list whose first member is the name of a variable; the second member of the list must be a number (or a Logo expression whose value is a number), which will be the *initial* value of that variable; and the third member must be another number (or numeric expression), which will be the *final* value of the variable. In the example above, the variable name is `side`, the initial value is `50`, and the final value is `53`. If there is a fourth member in the list, it's the amount to add to the named variable on each iteration; if there is no fourth member, as in the example above, then the *step* amount is either 1 or −1, depending on whether the final value is greater than or less than the initial value.

As an example in which expressions are used instead of constant numeric values, here's the `polyspi` procedure using `for`:

```
to polyspi :start :angle :number
for [side :start [:start+:number-1]] [forward :side right :angle]
end
```

Most of the work in writing `for` is in evaluating the expressions that make up the first input list. Here is the program:

```
to for :values :instr
localmake "var first :values
local :var
localmake "initial run first butfirst :values
localmake "final run item 3 :values
localmake "step forstep
localmake "tester ~
          ifelse :step < 0 [[:value < :final]] [[:value > :final]]
forloop :initial
end
```

```
to forstep
if (count :values)=4 [output run last :values]
if :initial > :final [output -1]
output 1
end

to forloop :value
make :var :value
if run :tester [stop]
run :instr
forloop :value+:step
end
```

One slightly tricky part of this program is the instruction

```
local :var
```

near the beginning of `for`. The effect of this instruction is to make whatever variable is named by the first member of the first input local to `for`. As it turns out, this variable isn't given a value in `for` itself but only in its subprocedure `forloop`. (A *loop,* by the way, is a part of a program that is invoked repeatedly.) But I'm thinking of these three procedures as a unit and making the variable local to that whole unit. The virtue of this `local` instruction is that a program that uses `for` can invent variable names for `for` freely, without having to declare them local and without cluttering up the workspace with global variables. Also, it means that a procedure can invoke another procedure in the instruction list of a `for` without worrying about whether *that* procedure uses `for` itself. Here's the case I'm thinking of:

```
to a
for [i 1 5] [b]
end

to b
for [i 1 3] [print "foo]
end
```

Invoking `A` should print the word `foo` fifteen times: three times for each of the five invocations of `B`. If `for` didn't make `I` a local variable, the invocation of `for` within `B` would mess up the value of `I` in the outer `for` invoked by `A`. Got that?

Notice that the `make` instruction in `forloop` has `:var` as its first input, not `"var`. This instruction does not assign a new value to the variable `var`! Instead, it assigns a new value to the variable whose name is the value of `var`.

The version of `for` actually used in the Berkeley Logo library is a little more complicated than this one. The one shown here works fine as long as the instruction list input doesn't include `stop` or `output`, but it won't work for an example like the following. To check whether or not a number is prime, we must see if it is divisible by anything greater than 1 and smaller than the number itself:

```
to primep :num
for [trial 2 [:num-1]] [if divisiblep :num :trial [output "false]]
output "true
end

to divisiblep :big :small
output equalp remainder :big :small 0
end
```

This example will work in the Berkeley Logo `for`, but not in the version I've written in this chapter. The trouble is that the instruction

```
run :instr
```

in `forloop` will make `forloop` output `false` if a divisor is found, whereas we really want `primep` to output `false`! We'll see in Chapter 12 how to solve this problem.

Logo: an Extensible Language

There are two ways to look at a program like `for`. You can take it apart, as I've been doing in these last few paragraphs, to see how it works inside. Or you can just think of it as an extension to Logo, an iteration command that you can use as you'd use `repeat`, without thinking about how it works. I think both of these perspectives will be valuable to you. As a programming project, `for` demonstrates some rather advanced Logo techniques. But you don't have to think about those techniques each time you use `for`. Instead you can think of it as a primitive, as we've been doing prior to this chapter.

The fact that you can extend Logo's vocabulary this way, adding a new way to control iteration that looks just like the primitive `repeat`, is an important way in which Logo is more powerful than toy programming languages like C++ or Pascal. C++ has several iteration commands built in, including one like `for`, but if you think of a new one, there's no way you can add it to the language. In Logo this kind of language extension is easy. For example, here is a preview of a programming project I'm going to develop later

in this chapter. Suppose you're playing with spirals, and you'd like to see what happens if you change the line length *and* the turning angle at the same time. That is, you'd like to be able to say

```
multifor [[size 50 100 5] [angle 50 100 10]] [forward :size right :angle]
```

and have that be equivalent to the series of instructions

```
forward 50 right 50
forward 55 right 60
forward 60 right 70
forward 65 right 80
forward 70 right 90
forward 75 right 100
```

`Multifor` should step each of its variables each time around, stopping whenever any of them hits the final value. This tool strikes me as too specialized and complicated to provide in the Logo library, but it seems very appropriate for certain kinds of project. It's nice to have a language in which I can write it if I need it.

No Perfect Control Structures

Among enthusiasts of the Fortran family of programming languages (that is, all the languages in which you have to say ahead of time whether or not the value of some numeric variable will be an exact integer), there are fierce debates about the "best" control structure. (A *control structure* is a way of grouping instructions together, just as a *data structure* is a way of grouping data together. A list is a data structure. A procedure is a control structure. Things like `if`, `repeat`, and `for` are special control structures that group instructions in particular ways, so that a group of instructions can be evaluated conditionally or repeatedly.)

For example, all of the Fortran-derived languages have a control structure for numeric iteration, like my `for` procedure. But they differ in details. In some languages the iteration variable must be stepped by 1. In others the step value can be either 1 or –1. Still others allow any step value, as `for` does. Each of these choices has its defenders as the "best."

Sometimes the arguments are even sillier. When Fortran was first invented, its designers failed to make explicit what should happen if the initial value of an iteration variable is greater than the final value. That is, they left open the interpretation of a

Fortran `do` statement (that's what its numeric iteration structure is called) equivalent to this `for` instruction:

```
for [var 10 5 1] [print :var]
```

In this instruction I've specified a positive step (the only kind allowed in the Fortran `do` statement), but the initial value is greater than the final value. (What will `for` do in this situation?) Well, the first Fortran compiler, the program that translates a Fortran program into the "native" language of a particular computer, implemented `do` so that the statements controlled by the `do` were carried out once before the computer noticed that the variable's value was already too large. Years later a bunch of computer scientists decided that that behavior is "wrong"; if the initial value is greater than the final value, the statements shouldn't be carried out at all. This proposal for a "zero trip `do` loop" was fiercely resisted by old-timers who had by then written hundreds of programs that relied on the original behavior of `do`. Dozens of journal articles and letters to the editor carried on the battle.

The real moral of this story is that there is no right answer. The right control structure for *you* to use is the one that best solves *your* immediate problem. But only an extensible language like Logo allows you the luxury of accepting this moral. The Fortran people had to fight out their battle because they're stuck with whatever the standardization committee decides.

In the remainder of this chapter I'll present various kinds of control structures, each reflecting a different way of looking at the general idea of iteration.

Iteration Over a List

Numeric iteration is useful if the problem you want to solve is about numbers, as in the `primep` example, or if some arbitrary number is part of the rules of a game, like the seven stacks of cards in solitaire. But in most Logo projects, it's more common to want to carry out a computation for each member of a list, and for that purpose we have the `foreach` control structure:

```
? foreach [chocolate [rum raisin] pumpkin] [print sentence [I like] ?]
I like chocolate
I like rum raisin
I like pumpkin
```

In comparing `foreach` with `for`, one thing you might notice is the use of the question mark to represent the varying datum in `foreach`, while `for` requires a user-specified variable name for that purpose. There's no vital reason why I used these different mechanisms. In fact, we can easily implement a version of `foreach` that takes a variable name as an additional input. Its structure will then look similar to that of `for`:

```
to named.foreach :var :data :instr
local :var
if emptyp :data [stop]
make :var first :data
run :instr
named.foreach :var (butfirst :data) :instr
end
```

```
? named.foreach "flavor [lychee [root beer swirl]] ~
                        [print sentence [I like] :flavor]
I like lychee
I like root beer swirl
```

Just as in the implementation of `for`, there is a recursive invocation for each member of the data input. We assign that member as the value of the variable named in the first input, and then we `run` the instructions in the third input.

In order to implement the version of `foreach` that uses question marks instead of named variables, we need a more advanced version of `run` that says "run these instructions, but using this value wherever you see a question mark." Berkeley Logo has this capability as a primitive procedure called `apply`. It takes two inputs, a *template* (an instruction list with question marks) and a list of values. The reason that the second input is a list of values, rather than a single value, is that `apply` can handle templates with more than one slot for values.

```
? apply [print ?+3] [5]
8
? apply [print word first ?1 first ?2] [Peter Dickinson]
PD
```

It's possible to write `apply` in terms of `run`, and I'll do that shortly. But first, let's just take advantage of Berkeley Logo's built-in `apply` to write a simple version of `foreach`:

```
to foreach :list :template
if emptyp :list [stop]
apply :template (list first :list)
foreach (butfirst :list) :template
end
```

`Apply`, like `run`, can be either a command or an operation depending on whether its template contains complete Logo instructions or a Logo expression. In this case, we are using `apply` as a command.

The version of `foreach` in the Berkeley Logo library can take more than one data input along with a multi-input template, like this:

```
? (foreach [John Paul George Ringo] [rhythm bass lead drums]
           [print (sentence ?1 "played ?2)]
John played rhythm
Paul played bass
George played lead
Ringo played drums
```

We can implement this feature, using a special notation in the title line of `foreach` to notify Logo that it accepts a variable number of inputs:

```
to foreach [:inputs] 2
foreach.loop (butlast :inputs) (last :inputs)
end

to foreach.loop :lists :template
if emptyp first :lists [stop]
apply :template firsts :lists
foreach.loop (butfirsts :lists) :template
end
```

First look at the title line of `foreach`. It tells Logo that the word `inputs` is a formal parameter—the name of an input. Because `:inputs` is inside square brackets, however, it represents not just one input, but any number of inputs in the invocation of `foreach`. The values of all those inputs are collected as a list, and that list is the value of `inputs`. Here's a trivial example:

```
to demo [:stuff]
print sentence [The first input is] first :stuff
print sentence [The others are] butfirst :stuff
end

? (demo "alpha "beta "gamma)
The first input is alpha
The others are beta gamma
```

As you know, the Logo procedures that accept a variable number of inputs have a *default* number that they accept without using parentheses; if you want to use more or fewer than that number, you must enclose the procedure name and its inputs in parentheses, as I've done here with the `demo` procedure. Most Logo primitives that accept a variable number of inputs have two inputs as their default number (for example, `sentence`, `sum`, `word`) but there are exceptions, such as `local`, which takes one input if parentheses are not used. When you write your own procedure with a single input name in brackets, its default number of inputs is zero unless you specify another number. `Demo`, for example, has zero as its default number. If you look again at the title line of `foreach`, you'll see that it ends with the number `2`; that tells Logo that `foreach` expects two inputs by default.

`Foreach` uses all but its last input as data lists; the last input is the template to be applied to the members of the data lists. That's why `foreach` invokes `foreach.loop` as it does, separating the two kinds of inputs into two variables.

Be careful when reading the definition of `foreach.loop`; it invokes procedures named `firsts` and `butfirsts`. These are not the same as `first` and `butfirst`! Each of them takes a *list of lists* as its input, and outputs a list containing the first members of each sublist, or all but the first members, respectively:

```
? show firsts [[a b c] [1 2 3] [y w d]]
[a 1 y]
? show butfirsts [[a b c] [1 2 3] [y w d]]
[[b c] [2 3] [w d]]
```

It would be easy to write `firsts` and `butfirsts` in Logo:

```
to firsts :list.of.lists
output map "first :list.of.lists
end

to butfirsts :list.of.lists
output map "butfirst :list.of.lists
end
```

but in fact Berkeley Logo provides these operations as primitives, because implementing them as primitives makes the iteration tools such as `foreach` and `map` (which, as we'll see, also uses them) much faster.

Except for the use of `firsts` and `butfirsts` to handle the multiple data inputs, the structure of `foreach.loop` is exactly like that of the previous version of `foreach` that only accepts one data list.

Like `for`, the version of `foreach` presented here can't handle instruction lists that include `stop` or `output` correctly.

Implementing `Apply`

Berkeley Logo includes `apply` as a primitive, for efficiency, but we could implement it in Logo if necessary. In this section, so as not to conflict with the primitive version, I'll use the name `app` for my non-primitive version of `apply`, and I'll use the percent sign (`%`) as the placeholder in templates instead of question mark.

Here is a simple version of `app` that allows only one input to the template:

```
to app :template :input.value
run :template
end

to %
output :input.value
end
```

This is so simple that it probably seems like magic. `App` seems to do nothing but `run` its template as though it were an ordinary instruction list. The trick is that a template *is* an instruction list. The only unusual thing about a template is that it includes special symbols (`?` in the real `apply`, `%` in `app`) that represent the given value. We see now that those special symbols are really just ordinary names of procedures. The question mark (`?`) procedure is a Berkeley Logo primitive; I've defined the analogous `%` procedure here for use by `app`.

The `%` procedure outputs the value of a variable, `input.value`, that is local to `app`. If you invoke `%` in some context other than an `app` template, you'll get an error message because that variable won't exist. Logo's dynamic scope makes it possible for `%` to use `app`'s variable.

The real `apply` accepts a procedure name as argument instead of a template:

```
? show apply "first [Logo]
L
```

We can extend `app` to accept named procedures, but the definition is somewhat messier:

```
to app :template.or.name :input.value
ifelse wordp :template.or.name ~
       [run list :template.or.name "%] ~
       [run :template.or.name]
end
```

If the first input is a word, we construct a template by combining that procedure name with a percent sign for its input. However, in the rest of this section I'll simplify the discussion by assuming that `app` accepts only templates, not procedure names.

So far, `app` takes only one value as input; the real `apply` takes a list of values. I'll extend `app` to match:

```
to app :template :input.values
run :template
end

to % [:index 1]
output item :index :input.values
end
```

No change is needed to `app`, but `%` has been changed to use another new notation in its title line. `Index` is the name of an *optional input*. Although this notation also uses square brackets, it's different from the notation used in `foreach` because the brackets include a *default value* as well as the name for the input. This version of `%` accepts either no inputs or one input. If `%` is invoked with one input, then the value of that input will be associated with the name `index`, just as for ordinary inputs. If `%` is invoked with no inputs, then `index` will be given the value 1 (its default value).

```
? app [print word first (% 1) first (% 2)] [Paul Goodman]
PG
```

A percent sign with a number as input selects an input value by its position within the list of values. A percent sign by itself is equivalent to `(% 1)`.

The notation `(% 1)` isn't as elegant as the `?1` used in the real `apply`. You can solve that problem by defining several extra procedures:

```
to %1                to %2                to %3
output (% 1)         output (% 2)         output (% 3)
end                  end                  end
```

Berkeley Logo recognizes the notation `?2` and automatically translates it to `(? 2)`, as you can see by this experiment:

```
? show runparse [print word first ?1 first ?2]
[print word first ( ? 1 ) first ( ? 2 )]
```

(The primitive operation `runparse` takes a list as input and outputs the list as it would be modified by Logo when it is about to be run. That's a handwavy description, but the internal workings of the Logo interpreter are too arcane to explore here.)

Unlike the primitive `apply`, this version of `app` works only as a command, not as an operation. It's easy to write a separate version for use as an operation:

```
to app.oper :template :input.values
output run :template
end
```

It's not so easy in non-Berkeley versions of Logo to write a single procedure that can serve both as a command and as an operation. Here's one solution that works in versions with `catch`:

```
to app :template :input.values
catch "error [output run :template]
ignore error
end
```

This isn't an ideal solution, though, because it doesn't report errors other than "`run` didn't output to `output`." It could be improved by testing the error message more carefully instead of just ignoring it.

Berkeley Logo includes a mechanism that solves the problem more directly, but it's not very pretty:

```
to app :template :input.values
.maybeoutput run :template
end
```

The primitive command `.maybeoutput` is followed by a Logo expression that may or may not produce a value. If so, that value is output, just as it would be by the ordinary `output` command; the difference is that it's not considered an error if no value is produced.

From now on I'll use the primitive `apply`. I showed you `app` for two reasons. First, I think you'll understand `apply` better by seeing how it can be implemented. Second, this implementation may be useful if you ever work in a non-Berkeley Logo.

Mapping

So far the iteration tools we've created apply only to commands. As you know, we also have the operation `map`, which is similar to `foreach` except that its template is an expression (producing a value) rather than an instruction, and it accumulates the values produced for each member of the input.

```
? show map [?*?] [1 2 3 4]
[1 4 9 16]
? show map [first ?] [every good boy does fine]
[e g b d f]
?
```

When implementing an iteration tool, one way to figure out how to write the program is to start with a specific example and generalize it. For example, here's how I'd write the example about squaring the numbers in a list without using `map`:

```
to squares :numbers
if emptyp :numbers [output []]
output fput ((first :numbers) * (first :numbers)) ~
            (squares butfirst :numbers)
end
```

`Map` is very similar, except that it applies a template to each datum instead of squaring it:

```
to map :template :values
if emptyp :values [output []]
output fput (apply :template (list first :values)) ~
            (map :template butfirst :values)
end
```

You may be wondering why I used `fput` rather than `sentence` in these procedures. Either would be just as good in the example about squares of numbers, because each datum is a single word (a number) and each result value is also a single word. But it's important to use `fput` in an example such as this one:

```
to swap :pair
output list last :pair first :pair
end

? show map [swap ?] [[Sherlock Holmes] [James Pibble] [Nero Wolfe]]
[[Holmes Sherlock] [Pibble James] [Wolfe Nero]]

? show map.se [swap ?] [[Sherlock Holmes] [James Pibble] [Nero Wolfe]]
[Holmes Sherlock Pibble James Wolfe Nero]
```

Berkeley Logo does provide an operation `map.se` in which `sentence` is used as the combiner; sometimes that's what you want, but not, as you can see, in this example. (A third possibility that might occur to you is to use `list` as the combiner, but that never turns out to be the right thing; try writing a `map.list` and see what results it gives!)

As in the case of `foreach`, the program gets a little more complicated when we extend it to handle multiple data inputs. Another complication that wasn't relevant to `foreach` is that when we use a word, rather than a list, as the data input to `map`, we must use `word` as the combiner instead of `fput`. Here's the complete version:

```
to map :map.template [:template.lists] 2
op map1 :template.lists 1
end

to map1 :template.lists :template.number
if emptyp first :template.lists [output first :template.lists]
output combine (apply :map.template firsts :template.lists)
               (map1 bfs :template.lists :template.number+1)
end

to combine :this :those
if wordp :those [output word :this :those]
output fput :this :those
end
```

This is the actual program in the Berkeley Logo library. One feature I haven't discussed until now is the variable `template.number` used as an input to `map1`. Its purpose is to allow the use of the number sign character `#` in a template to represent the position of each datum within its list:

```
? show map [list ? #] [a b c]
[[a 1] [b 2] [c 3]]
```

The implementation is similar to that of `?` in templates:

```
to #
output :template.number
end
```

It's also worth noting the base case in `map1`. When the data input is empty, we must output either the empty word or the empty list, and the easiest way to choose correctly is to return the empty input itself.

Mapping as a Metaphor

In this chapter, we got to the idea of mapping by this route: iteration, numeric iteration, other kinds of iteration, iteration on a list, iterative commands, iterative operations, mapping. In other words, we started thinking about the mapping tool as a particular kind of repetition in a computer program.

But when I first introduced `map` as a primitive operation, I thought about it in a different way. Never mind the fact that it's *implemented* through repetition. Instead think of it as extending the power of the idea of a list. When we started thinking about lists, we thought of the list as one complete entity. For example, consider this simple interaction with Logo:

```
? print count [how now brown cow]
4
```

`Count` is a primitive operation. It takes a list as input, and it outputs a number that is a property of the entire list, namely the number of members in the list. There is no need to think of `count` as embodying any sort of repetitive control structure. Instead it's one kind of handle on the *data* structure called a list.

There are other operations that manipulate lists, like `equalp` and `memberp`. You're probably in the habit of thinking of these operations as "happening all at once," not as examples of iteration. And that's a good way to think of them, even though it's also possible to think of them as iterative. For example, how does Logo know the `count` of a list? How would *you* find out the number of members of a list? One way would be to count them on your fingers. That's an iteration. Logo actually does the same thing, counting off the list members one at a time, as it would if we implemented `count` recursively:

```
to cnt :list
if emptyp :list [output 0]
output 1+cnt butfirst :list
end
```

I'm showing you that the "all at once" Logo primitives can be considered as iterative because, in the case of `map`, I want to shift your point of view in the opposite direction. We started thinking of `map` as iterative; now I'd like you to think of it as happening all at once.

Wouldn't it be nice if we could say

```
? show 1+[5 10 15]
[6 11 16]
```

That is, I'd like to be able to "add 1 to a list." I want to think about it that way, not as "add 1 to each member of a list." The metaphor is that we're doing something to the entire list at once. Well, we can't quite do it that way, but we can say

```
? show map [1+?] [5 10 15]
[6 11 16]
```

Instead of thinking "Well, first we add 1 to 5, which gives us 6; then we add..." you should think "we started with a list of three numbers, and we've transformed it into another list of three numbers using the operation add-one."

Other Higher Order Functions

Along with `map`, you learned about the higher order functions `reduce`, which combines all of the members of a list into a single result, and `filter`, which selects some of the members of a list. They, too, are implemented by combining recursion with `apply`. Here's the Berkeley Logo library version of `reduce`:

```
to reduce :reduce.function :reduce.list
if emptyp butfirst :reduce.list [output first :reduce.list]
output apply :reduce.function (list (first :reduce.list)
                                    (reduce :reduce.function
                                            butfirst  :reduce.list))
end
```

If there is only one member, output it. Otherwise, recursively reduce the butfirst of the data, and apply the template to two values, the first datum and the result from the recursive call.

The Berkeley Logo implementation of `filter` is a little more complicated, for some of the same reasons as that of `map`: the ability to accept either a word or a list, and the `#` feature in templates. So I'll start with a simpler one:

```
to filter :template :data
if emptyp :data [output []]
if apply :template (list first :data) ~
   [output fput (first :data)
                (filter :template butfirst :data)]
output filter :template butfirst :data
end
```

If you understand that, you should be able to see the fundamentally similar structure of the library version despite its extra details:

```
to filter :filter.template :template.list [:template.number 1]
localmake "template.lists (list :template.list)
if emptyp :template.list [output :template.list]
if apply :filter.template (list first :template.list) ~
   [output combine (first :template.list)
                   (filter :filter.template (butfirst :template.list)
                           :template.number+1)]
output (filter :filter.template (butfirst :template.list)
               :template.number+1)
end
```

Where `map` used a helper procedure `map1` to handle the extra input `template.number`, `filter` uses an alternate technique, in which `template.number` is declared as an optional input to `filter` itself. When you invoke `filter` you always give it the default two inputs, but it invokes itself recursively with three.

Why does `filter` need a local variable named `template.lists`? There was a variable with that name in `map` because it accepts more than one data input, but `filter` doesn't, and in fact there is no reference to the value of `template.lists` within `filter`. It's there because of another feature of templates that I haven't mentioned: you can use the word `?rest` in a template to represent the portion of the data input to the right of the member represented by `?` in this iteration:

```
to remove.duplicates :list
output filter [not memberp ? ?rest] :list
end

?  show remove.duplicates [ob la di ob la da]
[di ob la da]
```

Since `?rest` is allowed in `map` templates as well as in `filter` templates, its implementation must be the same for both:

```
to ?rest [:which 1]
output butfirst item :which :template.lists
end
```

Mapping Over Trees

It's time to move beyond the iteration tools in the Logo library and invent our own new ones.

So far, in writing operations on lists, we've ignored any sublist structure within the list. We do something for each top-level member of the input list. It's also possible to take advantage of the complex structures that lists make possible. For example, a list can be used to represent a *tree,* a data structure in which each branch can lead to further branches. Consider this list:

```
[[the [quick brown] fox] [[jumped] [over [the [lazy] dog]]]]
```

My goal here is to represent a sentence in terms of the phrases within it, somewhat like the sentence diagrams you may have been taught in elementary school. This is a list with two members; the first member represents the subject of the sentence and the second represents the predicate. The predicate is further divided into a verb and a prepositional phrase. And so on. (A representation something like this, but more detailed, is used in any computer program that tries to understand "natural language" interaction.)

Suppose we want to convert each word of this sentence to capital letters, using Berkeley Logo's `uppercase` primitive that takes a word as input. We can't just say

```
map [uppercase ?] ~
    [[the [quick brown] fox] [[jumped] [over [the [lazy] dog]]]]
```

because the members of the sentence-list aren't words. What I want is a procedure `map.tree` that applies a template to each *word* within the input list but maintains the shape of the list:

```
? show map.tree [uppercase ?]~
     [[the [quick brown] fox] [[jumped] [over [the [lazy] dog]]]]
[[THE [QUICK BROWN] FOX] [[JUMPED] [OVER [THE [LAZY] DOG]]]]
```

After our previous adventures in mapping, this one is relatively easy:

```
to map.tree :template :tree
if wordp :tree [output apply :template (list :tree)]
if emptyp :tree [output []]
output fput (map.tree :template first :tree) ~
            (map.tree :template butfirst :tree)
end
```

This is rather a special-purpose procedure; it's only good for trees whose "leaves" are words. That's sometimes the case but not always. But if you're dealing with sentence trees like the one in my example, you might well find several uses for a tool like this. For now, I've introduced it mainly to make the point that the general idea of iteration can take many different forms, depending on the particular project you're working on. (Technically, this is *not* an iteration, because it doesn't have a two-part structure in which the first part is to perform one step of a computation and the second part is to perform all the rest of the steps. `Map.tree` does have a two-part structure, but *both* parts are recursive calls that might carry out several steps. But `map.tree` does generalize the broad idea of dividing a large computation into similar individual pieces. We'll go into the nature of iteration more carefully in a moment.)

Iteration and Tail Recursion

If you look back at the introduction to recursion in the first volume, you'll find that some recursive commands seem to be carrying out an iteration, like `down`, `countdown`, or `one.per.line`. (In this chapter we've seen how to implement `countdown` using `for`, and you should easily be able to implement `one.per.line` using `foreach`. `Down` isn't exactly covered by either of those tools; can you see why I call it an iterative problem anyway?) Other recursive commands don't seem to be repeating or almost-repeating something, like `downup` or `hanoi`. The difference is that these commands don't do something completely, then forget about it and go on to the next repetition. Instead,

the first invocation of `downup`, for example, still has work of its own to do after all the lower-level invocations are finished.

It turns out that a command that is *tail* recursive is one that can be thought of as carrying out an iteration. A command that invokes itself somewhere before the last instruction is not iterative. But the phrase "tail recursive" doesn't *mean* "equivalent to an iteration." It just happens to work out, for commands, that the two concepts are equivalent. What "tail recursive" means, really, is "invokes itself just before stopping."

I've said before that this isn't a very important thing to worry about. The reason I'm coming back to it now is to try to clear up a confusion that has been part of the Logo literature. Logo implementors talk about tail recursion because there is a tricky way to implement tail recursion that takes less memory than the more general kind of recursion. Logo *teachers,* on the other hand, tend to say "tail recursive" when they really mean "iterative." For example, teachers will ask, "Should we teach tail recursion first and then the general case?" What's behind this question is the idea that iteration is easier to understand than recursion. (By the way, this is a hot issue. Most Logo teachers would say yes; they begin by showing their students an iterative command like `poly` or `polyspi`. I generally say no; you may recall that the first recursive procedure I showed you is `downup`. One reason is that I expect some of my readers have programmed in Pascal or C, and I want to make it as hard as possible for such readers to convince themselves that recursion is just a peculiar way to express the idea of iteration.)

There are two reasons people should stop making a fuss about tail recursion. One is that they're confusing an idea about control structures (iteration) with a Logo implementation strategy (tail recursion). The second is that this way of thinking directs your attention to commands rather than operations. (When people think of iterative procedures as "easier," it's always commands that they have in mind. Tail recursive operations are, if anything, less straightforward than versions that are non-tail recursive.) Operations are more important; they're what gives Logo much of its flexibility. And the best way to think about recursive operations isn't in implementation terms but in terms of data transformation abstractions like mapping, reduction, and filters.

Multiple Inputs to `For`

Earlier I promised you `multifor`, a version of `for` that controls more than one numeric variable at a time. Its structure is very similar to that of the original `for`, except that we use `map` or `foreach` (or `firsts` or `butfirsts`, which are implicit uses of `map`) in almost every instruction to carry out `for`'s algorithm for each of `multifor`'s numeric variables.

```
to multifor :values.list :instr
localmake "vars firsts :values.list
local :vars
localmake "initials map "run firsts butfirsts :values.list
localmake "finals map [run item 3 ?] :values.list
localmake "steps (map "multiforstep :values.list :initials :finals)
localmake "testers map [ifelse ? < 0 [[?1 < ?2]] [[?1 > ?2]]] :steps
multiforloop :initials
end

to multiforstep :values :initial :final
if (count :values)=4 [output run last :values]
if :initial > :final [output -1]
output 1
end

to multiforloop :values
(foreach :vars :values [make ?1 ?2])
(foreach :values :finals :testers [if run ?3 [stop]])
run :instr
multiforloop (map [?1+?2] :values :steps)
end
```

This is a very dense program; I wouldn't expect anyone to read and understand it from a cold start. But if you compare it to the implementation of `for` on page 184, you should be able to make sense of how each line is transformed in this version.

Here is an example you can try:

```
? multifor [[a 10 100 5] [b 100 10 -10]] ~
           [print (sentence :a "+ :b "= (:a + :b))]
10 + 100 = 110
15 + 90 = 105
20 + 80 = 100
25 + 70 = 95
30 + 60 = 90
35 + 50 = 85
40 + 40 = 80
45 + 30 = 75
50 + 20 = 70
55 + 10 = 65
?
```

The Evaluation Environment Bug

There's a problem with all of these control structure tools that I haven't talked about. The problem is that each of these tools uses `run` or `apply` to evaluate an expression that's provided by the calling procedure, but the expression is evaluated with the tool's local variables active, in addition to those of the calling procedure. This can lead to unexpected results if the name of a variable used in the expression is the same as the name of one of the local variables in the tool. For example, `forloop` has an input named `final`. What happens if you try

```
to grade :final
for [midterm 10 100 10] [print (sum :midterm :final) / 2]
end

? grade 50
```

Try this example with the implementation of `for` in this chapter, not with the Logo library version. You might expect each iteration to add 10 and 50, then 20 and 50, then 30 and 50, and so on. That is, you wanted to add the iteration variable `midterm` to the input to `grade`. In fact, though, the variable that contributes to the sum is `forloop`'s `final`, not `grade`'s `final`.

The way to avoid this problem is to make sure you don't use variables in superprocedures of these tools with the same names as the ones inside the tools. One way to ensure that is to rewrite all the tool procedures so that their local variables have bizarre names:

```
to map :template :inputs
```

becomes

```
to map :map.qqzzqxx.template :map.qqzzqxx.inputs
```

Of course, you also have to change the names wherever they appear inside the definition, not just on the title line. You can see why I preferred not to present the procedures to you in that form!

It would be a better solution to have a smarter version of `run`, which would allow explicit control of the *evaluation environment*—the variable names and values that should be in effect while evaluating `run`'s input. Some versions of Lisp do have such a capability.