# 12 Macros

I mentioned that the versions of `for` and `foreach` shown in Chapter 10 don't work if their instruction templates include `stop` or `output` commands. The problem is that we don't want, say, `foreach` to stop; we want the procedure that *invoked* `foreach` to stop.

What we need to fix this problem is a way for a subprocedure to *make its caller* carry out some action. That is, we want something like `run`, but the given expression should be run in a different context. Berkeley Logo includes a mechanism, called *macros,* to allow this solution. As I write this in 1996, no other version of Logo has macros, although this capability is commonly provided in most versions of Logo's cousin, the programming language Lisp.*

## Localmake

Before we fix `for` and `foreach`, and even before I explain in detail what a macro is, I think it's best to start with a simple but practical example. Throughout this book I've been using a command called `localmake` that creates a local variable and assigns it a value. The instruction

```
localmake "fred 87
```

is an abbreviation for the two instructions

```
local "fred
make "fred 87
```

* Readers who are familiar with Lisp macros should take note that Logo macros do not prevent argument evaluation.

Any version of Logo will allow those two separate instructions. It's tempting to write a procedure combining them:

```
to localmake :name :value                    ;; wrong!
local :name
make :name :value
end
```

What's wrong with this solution? If you're not sure, define `localmake` as above and try an example, like this:

```
to trial
localmake "fred 87
print :fred
end

? trial
fred has no value  in trial
[print :fred]
```

When `trial` invokes `localmake`, a local variable named `fred` is created *inside the invocation of* `localmake`! That variable is then assigned the value 87. Then `localmake` returns to `trial`, and `localmake`'s local variables disappear. Back in `trial`, there is no variable named `fred`.

Here's the solution. If `localmake` is an ordinary procedure, there's no way it can create a local variable in its caller. So we have to define `localmake` as a special kind of procedure:

```
.macro localmake :name :value
output (list "local (word "" :name) "make (word "" :name) :value)
end
```

The command `.macro` is like `to`, except that the procedure it defines is a macro instead of an ordinary procedure. (It's a Logo convention that advanced primitives that could be confusing to beginners have names beginning with a period.)

It's a little hard to read exactly what this procedure does, so for exploratory purposes I'll define an ordinary procedure with the same body:

```
to lmake :name :value
output (list "local (word "" :name) "make (word "" :name) :value)
end
```

```
? show lmake "fred 87
[local "fred make "fred 87]
```

As you see from the example, `lmake` outputs a list containing the instructions that we would like its caller to carry out.

The macro `localmake` outputs the same list of instructions. But, because `localmake` is a macro, that output is then *evaluated* by the procedure that called `localmake`. If `trial` is run using the macro version of `localmake` instead of the ordinary procedure version that didn't work, the effect is as if `trial` contained a `local` instruction and a `make` instruction in place of the one `localmake` invocation. (If you defined the incorrect version of `localmake`, you can say

```
erase "localmake
```

and then the official version will be reloaded from the library the next time you use it.)

You may find the expression `(word "" :name)` that appears twice in the definition of `localmake` confusing. At first glance, it seems that there is already a quotation mark in the first input to `localmake`, namely, `"fred`. But don't forget that that quotation mark is not part of the word! For example, when you say

```
print "fred
```

Logo doesn't print a quotation mark. What the quotation mark means to Logo is "use the word that follows as the value for this input, rather than taking that word as the name of a procedure and invoking that procedure to find the input value." In this example, the first input to `localmake` is the word `fred` itself, rather than the result of invoking a procedure named `fred`. If we want to construct an instruction such as

```
local "fred
```

based on this input, we must put a quotation mark in front of the word explicitly.

In fact, so far I've neglected to deal with the fact that a similar issue about quotation may arise for the value being assigned to the variable. In the `trial` example I used the value 87, a number, which is *self-evaluating;* when a number is typed into Logo as an expression, the number itself is the value of the expression. But if the value is a non-numeric word, then a quotation mark must be used for it, too. The version of `localmake` shown so far would fail in a case like

```
localmake "greeting "hello
```

because the macro would return the list

```
[local "greeting make "greeting hello]
```

which, when evaluated, would try to invoke a procedure named `hello` instead of using the word itself as the desired value.

The most straightforward solution is to write a procedure that will include a quotation mark only when it's needed:

```
.macro localmake :name :value
output (list "local (quoted :name) "make (quoted :name) (quoted :value))
end

to quoted :thing
if numberp :thing [output :thing]
if listp :thing [output :thing]
output word "" :thing
end
```

A somewhat less obvious solution, but one I find more appealing, is to avoid the entire issue of quotation by putting the inputs to `make` in a list, which we can do by using `apply`:

```
.macro localmake :name :value
output (list "local (word "" :name) "apply ""make (list :name :value))
end
```

On the other hand, it may take some thinking to convince yourself that the `""make` in that version is correct!

## Backquote

Even a simple macro like `localmake` is very hard to read, and hard to write correctly, because of all these invocations of `list` and `word` to build up a structure that's partly constant and partly variable. It would be nice if we could use a notation like

```
[local "NAME apply "make [NAME VALUE]]
```

for an "almost constant" list in which only the words in capital letters would be replaced by values of variables.

That particular notation can't work, because in Logo the case of letters doesn't matter when a word is used as the name of something. But we do have something almost as good. We can say

```
`[local ,[word "" :name] apply "make [,[:name] ,[:value]]]
```

The first character in that line, before the opening bracket, is a *backquote*, which is probably near the top left corner of your keyboard. To Logo, it's just an ordinary character, and happens to be the name of a procedure in the Berkeley Logo library. The list that follows the backquote above is the input to the procedure.

What the ` procedure does with its input list is to make a copy, but wherever a word containing only a comma (`,`) appears, what comes next must be a list, which is `run` to provide the value for that position in the copy. I've put the commas right next to the lists that follow them, but this doesn't matter; whenever Logo sees a bracket, it delimits the words on both sides of the bracket, just as if there were spaces around the bracket.

So if `:name` has the value `fred` and `:value` has the value 87, then this sample invocation of ` has the value

```
[local "fred apply "make [fred 87]]
```

Like macros, backquote is a feature that Berkeley Logo borrows from Lisp. It's not hard to implement:

```
to ` :backq.list
if emptyp :backq.list [output []]
if equalp first :backq.list ", ~
   [output fput run first butfirst :backq.list
               ` butfirst butfirst :backq.list]
if equalp first :backq.list ",@ ~
   [output sentence run first butfirst :backq.list
                    ` butfirst butfirst :backq.list]
if wordp first :backq.list ~
   [output fput first :backq.list ` butfirst :backq.list]
output fput ` first :backq.list ` butfirst :backq.list
end
```

This procedure implements one feature I haven't yet described. If the input to ` contains the word `,@` (comma atsign), then the next member of the list must be a list, which is `run` as for comma, but the *members* of the result are inserted in the output, instead of the result as a whole. Here's an example:

```
? show `[start ,[list "a "b] middle ,@[list "a "b] end]
[start [a b] middle a b end]
```

Using backquote, we could rewrite `localmake` a little more readably:

```
.macro localmake :name :value
output `[local ,[word "" :name] apply "make [,[:name] ,[:value]]]
end
```

In practice, though, I have to admit that the Berkeley Logo library doesn't use backquote in its macro definitions because it's noticeably slower than constructing the macro with explicit calls to `list` and `word`.

By the way, this implementation of backquote isn't as complex as some Lisp versions. Most importantly, there is no provision for *nested* backquotes, that is, for an invocation of backquote within the input to backquote. (Why would you want to do that? Think about a macro whose job is to generate a definition for another macro.)

## Implementing Iterative Commands

It's time to see how macros can be used to implement iterative control structures like `for` and `foreach` correctly. I'll concentrate on `foreach` because it's simpler to implement, but the same ideas apply equally well to `for`.

Perhaps the most obvious approach is to have the `foreach` macro output a long instruction list in which the template is applied to each member of the list. That is, if we say

```
foreach [a b c] [print ?]
```

then the `foreach` macro should output the list

```
[print "a print "b print "c]
```

To achieve precisely this result we'd have to look through the template for question marks, replacing each one with a possibly quoted datum. Instead it'll be easier to generate the uglier but equivalent instruction list

```
[apply [print ?] [a] apply [print ?] [b] apply [print ?] [c]]
```

this way:

```
.macro foreach :data :template
output map.se [list "apply :template (list ?)] :data
end
```

(To simplify the discussion, I'm writing a version of `foreach` that only takes two inputs. You'll see in a moment that the implementation will be complicated by other considerations, so I want to avoid unnecessary complexity now. At the end I'll show you the official, complete implementation.)

This version works correctly, and it's elegantly written. We could stop here. Unfortunately, this version is inefficient, for two reasons. First, it uses another higher order procedure, `map.se`, to construct the list of instructions to evaluate. Second, for a large data input, we construct a very large instruction list, using lots of computer memory, just so that we can evaluate the instructions once and throw the list away.

Another approach is to let the `foreach` macro invoke itself recursively. This is a little tricky; you'll see that `foreach` does not actually invoke itself within itself. Instead, it constructs an instruction list that contains another use of `foreach`. For example, the instruction

```
foreach [a b c] [print ?]
```

will generate the instruction list

```
[apply [print ?] [a] foreach [b c] [print ?]]
```

Here's how to write that:

```
.macro foreach :data :template
output `[apply ,[:template] [,[first :data]]
         foreach ,[butfirst :data] ,[:template]]
end
```

In this case the desired instruction list is long enough so that I've found it convenient to use the backquote notation to express my intentions. If you prefer, you could say

```
.macro foreach :data :template
output (list "apply :template (list (first :data))
             "foreach (butfirst :data) :template)
end
```

This implementation (in either the backquote version or the explicit list constructor version) avoids the possibility of constructing huge instruction lists; the constructed list has only the computation for the first datum and a recursive `foreach` that handles the entire rest of the problem.

But this version is still slower than the non-macro implementation of `foreach` given in Chapter 10. Constructing an instruction list and then evaluating it is just a slower

process than simply doing the necessary computation within `foreach` itself. And that earlier approach works fine unless the template involves `stop`, `output`, or `local`. We could have our cake and eat it too if we could find a way to use the non-macro approach, but notice when the template tries to stop its computation. This version is quite a bit trickier than the ones we've seen until now:

```
.macro foreach :data :template
catch "foreach.catchtag
      [output foreach.done runresult [foreach1 :data :template]]
output []
end

to foreach1 :data :template
if emptyp :data [throw "foreach.catchtag]
apply :template (list first :data)
.maybeoutput foreach1 butfirst :data :template
end

to foreach.done :foreach.result
if emptyp :foreach.result [output [stop]]
output list "output quoted first :foreach.result
end
```

To help you understand how this works, let's first consider what happens if the template does not include `stop` or `output`. In that case, the program structure is essentially this:

```
.macro simpler.foreach :data :template
catch "foreach.catchtag
      [this.stuff.never.invoked run [simpler.foreach1 :data :template]]
output []
end

to simpler.foreach1 :data :template
if emptyp :data [throw "foreach.catchtag]
apply :template (list first :data)
simpler.foreach1 butfirst :data :template
end
```

The instruction list that's evaluated by the `catch` runs a smaller instruction list that invokes `simpler.foreach1`. That procedure is expected to output a value, which is then used as the input to some other computation (namely, `foreach.done` in the actual version). But when `simpler.foreach1` reaches its base case, it doesn't output anything; it `throw`s back to the instruction after the `catch`, which outputs an empty list.

So all of the work of `foreach` is done within these procedures; the macro outputs an empty instruction list, which is evaluated by the caller of `foreach`, but that evaluation has no effect.

Now forget about the `simpler` version and return to the actual `foreach`. What if the template carries out a `stop` or `output`? If that happens, `foreach1` will never reach its base case, and will therefore not `throw`. It will either stop or output a value. The use of `.maybeoutput` in `foreach1` is what makes it possible for `foreach1` to function either as a command (if it stops) or as an operation (if it outputs) without causing an error when it invokes itself recursively. If the recursive invocation stops, so does the outer invocation. If the recursive invocation outputs a value, the outer invocation outputs that value.

`Foreach` invoked `foreach1` using Berkeley Logo's `runresult` primitive operation. `Runresult` is just like `run`, except that it always outputs a value, whether or not the computation that it runs produces a value. If so, then `runresult` outputs a one-member list containing the value. If not, then `runresult` outputs an empty list.

The output from `runresult` is used as input to `foreach.done`, whose job is to construct an instruction list as the overall output from the `foreach` macro. If the input to `foreach.done` is empty, that means that the template included a `stop`, and so `foreach` should generate a `stop` instruction to be evaluated by its caller. If the input isn't empty, then the template included an `output` instruction, and `foreach` should generate an `output` instruction as its return value.

This version is quite fast, and handles `stop` and `output` correctly. It does not, however, handle `local` correctly; the variable will be local to `foreach1`, not to the caller. It was hard to decide which version to use in the Berkeley Logo library, but slowing down every use of `foreach` seemed too high a price to pay for `local`. That's why, for example, procedure `onegame` in the solitaire program of Chapter 4 includes the instructions

```
local map [word "num ?] :numranks
foreach :numranks [make word "num ? 4]
```

instead of the more natural

```
foreach :numranks [localmake word "num ? 4]
```

That single instruction would work with the first implementation of `foreach` in this chapter, but doesn't work with the actual Berkeley Logo implementation!

## Debugging Macros

It's easy to make mistakes when writing a macro, because it's hard to keep straight what has to be quoted and what doesn't, for example. And it's hard to debug a macro, because you can't easily see the instruction list that it outputs. You can't say

```
show foreach ...
```

because the output from `foreach` is *evaluated,* not passed on to `show`.

One solution is to trace the macro.

```
? trace "foreach
? foreach [a b c] [print ?]
( foreach [a b c] [print ?] )
a
b
c
foreach outputs []
? foreach [a b 7 c] [if numberp ? [stop] print ?]
( foreach [a b 7 c] [if numberp ? [stop] print ?] )
a
b
foreach outputs [stop]
Can only use stop inside a procedure
```

In this case, I got an error message because, just as the message says, it doesn't make sense to use `stop` in a template unless this invocation of `foreach` is an instruction inside a procedure definition. Here I invoked `foreach` directly at the Logo prompt.

The Berkeley Logo library provides another solution, a `macroexpand` operation that takes as its input a Logo expression beginning with the name of a macro. It outputs the expression that the macro would output, without causing that expression to be evaluated:

```
? show macroexpand [foreach [a b 7 c] [if numberp ? [stop] print ?]]
a
b
[stop]
```

This time I didn't get an error message, because the instruction list that `foreach` outputs wasn't actually evaluated; it became the input to `show`, which is why it appears at the end of the example.

`Macroexpand` works by using `define` and `text` to define, temporarily, a new procedure that's just like the macro it wants to expand, but an ordinary procedure instead of a macro:

```
to macroexpand :expression
define "temporary.macroexpand.procedure text first :expression
...
end
```

You might enjoy filling in the rest of this procedure, as an exercise in advanced Logo programming, before you read the version in the library.

(What if you want to do the opposite, defining a macro with the same text as an ordinary procedure? Berkeley Logo includes a `.defmacro` command, which is just like `define` except that the resulting procedure is a macro. We don't need two versions of `text`, because the text of a macro looks just like the text of an ordinary procedure. To tell the difference, there is a primitive predicate `macrop` that takes a word as input, and outputs `true` if that word is the name of a macro.)

## The Real Thing

Here is the complete version of `foreach`, combining the macro structure developed in this chapter with the full template flexibility from Chapter 10.

```
.macro foreach [:foreach.inputs] 2
catch "foreach.catchtag ~
      [output foreach.done runresult [foreach1 butlast :foreach.inputs
                                                last :foreach.inputs 1]]
output []
end

to foreach1 :template.lists :foreach.template :template.number
if emptyp first :template.lists [throw "foreach.catchtag]
apply :foreach.template firsts :template.lists
.maybeoutput foreach1 butfirsts :template.lists ~
                      :foreach.template :template.number+1
end

to foreach.done :foreach.result
if emptyp :foreach.result [output [stop]]
output list "output quoted first :foreach.result
end
```

And here, without any discussion, is the actual library version of `for`. This, too, combines the ideas of this chapter with those of Chapter 10.

```
.macro for :for.values :for.instr
localmake "for.var first :for.values
localmake "for.initial run first butfirst :for.values
localmake "for.final run item 3 :for.values
localmake "for.step forstep
localmake "for.tester (ifelse :for.step < 0
                              [[(thing :for.var) < :for.final]]
                              [[(thing :for.var) > :for.final]])
local :for.var
catch "for.catchtag [output for.done runresult [forloop :for.initial]]
output []
end

to forloop :for.initial
make :for.var :for.initial
if run :for.tester [throw "for.catchtag]
run :for.instr
.maybeoutput forloop ((thing :for.var) + :for.step)
end

to for.done :for.result
if emptyp :for.result [output [stop]]
output list "output quoted first :for.result
end

to forstep
if equalp count :for.values 4 [output run last :for.values]
output ifelse :for.initial > :for.final [-1] [1]
end
```