5 Program as Data

In most programming languages there is a sharp distinction between *program* and *data*. Data are the things you can manipulate in your program, things like numbers and letters. These things live in variables, which can be given new values by your program. But the program itself is not subject to manipulation; it's something you write ahead of time, and then it remains fixed.

In Logo the distinction is not so sharp. We've made extensive use of one mechanism by which a program can manipulate itself: the instruction lists that are used as inputs to `run`, `if`, and so on are data that can be computed by a program. For example, the solitaire program in Chapter 4 constructs a list of Logo instruction lists, each of which would move a card to some other legal position, and then says

```
run first :onto
```

to move the card to the first such position.

`Text` and `Define`

In this chapter we'll use a pair of more advanced tools that allow a program to create more program. `Run` deals with a single *instruction;* now we'll be able to examine and create *procedures.*

`Text` is an operation that takes one input, a word. That word must be the name of a user-defined procedure. The output from `text` is a list. The first member of that list is a list containing the names of the inputs to the chosen procedure. (If the procedure has no inputs, the list will be empty.)* The remaining members of the output list are instruction lists, one for each line in the definition of the procedure.

Here is an example. Suppose we've defined the procedure

```
to opinion :yes :no
print sentence [I like] :yes
print sentence [I hate] :no
end
```

Here's what the text of that procedure looks like:

```
? show text "opinion
[[yes no] [print sentence [I like] :yes] [print sentence [I hate] :no]]
```

In this example the output from `text` is a list with three members. The first member is a list containing the words `yes` and `no`, the names of `opinion`'s inputs. (Note that the colons that are used to indicate inputs in a title line are *not* used here.) The second and third members of the output list are instruction lists, one for each line in the definition. (Note that there is no `end` line in the definition; as I've remarked before, that line isn't an instruction in the procedure because `end` isn't a command.)

The opposite of `text` is the command `define`. This command takes two inputs. The first must be a word and the second a list. The effect of `define` is to define a procedure whose name is the first input and whose text is the second input. You can use `define` to define a new procedure or to change the definition of an old one. For example, I might redefine `opinion`:

```
? define "opinion [[yes no] [print sentence :yes [is yummy.]]
                            [print sentence :no [is yucky.]]]
? opinion [Ice cream] "Cheese
Ice cream is yummy.
Cheese is yucky.
? po "opinion
to opinion :yes :no
print sentence :yes [is yummy.]
print sentence :no [is yucky.]
end
```

* Berkeley Logo allows user-defined procedures with *optional* inputs. For such a procedure, this first sublist may contain lists, representing optional inputs, as well as words, representing required inputs.

Instead of replacing an old definition with an entirely new one, we can use `define` and `text` together to change a procedure's definition:

```
? define "opinion lput [print sentence :no "stinks!] ~
                       butlast text "opinion
? opinion "Logo "Basic
Logo is yummy.
Basic stinks!
```

(Of course, I didn't have to redefine the same procedure name. I could have said

```
? define "strong.opinion ~
         lput [print sentence :no "stinks!] butlast text "opinion
```

and then I would have had two procedures, the unchanged `opinion` and the new version named `strong.opinion`.)

It may be instructive to consider the analogy between *variables,* which hold data, and *procedures,* which hold instructions. Variables are given values with the `make` command and examined with the operation `thing`. Procedures are given definitions with the `define` command and examined with the operation `text`. (There is no abbreviation for `text`-quote, however, like the dots abbreviation for `thing`-quote.)

To illustrate `define` and `text`, I've used them in instructions typed in at top level. In practice, you wouldn't use them that way; it's easier to examine a procedure with `po` and to change its definition with `edit`. `Text` and `define` are meant to be used not at top level but inside a program.

Automated Definition

Early in the first volume I defined the operation `second` this way:

```
to second :thing
output first butfirst :thing
end
```

Suppose I want more operations following this model, to be called `third`, `fourth`, and so on. I could define them all by hand or I could write a program to do it for me:

```
to ordinals
ord1 [second third fourth fifth sixth seventh] [output first butfirst]
end
```

```
to ord1 :names :instr
if emptyp :names [stop]
define first :names list [thing] (lput ":thing :instr)
ord1 (butfirst :names) (lput "butfirst :instr)
end

? ordinals
? po "fifth
to fifth :thing
output first butfirst butfirst butfirst butfirst :thing
end
```

(The name `ordinals` comes from the phrase *ordinal numbers,* which is what things like “third” are called. Regular numbers like “three” are called *cardinal numbers.*) This procedure automatically defined new procedures named `second` through `seventh`, each with one more `butfirst` in its instruction line.

A Single-Keystroke Program Generator

A fairly common thing to do in Logo is to write a little program that lets you type a single character on the keyboard to carry out some instruction. For example, teachers of very young children sometimes use a program that accepts `F` to move the turtle forward some distance, `B` for back, and `L` and `R` for left and right. What I want to write is a *program-writing program* that will accept a name and a list of keystrokes and instructions as inputs and define a procedure with that name that understands those instructions.

```
to onekey :name :list
local "text
make "text [[] [local "char] [print [Type ? for help]]
            [make "char readchar]]
foreach :list [make "text lput (sentence [if equalp :char]
                                         (word "" first ?)
                                         butfirst ?)
                               :text]
make "text lput (lput (list "foreach :list ""print)
                      [if equalp :char "?]) ~
                :text
make "text lput (list :name) :text
define :name :text
end
```

If we use this program with the instruction

```
onekey "instant [[F [forward 20]] [B [back 20]]
                 [L [left 15]] [R [right 15]]]
```

then it creates the following procedure:

```
to instant
local "char
print [type ? for help]
make "char readchar
if equalp :char "F [forward 20]
if equalp :char "B [back 20]
if equalp :char "L [left 15]
if equalp :char "R [right 15]
if equalp :char "? [foreach [[F [forward 20]] [B [back 20]]
                             [L [left 15]] [R [right 15]]]
                            "print]
instant
end
```

In addition to illustrating the use of `define`, this program demonstrates how `sentence`, `list`, and `lput` can all be useful in constructing lists, when you have to combine some constant members with some variable members.

Of course, if we only want to make one `instant` program, it's easier just to type it in. An automatic procedure like `onekey` is useful when you want to create several different procedures like `instant`, each with a different "menu" of characters. For example, consider these instructions:

```
onekey "instant [[F [forward 20]] [B [back 20]]
                 [L [left 15]] [R [right 15]] [P [pens]]]
onekey "pens [[U [penup stop]] [D [pendown stop]] [E [penerase stop]]]
```

With these definitions, typing `P` to `instant` prepares to accept a pen command from the second list. In effect, `instant` recognizes two-letter commands `PU` for `penup` and so on, except that the sequence `P?` will display the help information for just the pen commands. Here's another example:

```
onekey "tinyturns [[F [forward 20]] [B [back 20]]
                   [L [left 5]] [R [right 5]] [H [hugeturns]]]
onekey "hugeturns [[F [forward 20]] [B [back 20]]
                   [L [left 45]] [R [right 45]] [T [tinyturns]]]
```

Procedure Cross-Reference Listings

When you're working on a very large project, it's easy to lose track of which procedure invokes which other one. We can use the computer to help solve this problem by creating a *cross-reference listing* for all the procedures in a project. For every procedure in the project, a cross-reference listing tells which other procedures invoke that one. If you write long procedures, it can also be helpful to list which instruction line in procedure `A` invokes procedure `B`.

The general strategy will be to look through the `text` of every procedure, looking for the name of the procedure we're interested in. Suppose we're finding all the references to procedure `X` and we're looking through procedures `A`, `B`, and `C`. For each line of each procedure, we want to know whether the word `X` appears in that line. (Of course you would not really name a procedure `A` or `X`. You'd use meaningful names. This is just an example.) We can't, however, just test

```
memberp "x :instr
```

(I'm imagining that the variable `instr` contains an instruction line.) The reason is that a procedure invocation can be part of a *sublist* of the instruction list if `X` is invoked by way of something like `if`. For example, the word `X` is not a member of the list

```
[if emptyp :list [x :foo stop]]
```

But it's a member of a member. (Earlier I made a big fuss about the fact that if that instruction were part of procedure `A`, it's actually `if` that invokes `X`, not `A`. That's the true story, for the Logo interpreter. But for purposes of a cross-reference listing, it does us no good to know that `if` invokes `X`; what we want to know is which procedure definition to look at if we want to find the instruction that uses `X`.)

So the first thing we need is a procedure `submemberp` that takes inputs like those of `memberp` but outputs `true` if the first input is a member of the second, or a member of a member, and so on.

```
to submemberp :thing :list
if emptyp :list [output "false]
if equalp :thing first :list [output "true]
if listp first :list ~
   [if submemberp :thing first :list [output "true]]
output submemberp :thing butfirst :list
end
```

Now we want a procedure that will take two words as input, both of which are the names of procedures, and will print a list of all the references to the first procedure in the text of the second.

```
to reference :target :examinee
ref1 :target :examinee butfirst text :examinee 1
end

to ref1 :target :examinee :instrs :linenum
if emptyp :instrs [stop]
if submemberp :target first :instrs ~
   [print sentence "|    | (word :examinee "\( :linenum "\) )]
ref1 :target :examinee butfirst :instrs :linenum+1
end
```

`Reference` uses `butfirst text :examinee` as the third input to `ref1` to avoid the list of inputs to the procedure we're examining. That's because one of those inputs might have the same name as the `target` procedure, and we'd get a false indication of success. (In the body of the definition of `:examinee`, any reference to a variable named `X` will not use the word `X` but rather the word `"X` or the word `:X`. You may find that statement confusing. When you type an *instruction* like

```
print "foo
```

the Logo evaluator interprets `"foo` as a request for the word `foo`, quoted (as opposed to evaluated). So `print` won't print a quotation mark. But if we look at the *list*

```
[print "foo]
```

then we are not, right now, evaluating it as a Logo instruction. The second member of that list is the word `"foo`, quote mark and all.)

We can still get "false hits," finding the word `X` (or whatever procedure name we're looking for) in an instruction list, but not being used as a procedure name:

```
print [w x y z]
```

But cases like that will be relatively rare compared to the cases of variables and procedures with the same name.

The reason I'm printing spaces before the information is that I'm working toward a listing that will look like this:

```
target1
   proca(3)
   procb(1)
   procc(4)
target2
   procb(3)
   procb(4)
```

This means that the procedure named `target1` is invoked in each of the procedures `proca`, `procb`, and `procc`; procedure `target2` is invoked by `procb` on two different instruction lines.

Okay, now we can find references to one specific procedure within the text of another specific procedure. Now we want to look for references to one procedure within *all* the procedures making up a project.

```
to xref :target :list
print :target
foreach :list [reference :target ?]
end
```

We're almost done. Now we want to apply `xref` to every procedure in the project. This involves another run through the list of projects:

```
to xrefall :list
foreach :list [xref ? :list]
end
```

To use this program to make a cross-reference listing of itself, you'd say

```
xrefall [xrefall xref reference ref1 submemberp]
```

To cross-reference all of the procedures in your workspace, you'd say

```
xrefall procedures
```

If you try this program on a project with a large number of procedures, you should expect it to take a *long* time. If there are five procedures, we have to examine each of them for references to each of them, so we invoke `reference` 25 times. If there are 10 procedures, we invoke `reference` 100 times! In general, the number of invocations is the square of the number of procedures. The fancy way to say this is that the program "takes quadratic time" or that it "behaves quadratically."