4 Predicates

By introducing variables in Chapter 3, we made it possible for a procedure to operate on different data each time you invoke it. But the *pattern* of what the procedure does with the data remains constant. We can get even more variety out of our procedures if we can vary the *instructions* that the procedure executes. We need a way to say, "Sometimes do this; other times do that."

True or False

One helpful metaphor is this: When you invoke a command, you're giving the computer an order. "Now hear this! `Print` such-and-such!" But when you invoke an operation, you're asking the computer a *question*. "What is the `first` member of such-and-such?"

In real life we single out as a special category *yes-or-no questions*. For example, these special questions form the basis of the game Twenty Questions. The corresponding category in Logo is the *predicate*. A predicate is an operation whose output is always either the word `true` or the word `false`.

For example, `listp` (pronounced "list-pea") is a predicate that takes one input. The input can be any datum. The output from `listp` is `true` if the input is a list, `false` if the input is a word.

`Wordp` is another predicate that takes one input. The input can be any datum. The output from `wordp` is `true` if the input is a word, `false` if the input is a list. (This is the opposite of the output from `listp`.)

`Emptyp` is also a predicate with one input. The input can be any datum. The output from `emptyp` is `true` if the input is either the empty word or the empty list; if the input is anything else, the output is `false`.

You'll have noticed by now that predicates tend to have names ending in the letter `p`. This is not quite a universal rule, but almost. It's a good idea to follow the same convention in naming your own predicates.*

As I'm describing primitive predicates, you might want to try them out on the computer. You can do experiments like this:

```
? print wordp "hello
true
? print wordp [hello]
false
? print emptyp []
true
? print emptyp 0
false
```

Of course, most of the time you won't actually want to print the output from a predicate. You'll see in a few moments how we can use a predicate to control the instructions carried out in a procedure.

But first here are a few more primitive predicates. `Numberp` takes one input, which can be any datum. The output from `numberp` is `true` if the input is a number, `false` otherwise.

`Equalp` takes two inputs, each of which can be any datum. The output from `equalp` is `true` if the two inputs are identical or if they're both numbers and they're numerically equal. That is, 3 and 3.0 are numerically equal even though they're not identical words. A list is never equal to a word.

```
? print equalp 3 3.0
true
? print equalp "hello [hello]
false
? print equalp "hello first [hello]
true
? print equalp " []
false
? print equalp [] butfirst [hello]
true
```

* Many versions of Logo use a question mark at the end of names of predicates, instead of a `p`. For example, you may see `list?` instead of `listp`. Berkeley Logo accepts either form, but I prefer the `p` version.

The equal sign (=) can be used as an *infix* equivalent of `equalp`:

```
? print "hello = first [hello]
true
? print 2 = 3
false
```

As I mentioned in Chapter 2, if you use infix operations you have to be careful about what is grouped with what. It varies between versions of Logo. Here is an example I tried in Berkeley Logo:

```
? print first [hello] = "hello
f
```

Among current commercial implementations, Object Logo and Microworlds give the same answer `f`. But here is the *same* example in Logowriter:

```
? print first [hello] = "hello
true
```

You can avoid confusion by using parentheses. The following instructions work reliably in any Logo:

```
? print (first [hello]) = "hello
true
? print first ([hello] = "hello)
f
```

`Memberp` is a predicate with two inputs. If the second input is a list, then the first can be any datum. If the second input is a word, then the first must be a one-character word. The output from `memberp` is true if the first input is a member of the second input.

```
? print memberp "rain [the rain in Spain]
true
? print memberp [the rain] [the rain in Spain]
false
? print memberp [the rain] [[the rain] in Spain]
true
? print memberp "e "please
true
? print memberp "e "plain
false
```

`Lessp` and `greaterp` are predicates that take two inputs. Both inputs must be numbers. The output from `lessp` is `true` if the first input is numerically less than the second; the output from `greaterp` is true if the first is greater than the second. Otherwise the output is `false`. (In particular, both `lessp` and `greaterp` output `false` if the two inputs are equal.) The infix forms for `lessp` (`<`) and `greaterp` (`>`) are also allowed.

Defining Your Own Predicates

Here are two examples of how you can create new predicates:

```
to vowelp :letter
output memberp :letter [a e i o u]
end

? print vowelp "e
true
? print vowelp "g
false

to oddp :number
output equalp (remainder :number 2) 1
end

? print oddp 5
true
? print oddp 8
false
```

Conditional Evaluation

The main use of predicates is to compute inputs to the primitive procedures `if` and `ifelse`. We'll get to `ifelse` in a while, but first we'll explore `if`.

`If` is a command with two inputs. The first input must be either the word `true` or the word `false`. The second input must be a list containing Logo instructions. If the first input is `true`, the effect of `if` is to evaluate the instructions in the second input. If the first input is `false`, `if` has no effect.

```
? if equalp 2 1+1 [print "Yup.]
Yup.
? if equalp 3 2 [print "Nope.]
?
```

Here is an example of how `if` can be used in a procedure. This is an extension of the `converse` example in Chapter 3:

```
to talk
local "name
print [Please type your full name.]
make "name readlist
print sentence [Your first name is] first :name
if (count :name) > 2 ~
   [print sentence [Your middle name is] first bf :name]
print sentence [Your last name is] last :name
end

? talk
Please type your full name.
George Washington
Your first name is George
Your last name is Washington
? talk
Please type your full name.
John Paul Jones
Your first name is John
Your middle name is Paul
Your last name is Jones
```

`Talk` asks you to type your name and reads what you type into a list, which is remembered in the variable named `name`. Your first and last names are printed as in the earlier version. If the list `:name` contains more than two members, however, `talk` also prints the second member as your middle name. If `:name` contains only two members, `talk` assumes that you don't have a middle name.

☞ Write a procedure of your own that asks a question and uses `if` to find out something about the response.

You can use `if` to help in writing more interesting predicates.

```
to about.computersp :sentence
if memberp "computer :sentence [output "true]
if memberp "computers :sentence [output "true]
if memberp "programming :sentence [output "true]
output "false
end
```

```
? print about.computersp [This book is about programming]
true
? print about.computersp [I like ice cream]
false
?
```

This procedure illustrates something I didn't explain before about `output`: An `output` command finishes the evaluation of the procedure in which it occurs. For example, in `about.computersp`, if the input sentence contains the word `computer`, the first `if` evaluates the `output` instruction that is its second input. The procedure immediately outputs the word `true`. The remaining instructions are not evaluated at all.

☞ Write `past.tensep`, which takes a word as input and outputs `true` if the word ends in `ed` or if it's one of a list of exceptions, like `saw` and `went`.

☞ Write `integerp`, which takes any Logo datum as input and outputs `true` if and only if the datum is an integer (a number without a fraction part). Hint: a number with a fraction part will contain a decimal point.

Choosing Between Alternatives

`If` gives the choice between carrying out some instructions and doing nothing at all. More generally, we may want to carry out either of *two* sets of instructions, depending on the output from a predicate. The primitive procedure `ifelse` meets this need.* `Ifelse` is an unusual primitive because it can be used either as a command or as an operation. We'll start with examples in which `ifelse` is used as a command.

`Ifelse` requires three inputs. The first input must be either the word `true` or the word `false`. The second and third inputs must be lists containing Logo instructions. If the first input is `true`, the effect of `if` is to evaluate the instructions in the second input. If the first input is `false`, the effect is to evaluate the instructions in the third input.

```
? ifelse 4 = 2+2 [print "Yup.] [print "Nope.]
Yup.
? ifelse 4 = 3+5 [print "Yup.] [print "Nope.]
Nope.
?
```

* In some versions of Logo, the name `if` is used both for the two-input command discussed earlier and for the three-input one presented here.

Here is an example of a procedure using `ifelse`:

```
to groupie
local "name
print [Hi, who are you?]
make "name readlist
ifelse :name = [Ray Davies] ~
     [print [May I have your autograph?]] ~
     [print sentence "Hi, first :name]
end
```

```
? groupie
Hi, who are you?
Frank Sinatra
Hi, Frank
? groupie
Hi, who are you?
Ray Davies
May I have your autograph?
```

☞ Write an operation `color` that takes as input a word representing a card, such as `10h` for the ten of hearts. Its output should be the word `red` if the card is a heart or a diamond, or `black` if it's a spade or a club.

☞ Write a conversational program that asks the user's name and figures out how to address him or her. For example:

```
? converse
Hi, what's your name?
Chris White
Pleased to meet you, Chris.

? converse
Hi, what's your name?
Ms. Grace Slick
Pleased to meet you, Ms. Slick.

? converse
Hi, what's your name?
J. Paul Getty
Pleased to meet you, Paul.

? converse
Hi, what's your name?
Sigmund Freud, M.D.
Pleased to meet you, Dr. Freud.
```

```
? converse
Hi, what's your name?
Mr. Lon Chaney, Jr.
Pleased to meet you, Mr. Chaney.
```

What should the program say if it meets Queen Elizabeth II?

Conditional Evaluation Another Way

The use of `ifelse` in the `groupie` example above makes for a rather long instruction line. If you wanted to do several instructions in each case, rather than just one `print`, the `if` line would become impossible to read. Logo provides another mechanism that is equivalent to the `ifelse` command but may be easier to read.

`Test` is a command that takes one input. The input must be either the word `true` or the word `false`. The effect of `test` is just to remember what its input was in a special place. You can think of this place as a variable without a name. This special variable is automatically local to the procedure from which `test` is invoked.

`Iftrue` (abbreviation `ift`) is a command with one input. The input must be a list of Logo instructions. The effect of `iftrue` is to evaluate the instructions in its input only if the unnamed variable set by the most recent `test` command in the same procedure is `true`. It is an error to use `iftrue` without first using `test`.

`Iffalse` (abbreviation `iff`) is a command with one input, which must be an instruction list. The effect of `iffalse` is to evaluate the instructions only if the remembered result of the most recent `test` command is `false`.

`Iftrue` and `iffalse` can be invoked as many times as you like after a `test`. This allows you to break up a long sequence of conditionally evaluated instructions into several instruction lines:

```
to better.groupie
local "name
print [Hi, who are you?]
make "name readlist
test equalp :name [Ray Davies]
iftrue [print [Wow, can I have your autograph?]]
iftrue [print [And can I borrow a thousand dollars?]]
iffalse [print sentence [Oh, hello,] first :name]
end
```

About Those Brackets

I hope that the problem I'm about to mention won't even have occurred to you because you are so familiar with the idea of evaluation that you understood right away. But you'll probably have to explain it to someone else, so I thought I'd bring it up here:

Some people get confused about why the second input to `if` (and the second and third inputs to `ifelse`) is surrounded by brackets but the first isn't. That is, they wonder, why don't we say

```
if [equalp 2 3] [print "really??]          ; (wrong!)
```

They have this problem because someone lazily told them to put brackets around the conditionally evaluated instructions without ever explaining about brackets and quotation.

I trust *you* aren't confused that way. You understand that, as usual, Logo evaluates the inputs to a procedure before invoking the procedure. The first input to `if` has to be either the word `true` or the word `false`. *Before* invoking `if`, Logo has to evaluate an expression like `equalp 2 3` to compute the input. (In this case, the result output by `equalp` will be `false`.) But if the `print` instruction weren't quoted, Logo would evaluate it, too, *before* invoking `if`. That's not what we want. We want the instruction list *itself* to be the second input, so that `if` can decide whether or not to carry out the instructions in the list. So, as usual, we use brackets to tell Logo to quote the list.

actual argument expression	→	actual argument value
`equalp 2 3`	→	`false`
`[print "really??]`	→	`[print "really??]`

Logical Connectives

Sometimes the condition under which you want to evaluate an instruction is complicated. You want to do it if *both* this *and* that are true, or if *either* this *or* that is true. Logo provides operations for this purpose.

`And` is a predicate with two inputs. Each input must be either the word `true` or the word `false`. The output from `and` is `true` if both inputs are `true`; the output is `false` if either input is `false`. (`And` can take more than two inputs if the entire expression is

enclosed in parentheses. In that case the output from `and` will be `true` only if all of its inputs are `true`.)

`Or` is a predicate with two inputs. Each input must be either the word `true` or the word `false`. The output from `or` is `true` if either input is `true` (or both inputs are). The output is `false` if both inputs are `false`. (Extra-input `or` outputs `true` if any of its inputs are `true`, `false` if all inputs are `false`.)

`Not` is a predicate with one input. The input must be either the word `true` or the word `false`. The output from `not` is the opposite of its input: `true` if the input is `false`, or `false` if the input is `true`.

These three procedures are called *logical connectives* because they connect logical expressions together into bigger ones. (A *logical* expression is one whose value is `true` or `false`.) They can be useful in defining new predicates:

```
to fullp :datum
output not emptyp :datum
end

to realwordp :datum
output and wordp :datum not numberp :datum
end

to digitp :datum
output and numberp :datum equalp count :datum 1
end
```

`Ifelse` as an Operation

So far, we have applied the idea of conditional evaluation only to complete instructions. It is also possible to choose between two expressions to evaluate, by using `ifelse` as an operation.

When used as an operation, `ifelse` requires three inputs. The first input must be either the word `true` or the word `false`. The second and third inputs must be lists containing Logo expressions. The output from `ifelse` is the result of evaluating the second input, if the first input is `true`, or the result of evaluating the third input, if the first input is `false`.

```
? print sentence "It's ifelse 2=3 ["correct] ["incorrect]
It's incorrect
? print ifelse emptyp [] [sum 2 3] [product 6 7]
5
```

Here is one of the classic examples of a procedure in which `ifelse` is used as an operation. This procedure is an operation that takes a number as its input; it outputs the *absolute value* of the number:

```
to abs :number
output ifelse :number<0 [-:number] [:number]
end
```

Expression Lists and Plumbing Diagrams

`If` and `ifelse` require *instruction lists* or *expression lists* as inputs. This requirement is part of their semantics, not part of the syntax of an instruction. Just as the arithmetic operators require numbers as inputs (semantics), but those numeric values can be provided either as explicit numbers in the instruction or as the result of an arbitrarily complicated subexpression (syntax), the procedures that require instruction or expression lists as input don't interpret those inputs until after Logo has set up the plumbing for the instructions that invoke them.

What does that mean? Consider the instruction

```
ifelse "false ["stupid "list] [print 23]
```

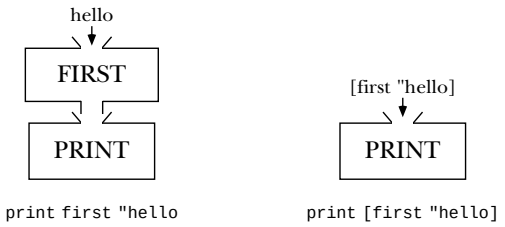
Even though the second input to `ifelse`—that is, the first of the two literal lists—makes no sense as an instruction list, this instruction will work correctly without printing an error message. The Logo interpreter knows that `ifelse` accepts three inputs, and it sees that the three input expressions provided are a literal (quoted) word and two literal lists. It sets up the plumbing without paying any attention to the semantics of `ifelse`; in particular, Logo doesn't care whether the given inputs are meaningful for use with `ifelse`. Then, once `ifelse` starts running, it examines its first input value. Since that input is the word `false`, the `ifelse` procedure ignores its second input completely and executes the instruction in its third input.

The use of quotation marks and square brackets to indicate literal inputs is part of the plumbing syntax, not part of the procedure semantics. Don't say, "`Ifelse` requires one predicate input and two inputs in square brackets." The instruction

```
ifelse last [true false] list ""stupid ""list list bf "sprint 23
```

has a very different plumbing diagram (syntax) from that of the earlier example, but provides exactly the same input values to `ifelse`.

Consider these two instructions:



Since the effect of `print` is easy to observe, it's not hard to see the relationship among the instructions, the plumbing diagrams, and the effects when these instructions are run. Why are brackets used around the `first` expression in one case but not in the other? Because in one case the expression is how we tell Logo to set up the plumbing diagram, while in the second case we are giving `print` as input a literal list that just happens to look like an expression. When the context is something like `ifelse` instead of `print`, the syntactic situation is really quite similar, but may be harder to see. Consider this instruction:

```
print ifelse emptyp :a [emptyp :b] [emptyp :c]
```

Why do we put brackets around two `emptyp` expressions but not around another similar-looking one? ☞ Draw a plumbing diagram for this instruction, paying no attention to your mental model of the meaning of the `ifelse` procedure, treating it as if it were the nonsense procedure `zot3`. You will see that the first input to `ifelse` is an expression whose value will be the word `true` or the word `false`, because Logo will carry out that first `emptyp` computation before invoking `ifelse`. The remaining two inputs, however, are literal lists that happen to contain the word `emptyp` but do not involve an invocation of `emptyp` in the plumbing diagram. Once `ifelse` is actually invoked, precisely one of those two list inputs will be interpreted as a Logo expression, for which a *new* plumbing diagram is (in effect) drawn by Logo. The other input list is ignored.

Stopping a Procedure

I'd like to examine more closely one of the examples from the first chapter:

```
to music.quiz
print [Who is the greatest musician of all time?]
if equalp readlist [John Lennon] [print [That's right!] stop]
print [No, silly, it's John Lennon.]
end
```

You now know about almost all of the primitive procedures used in this example. The only one we haven't discussed is the `stop` command in the second instruction line.

`Stop` is a command that takes no inputs. It is only allowed inside a procedure; you can't type `stop` to a top-level prompt. The effect of `stop` is to finish the evaluation of the procedure in which it is used. Later instructions in the same procedure are skipped.

Notice that `stop` does not stop *all* active procedures. If procedure A invokes procedure B, and there is a `stop` command in procedure B, then procedure A continues after the point where it invoked B.

Recall that the `output` command also stops the procedure that invokes it. The difference is that if you're writing an operation, which should have an output, you use `output`; if you're writing a command, which doesn't have an output, you use `stop`.

In `music.quiz`, the effect of the `stop` is that if you get the right answer, the final `print` instruction isn't evaluated. The same effect could have been written this way:

```
ifelse equalp readlist [John Lennon] ~
    [print [That's right!]] ~
    [print [No, silly, it's John Lennon.]]
```

The alternative form uses the three-input `ifelse` command. One advantage of using `stop` is precisely that it allows the use of shorter lines. But in this example, where there is only one instruction after the `if`, it doesn't matter much. `Stop` is really useful when you want to stop only in an unusual situation and otherwise you have a lot of work still to do:

```
to quadratic :a :b :c
local "discriminant
make "discriminant (:b * :b)-(4 * :a * :c)
if :discriminant < 0 [print [No solution.] stop]
make "discriminant sqrt :discriminant
local "x1
local "x2
make "x1 (-:b + :discriminant)/(2 * :a)
make "x2 (-:b - :discriminant)/(2 * :a)
print (sentence [x =] :x1 [or] :x2)
end
```

This procedure applies the quadratic formula to solve the equation

$$ax^2 + bx + c = 0$$

The only interesting thing about this example for our present purpose is the fact that sometimes there is no solution. In that case the procedure `stop`s as soon as it finds out.

Don't forget that you need `stop` only if you want to stop a procedure before its last instruction line. A common mistake made by beginners who've just learned about `stop` is to use it in every procedure. If you look back at the examples so far you'll see that many procedures get along fine without invoking `stop`.

Improving the Quiz Program

When I first introduced the `music.quiz` example in Chapter 1, we hadn't discussed things like user procedures with inputs. We are now in a position to generalize the quiz program:

```
to qa :question :answer
print :question
if equalp readlist :answer [print [That's right!] stop]
print sentence [Sorry, it's] :answer
end

to quiz
qa [Who is the best musician of all time?] [John Lennon]
qa [Who wrote "Compulsory Miseducation"?] [Paul Goodman]
qa [What color was George Washington's white horse?] [white]
qa [how much is 2+2?] [5]
end
```

Procedure `qa` is our old friend `music.quiz`, with variable inputs instead of a fixed question and answer. `Quiz` uses `qa` several times to ask different questions.

☞ Here are a couple of suggestions for further improvements you should be able to make to `quiz` and `qa`:

1. `Qa` is very fussy about getting one particular answer to a question. If you answer `Lennon` instead of `John Lennon`, it'll tell you you're wrong. There are a couple of ways you might fix this. One is to look for a single-word answer *anywhere within* what the user types. So if `:answer` is the word `Lennon`, the program will accept "`Lennon`," "`John Lennon`," or "`the Lennon Sisters`." The second approach would be for `qa` to take a *list* of possible answers as its second input:

```
qa [Who is the best musician of all time?] ~
   [[John Lennon] [Lennon] [the Beatles]]
```

`Qa` then has to use a different predicate, to see if what the user types is any of the answers in the list.

2. By giving `quiz` a local variable named `score`, you could have `quiz` and `qa` cooperate to keep track of how many questions the user gets right. At the end the score could be printed. (This is an opportunity to think about the stylistic virtues and vices of letting a subprocedure modify a variable that belongs to its superprocedure. If you say

```
make "score :score+1
```

inside `qa`, doesn't that make `quiz` somewhat mysterious to read? For an alternative, read the next section.)

Reporting Success to a Superprocedure

Suppose we want the quiz program to give the user three tries before revealing the right answer. There are several ways this could be programmed. Here is a way that uses the tools you already know about.

The general idea is that the procedure that asks the question is written as an *operation,* not as a command. To be exact, it's a predicate; it outputs `true` if the user gets the right answer. This asking procedure, `ask.once`, is invoked as a subprocedure of `ask.thrice`, which is in charge of allowing three tries. `ask.thrice` invokes `ask.once` up to three times, but stops if `ask.once` reports success.

```
to ask.thrice :question :answer
repeat 3 [if ask.once :question :answer [stop]]
print sentence [The answer is] :answer
end

to ask.once :question :answer
print :question
if equalp readlist :answer [print [Right!] output "true]
print [Sorry, that's wrong.]
output "false
end
```

You've seen `repeat` in the first chapter, but you haven't been formally introduced. `Repeat` is a command with two inputs. The first input must be a non-negative whole number. The second input must be a list of Logo instructions. The effect of `repeat` is

to evaluate its second input, the instruction list, the number of times given as the first input.

The programming style used in this example is a little controversial. In general, it's considered a good idea not to mix effect and output in one procedure. But in this example, `ask.once` has an effect (it prints the question, reads an answer, and comments on its correctness) and also an output (`true` or `false`).

I think the general rule I've just cited is a good rule, but there are exceptions to it. Using an output of `true` or `false` to report the success or failure of some process is one of the situations that I consider acceptable style. The real point of the rule, I think, is to separate *calculating* something from *printing* it. For example, it's a mistake to write procedures like this one:

```
to prsecond :datum
print first butfirst :datum
end
```

A more powerful technique is to write the `second` operation from Chapter 2; instead of

```
prsecond [something or other]
```

you can then say

```
print second [something or other]
```

It may not be obvious from this example why I call `second` more powerful than `prsecond`. But remember that an operation can be combined with other operations, as in the plumbing diagrams we used earlier. For example, the operation `second` can extract the word `or` from the list as shown here. But you can *also* use it as part of a more complex instruction to extract the letter `o`:

```
print first second [something or other]
```

If you'd written the command `prsecond` to solve the first problem, you'd have to start all over again to solve this new one. (Of course, both of these examples must seem pretty silly; why bother extracting a word or a letter from this list? But I'm trying to use examples that are simple enough not to obscure this issue with the kinds of complications we'll see in more interesting programs.)

☞ If you made the improvements to `quiz` and `qa` that I suggested earlier, you might like to see if they can fit easily with a new version of `quiz` using `ask.thrice`.