Leone Learning Systems, Inc.
Wonder. Create. Grow.

Leone Learning Systems, Inc.
237 Custer Ave
Evanston, IL 60202
Emal tj@leonelearningsystems.com

Phone 847 951 0127
Fax 847 733 8812

Predicates and Instruction Lists

TJ Leone

November 2004

Introduction

This unit is based on Chapter 4 of Brian Harvey's book, *Symbolic Computing*, which is available on the web at http://www.cs.berkeley.edu/~bh/v1-toc2.html, and the section called "Predicates and Filters" in the Appendix of Investigation in Algebra by Albert Cuoco.

We have seen that in Logo, numbers are special class of words. Any command or operation that needs words as inputs can take numbers as inputs. For example, the `show` command takes a word or list as input and displays the word or list in the Recall List Box. For example, the following instructions are all legal Logo instructions:

```
show "pie
show [apple pie]
show 72
```

Since numbers are words, `72` is valid input for the `show` command.

We also saw that some commands and operations will work *only* with numbers as inputs. For example,

```
fd 70
```

works OK, but

```
fd "seventy
```

reports the error:

```
fd doesn't like seventy as input
```

Finally, we also know that some operations only output numbers. For example, `sum` outputs the number that is the sum of its two inputs.

There is a pair of words that make up a special class of words, just like numbers make up a class. This pair of words is "true and "false. *Predicates* are operations that output one or the other of these words. In this unit, we will discuss predicates and we will also discuss commands that need "true or "false as input.

Lists can also be used as input to different Logo commands. In this unit, we will also talk about a special class of lists called *instruction lists*.

Copyright © 2004 Leone Learning Systems, Inc.

Predicates are operations

In *Symbolic Computing*, Brian Harvey explains the difference between commands and operations this way:

"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 `sum` of this number and that number?']

Predicates are used to answer yes-or-no type questions, like "Is this number equal to that one?" or "Is this a number?" These probably seem like weird questions to ask. After all, we can look at two numbers and see if they are equal, or look at a word and tell if it is a number. However, just as operations like `sum` and `product` become more useful as procedures get more complicated, we will see that the same is true for predicates.

In the following section, we will look at some simple examples of predicates. Later, we will see how these predicates can combine with variables, commands, and other operations to do more interesting work.

Copyright © 2004 Leone Learning Systems, Inc.

equal?, less?, greater?

There are three important predicates used to compare numbers:

```
equal?
less?
greater?
```

It is common practice to end the names of predicates with a question mark.

Try out these instructions:

```
show equal? 5 sum 2 3
show equal? sum 2 2 product 2 2
show equal? 4 12
show less? 5 difference 10 7
show greater? 12 sum 6 6
```

Since `equal?`, `less?` and `greater?` are all predicates, they all output either "true or "false. The `show` command displays the output in the Recall List Box.

The equal? operation needs two inputs. If the inputs are equal, the equal? operation outputs "true. Otherwise, it outputs "false.

The less? operation needs two inputs. If the first input is less than the second input, it outputs "true. Otherwise, it outputs "false.

The greater? operation needs two inputs. If the first input is greater than the second input, it outputs "true. Otherwise, it outputs "false.

All three of these predicates have infix forms. We can rewrite the instructions above as:

```
show 5 = 2 + 3
show 2 + 2 = 2 * 2
show 4 = 12
show 5 < 10 - 7
show 12 > 6 + 6
```

not, and, or

The predicates

```
not
and
or
```

are special because they use “true or “false as inputs as well as producing “true or “false as outputs. They don’t have question marks at the end of their names because their job is more to convert outputs from other predicates rather than answer questions like “is this number bigger than that one?”

For example, take the predicate `not`, which needs one input. The input can be either “true or “false. If the input is “true, then `not` outputs “false. If the input is “false, then `not` outputs “true.

Try executing these instructions:

```
show not “true
show not 5 = 7
show not greater? 21 3
show not “false
show not less? 10 6
```

The `and` predicate returns “true if all of its inputs are “true. Otherwise, it returns “false. The `or` predicate returns “false if all of its inputs are “false. Otherwise, it returns “true. Try these:

```
show and “true “true
show and “true “false
show or “true “false
show or “false “false
show and 5 = 5 less? 4 12
show not or less? 5 3 equal? 7 8
```

Normally, `and` and `or` expect two inputs. However, you can use parentheses to provide more than two inputs:

```
show (and 4 < 5 “true 9 > 3)
show (or 3 = 3 5 > 12 “false)
```

if, ifelse

In this section, we will talk about two commands—`if` and `ifelse`—that use "true or "false as inputs. Besides "true or "false, these commands need a special kind of input called an *instruction list*. First we will explain what instruction lists are, and then we will show how predicates and instruction lists are used as inputs to `if` and `ifelse`.

In an earlier unit, we saw that a *list* is a group of zero or more words enclosed in brackets. For example, the following are lists:

```
[0 50]
[You earned the highest score yet!]
[]
[fd 50 rt 90]
```

The last list above contains Logo instructions. A list with Logo instructions in it is called an *instruction list*.

The `if` command needs two inputs. As its first input, `if` likes a "true or a "false. As its second input, if likes an instruction list. The `if` command looks at its first input. If the first input is "true, it tells Logo to execute all the instructions in the instruction list. If the first input is "false, it does nothing.

Try these examples:

```
if 5 = sum 2 3 [show [2 plus 3 is 5]]
if "false [show [How now brown cow?]]
if 7 > 3 [fd 80 rt 90 fd 20]
if "true [pd fd 50 rt 120 fd 50 rt 120 fd 50 rt 120 pu]
if 1 > 9 [print [1 is bigger than 9]]
```

Remember, if the first input is "false then nothing should happen.

Copyright © 2004 Leone Learning Systems, Inc.

if, ifelse (continued)

The `ifelse` command is similar to the `if` command. `ifelse` needs three inputs—a "true or "false, and two instruction lists. If the first input is "true, `ifelse` tells Logo to execute all the instructions in the first instruction list. If the first input is "false, `ifelse` tells Logo to execute all the instructions in the second instruction list.

Try these:

```
ifelse 5 = sum 2 3 [show [2 plus 3 is 5]] [show "hello]
ifelse "false [show [How now brown cow?]] [pd fd 100 pu]
ifelse 7 > 3 [fd 80 rt 90 fd 20] [show "huh?]
ifelse "true [pd fd 50 rt 120 pu][pd bk 100 lt 90 pu]
ifelse 1 > 9 [print [1 is bigger]][print [1 is not bigger]]
```

Copyright © 2004 Leone Learning Systems, Inc.

repeat, for

In this section, we will talk about two commands—`repeat` and `for`—that need instruction lists as inputs but not "true or "false.

Like the `if` command, the `repeat` command needs two inputs. Like the `if` command, the second input of the `repeat` command is an instruction list. Unlike the `if` command, first input to repeat is not true or false. Instead, the first input to `repeat` is a number. The number is used to tell repeat how many times the instruction list should be executed.

Try these examples:

```
pd
repeat 5 [fd 80 rt 72]
repeat 3 [show "hello]
```

Remember:

The first input to `if` tells *whether or not* the second input should be executed.
The first input to `repeat` tells *how many times* the second input should be executed.

The `for` command is kind of like `repeat`. Its second input is an instruction list that is executed over and over. However, `for` provides some extra information. Here's a sample `for` command with its output:

```
for [i 4 16 2] [show :i]
4
6
8
10
12
14
16
```

In this example, the `for` command says, "Put 4 into the variable :i and execute the instruction list. Add 2 to :i and execute the instruction list again. Keep going until :i is 16. When :i is 16, execute the instruction list one last time and then stop."

If you only put a variable name and two numbers into the first input list, then `for` assumes that you want to add 1 to the variable each time. Try this:

```
pd for [i 90 360] [fd 30 rt :i show :i]
```

Solved Problems

1. *Problem:* Write the following predicates:

   a. `big?` *number* outputs true if the number is more than 10, otherwise false
   b. `neg?` *number* outputs true if the number is negative, otherwise false
   c. `even?` *number* outputs true if the number is even, otherwise false

   Solution:

   Predicates are operations, so the procedure needs to figure out what the output should be and use the `output` command to report its output.

   a. For this example, we just need to check to see if the input is greater than 10.

```
to big? :n
output :n > 10
end
```

   b. A number is negative if it is less than zero.

```
to neg? :n
output :n < 0
end
```

   c. This procedure requires a special primitive operation called `remainder`. The `remainder` procedure gives you the remainder that's left when you divide one number by another. For example, try these:

```
remainder 16 3
remainder 12 4
```

   How can `remainder` help us here? Check out these examples:

```
remainder 6 2
remainder 29 2
remainder 44 2
remainder 3 2
```

Solved Problems (continued)

Do you see a pattern? When an even number is divided by 2, the remainder is zero. When an odd number is divided by 2, the remainder is 1. Why?

Here's our `even?` procedure:

```
to even? :n
output (remainder :n 2) = 0
end
```

The parentheses here are important. Try the procedure without the parentheses to see what happens. If you don't include them, then Logo thinks that `2 = 0` is the second input to `remainder`. Since `2 = 0` evaluates to "false, Logo will try to execute `remainder` with "false as the second input. Logo will send you a message telling you that `remainder` doesn't like "false as an input and the procedure won't work.

If you don't want to worry about parentheses, you can use the prefix form of the predicate that checks for equality:

```
to even? :n
output equal? remainder :n 2 0
end
```

2. *Problem:* Use `not`, `and`, or `or` as required to create the following predicates:

a.	`less.or.equal?` *number1 number2*	is *number1* ≤ *number2*?
b.	`big.and.even?` *number*	is *number* > 10 and even?
c.	`odd.or.neg?` *number*	is *number* negative or odd?

Solved Problems (continued)

Solution:

a. This could be written with `or`, `equal?` and `less?` as follows:

```
to less.or.equal? :n1 :n2
op or equal? :n1 :n2 less? :n1 :n2
end
```

This predicate has two inputs, `:n1` and `:n2`. The inputs are compared twice—once with the `equal?` predicate and once with the `less?` predicate. The `or` predicate combines the two results. Here's a detailed look at the evaluation of an instruction that uses `less.or.equal?`:

```
show less.or.equal? 4 5
```

1. Logo sees the command `show` which needs one input. So Logo looks for the next word over to get the input for `show`.
2. The next word over is `less.or.equal?`. This is an operation which requires two inputs. Logo collects the two inputs, 4 and 5, and then evaluates the `less.or.equal?` procedure.
3. The top line of the procedure tells us that less.or.equal? has two inputs, :n1 and :n2. Logo assigns the number 4 to :n1 and the number 5 to :n2. From now on, whenever we see :n1 in the procedure, Logo replaces it with 4. Whenever we see :n2 in the procedure, Logo replaces it with 5.
4. The next line in the procedure starts with the `op` command. This command tells Logo that the procedure is an operation and will output whatever comes next.
5. After the word `op` is the word `or`. Logo knows that or is an operation that needs two inputs, each with the value "true or "false.
6. The first word after `or` is `equal?`. Since or needs a "true or "false as its first input, we know that `equal?` must output a "true or a "false. Otherwise, Logo will send us an error message. Logo knows that `equal?` needs two inputs. Since :n1 is replaced by 4 and :n2 is replaced by 5, the two inputs that Logo gathers are 4 and 5. Since 4 and 5 are not equal, the predicate `equal?` outputs "false, which is the first input to `or`.
7. The next word Logo sees is `less?`. Like equal?, the word `less?` is followed by :n1 and :n2. So, like `equal?`, the inputs to less? are 4 and 5. Since 4 is less than 5, `less?` output "true, which becomes the second input to `or`.
8. Given the inputs "false and "true, `or` outputs "true. This is the input that `op` needed, so now the procedure can output "true, which is shown by `show`.

Copyright © 2004 Leone Learning Systems, Inc.

Solved Problems (continued)

An equivalent function could be written with `not` and `greater?` like this:

```
to less.or.equal? :n1 :n2
op not greater? :n1 :n2
end
```

Again, if we invoke this procedure with inputs 4 and 5, the predicate `greater?` outputs "false, and the predicate `not` takes the input "false and outputs "true.

What would happen with both procedures if the inputs were 5 and 4 instead of 4 and 5? What if the inputs were 4 and 4?

Both ways of writing the procedure are correct. In general, when you have a choice between equivalent functions, you should pick the one that's clearest to you. That way, when you go back to look at your code at a later date, you'll have a better chance of understanding what you wrote.

b. Since we have already written the predicates `big?` and `even?`, we can just `and` them together:

```
to big.and.even? :n
op and big? :n even? :n
end
```

When does this predicate output "true? Only when the input is both big (greater than 10) *and* even. Here are examples:

```
show big.and.even? 3
false
show big.and.even? 4
false
show big.and.even? 15
false
show big.and.even? 12
true
```

Solved Problems (continued)

c. This predicate outputs "true when the input is odd or when the input is negative. Otherwise, it outputs "false.

Remember how we used `remainder` in writing the `even?` procedure to tell us if something was divisible by 2? The remainder of an even number divided by 2 is zero. In this example, we can use the fact that the remainder of an odd number divided by 2 is 1:

```
to odd.or.neg? :n
op or equal? 1 remainder :n 2 less? :n 0
end
```

We could write an equivalent function by combining predicates we wrote earlier:

```
to odd.or.neg? :n
op or not even? :n neg? :n
end
```

Do you see why these functions are equivalent? If not, think how Logo would evaluate these functions for particular inputs.

3. *Problem:* Write a Logo function for the following algebraic equation:

$f(n) = 3n - 2n + 7$

Write another procedure called `verifyf` that tests to see if $f(n) = 10$ for different values of n. The procedure should be a command that shows a different message depending on whether or not $f(n) = 10$.

Solved Problems (continued)

Solution:

The first function `f` is a kind of function we've done before:

```
to f :n
output 3 * :n – 2 * :n + 7
end
```

You might have noticed that an equivalent function is:

```
to f :n
ouput :n + 7
end
```

We can use the `ifelse` command in `verifyf` to take different actions depending on whether or not the output of `f` is 10:

```
to verifyf :n
ifelse equal? 10 f :n [
     show se :n [is a root of 3n – 2n + 7 = 10]
] [
     show se :n [is not a root of 3n – 2n + 7 = 10]
]
end
```

The operation `se` is a special operation that takes words and lists as input and puts them together into one list which is the output of the operation. Here are examples:

```
show se "hello "there
[hello there]
show se [my heading is] heading
[my heading is 0]
show se [my pen color is] pencolor
[my pen color is 0 0 0]
```

We say a number is a *root* of an equation if it makes the equation true.

Solved Problems (continued)

4. *Problem:* Use `repeat` to rewrite the following procedure:

```
to square :n
fd :n
rt 90
fd :n
rt 90
fd :n
rt 90
fd :n
rt 90
end
```

Solution:

The part of the code that's being repeated is `fd :n rt 90`, so that's what goes into the instruction list. The number of times we see `fd :n rt 90` is 4, so that's the first input to `repeat`:

```
to square :n
repeat 4 [fd :n rt 90]
end
```

Solved Problems (continued)

5. *Problem:* Use `for` to display numbers from 0 to 100 by fives.

   Solution:

```
for [n 0 100 5] [show :n]
```

   Here's an explanation of the items in the list `[n 0 100 5]`:

`n`	Name of variable used in instruction list
`0`	First value given to n
`100`	Last value given to n
`5`	Amount added to n each time instruction list is executed

   So Logo executes

```
show 0
show 5
show 10
…
show 100
```

   The "…" means "and so on" until we get up to `show 100`.

The Author

TJ Leone owns and operates Leone Learning Systems, Inc., a private corporation that offers tutoring and educational software. He has a BA in Math and an MS in Computer Science, both from the City College of New York. He spent two years in graduate studies in education and computer science at Northwestern University, and six years developing educational software there. He is a former Montessori teacher and currently teaches gifted children on a part time basis at the Center for Talent Development at Northwestern University in addition to his tutoring and software development work. His web site is http://www.leonelearningsystems.com.