The Logo Programming Language

The Logo programming language was developed in the late 1960's as a tool for teaching programming. Its simple syntax, conversational style, high-level data types, and informative error messages helped Logo to gain popularity as a professional language as well. Several commercial and public-domain implementations of Logo exist today, including UCB Logo developed here at Berkeley by Brian Harvey and his students. Logo serves as the basis for the Scratch and BYOB graphical programming languages used in CS 10.

Logo Features

We'll first present a sample Logo program, and then discuss each component separately.

```
print "Hello!
print [Starting a program]
to fib :n
   if :n = 0 [output 0]
   if :n = 1 [output 1]
   output (fib :n - 1) + (fib :n - 2)
end
print sentence [fib(4) is:] (fib 4)
```

Running this through logo outputs:

```
Hello!
Starting a program
fib(4) is: 3
```

Print

`print` is a built-in procedure that, given a single argument, outputs the argument to the screen. Notice that giving a list (sentence) to `print` will result in the brackets being removed:

```
? print [hello world]
hello world
```

To get the brackets to show up, we can use the built-in `show` procedure:

```
? show [hello world]
[hello world]
```

Parentheses

Logo doesn't require any punctuation for call expressions, but parentheses can be added for clarity. Unlike Python, which puts parentheses around only the operand sub-expressions of a call expression, Logo allows you to put parentheses around the *whole* call expression, including the operator. These parentheses are required in other dialects of Lisp.

```
? (print (sum 1 2))
3
```

Parentheses also affect how infix operators are applied, just like in algebra. Adding parentheses liberally when using infix operators is always a good idea in Logo.

```
? (print ((5 - 1) + 4))
8

? (print (5 - (1 + 4)))
0
```

Defining Procedures

To define a procedure, we use the `to` special form, which is of the form:

```
to <procedure name> <arg_1> … <arg_n>
   <body>
end
```

Notice the use of the ending '`end`' keyword - don't forget this!

In Logo, you return a value by using the `output` keyword. Here's a side-by-side comparison of `fib` in Logo versus `fib` in Python:

Logo	`to fib :n`
	`  if :n = 0 [output 0]`
	`  if :n = 1 [output 1]`
	`  output (fib :n - 1) + (fib :n - 2)`
	`end`

Python	`def fib(n):`
	`  if n == 0:`
	`  return 0`
	`  if n == 1:`
	`  return 1`
	`  return fib(n - 1) + fib(n - 2)`

Conditional Expressions: `if`, and `ifelse`

One pretty neat feature of Logo is how it handles `if`/`ifelse` - unlike most programming languages out there, Logo does not make `if`/`ifelse` a special form - instead, it's just a regular procedure!

```
? (print (if 3 = 3 [4 + 5]))
9
```

The `if` procedure takes two arguments: the first argument is either the word True or the word False, and the second argument is a list that contains a Logo expression to be evaluated if the first argument is True.

Logo handles delaying the conditional expression of the `if`/`ifelse` constructs by treating the conditional expressions as simply lists of words. If the first argument is True, Logo will then treat that list as Logo code. How does Logo 'execute' a list of words? With the `run` built-in procedure!

```
? (print (run (list "first [this is a list])))
this
```

Given an argument list, `run` will execute the list as Logo code, i.e. as if you had typed the list into the interpreter (without the brackets).

`ifelse` is essentially the same as `if`, just with a third argument list representing the line of Logo to evaluate if the first argument is False:

```
? print ifelse (1 = 3) [2 + 5] [sentence "was "false]
was false
```

Higher-Order Functions in Logo

In Logo, procedures are not first-class - in other words, you can't pass functions around as arguments like in Python. However, we can still get the same effect by passing the name of the function, and using `run`. Here's a function that applies a given function to an argument:

```
to apply_fn :fn :arg
   output run list :fn :arg
end
? print apply_fn "first [1 2]
1
```

Here, I'm constructing a list out of `:fn` and `:arg` (creating `[first [1 2]]`), then calling `run` on that list.

Due to this technique of using lists and `run` to simulate passing functions, we say that expressions are first-class objects, since they are just lists (i.e., sentences).

Logo is Case-Insensitive

One final note - Logo is a case-insensitive language. For instance, the following lines are equivalent:

```
? print ifelse 1 = 1 ["moo] ["baa]
? PRINT IFELSE 1 = 1 ["moo] ["baa]
? pRiNt ifElse 1 = 1 ["moo] ["baa]
```

Exercises

1. To prove that `if`/`ifelse` don't have to be special forms in Logo, define a Logo procedure `ifelse_2` that behaves just like the built-in `ifelse`:

```
? ifelse_2 "hi = "hi [print sentence "was "true] [print sentence "was "false]
was true
```

2. Define the procedure `add_s` that, given a sentence of words, adds the letter `s` to the end of each word:

```
? print add_s [lion tiger bear oh my]
lions tigers bears ohs mys
```

3. Define the `map_fn` procedure in Logo – `map_fn` should take a function name and a sentence, and apply the function to each argument:

```
? map_fn "first [i wish today was just like every other day]
[i w t w j l e o d]
```

4. Redefine `add_s` (from question 2) to instead call `map_fn`.

5. Say I mistyped, and I forgot to quote the name of the procedure when making a call to `map`:

```
? map first [this is the story of a girl]
not enough inputs to map
```

What happened here?

Procedure Calling in Logo

Perhaps one of the more difficult things to get used to in Logo is the 'mysterious' lack of parenthesis when calling procedures:

```
? print first ifelse run [1 = 1] [word word 1 2 3] [sentence "is "awesome]
```

How can this expression possibly work? Ahh! (it returns `1`, by the way)

The key to understanding this line is that you have to understand how Logo interprets lines. When Logo sees the name of a function, it performs an internal lookup to see how many arguments the function takes (for now, let's omit functions that take any number of arguments). It then takes the number k that it gets back, decides that the next k values shall be arguments to this function, and then proceeds to evaluate the rest of the line.

For instance, to evaluate the line:

```
? word word 1 2 3
```

We're calling the `word` procedure, and Logo figures out that `word` takes 2 arguments. It then goes to evaluate the rest of the line (hopefully resulting in 2 values being produced!).

`word word 1 2 3`

Logo sees that we're calling the `word` procedure, which takes 2 arguments:

`word word 1 2 3`

Here, `1` is self-evaluating.

`word word` **`1`** `2 3`

Here, `2` is self-evaluating.

`word word` **`1 2`** `3`

At this point, Logo notices that the two values we just evaluated are the arguments for the second `word` function, so we get:

`word` **`word 1 2`** `3` `=>` `word` **`12`** `3`

Then, `3` is self-evaluating.

`word 12` **`3`**

At this point, Logo notices that the two values we just evaluated are the arguments for the first `word` function, so we finally get:

word 12 3 => 123

If it helps you, you can parenthesize expressions to see how things fit together (using prefix notation from Scheme):

```
(word (word 1 2) 3)
```

Exercises

1. What Would Logo Print? (WWLP)

Indicate what Logo would print - if an error occurs, please briefly describe the error.

Note: `se`, `bf`, and `bl` are shorthands for `sentence`, `butfirst`, and `butlast` respectively.

a.

```
? show run [run [list "run word "L "OL]]
```

b.

```
? show run [run [list run word "L "OL]]
```

c.)

```
? show run [run [list run [word "L "OL] "HAI]]
```

d.)

```
? print se se first bf [are fezzes cool] word word "a "r "e map "first [come on over larry]
```

2. Louis Reasoner is practicing with Logo, and decides to define the `fib` procedure in order to get used to Logo syntax:

```
to fibo :n
   if :n = 0 [output 0]
   if :n = 1 [output 1]
   output fibo :n - 1 + fibo :n - 2
end

? fibo 4
```

However, this code infinite loops! Louis is puzzled. What is happening here? What is the simplest fix you can make?

Dynamic Scoping in Logo

Recall that Python uses Lexical Scoping, which controls what frame a newly created frame points to. Lexical Scoping says: when calling a function, create a new frame that extends the environment that the function was defined in. Logo, however, uses a different rule - it uses Dynamic Scoping. Dynamic Scoping says: when calling a function, create a new frame that extends the **current** frame.

```
to repeat_name :name :num
   output repeat_name_helper :num
end

to repeat_name_helper :num
   if :num = 0 [output []]
   output sentence :name (repeat_name_helper :num - 1)
end

? print repeat_name "brian 4
brian brian brian brian
```

This result might be a little surprising to you - at first glance, it seems as if '`:name`' shouldn't be bound in `repeat_name_helper`'s environment. After all, this would be true in the equivalent Python program:

```
def repeat_name(name, num):
   return repeat_name_helper(num)
def repeat_name_helper(num):
   if num == 0:
       return []
   else:
       return [name] + repeat_name_helper(num - 1)
>>> repeat_name("brian", 4)
```

The above Python program throws an unbound variable error for the variable `name` in the body of `repeat_name_helper`. The key difference, then, is in the fact that Logo uses Dynamic Scoping (whereas Python uses Lexical Scoping).

In the Logo program, when `repeat_name` calls `repeat_name_helper`, `repeat_name_helper` has access to all variables visible to the caller - in this case, `repeat_name`. Interesting! This implies that `repeat_name_helper` will not throw an unbound variable error if the function calling `repeat_name_helper` has a `:name` variable defined. This new scoping mechanism fundamentally changes the way we look at code. In Lexical Scoping, when you look at a procedure definition, you always know which variable points to which value. The order/manner in which you call functions doesn't change what a variable points to. However, in Dynamic Scoping the order/manner in which function calls happen does matter, since variable names will get resolved differently depending on who calls what.

Exercises

1. Come up with a short Logo program that prints Alice if Lexical Scoping is used, and Bob if Dynamic Scoping is used.

2. Consider the following code:

Note: the '`make`' procedure creates variables

```
make "x 2
to foo :x
   output bar [5 + garply]
end
to bar :exp
   output :x * (run :exp)
end
to garply
   output :x * 2
end

? print foo 4
```

a. If Logo were to use Lexical Scope (i.e. the scoping rule that Python uses), what would be printed out?

b. As you now know, Logo uses Dynamic Scope. What does Logo actually print out?