CS61A Project 4, Part II: The Interpreter

Chapter 4 of *Structure and Interpretation of Computer Programs* presents a Scheme interpreter written in Scheme, the metacircular evaluator. In this project, you will write an interpreter for a different programming language, Logo, in Scheme. You will need to write a good deal of code for this project, because very little code is given to you. However, you do have a working Scheme interpreter to use as a reference and to borrow code from as you see fit.

To see a working Logo interpreter, run Berkeley Logo by typing `logo` at the shell:

```
% logo
? Welcome to Berkeley Logo version 2.0
```

Our interpreter will differ from Berkeley Logo in several ways:

- Berkeley Logo allows you to use `+`, `-`, `*`, `/`, `<`, `>` and `=` as infix operators. In our Logo interpreter all functions will be invoked in prefix form.
- In Berkeley Logo, you can use parentheses to allow functions like `sum` and `sentence` to take an arbitrary number of arguments. In our Logo interpreter, parentheses shall be completely meaningless, and all functions take a fixed number of arguments.
- Berkeley Logo prints error messages in a casual way, without even telling you it's a bona-fide error (probably to avoid scaring little kids to death):

  ```
  ? 3
  You don't say what to do with 3
  ```

  Ours will officially announce errors:

  ```
  ? 3
  *** Error:  You don't say what to do with 3
  ```

  The actual content of the error messages you create does not have to match those of Berkeley Logo, or those you shown here.
- Berkeley Logo has many more features than will our interpreter. For example, it allows for a funny form of higher-order functions with `map` and things called *templates* that have question marks in them. For example:

  ```
  ? print map [product ? ?]  [1 2 3]
  1 4 9
  ```

  Our interpreter will have these features.
- Berkeley Logo allows a single Logo expression to stretch for several lines, by putting a ~ at the end of each continuing line. For example (from Lab 7.1):

  ```
  ? print ifelse 2=1+1 ~
          [second [your mother should know]] ~
          [first "help]
  ```

  In our interpreter, a function call **must fit on a single line**. That is, all arguments to a function must be on the same line as the function itself. So the above would be:

  ```
  ? print ifelse 2=1+1 [second [your mother should know]] [first "help]
  ```

  It is not too difficult to fix this, but we're not asking you to.

Although you should model your interpreter on Berkeley Logo, the specifications here trump the behavior of Berkeley Logo.

It is important that you understand that this is *your* Logo interpreter. There is no right or wrong way to build it; if it works it works. There is no "wrong answer." You do not need to use *any* of the code already provided; you do not need to name functions you write in any standard way; go ahead and do your own thing, if you want to. Just make sure that it behaves as specified here.

This monster of a project description is organized as follows:

Section 1: An introduction to Logo, hinting slightly at implementation aspects.

Section 2: General notes about your implementation, as well as explanation of the code already provided.

Section 3: Detailed instructions, hints, and requirements, following roughly the project outline on Page 13.

1 Scheme vs. Logo

Although Logo is a dialect of Lisp, just like Scheme, it is a more distant dialect. There are several profound differences between Logo and Scheme.

1.1 Syntax

Logo code was never meant to look like a list; you won't see zillions of nested parentheses in Logo source. In Scheme, the parentheses served to group the operator with its operands. You could easily tell how many arguments a procedure is given by determining the length of the list that represents its invocation. For example, consider the following Scheme expression:

```
(foo 2 (bar 3 4) 7)
```

It is immediately clear that the procedure `bar` is given two arguments, `3` and `4`, and `foo` is given three arguments, `2`, `(bar 3 4)` and `7`.

Contrast this with the following Logo expression:

```
foo 2 bar 3 4 7
```

Like Scheme, Logo uses prefix notation for procedure calls: *operator*, arg_1, arg_2, arg_3, etc. However, the expression above might correspond to any of the following Scheme expressions:

```
(foo 2 (bar 3 4 7))

(foo 2 (bar 3 4) 7)

(foo 2 (bar 3) 4 7)

(foo 2 (bar) 3 4 7).
```

Even worse, `foo 2` and `bar 3 4 7` might be two distinct Logo expressions, corresponding to the sequence of Scheme expressions `(begin (foo 2) (bar 3 4 7))`. This ambiguity is resolved by having each Logo procedure know the number of arguments it takes. For example, if we know that `foo` takes two arguments

and `bar` takes three, then the equivalent Scheme expression can only be `(foo 2 (bar 3 4 7))`. On the other hand, if we knew that both `foo` and `bar` take two arguments, then the Logo interpreter should signal an error since it won't know what to do with the `7` on the end.

As another example, consider:

```
print word sum first 27 27 word "two word "- "nine
```

The interpreter figures out which arguments belong to what procedure by looking at the line from left to right.

`print` takes one argument: `print word sum first 27 27 word "two word "- "nine`

`word` takes two arguments: `word sum first 27 27 word "two word "- "nine`

`sum` takes two arguments: `sum first 27 27`

and and so does `word`: `word "two word "- "nine`

So equivalent Scheme expression would be:

```
(print (word (sum (first 27) 27) (word 'two (word '- 'nine))))
```

Very early in the project you will need to implement this argument-gathering mechanism. That is, you will know how many arguments a given procedure takes, and it will be your job to collect them. Our Logo interpreter will not handle procedures that take a varying number of arguments.

Another thing to remember is that there can be any number of unrelated Logo expressions on one line:

```
? print [hello] print [there] make "a "aye
hello
there
? print :a print :a print :a print :a     ;; :a means lookup value of a
aye
aye
aye
aye
```

To get the same effect in Scheme, you'd need to enclose all the Scheme expressions with `begin`, as in:

```
(begin (print '(hello)) (print '(there)) (define a 'aye))
```

In Logo you just put them on the same line. We have provided the function `logo-eval-line` to handle this. It's not very complicated. It just keeps calling `logo-eval` to "eat up" Logo expressions until the line becomes empty or an error is encountered.

1.1.1 Procedures vs. variables

Going back to the example of `foo 2 bar 3 4 7`, you may have noticed that we immediately assumed `foo` and `bar` were procedure calls. What about variables? In Logo, any symbol that does not begin with a colon is taken to be a procedure call:

```
? print [the matrix has you]
the matrix has you
? prant [knock knock]
*** Error:  I don't know how to prant
```

And if the symbol, like `prant` above, does not correspond to the name of a procedure the "I don't know how to" error is reported. Logo variables are distinguished from procedure calls by prefixing their name with a colon. Hence, `:foo` is a variable, and `foo` is a procedure call:

```
? make "foo [oh my god]      ;; create a variable "foo"
? to foo                     ;; create procedure of the same name
-> output [holy cow]
-> end
foo defined
? print :foo                 ;; lookup the value of the variable "foo" and print it
oh my god
? print foo                  ;; call procedure "foo" and print its return value
holy cow
```

This leads to the conclusion that in Logo a procedure named x can coexist with a variable named x. In Scheme this is impossible since procedures *are* variables. Think about how many times you named a formal parameter `lst` or `wd` to avoid loosing the `list` or `word` functions, respectively! More on this later.

If a variable is unbound, Berkeley Logo prints the "has no meaning" error message. You can have a lot of fun with this:

```
? :time
time has no meaning
? :love                    ;; variable!
love has no meaning
? love                     ;; procedure call!
I don't know how to love
```

Feel free to keep or modify these error messages for your interpreter.

1.1.2 Infix operators

The last piece of syntax that will be unfamiliar to the Scheme programmer is infix operators. In real Logo, the seven functions `+`, `-`, `*`, `/`, `<`, `>` and `=` are placed between their two arguments:

```
? print 2 > 17
false
? print 2 + 5 * 6 - 4        ;; note precedence
28
```

The Logo interpreter we are building will not handle infix operators. All function calls must be in prefix form. This means more typing for the user. To compute the same things as above you'd say:

```
? print greaterp 2 17
false
? print sum 2 difference product 5 6 4
28
```

But having just prefix operators simplifies the interpreter a good deal.

1.2 First-Class Procedures

In Logo procedures are not *first-class data*. You cannot pass Logo procedures as arguments to other Logo procedures, return procedures or store procedures in aggregates. In Scheme, procedures are just variables

that happen to be bound to the results of `lambda` expressions. In Logo, procedures are beasts of a different nature. In fact, Logo procedures and variables are stored in separate name spaces. Like Scheme variables, Logo variables are stored in environments. Logo procedures, however, are stored in a single global data structure (one that you'll have to implement).

Since there is no `lambda` equivalent in Logo, there are no anonymous procedures. All Logo procedures have names. The primitive procedures have their names built in. User-defined procedures are created and named using the `to` special form, the *only* special form in Logo (see below):

```
? to factorial :n
-> if equalp :n 0 [output 1]
-> output product :n factorial difference :n 1
-> end
factorial defined
? factorial 5
120
```

When a procedure is created with `to`, its name and the number of arguments it takes (just one in the case of `factorial`) are set, and the procedure is entered into some global data structure. Implementing `to` will be an important part of this project, since it will allow you to define your own procedures.

1.3 Dynamic Scope

A more profound difference between Scheme and Logo is that Logo uses *dynamic scope* whereas Scheme uses lexical—also known as *static*—scope. Under lexical scope, compound procedure calls are handled by extending "the environment to which the procedure points" on an environment diagram. This is implemented in the metacircular evaluator by having every compound procedure carry with it the environment in which it was created. It is this environment that is extended when the procedure is called. That's why there is a `procedure-environment` selector in the MCE. Hence, regardless of what the current environment is, calling some procedure *foo* always extends the environment to which *foo* points. This means that any given call to *foo* has access to the same, unchanging set of local variables. That's why they call it *static* scope!

Under dynamic scope, compound procedure calls are handled by extending the *current* environment, whatever it may be. Another way to think of it is that each procedure call extends the environment from which it was invoked, the "calling" environment. Consider the following Logo procedure:

```
to foo :a :b :c
print bar
end
```

Note the call to `bar` in the body of `foo`; `bar` takes zero arguments. When the Logo interpreter gets to this invocation of `bar` it will extend the environment of `foo`. That is, it will extend the environment where the arguments to `foo` are bound. The body of `bar` will be evaluated in an environment containing bindings of `a`, `b` and `c`. The following are all valid definitions of `bar`:

```
to bar
output :a
end

to bar
output sum :a quotient :b :c
end

to bar
output ifelse equalp :a :b [[a and b are the same]] [[a and b are not the same]]
```

```
end
```

When called from within the body of `foo`, `bar` will have access to three local variables: `a`, `b` and `c`. Of course, you can also invoke `bar` from other environments, such as the global environment. In this case, the new frame will extend the global environment and `bar` will see a different set of local variables (namely, no local variables at all). Thus, the scope of `bar` changes with the current environment—it is *dynamic*.

Here is another example of dynamic scope:

```
? make "prefix "sub
? to attach.prefix :word
-> output word :prefix :word
-> end
attach.prefix defined
? print attach.prefix "marine
submarine
? to prefixify :sent :prefix
-> if emptyp :sent [output []]
-> output sentence attach.prefix first :sent prefixify butfirst :sent :prefix
-> end
prefixify defined
? print prefixify [do make cork] "un
undo unmake uncork
```

Note that the `attach.prefix` function gets the value of the variable `prefix` from the current environment, not the environment in which it was created.

Here is Scheme version:

```
STk> (define prefix 'sub)
STk> (define (attach-prefix wd)
        (word wd prefix))
STk> (attach-prefix 'marine)
submarine
STk> (define (prefixify sent prefix)
        (if (empty?  sent)
            '()
            (sentence (attach-prefix (first sent) (prefixify (butfirst sent) prefix)))))
STk> (prefixify '(do make cork) 'un)
(subdo submake subcork)
```

Make sure you understand why the results are different, since you won't be able to implement dynamic scope if you don't get it.

Part of the project will be to write `logo-apply`, the function that applies both primitive and compound Logo procedures. Just like the `apply` procedure in the metacircular evaluator, `logo-apply` should do different things for primitive and compound procedures. Primitive procedures should be applied in underlying Scheme, just like in the MCE. The application of compound procedures must follow the rules of dynamic scope.

1.4 Operators and Commands

In Scheme, every function returns a value, even if it is not a very useful one. For example, the return value of `define` on STk is the name of the variable that was created:

```
STk> (first (define (square x) (* x x)))
s
```

We use `define` only for its side-effect, the creation of a variable. Its return value is garbage. Same with `display`. We use it to print stuff, not for the `okay` it returns:

```
STk> (display "There is no spoon.\n")
There is no spoon.                          ;; side-effect
okay                                        ;; return value
```

According to the Scheme standard, the formal document that defines the Scheme programming language, the return values of procedures like `define`, `display`, `newline` and `set!`, which are used only for their side effects, are "unspecified." This means that different Scheme interpreters may return different values from these procedures. The programmer is never intended to use them!

Although the presence of the garbage return values may be a slight annoyance to the Scheme programmer, it is a boon for the implementer! Not having to distinguish between procedures that return something and procedures that do not makes a Scheme interpreter, like the MCE, easier to write.

Logo does make this distinction. In Logo, procedures that return useful values are called *operators*. Examples would be `sum`, `word` and `numberp`. The other class of procedures, the ones used solely for their side-effects, are called *commands*. Logo commands include `print`, `make` and `to`. And commands really do return *nothing*:

```
? make "pi 3.14
?                                           ;; no return value printed!
```

Well, to be more precise, Logo gives the user the impression that commands returned nothing: nothing is printed by the main loop, and using `print`, `to` or `make` as subexpressions causes an error. For example:

```
? first print "hello
hello
*** Error: print did not output to first
```

Notice that the argument to `first` was evaluated. This is why "hello" was printed (`print` also prints a newline). But afterwards the interpreter realized that we were trying to make use of the return value of `print` and complained.

Since the Logo interpreter will be written in Scheme, you will need to return some object that shall represent the lack of a return value in the Logo world. In fact, the file "logo.scm" already contains a definition of this object:

```
(define void (cons 'no 'value))
```

The reason `void` is a pair is to allow us to test for it using `eq?`; nothing except `void` can possibly be `eq?` to it. The predicate that checks for `void` also is given:

```
(define (void? x) (eq? x void))
```

The `driver-loop` of the interpreter, which we have provided, is already set up to not print `void`:

```
(define (driver-loop)
  (display "?  ")
  (flush)                       ;; makes sure ? is printed (unbuffers output)
  (let ((line (logo-read)))
    (cond ((null?  line) (driver-loop))
          ((equal?  line '(bye))
           (display "Thank you for using Logo.\n"))
```

```
(else
 (call/cc (lambda (cont)                    ;; this is explained later
            (set!  back-to-driver cont)
            (let ((result (logo-eval-line (make-line-obj line) logo-global-environment)))
              (if (not (void?  result))
                  (logo-error "You did not say what to do with " result)))))
 (newline)
 (driver-loop)))))
```

Actually, the `driver-loop` complains if it gets anything other than `void` as a return value. This is because, as you may have already realized by playing with Logo, every line of Logo code that is evaluated at the prompt *must* result in `void` or you get the "you don't say what to do with" error. Whereas Scheme will always print the return value of any expression typed at the prompt...

```
STk> (+ 5 5)
10
```

... Logo will never print it. Instead, it will complain that you did not explicitly say to print it (or use it in `make`, or some other command):

```
? sum 5 5
*** Error: You did not say what to do with 10
```

Whereas Scheme has a read-eval-print loop, Logo has only a read-eval loop, without the print. If you want something printed, you have to say so explicitly:

```
? print sum 5 5
10
```

Similarly, if you want something returned, you have to say so. Scheme always returns the value of the last expression in a procedure body. You don't need to tell Scheme to do this; it happens automatically.

```
STk> (define (square x)
       (* x x))
STk> (square 5)
25
```

A similar definition of `square` will not work in Logo:

```
? to square :x
-> product :x :x
-> end
square defined
? square 5
*** Error: You did not say what to do with 25 in square
```

There are two things you can do with 25. You can `print` it, making `square` a command:

```
? to square :x
-> print product :x :x
-> end
square defined
? square 5
25
```

Or, you can return it with `output`, making `square` an operator (in which case you need to `print` the return value of the `square` explicitly):

```
? to square :x
-> output product :x :x
-> end
square defined
? print square 5
25
```

1.4.1 Output

Just like you have to "tell" Logo to print a value, you have to "tell" logo to return it as well. That's where `output` comes in. Here is how you would define a recursive function in Logo that makes use of the return value of the recursive call:

```
? to factorial :n
-> if equalp :n 0 [output 1]
-> output product :n factorial difference :n 1
-> end
factorial defined
? print factorial 5
120
```

Since `factorial` is an operator—it returns a number—we must use `output` in the recursive case as well as the base case, otherwise Logo will not know what to do with the value returned by the recursive call.

When a call to `output` is evaluated, evaluation of the rest of the procedure body stops. Hence, when `n` is zero, `output 1` is evaluated causing 1 to be returned, and causing the recursive call to *not* be evaluated. As another example, consider this operator:

```
? to play.with.output
-> print [before output]
-> print [still before output]
-> output "foo
-> print [after output]
-> foo bar baz
-> sum "not.a.number 15
-> end
play.with.output defined
? print play.with.output
before output
still before output
foo
```

As you can see, the instant we hit the call to `output`, evaluation of the rest of the procedure body is aborted. Although `play.with.output` prints stuff, it is still considered an operator because it returns a value. As with all operators, you better say what you want to do with it:

```
? play.with.output
before output
still before output
***Error: You did not say what to do with foo
```

It may surprise you to learn that `output` is not a special form (see the following section). It is just a primitive procedure that takes one argument. Part of your task will be to implement `output`.

1.4.2 Stop

As we have seen, `output` must be used in the base cases of Logo operators to stop the computation. What about Logo commands? For example, say we want to define a command `count-down` that prints the numbers from n to 1:

```
? to count.down :n
-> if equalp :n 0 [print "blastoff]
-> print :n count.down difference :n 1
-> end
count.down defined
? count.down 3
3
2
1
blastoff
-1
-2
-3
-4
-5
...
```

Why did it go into an infinite loop? Because saying `print "blastoff` does not terminate the program like `output` does; it just prints "blastoff." One way to fix this problem is to re-structure `count.down` and use `ifelse` instead of `if`:

```
to count.down :n
ifelse equalp :n 0 [print "blastoff] [print :n count.down difference :n 1]
end
```

But this can become unweieldly because, in our interpreter, all inputs to `ifelse` must be on the same line. (In Berkeley Logo, you can create Logo expressions that span multiple lines by placing a ~ at the end of each continuing line.)

There is another solution. You can use the built-in function `stop`, which takes no arguments. Like `output`, the job of `stop` is to halt further computation. But `stop` does this without returning anything (well, it returns `void`). Here is a working version of `count.down` that uses it:

```
to count.down :n
if equalp :n 0 [print "blastoff stop]   ;; like (begin (print 'blastoff) (stop))
print :n count.down difference :n 1
end
```

You will need to make `stop` work in your Logo interpreter.

1.5 Special Forms

One of the design principles of Logo was to minimize the number of special forms in the language. This is a worthy goal because they introduce "special cases" in the evaluation rules, complicating the interpreter. While STk bends over backwards to make special forms behave like first-class data (it's pretty easy to break—try mapping `if`), the metacircular evaluator gives a truer picture of the limitations of special forms. In the MCE, special forms are not variables bound to procedures; they cannot be passed as arguments to functions nor returned as the results of functions. This is the cost of deviating from the standard evaluation

rules.

The funny thing is that Logo does not have first-class procedures anyway, so having `if`, `ifelse` and `make` as ordinary procedures does not provide any additional expressive power to the language. However, minimizing the number of special forms does simplify the interpreter. The following is the code for `logo-eval`, which is analogous to `mc-eval` in the metacircular evaluator:

```
(define (logo-eval line-obj env)
  (let ((token (ask line-obj 'next-token)))
    (cond ((self-evaluating? token) token)
          ((variable? token)
           (lookup-variable-value (variable-name token) env))
          ((quoted? token) (text-of-quotation token))
          ((to? token) (eval-to line-obj))
          (else
            ;; handle procedure application
            ;; FILL IN
```

As in the MCE, the code first checks for self-evaluating expressions, quoted expressions and variables. The `else` is the procedure-application clause. Between the `quoted?` clause and the `else` clause are all the special forms—in Logo, there is just one: `to`, the special form that defines procedures. **You may not add any other special forms to your interpreter.**

But wait a second, don't `if` and friends *have* to be special forms? How do you keep their arguments from being evaluated? You quote them! There are two ways to quote expressions in Logo. To quote a word, use a single double quote, as in:

```
? print "cs61a
cs61a
```

Don't confuse these with the string type you have seen in Scheme. Scheme strings like `"Happy Birthday!"` are delimited by matching double quotes. In Logo, there are no strings, and the double quote when used on words serves the same function as Scheme's single quote. To quote a list in Logo, enclose it in square brackets:

```
? print [cs61a is cool]
cs61a is cool                ;; lists are printed without outermost brackets
```

This is analogous to saying `'(cs61a is cool)` in Scheme.

How does this turn special forms into regular procedures? In Logo, you just quote the arguments that in Scheme would be unevaluated. Take `make`, which is `define` and `set!` rolled into one. In Scheme, the first argument to `define` and `set!` is the *name* of the variable and is not evaluated. To get the same effect in Logo, quote it:

```
? make "pi 3.14
? print :pi
3.14
```

Quoting the first argument to `make` means that evaluating it yields the symbol `pi`, which is exactly what we want. The second argument to `make` is, of course, evaluated also. This means that variable names can be the results of arbitrarily complicated Logo expressions:

```
? make first [the matrix has you] "foo
? print :the
foo
? make :the "bar
```

```
? print :foo
bar
```

The Logo functions `if` and `ifelse` use a similar trick. But instead of words, they're given lists that look like Logo code:

```
? ifelse equalp 2 3 [print [2 is equal to 3]] [print [2 is not equal to 3]]
2 is not equal to 3
```

The expression `[print [2 is not equal to 3]]` is a list of two elements, the first is the word `print` and the second is the list `[2 is not equal to 3]`; `ifelse` executes this list as Logo code when the first argument to it is the word `false` (as in this case).

Here is another example using `if` instead of `ifelse`:

```
? if "false [print [la la la la]]
?
```

Nothing was printed because the first argument was the word `false`. Was `[print [la la la la]]` evaluated? Sure it was! It is a quoted list, so it evaluates to itself:

```
? [print [la la la la]]
*** Error: you did not say what to do with [print [la la la la]]
? print first [print [la la la la]]
print
```

Remember, `[print [la la la la]]` in Logo means the same as `'(print (la la la la))` in Scheme.

There is one more primitive procedure in Logo that is worth knowing about, `run`. This function, like `if` and `ifelse` takes a list that looks like Logo code and evaluates it:

```
? run [print "wassup print "dog]
wassup
dog
```

The results of saying `run` *expr* are the same as typing *expr* at the prompt directly:

```
? print "wassup print "dog
wassup
dog
```

With `run`, we can write our own `ifelse` procedure that is identical to the one that is built-in:

```
to my.ifelse :pred :consequent :alternative
ifelse :pred [run :consequent] [run :alternative]
end
```

Unlike the `my-if` we wrote in Homework 1.1 in Scheme, `my.ifelse` works just fine with recursive procedures—as long as the last two arguments to it are *lists of Logo code*.

Having what we'd think of as special forms be regular procedures (they don't even need to be primitive procedures—we can make our own, as with `my.ifelse` above) means that `logo-eval` becomes a pretty straightforward procedure. However, there is still a small complication. Procedures like `make`, `if`, `ifelse` and `run` need access to the current environment; `make` needs it so it can create or change variable bindings, and the other three need it when it comes time for them to evaluate the Logo-code-in-a-list. You will need to design a mechanism to pass the current environment to these procedures. An easy way to do this is to give all primitive procedures access to the current environment; regular procedures like `sum` and `print` can just ignore it.

2 The Project

Before starting the interpreter, write the Logo program `numspell`. Details are on the Web site. Put your work into a file `numspell.lg` and submit it along with your interpreter.

The file `~cs61a/lib/logo.scm` contains the start of a Logo interpreter. Add your code to this file. You will need to implement the following, roughly in the order shown:

I Primitive Procedures

a. Design representation of primitive procedures
b. Decide on a structure to hold them
c. Add certain primitive procedures to the Logo interpreter
d. Implement an argument-gathering mechanism for procedures
e. Write the part of `logo-apply` that handles primitive procedure applications.

II Variables

a. Choose a representation for environments
b. Write `lookup-variable-value`
c. Implement the Logo procedure `make`

III Compound Procedures

a. Choose a representation for compound procedures
b. Design a way to store them (may or may not be together with primitive procedures)
c. Implement the Logo special form `to`
d. Implement a way of extending environments
e. Implement dynamic scope in `logo-apply`
f. Implement `eval-sequence`, handle `stop` and `output`

IV Extra Features

a. Implement the Logo procedure `local`
b. Add a `read` function to Logo

The rest of this document describes in more detail the requirements of each step in the outline above, and gives some tips for their implementation. The order of the outline is followed. But before we do that, we need to discuss three general aspects of this project that will affect the interpreter as a whole.

2.1 Error Handling

As you have no doubt noticed, the metacircular evaluator does not handle errors at all. Any error—such as an unbound variable, a division by zero or an application of a non-procedure—crashes the entire program causing control to return to STk. While ignoring error handling simplifies the interpreter a good deal, it makes it a pain to use. Production quality interpreters such as STk or Berkeley Logo do not crash when the programmer mistypes an identifier.

The Logo interpreter you are building must be bulletproof. That is, it must recover from *all* errors (except infinite loops). Nothing except `bye` typed at the Logo prompt should cause control to return to STk. Each of the stages in the above project outline introduces a new set of error situations. We will point out most of

them, and you will need to catch them and recover from them. By "recover from them" we mean print an error message and return to the main loop. We have provided a procedure `logo-error` that does exactly this. This procedure takes any number of arguments and forms them into an error message that is printed, then returns to the `driver-loop`. Therefore, when you detect that an error has occurred—the arguments to `sum` were not numeric, a variable was not found in the environment, a procedure was not given the right number of arguments—you need only call `logo-error` with a helpful message, and it does the rest. The `logo-error` procedure is to Logo what the `error` procedure is to Scheme: a way to abort the current evaluation and return control to the main loop of the interpreter (print the prompt and continue reading input).

You do not need to understand how logo-error works, just how to use it. The code that makes it work is in the `driver-loop`:

```
(define (driver-loop)
  (display "? ")
  (flush)
  (let ((line (logo-read)))
    (cond ((null? line) (driver-loop))
          ((equal? line '(bye))
           (display "Thank you for using Logo.\n"))
          (else
           (call/cc (lambda (cont)                    ;; code that makes LOGO-ERROR
                      (set! back-to-driver cont)      ;; work on these lines
                      (let ((result (logo-eval-line (make-line-obj line))))))))
                        do stuff with result ...
```

Make sure not to modify the lines commented above since you might break `logo-error`. If you would like to understand how `logo-error` works, you can read up on Scheme continuations. Google for `call-with-current-continuation`, abbreviated as `call/cc`.

Do not modify `logo-error` either. The text of the error message you give it is completely up to you. It does not have to match the error messages in this document, nor those in Berkeley Logo. It should, however, be informative and genuinely helpful. However, the format of error messages, specifically the "*** Error:" that `logo-error` prints must remain as is. We may do some automatic testing of this project, so errors must be reported in a consistent format for the autograder to recognize them.

Lastly, it is not exactly true that nothing you type at the Logo prompt should crash the interpreter. There is at least one thing we know of that will crash Logo. Typing a single `"]"` at the very start of your Logo session will crash the interpreter:

```
STk> (initialize-logo)
? ]
*** Error: bad function in #f     ;; this is an STk error
```

This is only a problem if typed immediately after Logo is initialized. After one normal evaluation, or even one press of the Enter key, not even the single closing bracket will crash your interpreter. The reason for this behavior is complicated, and we felt fixing it would needlessly complicate the `driver-loop`.

Although most of the errors you have to catch will be pointed out in this project description, **it is your responsibility to thoroughly test your interpreter and catch other errors not mentioned here**. Please report any error cases you find on the class newsgroup so others can know about them, and so that the course staff can verify whether or not they are bona-fide errors and give hints on how to handle them.

2.2 The Logo Lexer

The metacircular evaluator uses the Scheme primitive `read` to read in a Scheme expression. The `read` function is specifically designed for this purpose; it knows to do things like turn expressions enclosed in parentheses into lists, and to turn the single character `'` into a call to the `quote` special form:

```
STk> (define what-i-typed (read))
'foo                                    ;; this is my input
STk> what-i-typed
(quote foo)
```

Actually, `read` does a great deal more work then you might think. In addition to making lists, it looks at the characters you type in and decides which ones should be "lumped together" to form atomic values. For example, `read` knows that a sequence of numeric characters like 362 constitutes a Scheme integer, and a sequence of numeric characters with a period in it like 3.002 constitutes a floating-point number. It knows that a pound sign `#` marks the start of a boolean literal. It knows that a sequence of characters that cannot be a number (contains at least one non-digit) like `hello` or `123hello4` or even `f32437f` makes up a Scheme symbol. This process is known as *lexing*.

Using `read` on Logo expressions does not work because Logo has a very different syntax from Scheme. As mentioned in Section 1.1, Logo expressions are not lists. Moreover, Logo has two different quoting mechanisms. The single double quote in Logo quotes words:

```
? print word "abra "cadabra
abracadabra
? print "hello
hello
```

However, to quote a list in Logo you enclose it in square brackets:

```
? print first [look out trinity]
look
? print sentence [the first] "matrix
the first matrix                    ;; lists are printed without outermost brackets
```

A lot of students become confused by this! In Scheme, to treat an entire list literally, you just precede it with a single quote as in `'(a b c)`. To get the same effect in Logo you'd say `[a b c]`. Saying `"[a b c]` in Logo does not make sense since the double quote only works on words.

Because Logo source code looks so different from Scheme, a different lexer is needed for it. That's where `logo-read`, the procedure we've provided to read in a Logo expression, comes in. It is a rather complicated procedure, and you do not need to understand how it works. You do need to understand what it does.

The `logo-read` procedure always returns a list of *tokens*, which are the various components of Logo code. To play with `logo-read`, you must call it and type your input to it on the same line, like this:

```
STk> (logo-read)1 324 3.4
(1 324 3.4)
```

In the preceding expression, `logo-read` returned a list of three tokens, each of them a number.

```
STk> (logo-read)print sum 2 product 3 17
(print sum 2 product 3 17)
```

Here we get a list of 6 tokens, some of which are words and some are numbers.

But the main thing that `logo-read` does for you is turn literal Logo lists, typed by enclosing them in square

brackets, into the familiar Scheme lists:

```
STk> (logo-read)print list [a b c] [d e f]
(print list (a b c) (d e f))
```

The return value is a list of *four* tokens, the first is the word `print`, the second is the word `list`, the third is the *list* `(a b c)` and the fourth is the list `(d e f)`. As you can see, `logo-read` handles nested Logo lists just fine:

```
STk> (logo-read)print [[a [b [[] c] [d] e] f] g]
(print ((a (b (() c) (d) e) f) g))
```

It has turned the deep Logo list `[[a [b [[] c] [d] e] f] g]` into the deep Scheme list `((a (b (() c) (d) e) f) g)`.

The square brackets are used a lot in Logo to enclose code that is not to be run until later, if at all. For example, Logo's `if` procedure takes two arguments: the first must evaluate to the words `true` or `false` and the second must evaluate to a list that will be executed as Logo code if the value of the first argument is `true`:

```
? if emptyp [] [print [[] is empty]]
[] is empty
? if oddp 7 [print [7 is odd]]
7 is odd
? if emptyp [a b c] [print [[a b c] is empty]]    ;; not empty
?
```

Here is how each of these expressions will look like when read in by `logo-read`:

```
STk> (logo-read)if emptyp [] [print [[] is empty]]
(if emptyp () (print (() is empty)))
STk> (logo-read)if oddp 7 [print [7 is odd]]
(if oddp 7 (print (7 is odd)))
STk> (logo-read)if emptyp [a b c] [print [[a b c] is empty]]
(if emptyp (a b c) (print ((a b c) is empty)))
```

As you can see, each Logo expression in square brackets is turned into a regular Scheme list. Don't think "`logo-read` turns square brackets into parentheses." What really happens is that `logo-read` turns square brackets into box-and-pointer lists, which are then printed by Scheme in the normal way with parentheses around them.

One important difference between `read` and `logo-read` is that Scheme expressions can be several lines long because they are delimited by matching parentheses; `read` keeps consuming input until it reaches the closing parenthesis that matches the first opening one. Because it is more difficult to determine where a Logo expression ends, `logo-read` makes the simplifying assumption that Logo expressions fit on one line. Hence, `logo-read` will always read exactly one line of Logo code. All procedures called on this line better see their arguments on the same line, or an error will occur when the line is evaluated.

For example, `ifelse` takes three arguments:

```
? ifelse listp [] [print [[] is a list]] [print [[] is not a list]]
[] is a list
```

When using `ifelse`, make sure that all three arguments to it appear on the same line, as shown above.

2.3 Working with Line-Objects

We will represent each line of Logo code as an instance of the following class:

```
(define-class (line-object text)
  (method (next-token)
          (if (null? text) (error "Empty line object"))
          (let ((token (car text)))
            (set!  text (cdr text))
            token))
  (method (peek)
          (car text))
  (method (has-more?)
          (not (null? text))))
```

Hence, when you see "line-obj" as a parameter name (as in `logo-eval` and `logo-eval-line`), know that it refers to an instance of this class. You are able to remove the tokens from the line one-by-one, and find out when there are no more. You may, of course, modify this class as you see fit.

The reason we are representing Logo lines as objects instead of as regular lists is a bit difficult to explain. The basic reason is that since Logo expressions are not parenthesized, it is difficult to know where one subexpression ends and another begins. Unlike Scheme code, which can be picked apart one subexpression at a time, we shall examine Logo code one token at a time. And we will need a way to tell how many tokens a recursive call to `logo-eval` has "consumed" from the line. The easiest way to do this is modify the state of the line to reflect how many tokens remain to be evaluated. That's why we "wrap" each line of Logo in this object. Trust us on this one.

As you can see, the `driver-loop` reads a line of Logo code, wraps it in one of these line-object things and hands it off to `logo-eval-line` to be evaluated. (We call `logo-eval-line` instead of `logo-eval` because the line may contain several distinct Logo expressions; `logo-eval` will evaluate exactly one Logo expression.)

3 Okay, Get to Work

Time to write the actual interpreter.

3.1 Primitive Procedures

The first thing the interpreter will need is a set of primitive operators. Like in the metacircular evaluator, primitive procedures should be STk procedures, and should be applied in underlying Scheme. You need to design a representation for primitive procedures, and devise a method to store them somewhere. We recommend making good use of data abstraction in your representation, in case you'll need to tweak it later.

The arguments to primitive procedures must be of the appropriate types, or an error should be triggered with `logo-error`. For example:

```
? print sum product 4 5 "foo
*** Error: arguments to sum not numeric: 20 "foo
? word [what time is] "it
*** Error: arguments to word not words: [what time is] it
```

As you can see, the arguments to any of the mathematical operations must pass the `number?` predicate; both arguments to `word` must pass the `word?` predicate. Some primitives are more picky still because they require that only one of their arguments meet some condition:

```
? if "maybe-true [print "yes]
*** Error: first argument to if not true or false
? quotient 10 0
*** Error: division by zero
```

Here is a listing of the required primitive procedures, what they do and hints on error handling. You should feel free to add more primitives to your Logo interpreter, but you must have at least these:

- `sum`, `product`, `difference` — These correspond to the Scheme functions `+`, `*` and `-`. In Logo, they each take only two arguments, both of which have to pass the `number?` predicate.
- `unary-minus` — This procedure takes one argument, which has to be a number, and negates it, as in:

  ```
  ? print sum 3 -3
  0
  ```

  You won't ever have to call it directly. Instead, `logo-read` turns some occurrences of `-` into `unary-minus`:

  ```
  STk> (logo-read) print sum 3 -3
  (print sum 3 unary-minus 3)
  ```

  Remember, we are not handling `+`, `*` and `-` as infix operators.
- `quotient` — Despite it's name, this procedure corresponds to `/` in Scheme, not to `quotient`. Both arguments to it must be numeric, and the second one cannot be zero. One way to catch the division-by-zero error is to put a wrapper around `/`:

  ```
  (lambda (a b) (if (= b 0) (logo-error "division by zero") (/ a b)))
  ```

  Unfortunately, calling `logo-error` outside the Logo interpreter might not work, so you won't be able to test this in isolation.
- `first`, `butfirst` — Remember these from the start of the semester? Well, Brian Harvey stole these Logo procedures and put them into Scheme. In Logo, however, they work a little differently. Since Logo has lists, sentences and words `first` and `butfirst` work on all three types. They each take one argument, and if that argument is a word, call the Scheme version of `first` or `butfirst`; if it is a list (or sentence) call `car` and `cdr`. Remember, numbers *are* words, so the following is not a bug:

  ```
  ? print butfirst 3454
  454
  ```

  While there is no empty word in Logo, there is an empty list (sentence), `[]`. Using any of the above on it is an error.
- `word` — Takes two arguments, both of which have to satisfy the `word?` predicate, and works just like the similarly named function in Scheme.
- `list` — Takes two arguments, which can be of *any* type, and works like the similarly named Scheme primitive.
- `sentence` — Takes two arguments, which have to be either words or sentences, and works just like Scheme's `sentence` function. Use the Scheme function `sentence?` to test the arguments.
- `fput` — This is `cons`, but with a slight twist. Logo does not allow arbitrary pairs:

  ```
  ? fput 3 10
  *** Error: second argument to fput not a list
  ```

  In our implementation, arbitrary pairs (pairs that are not lists) are fatal since the given `logo-print` procedure will not display them correctly. Make sure that the second argument to `fput` is a list—it must pass the `list?` predicate. The first argument can be anything.

- `wordp`, `numberp`, `listp`, `emptyp`, — These are general Logo predicates. They each take one argument, which can be of any type (after all, the point of a predicate is to test the type of its argument), and return the *word* `true` or `false`. Unlike Scheme, Logo does not have a special boolean type. For example:

```
? print numberp sum 2 3
true
? print wordp listp []
true
```

  The corresponding Scheme function should be obvious for most of these, but you'll need to convert Scheme's notion of truth to Logo's.

- `equalp`, `lessp`, `greaterp` — These are the binary predicates; `equalp` is Scheme's `equal?` and works on any types:

```
? print equalp 7 [[a] [[deep] list]]
false
? print equalp numberp "hello emptyp [not empty]
true
```

  The other two, `lessp` and `greaterp`, correspond to `<` and `>` and must be given two numbers. Again, don't forget to convert Scheme booleans into the words `true` and `false`.

- `print` — This one-argument function corresponds to the `logo-print` procedure that is given. The argument to it can be any Logo type.

- `load` — This function corresponds to the provided `meta-load` procedure; it takes the name of a file, which must be a word, and loads the contents of the file into Logo. As you can see, the error-checking is already done in the function, so you need only add it as a primitive procedure. To use it, type:

```
? load "numspell.lg
```

- `if`, `ifelse` — These procedures correspond to the procedures `logo-if` and `logo-ifelse` that are provided; `if` takes two arguments and `ifelse` takes three. See Section 2.5 on Special Forms for details on using `if` and `ifelse`. For example, here is the code for `logo-if`:

```
(define (logo-if env pred consequent)
  (if (not (true-or-false? pred))
      (logo-error "argument to if not true/false: " pred))
  (if (logo-true? pred)
      (logo-run env consequent)
       void))
```

  As you can see, it takes an extra argument `env` that is not provided by the Logo programmer. It is the current environment, and must be supplied to `logo-if` so that *consequent* may be evaluated with respect to it. **It is up to you to figure out how to pass the environment to procedures that need it.** Making it an additional argument to `logo-if` is just a suggestion. Same for `logo-ifelse` and `logo-run`. This is *your* interpreter and you can design things the way you want to. (The only thing you cannot do is add any special forms.)

- `run` — This procedure corresponds to the `logo-run` procedure that is provided. It takes one argument, a list that looks like Logo code, and evaluates it *in the current environment.*

```
? print run [sum 1 butfirst 110]
11
? run [print fput "one ifelse "false [[foo bar]] [[2 3 4]]]
one 2 3 4
```

Just like `logo-if` and `logo-ifelse` above, `logo-run` needs access to the environment. How you do this is up to you.

After you have added these primitive procedures, **you must implement a way to give them their arguments**. Essentially, write a `list-of-values` function (from MCE) for Logo. Since you will want to use it for compound procedures too, it would be a good idea for this function to take as a parameter the number of arguments required. **Hint:** Take in the line-object, but let `logo-eval` do the work of evaluating the arguments.

Test your work on the nastiest compositions of primitives you can think of. Here are some ideas:

```
? print numberp sum first 324 quotient 20 first [2 4 6]
true
? ifelse first [true false] [run [print "yes]] [run [print "no]]
yes
? print quotient 27 run [3]
9
? if word "t word "r word "u "e [print quotient product 10 10 0]
*** Error: division by zero
? print fput [hello] sum "one word "tw "o
*** Error: bad arguments to sum: one two
```

In addition to the errors described above, there are two more error cases we want you to catch. Handle the case when a procedure is not given enough arguments:

```
? sum
*** Error: not enough inputs to sum
? print product first [1 2 3]
*** Error: not enough inputs to product
? ifelse "true [print "yes]
*** Error: not enough inputs to ifelse
```

This situation occurs when the line-object becomes empty, but you're still gathering arguments. The text of the error message is up to you. You don't have to include the name of the procedure that is lacking inputs, although that would make your interpreter more user-friendly.

Note that the case of too many arguments to a procedure is already handled by `logo-eval-line` and `driver-loop`:

```
? print [a] [b] [c] [d] [e]
a
*** Error: you did not say what to do with [b]
```

The other error situation concerns Logo commands. Recall that Logo commands, like `print`, are not supposed to return a value. They are used only for their side-effects. We implement this behavior by having `print` return `void`. It does not make sense, then, to use the return value of `print` as a subexpression, like this:

```
? list print [hello] "there
hello
*** Error: print did not output to list
? print print 7
7
*** Error: print did not output to print
```

Please catch this error. An easy way to do this is to check that none of the arguments to a procedure are `void`. Use the `void?` predicate that is given.

3.2 Variables

The next step is to create variables. We have not needed them so far because procedures are not variables in Logo. However, we'll need them in the next section to implement compound procedures.

3.2.1 Environments

We need to represent environments in some way. You should be familiar with the basic operations on environments: looking up, creating and altering variable bindings. Currently the value of `logo-global-environment` is `foo`, which is not a good environment. Once you have decided how to represent environments, you should change `logo-global-environment` to be something more reasonable.

If you like the way environments are represented in the metacircular evaluator, feel free to steal the implementation and to modify it as you see fit. You can use Kurt's environments from lecture. You may instead want to implement environments from scratch in your own way (maybe as OOP objects!). The implementation of environments is completely and utterly up to you. The Logo programmer will be oblivious to their existence.

A good place to put any initialization code for environments (or other aspect of the Logo interpreter) is in the `initialize-logo` function, just before the call to `driver-loop`:

```
(define (initialize-logo)
  ;;
  ;; ADD ANY INITIALIZATION CODE HERE
  ;;
  (logo-read)        ;; hack to avoid repeated ? prompt
  (driver-loop))
```

3.2.2 Operations on Environments

As you can see, the `variable?` clause already exists in `logo-eval`, along with the definitions of `variable?` and `variable-name`. Variables in Logo are prefixed with a colon, but the colon is not part of their name. Hence, `:n` is a request to look up the value of the variable `n`. You need to write `lookup-variable-value`. A variable that is not bound should produce an error:

```
?  list :a :b
*** Error: a has no value
```

When we create compound procedures in the next section, we will require an ability to extend environments. So you may want to provide an `extend-environment` function now. If you are using the metacircular evaluator's representation of environments, it is already written.

3.2.3 Make

How does one actually create variables in Logo? You use the `make` primitive; it plays the roles of both `define` and `set!`. The first argument to `make` is the name of the variable, and the second argument is the value. If the variable already exist in some environment (possibly an enclosing environment), `make` should change its value there; otherwise, `make` must create the variable **in the global environment**. For example:

```
? print :plus
*** Error: plus has no value
? make "plus "sum
```

```
? print :plus
sum
? run fput "print fput :plus [10 30]         ;; (cons 'print (cons plus '(10 30)))
40
? make "plus "equalp
? run fput "print fput :plus [10 30]
false
```

Your job is to implement `make`, then add it as a primitive procedure to your Logo interpreter. **Do not add make as a special form!** See Section 1.5 on Special Forms for more information on `make`. Check that the first argument to `make` is a word, and call `logo-error` if this is not the case:

```
? make [a b c] 6
*** Error: bad variable name [a b c]
```

Berkeley Logo actually allows you to make variables out of numbers:

```
? make 74 7
? print :74
7
```

The decision to allow or disallow fully numeric variable names is left to you; we will not test this case.

Note that `make` is a command, so it should return `void`.

Since we don't yet have compound procedures working, we don't have local environments, so it'll not be possible to test `make` fully through the Logo interpreter. **Test make in isolation by creating environments by hand.** Also, test the interaction of variables with primitive procedures and error handling. Here are some ideas to get your started:

```
? make "true.value "true
? make "false.value "false
? print list fput :true.value [] fput :false.value []
[true] [false]
? if :false.value [print product 2 :six]
? make first [six seven] 6
? if :true.value [print product 2 :six]
12
```

3.3 Compound Procedures

Time to invent compound procedures! You will need to come up with a representation for compound procedures, and a place to put them (maybe together with primitive procedures?). One thing that a compound procedure should probably know is how many arguments it takes. You should be able to employ the same argument-gathering mechanism for primitive and compound procedures. To simplify our interpreter, all procedures take a set number of arguments.

The next step is to write Logo's only special form, `to`. The Scheme procedure that implements it is `eval-to`. A Logo procedure definition looks like this:

```
to fib :n
if equalp :n 0 [output 1]
if equalp :n 1 [output 1]
output sum fib difference :n 1 fib difference :n 2
end
```

Immediately after the `to` token you should expect to find the name of the procedure, a word. The rest of the line contains formal parameters to the procedure, each in the form of a Logo variable.

The body of the procedure consists of zero or more lines of Logo code. The job of `eval-to` will be to read in these lines (using `logo-read`) stopping when a line containing just `end` is encountered. How you store the body of a Logo procedure is completely up to you. As `eval-to` reads in these lines, it should prompt the user by printing an arrow (`"-> "`), so the actual interaction would look like this:

```
? to fib :n
-> if equalp :n 0 [output 1]
-> if equalp :n 1 [output 1]
-> output sum fib difference :n 1 fib difference :n 2
-> end
fib defined
```

The word "end" is not part of the body of the procedure. It is simply a marker telling `eval-to` that the body is finished. After you see "end," print out "⟨*name of procedure*⟩ defined" and store the procedure somewhere.

You should perform some error-checking at definition-time. Make sure that the name of the procedure (the token that follows `to`) is a word; make sure that all the formal parameters are Logo variables (use the given `variable?` procedure).

Berkeley Logo prevents users from re-defining procedures. Once you define `square` you are stuck with it—you cannot define `square` again. It is up to you to allow or disallow procedure re-definition.

The body of a procedure can consist of zero or more lines of Logo, so the following is a valid Logo procedure:

```
to do.nothing :a :b :c :d
end
```

3.3.1 Evaluating the procedure body

The next task will be to write a version of `eval-sequence` for Logo, which will evaluate the body of a Logo procedure in a given environment. The job of `eval-sequence` will be to evaluate each line in the body of a procedure. Each line in the body of a procedure must result in `void`, `stop` or `output`—otherwise print the "you don't say what to do with X in FOO" error. The details of this process are left to you.

To make `stop` and `output` work, you will need to add them as primitive procedures to the interpreter. The `stop` procedure takes zero arguments and corresponds to the given `logo-stop` function. The `output` procedure corresponds to `logo-output` and takes one argument. Here they are:

```
(define (logo-stop) stop)

(define (logo-output x) (set-cdr! output x) output)
```

As you can see, they return one of the following pairs:

```
(define output (cons 'output 'whatever))
(define stop (cons 'logo 'stop))
```

The reason they're pairs is so that, as with `void`, we can test for them using the provided predicates `stop?` and `output?`.

When `stop` is evaluated in a procedure body, you should stop all further evaluation of the body and return `void`. When `output` is evaluated in a procedure body, you should stop all further evaluation of the body

and return the thing in the `cdr` of the pair. The details of this process are left to you to figure out. You may even implement `stop` and `output` without using any of the code we've given you; you may wish to discuss your approach with a TA first, however.

Try to test your `eval-sequence` procedure in isolation first; make up a non-recursive procedure body and run `eval-sequence` on it.

The last thing to do is to modify `logo-apply` to handle compound procedures. This is where you will implement dynamic scope.

First test your work on non-recursive procedures, then try things like `factorial`, `fib` and `count-down`. Once you have gotten compound procedures to work, you have pretty much completed the project. The rest is fluff. **Thoroughly test your implementation of compound procedures, as it will be worth the bulk of the project points.**

3.3.2 Error handling with stop, output and to

The addition of `stop` and `output` means a host of new errors need to be caught. The two procedures `stop` and `output` are only allowed inside the body of a procedure. Invoking them at the prompt should cause an error:

```
? stop
*** Error: you can only use stop inside a procedure
? output "hello
*** Error: you can only use output inside a procedure
? run [output []]
*** Error: you can only use output inside a procedure
? run [print "a print "b stop]
a
b
*** Error: you can only use stop inside a procedure
? print if "true [output 4]
*** Error: you can only use output inside a procedure
? if stop stop stop
*** Error: argument to if not true or false
```

Model your error-handling on Berkeley Logo. Things that cause errors in Berkeley Logo should also cause errors in your interpreter—but the error messages don't need to match. In fact, you don't even have match the error. In the case of `if stop stop stop` above, for example, the "you can only use stop inside a procedure" is also appropriate.

Moreover, inside or outside a procedure body, `stop` and `output` cannot appear as an argument to other functions.

```
? print sum output 17 2
*** Error: output not valid as subexpression
? first stop
*** Error: stop not valid as subexpression
? to mess.up
-> print stop
-> end
mess.up defined
? mess.up
*** Error:  stop not a valid subexpression
```

Here are more procedure definitions that should produce errors when run (the text of the error message is up to you):

```
to mess.up1 :a :b :c
output stop
end

to mess.up2 :a :b :c
output output output sum :a :b
end

to mess.up3 :a :b :c
output print :a
end
```

Note that saying `if emptyp :lst [output []]` inside the body of a procedure is okay because we're not using `output` as an argument to `if`. The second argument to `if` is the two-element list `[output []]`.

Doing error checking on `to` is harder, because it is a special form. Berkeley Logo prevents users from using `to` inside a procedure, as in:

```
to huh :a :b
to bar
end
```

Berkeley Logo also prevents using `to` as a subexpression, as in:

```
print to square
```

You are not responsible for doing error-handling of `to` because it is difficult. You may assume that `to` will be typed at the Logo prompt and nowhere else. Still, give it a try if you're up for a challenge. We may even offer extra credit for people who do error-handling of `to`. Model your error-handling on Berkeley Logo.

3.4 Extra Features

We are going to add two more features to our Logo interpreter. Neither of them should require more than five lines of code to implement.

3.4.1 Local

Add `local` to your Logo interpreter. This procedure creates local variables. That does not say "local state" variables, the kind that persist across procedure calls; that would be much, much harder in a dynamically scoped language. (Why?) This procedure takes either a single word, or a list of words. A variable is created in the *current* environment for each of these words, with that word as its name. Unlike `make`, a variables made by `local` is not immediately assigned a value. The value must subsequently be assigned with `make`. It is an error to use a variable made by `local` before it has a value.

```
? local [fluffy buffy love]            ;; creates three variables,
? print :fluffy                        ;; but they are uninitialized
*** Error: fluffy has no value
? make "fluffy "pink
? print :fluffy
pink
? print :buffy
```

```
*** Error: buffy has no value
? make "love "green
? print :love
green
```

Here is a more useful example of `local` in action:

```
? to factorial :n
-> local "result make "result 1
-> fact.help
-> output :result
-> end
factorial defined
? to fact.help
-> if equalp :n 0 [stop]
-> make "result product :result :n
-> make "n difference :n 1
-> fact.help
-> end
fact.help defined
? print factorial 6
720
? print :result
*** Error: result has no value             ;; result was never a global variable
```

The tricky part will be to figure out what to make the initial value of the variable, the value it is given immediately after the call to `local`, before it is set with `make`. It must be a special kind of value that denotes the *lack* of any real value. Importantly, there should not be a way of creating this value from the Logo side. That is, it *must not* be possible to do the following:

```
? make "x ⟨some Logo expression⟩
? print :x
*** Error: x has no value
```

3.4.2 Read

The interpreter is almost complete. Almost. If you take a step back and survey what you've done you'll realize that the Logo interpreter is missing something rather fundamental. You can't write any Logo programs that interact with the user! This is because there is no way for the Logo programmer to read user input. Fix this situation by adding the Scheme procedure `logo-read` as a primitive. It should correspond to a zero-argument Logo procedure called `read`. We are not going to provide any sample calls for this one, since you should be able to figure out for yourself when you have done this correctly.

Please demonstrate our new ability to interact with the user by writing a Logo version (it doesn't have to be identical) of the following Scheme program:

```
(define secret 0)

(define (guess)
  (set!  secret (random 100))
  (print "I am thinking of a number ...  can you guess what it is?")
  (guess-loop))

(define (guess-loop)
```

```
(display ">>> ")
(let ((input (read)))
  (cond ((= input secret) (print "You've got it!"))
        ((< input secret) (print "Too low.") (guess-loop))
        (else (print "Too high.") (guess-loop)))))
```