CS 61A Programming Project #4: A Logo Interpreter
Summer 2005

In Chapter 4 we study a Scheme interpreter written in Scheme, the metacircular evaluator. In this project we modify that evaluator to turn it into an interpreter for a different programming language. This project is valuable for several reasons: First, it will make you more familiar with the structure of the metacircular evaluator because you'll have to understand which procedures to modify in order to change some aspect of its operation. Second, working with another language may help overcome some of the confusion students often have about talking to two different versions of Scheme at once. In this project, it'll be quite obvious when you're talking to Scheme and when you're talking to Logo. Third, this project will encourage you to think about the design of a programming language. Why did Scheme's designers and Logo's designers make different choices?

This is approximately a two-week project. As in the adventure game project, you'll have a group of two people, person A and person B. You will do most of the work separately and then meet together for the final steps. After the first part you should be able to enter instructions using primitive procedures with constant arguments. In the second part you will add variables and user-defined procedures.

Before you begin working on the project, you have to know something about the Logo programming language. The Logo-in-Scheme interpreter is structured like the metacircular evaluator, so to run it you say

```
nova ~ > stk
STk> (load "~cs61a/lib/obj.scm")
STk> (load "~cs61a/lib/logo.scm")
STk> (load "~cs61a/lib/logo-meta.scm")
STk> (initialize-logo)
?
```

and the question-mark prompt means that you're talking to Logo. (The versions in the library are incomplete; you'll have to do the project before you can really run it!) Errors in your Logo instructions can cause the interpreter to get a Scheme error message and return you to the Scheme prompt. If this happens, type `(driver-loop)` to return to Logo. You should only use `(initialize-logo)` once, or else you will lose any Logo variables or procedures you've defined.

>>> NOTE TO MACINTOSH GAMBIT USERS: Before running this project you must tell Gambit to read a line, not a Scheme expression, in response to the ENTER key. To do this, look in the Edit menu and select Window Styles. Near the bottom right corner of the window that will appear are three check boxes; the middle one is labelled "Enter = Send Line". Check that box (so that you see an X in the box), then click OK.

If you want to experiment with a *real* Logo interpreter, to see how it's supposed to work, just say

```
nova ~ > logo
```

to the shell. You exit Logo by saying `bye`.

General Logo

Logo is essentially a dialect of Lisp. That makes it a good choice for this project, both because it'll be easy to teach you the language and because the modifications to the evaluator are not as severe as they would be for an unrelated language. However, Logo was designed for educational use, particularly with younger children. Many design decisions in Logo are meant to make the language more comfortable for a beginner. For example, most Logo procedures are invoked in **prefix form** (first the procedure name, then the arguments) as in Lisp, but the common arithmetic operators are also provided in the customary **infix form**:

```
? print sum 2 3
5
? print 2+3
5
```

(Note: As you work with the Logo-in-Scheme interpreter, you probably won't be impressed by its comfort. That's because our interpreter has a lot of rough edges. The most important is in its error handling. A real Logo interpreter would not dump you into Scheme with a cryptic message whenever you make a spelling mistake! Bear in mind that this is only a partial implementation. Another rough edge is that **there is no precedence among infix operators**, unlike real Logo, in which (as in most languages) multiplication is done before addition. In this interpreter, infix operators are carried out from left to right, so 3+4*5 is 7*5 or 35, not 3+20.

Even in the trivial example above, adding two numbers, you can see several differences between Scheme and Logo. The most profound, in terms of the structure of the interpreter, is that expressions and their subexpressions are not enclosed in parentheses. (That is, each expression is not a list by itself.) In the metacircular evaluator, `eval` is given one complete expression as an argument. In the Logo interpreter, part of `eval`'s job will be to figure out where each expression begins and ends, by knowing how many arguments are needed by each procedure, for example:

```
? print sentence last [UPS driver Jeff] word "was "here
```

Logo must understand that `word` requires two arguments (the quoted words that follow it) while `last` requires one, and that the values returned by `word` and `last` are the two required arguments to `sentence`. (Also, `print` requires one argument.)

Another important difference between Scheme and Logo is that in the latter you must explicitly say PRINT to print something:

```
? print 2+40
42
? 2+40
You don't say what to do with 42
```

An expression that produces an unused value causes an error message. Unlike Scheme, in which every procedure returns a value, Logo makes a distinction between **operations** that return a value and **commands** that are used for effect. `print` is a command; `sum` is an operation. This distinction means that Logo has less of a commitment to functional programming style, and it makes the interpreter a little more complicated because we have to keep track of whether we have a value or not. But in some ways it's easier for the user; we don't keep saying things like "`set!` returns some value or other, but the value is unspecified and you're not supposed to rely on it in your programs." Also, Logo users don't see the annoying extra values that Scheme programs sometimes print because some procedure that was called for effect happens to return `()` or `#f` as a value that gets printed.

One implication for the interpreter is that instead of Scheme's read-eval-print loop

```
(define (driver-loop)
  (display "> ")
  (print (eval (read) the-global-environment))
  (driver-loop))
```

Logo just has a read-eval loop without the print.

In Scheme something like 2+3 would be considered a single symbol, not a request to add two numbers. The plus sign does not have special status as a delimiter of expressions; only spaces and parentheses separate expressions. Logo is more like most other programming languages in that several characters are always considered as one-character symbols even if not surrounded by spaces. These include arithmetic operators, relational operators, and parentheses:

```
+ - * / = < > ( )
```

Remember that in Scheme, parentheses are used to indicate list structure and are not actually part of the internal representation of the structure. (In other words, there are no parentheses visible in the box-and-pointer diagram for a list.) In this Logo interpreter, parentheses are special symbols, just like a plus sign, and are part of the internal representation of an instruction line. Square brackets, however, play a role somewhat like that of parentheses in Scheme, delimiting lists and sublists. One difference is that a list in square brackets is automatically quoted, so `[...]` in Logo is like `'(...)` in Scheme:

```
? print [hi there]
```

Logo uses the double quotation mark `(")` to quote a word, so `"foo` in Logo is like `'foo` in Scheme. Don't get confused -- these quotation marks are not used in pairs (i.e. two at a time), as in Scheme string constants ("error message"); a single one is used before a word to be quoted. **(Note: This will not work in Windows STk)**

```
? print "hello
```

Just as the Scheme procedure READ reads an expression from the keyboard and deals with parentheses, spaces, and quotation marks, you are given a LOGO-READ procedure that handles the special punctuation in Logo lines.
One important difference is that a Scheme expression is delimited by parentheses and can be several lines long; LOGO-READ reads a single line and turns it into a list. If you want to play with it from Scheme, first get out of Logo (if you're in it), then type the invocation `(logo-read)` and the Logo stuff you want read all on the same line:

```
STk> (logo-read)print 2+(3*4)
(print 2 + ( 3 * 4 ))
STk> (logo-read)print se "this [is a [deep] list]
(print se "this (is a (deep) list))
```

Remember that the results printed in these examples are Scheme's print representation for Logo data structures! Don't think, for example, "`logo-read` turns square brackets into parentheses." What really happens is that `logo-read` turns square brackets into box-and-pointer lists, and Scheme's print procedure displays that structure using parentheses. Note: In the first of these two examples, the inner parentheses in the returned value
are *not* the boundaries of a sublist! They are parenthesis symbols.

What logo-read returned was a sentence with eight words:
`print, 2, +, (, 3, *, 4, and ).`
This makes it a little tricky to be sure what you're seeing.

If you want to include one of Logo's special characters in a Logo word, you can use backslash before it:

```
?print "a\+b
a+b
```

Also, as a special case, a special character other than square bracket is considered automatically backslashed if it's immediately after a quotation mark:

```
?print "+
+
```

All of this is handled by `logo-read`, a fairly complicated procedure. You are not required to write this, or even to understand its algorithm, but you'll need to understand the results in order to work on `eval`.

Procedures and Variables

Here is a Scheme procedure and an example of defining and using the corresponding Logo procedure:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

```
? to factorial :n
-> if :n=0 [output 1]
-> output :n * factorial :n-1
-> end
? print factorial 5
120
```

There are several noteworthy points here. First, a procedure definition takes several lines. The procedure name and formal parameters are part of the first instruction line, headed by the `to` special form. (This is the *only* special form in Logo!) The procedure body is entered on lines in response to a special `->` prompt. These instruction lines are not evaluated, as they would be if entered at a `?` prompt, but are stored as part of the procedure text. The special keyword `end` on a line by itself indicates the end of the body.

Unlike Scheme, Logo does not have first-class procedures. Among other things, this means that a procedure name is not just a variable name that happens to be bound to a procedure. Rather, *procedures and variables are looked up in separate data structures*, so that there can be a procedure and a variable with the same name. (This is sometimes handy for names like `list` and `word`, which are primitive procedures but are also convenient formal parameter names. In Scheme we resort to things like `l` or `lst` to avoid losing access to the `list` procedure.) Variable names are part of a Scheme-like environment structure (but with dynamic rather than lexical scope); procedure names are always globally accessible. To distinguish a procedure invocation from a variable reference, the rule is that a word `foo` without punctuation is an invocation of the procedure named `foo`, while the same word with a colon in front `(:foo)` is a request for the value of the variable with that name.

A Logo procedure can be either a command (done for effect) or an operation (returning a value). In this example we are writing an operation, and we have to say so by using the `output` command to specify the return value.
Once an `output` instruction has been carried out, the procedure is finished; in this example, if the `if` in the first line of the body outputs 1, the second line of the body is not evaluated.

The file `~cs61a/lib/test.logo` contains definitions of several Logo procedures that you can examine and test to become more familiar with the language. You can load these definitions into your Logo interpreter by copying it to your directory and then using Logo's `load` command:

```
? load "test.logo
```

(Notice that if you want to use a filename including slashes you have to backslash them to make them part of the quoted word.)

Unlike Scheme's `if`, Logo's `if` is not a special form. You probably remember a homework exercise that proved that it had to be, but instead Logo takes advantage of the fact that square brackets quote the list that they delimit. The first argument to `if` must be the word `true` or the word `false`. (Predicate functions in Logo always return one of these two words. Logo does not accept any non-`false` value as true; anything other than these two specific words is an error.) The second argument is a list containing instructions that should be run conditionally. Because the list is enclosed in square brackets, the instructions are not evaluated before `if` is invoked. In general, anything that shouldn't be evaluated in Logo must be indicated by explicit quotation, with `"xxx` or `[xxx]`. The only special form is `to`, in which the procedure name and formal parameter names are not evaluated.

The procedures `first`, `butfirst`, etc. that we've been using to manipulate words and sentences were invented in Logo. The Scheme versions don't quite work as smoothly as the real Logo versions, because Scheme has four distinct data types for numbers, symbols, strings, and characters; all of these are a single type (words) in Logo. If you evaluate `(bf 104)` in Scheme you get `"04"`, not just `04`, because the result has to be a Scheme string in order not to lose the initial zero.

Our Logo interpreter does manage to handle this:

```
? print bf 104
04
? print bf bf 104
4
```

The interpreter represents 04 internally as a Scheme symbol, not as a number. We can nevertheless do arithmetic on it

```
? print 7+bf 104
11
```

because all the Logo arithmetic functions have been written to convert symbols-full-of-digits into regular numbers before invoking the actual Scheme arithmetic procedure. (This is the job of `make-logo-arith`.)

Actual Project

You will need these files:

```
~cs61a/lib/obj.scm        object-oriented tools
~cs61a/lib/logo.scm       various stuff Logo needs
~cs61a/lib/logo-meta.scm  modified metacircular evaluator
```

These files (or your modified versions of the last two) must be loaded into Scheme in this order; each one depends on the one before it. Much of the work has already been done for you. (The names `logo-eval` and `logo-apply` are used so as not to conflict with Scheme's built-in eval and apply functions.)

For reference, `~cs61a/lib/mceval.scm` is the metacircular evaluator without the modifications for Logo.

Start by examining `logo-eval`. It has two parts: `eval-prefix`, which is comparable to the version of `eval` in the text, handles expressions with prefix operations similar to Scheme's syntax. The result of evaluating such an expression is then given as an argument to `handle-infix`, which checks for a possible infix operator following the expression. For now, we'll ignore `handle-infix`, which you'll write later in the project, and concentrate on `eval-prefix`. Compare it with the version of `eval` in the text. The Scheme version has a `cond` clause for each of several special forms. (And the real Scheme has even more special forms than the version in the book.) Logo has only one special form, used for procedure definition, but to make up for it, `eval-prefix` has a lot of complexity concerning parentheses. An ordinary application (handled by the else clause) is somewhat more complicated than in Scheme because we must figure out the number of arguments required before we can collect the arguments. Finally, an important subtle point is that the Logo version uses `let` in several places to enforce a left-to-right order of evaluation. In Scheme the order of evaluation is unspecified, but in Logo we don't know where the second argument expression starts, for example, until we've finished collecting and evaluating the first argument expression.

PART I.

There are four problems: one per person, two together. The ones done separately must be completed before you'll be able to run the Logo interpreter at all. It's important that you understand each other's parts because you will be building off them in the future.

PERSON A:

A1. As explained above, `eval` can't be given a single complete expression as its argument, because expressions need not be delimited by parentheses and so it's hard to tell where an expression ends. Instead, `eval` must read through the line, one element at a time, to figure out how to group things. `logo-read`, you'll recall, gives us a Logo instruction line in the form of a list. Each element of the list is a "token" (a symbol, a number, a punctuation character, etc.) and `eval` reads them one by one. You might imagine that `eval` would accept this list as its argument and would get to the next token by `cdr`ing down, like this:

```
(define (eval-prefix line-list env)
  ...
  (let ((token (car line-list)))
    ...
    (set! line-list (cdr line-list))
    ...)
  ...)
```

but in fact this won't quite work because of the recursive invocation of `eval-prefix` to evaluate subexpressions.

Consider a line like:

```
print sum (product 2 3) 4
```

One invocation of eval-prefix would be given the list

```
(sum ( product 2 3 ) 4)
```

as argument. It would cheerfully `cdr` down its local line-list variable, until it got to the word `"product"`; at that point, another invocation of `eval-prefix` would be given the ENTIRE REMAINING LIST as its argument (since we don't know in advance how much of that list is part of the subexpression).

When the inner `eval-prefix` finishes, the outer one still needs to read another argument expression, but it has no way of knowing how much of the list was read by the inner one.

Our solution is to invent a `line-object` data type. This object will be used as the argument to `logo-eval`, which in turn uses it as argument to `eval-prefix`; the `line-object` will remember, in its local state, how much of the line has been read. The very same `line-object` will be the argument to the inner `eval-prefix`. When that finishes, the line object (still available to the outer invocation of `eval-prefix`) has already dispensed some tokens and knows which tokens remain to be processed.

Your job is to define the `line-object` class. It has one instantiation variable, a list containing the text of a line. Objects in the class should accept these messages:

`(ask line-obj 'empty?)`	`should return #T if there is nothing left to read in the line-list, #F if there are still tokens unread.`
`(ask line-obj 'next)`	`should return the next token waiting to be read in the line, and remove that token from the list.`
`(ask line-obj 'put-back token)`	`should stick the given token at the front of the line-list, so that the next NEXT message will return it. This is used when EVAL has to read past the end of an expression to be sure that it really is the end, but then wants to un-read the extra token.`

There are several places in `logo-meta.scm` that send these messages to the objects you'll create, so you can see examples of their use. You'll get `ask` from `obj.scm` and should use its syntax conventions.

Also write a short procedure `(make-line-obj text)` that creates and returns a line object instance with the given text. This procedure is invoked in several places within the Logo interpreter.

****When you've finished, you must combine your work with that of person B. When you've
****done that, you should be able to run the interpreter and carry out instructions involving only primitive
****procedures and constant (quoted or self-evaluating) data. (You aren't yet ready for variables, conditionals,
****or defining procedures, and you can only use prefix arithmetic operators.)

(There are some suggestions for things to test at the end of person B's problems for this week.)

PERSON B:

B1. A Logo line can contain more than one instruction:

```
? print "a print "b
a
b
?
```

This capability is important when an instruction line is given as an argument to something else:

```
? to repeat :num :instr
-> if :num=0 [stop]
-> run :instr
-> repeat :num-1 :instr
-> end
? repeat 3 [print "hi print "bye]
hi
bye
hi
bye
hi
bye
?
```

On the other hand, an instruction line used as argument to something might not contain any complete instructions at all, but rather an expression that provides a value to a larger computation:

```
? print ifelse 2=3 [5*6] [8*9]
72
?
```

In this example, when the `ifelse` procedure is invoked, it will turn the list [8*9] into an instruction line for evaluation. (Note: This example is here to explain to you why you need to handle an "instruction line" without a complete instruction. You can't actually type this into your Logo interpreter yet; you haven't invented infix notation or predicates.)

`logo-eval`'s job is to evaluate one instruction or expression and return its value. (An instruction, in which a command is applied to arguments, has no return value. In our interpreter this is indicated by `logo-eval` returning the symbol `=NO-VALUE=` instead of a value.) We need another procedure that evaluates an entire line, possibly containing several instructions, by repeatedly invoking logo-eval until either the line is empty (in which case `=NO-VALUE=` should be returned) or `logo-eval` returns a value (i.e., a value other than `=NO-VALUE=`), in which case that value should be returned. You will write this procedure, called `eval-line`, like this:

```
(define (eval-line line-obj env)
  ...)
```

You'll find several invocations of `eval-line` in the interpreter, most importantly in driver-loop where each line you type after a `?` prompt is evaluated.

****When you've finished, you must combine your work with that of person A. When you've done that, you
****should be able to run the interpreter and carry out instructions involving only primitive procedures and
****constant (quoted or self-evaluating) data. (You aren't yet ready for variables, conditionals, or defining
****procedures, and you can only use prefix arithmetic operators.)

TESTING: PART I

Try these examples and others:

```
? print 3
3
? print sum product 3 4 8
20
? print [this is [a nested] list]
this is [a nested] list
? print 3 print 4
3
4
? print equalp 4 6
false
?
```

COMMON EXERCISES: PART I

2. Ordinarily, each Logo procedure accepts a certain fixed number of arguments. There are two exceptions to this rule. First, some primitive procedures (but only primitives) can accept variable numbers of arguments, just as in Scheme. In Logo, such a procedure has a "default" number of arguments -- this is the number that `logo-eval` will ordinarily look for. If you want to use a different number of arguments, you must enclose the entire expression in parentheses as you would in Scheme:

```
? print sum 2 3
5
? print sum 2 3 4               ; this is an error
5
You don't say what to do with 4
? print (sum 2 3 4)
9
?
```

Second, certain primitive procedures need access to the current environment in order to do their job. For example, `make`, which is Logo's equivalent to `set!`, takes two arguments, a variable name and a new value, but the procedure that implements it requires a third argument, the current environment, since the job is done by modifying that environment. In the Scheme metacircular evaluator, this problem is less noticeable because `set!` is a special form anyway -- its first argument isn't evaluated -- and so it is handled directly by `eval` itself. In Logo we have no special forms except for `to`, so `make` is an ordinary procedure handled by `logo-apply`, but we still need to indicate that it needs the environment as an extra "hidden" argument.

In this interpreter a procedure is represented as a four-element list:

```
(name type-flag arg-count text)
```

- `name` is the procedure's name. (Unlike Scheme's first-class procedures which can be created by `lambda` without a name, every Logo procedure must have a name in order to exist at all.)
- `type-flag` is a symbol, either `primitive` or `compound`. The former means that the procedure is written in Scheme (or is a Scheme primitive); the latter means that the procedure was defined in Logo, using `to`.
- `arg-count` is the number of arguments that the procedure expects. For most procedures, this is a straightforward nonnegative integer. In this part of the project, we are going to deal with the exceptions discussed above. *For a procedure that accepts variable numbers of arguments, ARG-COUNT will be a negative integer,* the negative of the default number of arguments. For a procedure that requires the environment as an extra argument, ARG-COUNT will be a list whose only element is the number of visible arguments, before the environment is added. (No procedure is in both categories.)

  Examples:

  ```
  (list 'type 'primitive 1 logo-type)    ;ordinary case
  (list 'word 'primitive -2 word)        ;variable # of args
  (list 'make 'primitive '(2) make)      ;2 visible args plus env
  ```

  These lists are generated by the `add-prim` procedure that you can see in `logo-meta.scm` along with entries for all the existing primitives.
- `text` is either a Scheme procedure, for a primitive, or a list whose first element is a list of formal parameters and whose remaining elements are instruction lines making up the body of the procedure, for a user-defined Logo procedure.

The actual collection of argument values, corresponding to `list-of-values` in the metacircular evaluator, is called `collect-n-args` in the Logo interpreter. It has an extra argument, `n`, which is the number of arguments to be collected from the `line-object`. If that argument is negative, then `collect-n-args` will keep evaluating argument expressions until it sees a right parenthesis. (Remember that we allow a variable number of arguments only if the expression is in parentheses.)

Your job is to modify the **invocation** of `collect-n-args` to handle both of the special cases described here.

- If the `arg-count` in the procedure is a list, call `collect-n-args` with its `car` as the number, and `cons` the current environment onto the front of the resulting argument list.
- If the `arg-count` is negative, you should use its absolute value as the number unless this invocation is inside parentheses. (There is a local variable paren-flag that will be #T in this situation, #F otherwise.)
  The function `abs` takes the absolute value of its input argument

[**Important Note:** You will be modifying the *invocation* of `collect-n-args`. You do not have to modify the *definition* of `collect-n-args` at all] Once you've done this, **modify the primitive table** entries for sum, product, word, sentence, and list so that they can accept variable numbers of arguments.

**** Then test your work:

```
? ifelse equalp 2 3 [print "yes] [print "no]
? ifelse equalp 3 3 [print "yes] [print "no]
? print (sum 4 5 6 7 8)
? print (word "a "b "c)
? print (sum 4 5 product 6 7 8)
```

3. We are going to invent variables. Most of the work has already been done, because the environment structure is exactly like that of the Scheme metacircular evaluator. There are two things left for you to handle: First, `eval-prefix` uses data abstraction procedures `variable?` And `variable-name` to detect and process a variable reference. In Scheme, any symbol is a variable reference, since procedure names are variables too. In Logo, a variable reference is a **symbol** whose first character is a colon (`:`) and the actual variable name is all but the first character of that symbol. First, write these aforementioned procedures.

Second, Scheme provides two different special forms, `define` and `set!`, for creating a new variable binding and for changing an existing binding. In Logo there is one procedure, `make`, that serves both purposes. If there is already a binding for the given name in the current environment, then `make` acts like `set!`. If not, then `make` creates a new binding in the global environment. (Note, this is not necessarily the current frame.) Make the `make` procedure in `logo.scm` call the right `logo-meta.scm` procedures to accomplish this, **modifying those procedures if necessary.**

****Test your work:

```
? make "foo 27
? print :foo
27
?
```

(Why the quotation mark? Remember, `make` isn't a special form. The *value of* its first actual argument expression has to be the name we want to bind.)

Note: You can't fully test this yet, because you won't know if it does the right thing for local variables until we can define and invoke procedures. For now, just test that it works for global variables.

PART II.

There are five problems: one per person and three common ones. When you're done with them, the Logo interpreter will be complete. The common problems are hard, so don't wait until the last minute to merge your work!

PERSON A:

A4. Infix arithmetic. `logo-eval` calls `eval-prefix` to find a Scheme-style expression and evaluate it. Then it calls

```
(handle-infix value line-obj env)
```

We have provided a "stub" version of `handle-infix` that doesn't actually handle infix, but merely returns the value you give it. Your task is to write a version that really works. The situation is this. We are dealing with the instruction line

```
? print 3 + 2
```

We are inside the `logo-eval` that's preparing to invoke `print`. It knows that `print` requires one argument, so it recursively called `logo-eval`. (Actually `logo-eval` calls `eval-prefix`, which calls `collect-n-args`, which calls `logo-eval`.) The inner `logo-eval` called `eval-prefix`, which found the expression 3, whose value is 3. But the argument to `print` isn't really just 3; it's 3 + 2.

The job of `handle-infix` is to notice that the next token on the line is an infix operator (one of `+ - * / = < >`), find the corresponding procedure, and apply it to the already-found value (in this case, 3) and the value of the expression after the infix operator (in this case, 2). Remember that this following expression need not be a single token; you have to evaluate it using `eval-prefix`. If the next token isn't an infix operator, you must put it back into the line and just return the already-found value. Remember that there may be another infix operator after you deal with the first one, as in the instruction

```
? print 3 * 4 + 5
17
```

We've provided a procedure called `de-infix` that takes an infix operator as argument and returns the name of the corresponding Logo primitive procedure.

To further your understanding of this problem, answer the following question: What difference would it make if your `handle-infix` invoked `logo-eval` instead of `eval-prefix` as suggested above? Think of an example instruction for which this change would give a different result.

By the way, don't forget that **we are not asking you to handle the precedence of multiplication over addition correctly**. Your `handle-infix` will do all infix operations from **left to right**, unless parentheses are used. (You don't have to deal with parentheses in `handle-infix`. `logo-eval` already knows about them.)

****Now skip over person B's problem to get to the common problems 5, 6, and 7. You must merge
****the results of A4 and B4 before you can solve the common problems.

PERSON B:

B4. Time to define procedures! You are going to write `eval-definition`, a procedure that accepts a `line-obj` as argument. (The corresponding feature in the metacircular evaluator also needs the environment as an argument, but recall that in Logo procedures are not part of the environment; they go in a separate, global list.) The `line-obj` has just given out the token `to`, and is ready to provide the procedure name and formal arameters.

Your job is to read those, then enter into an interactive loop in which you read Logo lines and store them in a list, the procedure body. You keep doing this until you see a line that contains only the word `end`. You put together a procedure representation in the form

```
(list name 'compound arg-count (cons formals body))
```

and you prepend this to the procedure list in the (Scheme) variable `the-procedures`.

- The `arg-count` is the number of formal parameters you found.
- `formals` is a list of the formal parameters, without the colons.
- `body` is the list of instruction lines, not including the `end` line.

Do not turn the lines into `line-objects`; that'll happen when the procedure is invoked.

To print the prompt, say `(prompt "-> ")`.

It's going to be a little hard to test the results of your work until you can invoke user-defined procedures, which requires one more step. Meanwhile you could leave Logo, and ask Scheme to look at the first element of `the-procedures` to see if you've done it right.

****You must merge the results of A4 and B4 before you can solve the common problems 5, 6, and 7.

COMMON EXERCISES: PART II

5. Evaluating procedure bodies. In the metacircular evaluator, `apply` sets up an environment and uses `eval-sequence` to evaluate each expression in the procedure body. The Logo interpreter does the same, except that the job of `eval-sequence` is different. Its argument is a list of instruction lists. Each of those lists must be turned into a `line-object` before it can be evaluated. Also, we must take into account the fact that instructions are different from expressions; the instruction lines in the procedure body should generally return `=NO-VALUE=` when evaluated. If not, `eval-sequence` must signal the error "You don't say what to do with" the value.

The exceptions are the two primitive commands that can end a procedure invocation early, `stop` (for commands) and `output` (for operations). If `stop` is invoked, it will return the symbol `=STOP=`; if `output` is invoked, it will return a pair whose car is `=OUTPUT=` and whose `cdr` is the desired return value:

```
(add-prim 'stop 0 (lambda () '=stop=))
(add-prim 'output 1 (lambda (x) (cons '=output= x)))
```

If `eval-sequence` evaluates a `stop`, it should immediately return `=NO-VALUE=`. If it gets an `output`, it should immediately return the value provided (the cdr of that pair).

6. Dynamic scope. In the metacircular evaluator, mc-apply handles compound (user-defined) procedures by setting up an environment and evaluating the procedure body (using `eval-sequence`) in that environment. `logo-apply` must do the same thing, but instead of Scheme's lexical scope, in which the new environment extends the one in which the procedure was created, it must follow Logo's dynamic scope, in which the new environment extends the current environment.

The version of `logo-apply` we give you doesn't handle compound procedure calls. Modify it as needed, along with **any** other changes required to go along with this one. (Hint: Start by looking at the `mc-apply` version.)

****Once you've solved these problems, you should be able to define and invoke procedures:

```
? make "x 3
? to scope :x
> helper 5
> end
? to helper :y
> print (sentence :x :y)
> end
? scope 4
```

7. 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 variable 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
25
*** 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
```

[Hint: thing `eq?`]

****The interpreter is now complete. Congratulations!

Miscellaneous

- We need to be able to print the results of Logo computations. Logo provides three primitive procedures for this purpose:

  ```
  ? print [a [b c] d]      ; don't show outermost brackets
  a [b c] d
  ? show [a [b c] d]       ; do show outermost brackets
  [a [b c] d]
  ? type [a [b c] d]       ; don't start new line after printing
  a [b c] d?
  ```

  Normally, you are asked to write these functions, but we have provided them for you.

- The Logo primitives `if` and `ifelse` require as argument the word `true` or the word `false`. Of course our implementation of the Logo predicates will use Scheme predicates, which return `#t` and `#f` instead.

  Thus, we'd like a higher-order function `logo-pred` whose argument is a Scheme predicate and whose return value is a Logo predicate, i.e., a function that returns `true` instead of `#t` and `false` instead of `#f`. This higher-order function is used in the table of primitives like this:

  ```
  (add-prim 'emptyp 1 (logo-pred empty?))
  ```

  That is, the Scheme predicate `empty?` becomes the Logo predicate `emptyp`. (The "P" at the end of the name stands for "Predicate," by the way. Some versions of Logo use this convention, while others use ? at the end the way Scheme does. The educational merits of the two conventions are hotly debated in the Logo community.)

  The spiffiest way to do this is to create a logo-read that works for predicate functions of any number of arguments. To do that you have to know how to create a Scheme function that accepts any number of arguments. You do it with (lambda args (blah blah)). That is, the formal parameter name `args` is not enclosed in parentheses. When the procedure is called, it will accept any number of actual arguments and they will all be put in a list to which `args` is bound. (This is discussed in exercise 2.20.)

  Again, you are normally asked to write these functions, but we have provided it for you as well.

Project Submission

This project is due at 3:00 am on Sunday, August 14.

You need to turn in the following files:

- `logo.scm` – with all your changes
- `logo-meta.scm` – with all your changes
- `testing.txt` – contains testing of all your code – in isolation as well as running the entire interpreter

Please clearly indicate which person is Partner A and which person is Partner B.

You must also turn in a paper copy to 283 Soda. On the paper copy, please **HIGHLIGHT all changes** you made so that we can easily find and grade your work.