Background

**Code to load for this project:** A link to the system code files eval_ft_06.scm is provided from the Projects link on the course web page.

As usual, you should begin working on the assignment once you receive it. It is to your advantage to get work done early, rather than waiting until the night before it is due. You should also read over and think through each part of the assignment (as well as any project code) before you sit down at the computer. It is generally much more efficient to test, debug, and run a program that you have thought about beforehand, rather than doing the planning "online." Diving into program development without a clear idea of what you plan to do generally guarantees that the assignments will take much longer than necessary.

The purpose of this project is to familiarize you with evaluators. We recommend that you first skim through the project to familiarize yourself with the format, before tackling problems.

Word to the wise: This project doesn’t require a lot of actual programming. It does require understanding a body of code, however, and thus it will require careful preparation. You will be working with evaluators such as those described in chapter 4 of the text book, and variations on those evaluators. If you don’t have a good understanding of how the evaluator is structured, it is very easy to become confused between the programs that the evaluator is interpreting, and the procedures that implement the evaluator itself. For this project, therefore, we suggest that you do some careful preparation. Once you’ve done this, your work in the lab should be fairly straightforward.

Understanding the evaluator

Load the code for this project. This file has three parts, and contains a version of the meta-circular evaluator similar to that described in lecture (there are a few minor differences) and in the textbook. The first part defines the syntax of the evaluator, the second part defines the actual evaluator, and the third part handles the environment structures used by the evaluator. Because this evaluator is built on top of the underlying Scheme evaluator, we have called the procedure that executes evaluation m-eval (with associated m-apply) to distinguish it from the normal eval.

You should look through these files to get a sense for how they implement a version of the evaluator discussed in lecture (especially the procedure m-eval).
You will be both adding code to the evaluator, and using the evaluator. Be careful, because it is easy to get confused. Here are some things to keep in mind:

When adding code to be used as part of eval-f06.scm, you are writing in Scheme, and can use any and all of the procedures of Scheme. Changes you make to the evaluator are changes in defining the behavior you want your new evaluator to have.

After loading the evaluator (i.e., loading the file eval-f06.scm and any additions or modifications you make), you start it by typing `(driver-loop)`. In order to help you avoid confusion, we've arranged that each driver loop will print prompts on input and output to identify which evaluator you are typing at. For example,

```scheme
;;; M-Eval input:
(+ 3 4)

;;; M-Eval value:
7
```

shows an interaction with the m-eval evaluator. To evaluate an expression, you type the expression and hit Enter. Note that DrScheme provides you with an input box into which you can type your expressions.

The evaluator with which you are working does not include an error system. If you hit an error you will bounce back into ordinary Scheme. You can restart the driver-loop by running the procedure `(driver-loop)`. Note that the driver loop does not re-initialize the environment, so any definitions you have made should still be available if you have to re-run driver-loop. In case you do want a clean environment, you should evaluate `(refresh-global-environment)` while in normal Scheme.

To quit out of the new evaluator, simply evaluate the expression `**quit**`. This will return you to the underlying Scheme evaluator and environment.

**Computer Exercise 1: Exploring Meval**

Load the code files into your Scheme environment. To begin using the interpreter defined by this file, evaluate `(driver-loop)`. Notice how it now gives you a new prompt, in order to alert you to the fact that you are now "talking" to this new interpreter. Try evaluating some simple expressions in this interpreter.

You will probably very quickly notice that some of the primitive procedures about which Scheme normally knows are missing in m-eval. These include some simple arithmetic procedures (such as *) and procedures such as `cadr`, `cddr`, `newline`, `length`. Extend your evaluator by adding these new primitive procedures (and any others that you think might be useful). Check through the code to figure out where this is done. In order to make these changes visible in the evaluator, you'll need to rebuild the global environment:
or you will need to re-evaluate your file and start fresh (which is probably the less confusing option). Show your changes to the evaluator, and turn in a demonstration of your extended evaluator working correctly.

Adding new special forms to Scheme

Computer Exercise 2: Changing style

In our standard evaluator, if we define a variable with a value for a second time, we lose the previous value of the variable. Similarly, if we set! a variable, we lose the previous value. We would like to change this behavior, so that we keep track of all previous bindings of a variable. This would allow us to introduce a new special form, reset!. Evaluating (reset! var) in some environment should have the following behavior:

- if the variable var had been previously defined, its value is reset to the most recent prior value,
- if the variable var had only been defined once, then this evaluation results in an error,
- reset! can be used multiple times, that is, evaluating (reset! var) (reset! var) will “undo” the two previous bindings for var (assuming that it had been set more than twice).

To implement this change in our evaluator, we need to do several things:

- Change bindings of variables to associate a list of values with a variable, rather than a single value (note that you can do this by simply changing set-binding-value!).
- Add a new special form called reset!. Evaluating this special form should change the binding of the variable to include the new value in the list of bindings that have been associated with the variable. (You may find lookup-variable-value to be a useful template for this change.) Be sure to think about where a special form should go in m-eval as well as creating a syntactic abstraction to handle reset! expressions.

Implement this change and demonstrate your new evaluator showing this new behavior.

Computer Exercise 3: Adding a new special form

We have seen that our evaluator treats any compound expression as an application of a procedure, unless that expression is a special form that has been explicitly defined to obey a different set of rules of evaluation (e.g., define, lambda, if). We are going to add a special form to our evaluator in the next exercise.

A common construct in other languages is a loop. Our version of a loop has the following syntax:
(loop <until> <return>
  <exp1>
  <exp2>
  ...
  <expN>)

The behavior is as follows: Each of the exp1 ... expN expressions is evaluated in order. Then the until expression is evaluated. If the until expression evaluates to #t then we stop executing the loop and return the value obtained by evaluating the return expression. Otherwise we evaluate the expressions exp1 through expN again, and repeat the whole process. In short, a loop repeatedly evaluates exp1, exp2, ... expN until the expression until evaluates to #f, whereupon the entire loop expression evaluates to the value of the return expression.

Here's an example:

(let ((x '()))
  (loop (> (length x) 3) x
    (set! x (cons 'x x))
    (display x))

(*)(* *)(* *)(* * *)(* * *)

; M-eval value: (* * *)

Your task is to add this special form to m-eval, showing your changes to the code, and demonstrating that it works by providing some test cases.

To do this, you should include the following:

- Create a data abstraction for handling loops, that is, selectors for getting out the parts of the loop that will be needed in the evaluation.
- Add a dispatch clause to the correct part of m-eval.
- Write the procedure(s) that handle the actual evaluation of the loop.

Be sure to turn in a listing of your additions and changes, as well as examples of your code working on test cases.

Transformers: More than meets the eye

In some cases, it is easy to think of implementing a special form in terms of more primitive expressions that are already covered by the evaluator. As an example, to evaluate cond expressions, m-eval uses the following code in the dispatcher:
((cond? exp) (m-eval (cond->if exp) env))

together with the following method for converting a cond statement to an if statement:

(define (cond->if expr)
 (let ((clauses (cond-clauses expr)))
  (if (null? clauses)
    (if (eq? (car (first-cond-clause clauses)) 'else)
        (make-begin (cdr (first-cond-clause clauses)))
        (make-if (car (first-cond-clause clauses))
           (make-begin (cdr (first-cond-clause clauses)))
           (make-cond (rest-cond-clauses clauses))))))

What would the following expression evaluate to?:

(cond->if '(cond ((= x 0) 5)
               ((> x 0) 10)
               (else 15)))

Computer Exercise 4: Transforming boolean combinations

In scheme, and and or are special forms because all of the arguments to these forms are not necessarily evaluated. The following example exhibits this behaviour:

(define (safe-list-ref lst n)
  (if (and (integer? n) (list? lst) (>= n 0) (< n (length lst)))
      (list-ref lst n)
      'invalid-list-reference))

So, for example, if (list? 1st) is false, then the expression (length 1st) is not evaluated, which is good. This is called 'short-circuiting:' as soon as an argument to and evaluates to false we return true, without evaluating the remaining arguments; or behaves almost the same way. You already implemented this behavior in the problem set, but there’s one more feature we want you to add.

The result of an or special form is not necessarily #t or #f, but rather the value of the first non-#f argument. (For example, (or #f 3 4) => 3.) Likewise, if an and special form has no arguments which evaluate to #f, then its return value is the value of its last argument. (For example, (and 2 3 4) => 4.)

As before, add a pair of clauses to m-eval that transform or and and expressions into expressions that m-eval already knows how to evaluate. Remember that the individual expressions in the body of or are supposed to be evaluated at most once:
(or (begin (display "once ") #f)
    (begin (display "and ") #f)
    (begin (display "only ") 'done)
    (begin (display "adbmal") #t))

once and only
;Value: done

(and (begin (display "a ") (> 6 5))
    (begin (display "b ") (< 6 5))
    (begin (display "c ") (+ 6 5)))
a b
;Value: #f

(One way to desugar or involves using let and if. In this case there can be "name capture" errors if the let variable already appears in the or expression being desugared. For this problem, it is okay to assume that any variables you use in desugaring do not already appear. Note that there is a better way to desugar and and or that avoids this problem.)

**Computer Exercise 5: Implementing loop as syntactic sugar**

Show how to implement loop using a similar strategy to that for cond, i.e., convert a loop expression into an associated if expression. You should have the following code in the dispatcher:

```
((loop? exp) (m-eval (loop->if exp) env))
```

together with a definition for loop->if (hint: loop->if may return an expression which includes a loop expression as one of its sub-expressions).

**Implementing an Evaluator with Memoization**

A few times in the course we’ve seen the idea of memoization. This involves using some kind of data structure to store previously computed values of a procedure application.

In this part of the project we’re going to extend the evaluator to allow automatic implementation of memoization. In particular, we are going to extend the syntax of lambda expressions so that they allow us to specify if we want to use memoization with the compound procedure created by the lambda expression.

The new syntax will be as follows:

```
(lamda <memo?> <parameters> <body>)
```
The `<parameters>` and `<body>` are the same as in regular lambda expressions. `<memo?>` is either `memo` or `no-memo`, and specifies whether or not memoization is desired with this lambda expression.

As an example, we might do the following:

```scheme
(define myplus (lambda memo (x y) (+ x y)))

(myplus 3 4) ;-> return value is 7
(myplus 3 4) ;-> return value is 7
```

In the first call to `myplus` with the arguments 3 and 4, the procedure operates in its usual way. In the second call, however, the evaluator realizes that the arguments 3 and 4 have been used before with this procedure, and the value 7 is retrieved from a data structure that stores the result of previous computations with `myplus`.

On the other hand, if we do the following:

```scheme
(define myplus2 (lambda no-memo (x y) (+ x y)))

(myplus2 3 4) ;-> return value is 7
(myplus2 3 4) ;-> return value is 7
```

then memoization is not used (because `no-memo` was specified in the original lambda expression), and in both cases the value 7 has to be computed from scratch.

Notice that this use of memoization can be very useful in some cases, take for example the following definition of fibonacci:

```scheme
(define fib (lambda memo (n)
    (cond ((= n 1) 1)
          ((= n 2) 1)
          (else (+ (fib (- n 1)) (fib (- n 2)))))))
```

This will immediately give an implementation of fibonacci with memoization; this procedure should run in $\Theta(n)$ time, rather than the exponential time required for a version without memoization.

As one additional point, note that we'll have to alter the syntax for `define` statements as well. In regular scheme, we can define procedures in the following way:

```scheme
(define (myplus x y) (+ x y))
```

In the new version with memoization, in this case we could use either

```scheme
(define (myplus memo x y) (+ x y))
```
or

\[
\text{(define (myplus no-memo x y) (+ x y))}
\]
to define versions of myplus with or without memoization.

To implement memoization, a crucial idea will be the following: a \text{lambda} expression evaluates to a triple bubble \textit{in the new evaluator}. The first two bubbles are the same as in regular scheme: the first bubble points to the parameters and body of the lambda expression, the second bubble points to the environment in which the lambda expression was evaluated. The third bubble points to a data structure if the \text{lambda} expression had the \text{memo} option, or the symbol \text{none} if it had the \text{no-memo} option. Initially the data structure is empty.

You will need to implement a data structure to store information about previous calls to a procedure. One easy way to do this is to use a list, where each entry in the list is itself a list consisting of the values supplied to the procedure and the associated value. For example, if \text{fib} had been called with the arguments 4, 3, 2 and 1, we \textbf{might} choose to represent that information as:

\[
(((5) 5) ((4) 3) ((3) 2) ((2) 1) ((1) 1))
\]

Note that the strange form of this list is because a procedure might take more than one argument, so we are capturing in each element a list of the list of arguments and the value.

When using \text{m-apply}, for compound procedures a triple bubble will be applied to a list of arguments. An application of a triple bubble involves the following steps:

- If the third bubble points to \text{none}, then apply the compound procedure in the usual way (no memoization is needed).

- Otherwise, the third bubble points to a memoization data structure. In this case, first check to see if the \text{arguments} are stored in the data structure with some value. If so, return that value as the value of the application. Otherwise, evaluate the compound procedure in the usual way, \textit{but make sure to store an entry in the data structure that associates the arguments with the value that is computed, before returning that value}.

\textbf{Computer Exercise 6: Adding memoization to the evaluator}

To implement this style of memoization in the mc-evaluator, you’ll need to carry out the following steps:

- Change and add to the code defining the syntax of \text{lambda} expressions, to take into account the modification that allows the additional \text{memo} or \text{no-memo} syntax. (Note that you should change \text{make-lambda} to take an additional argument, \text{memo}?, and that the call to \text{make-lambda} within \text{let->application} should use the \text{no-memo} option.)

- Change the code for the \text{define} syntax to take into account the additional \text{memo} or \text{no-memo} option.
• Create a data structure for storing memoization information about application of a procedure – what values of arguments has it been called on, and what was the resulting value.

• Change the code for make-procedure so that it takes an additional argument that is either memo or no-memo. Add a function procedure-mlist which returns a memoization data structure from the third bubble if the memo option was used, or the symbol none otherwise. Change the call to make-procedure in the dispatcher.

• Change m-apply to implement memoization. See the description above for how this should proceed.

Be sure to turn in a listing of your additions and changes, as well as examples of your code working on test cases.

Finally, note that code with and without memoization may give different values, in particular when mutation or side-effects are used. Give three examples of code that gives different behavior (i.e., different return values, or different output to the terminal) when the memo and no-memo options are used.

Submission

For each problem, include your code (with identification of the exercise number being solved), as well as comments and explanations of your code, and demonstrate your code's functionality against a set of test cases. Once you have completed this project, your file should be submitted electronically on the 6.001 on-line tutor, using the Submit Project Files button.

We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout) and so long as you observe the rules on collaboration and using “bibles”. If you cooperated with other students, LA’s, or others, please indicate your consultants’ names and how they collaborated. Be sure that your actions are consistent with the posted course policy on collaboration.

Remember that this is Project 5; when you have completed all your work and saved it in a file, upload that file and submit it for Project 5.