

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001 Structure and Interpretation of Computer Programs
Spring Semester, 2005

Project 5 - The Meta-Circular Evaluator

- Issued Monday, April 25th.
- To Be Completed By: Friday, May 6th, 6:00 pm
- Code to load for this project:
 - Links to the system code files `meval.scm`, `syntax.scm`, and `environment.scm` are provided from the Projects link on the course web page.

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 ensures that the assignments will take much longer than necessary.

The purpose of this project is to familiarize you with evaluators. There is a fair amount to read and understand; 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 (`meval.scm`, then `syntax.scm`, finally `environment.scm`) for this project. These files contain a version of the meta-circular evaluator described in

lecture and in the textbook. 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 `meval.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 `meval.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,

```
;;; 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 press `ctrl-x ctrl-e`. Don't use `M-z` to evaluate the expression; the presence of the prompt confuses the `M-z` mechanism. Also notice that if you have started up the evaluator in a `*scheme*` buffer, you may go to any other buffer, write definitions or expressions and evaluate them from that buffer. This will cause the expressions to be evaluated in the **new** evaluator. On the other hand, if you have not started up the evaluator, evaluating expressions from a buffer will cause them to be evaluated in the normal Scheme evaluator.

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.

It is instructive to run the interpreter while tracing `m-eval`. (You will also probably need to do this while debugging your code for this assignment.)

Since environments are, in general, complex circular list structures, we have set Scheme's printer so that it will not go into an infinite loop when asked to print a circular list. (See the descriptions of `*unparser-list-depth-limit*` and `*unparser-list-breadth-limit*` in the Scheme reference manual.)

You will notice that when you have run `driver-loop`, the mode line (bar at the bottom of edwin with the buffer name, along with `(Scheme: listen)`) will display `(Scheme Meval: listen)` instead. This indicates that `meval` will be receiving anything you `C-x C-e`, instead of (regular) MITScheme.

In order to give you further warning when you accidentally send things to Meval instead of MITScheme, the `*warn-meval-define*` variable controls whether Meval will print out warnings if the `define` you're evaluating is binding a variable that is also bound in MITScheme.

Computer Exercise 1: Exploring `meval`.

Load the code files into your Scheme environment. To begin using the Scheme 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. **Note that this interpreter has no debugging mechanism** -- that is, if you hit an error, you will be thrown into the debugger for the underlying Scheme. This can be valuable in helping you to debug your code for the new interpreter, but you will need to restart the interpreter.

You may 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`. 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:

```
(refresh-global-environment)
```

Show your changes to the evaluator, and turn in a demonstration of your extended evaluator working correctly.

Computer Exercise 2: Changing style.

In MITScheme, `if` expressions are allowed to lack an *alternative* expression, as in the following:

```
(define x 5)
(if (= x 0)
    (set! x 0.000001))
```

At present, what happens if you evaluate this expression in `m-eval`?

Modify the evaluator so that `ifs` may lack an *alternative*, but if they are lacking the alternative and the *test* evaluates to false, then the value of the `if` should be `#f`.

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 3: Adding a 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 some special forms to our evaluator in the next few exercises.

A common loop construct in other languages is the `do...while` loop. It has the following syntax:

```
(do exp1
    exp2
    ...
    expN
    while test)
```

The behavior is as follows: `exp1` through `expN` are evaluated in sequence. Then `test` is evaluated. If `test` evaluates to `#f` then we stop executing the loop and return the symbol `done`; otherwise the loop is repeated from the start. For example,

```
(let ((x '()))
  (do (set! x (cons '* x))

      (write-line x)
      while (< (length x) 3))) ;evaluate this to get
(*)
(* *)
(* * *)
; Value: done
```

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.

Caveat: avoid mutation on the expression that you are desugaring; for example, when the expression which you are passed is the code pointer of a compound expression, mutating the expression will actually change what the procedure does.

To do this, you should include the following:

- Create a data abstraction for handling `do...while` 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 right part of `m-eval`. We have actually included this clause, you simply need to uncomment it.
- Write the procedure(s) that handle the actual evaluation of the `do...while` loop.

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

Computer Exercise 4: The `let*` special form.

Now let's try something a bit tougher. We have already seen the `let` special form. A variation on this is the `let*` special form. This form acts much like `let`, however, the clauses of the `let*` are evaluated in order (rather than all at the same time as in a `let`), and any reference in a later clause to the variable of an earlier clause should use the new value of that variable. For example

```
(define i 1)
(let ((i 3)
      (j (factorial i)))
  (list i j))
```

would return the value `(3 1)` because the `let` variables `i`, `j` are bound in parallel, and thus the argument to `fact` is the value of `i` outside the expression, namely 1. On the other hand:

```
(define i 1)
(let* ((i 3)
       (j (factorial i)))
  (list i j))
```

would return the value `(3 6)` since the variable `i` is first bound to 3, and then the variable `j` is bound, and in this case the input to `fact` is the new value 3.

Add `let*` to the evaluator as a special form. To do this, you will need to

- create some syntax procedures to extract out the parts of a `let*` expression
- add a dispatch clause to `m-eval` to handle `let*` expressions
- create procedure(s) to handle the evaluation of the `let*`. We suggest that you use the following idea. If the `let*` has no clauses, then you simply need to evaluate the body in the current environment. Otherwise you need to create a new environment, extending the current one, in which the variable of the first clause is bound to the value (obtained by evaluation) of the corresponding part of the first clause. Within that environment, you need to continue the process, either binding the next variable to its value, or if there are not more variables, evaluating the body.

Add this special form to your evaluator. Turn in a listing of your changes, and examples of your procedures working on test cases.

Computer Exercise 5:

As a final exercise in adding special forms directly to the evaluator, we would like to add a way for `set!`s to be undone. To do this, we will need a new special form, `unset!` which is used as follows:

```
(define x 4)
x          ; 4
(set! x 5)
x          ; 5
(set! x 6)
x          ; 6
(unset! x)
x          ; 5
(unset! x)
x          ; 4
(unset! x)
x          ; 4
```

You'll notice that any number of `set!`s can be undone, and that `unset!`ing a variable that hasn't been set does nothing.

- create some syntax procedures to extract out the parts of a `unset!` expression
- add a dispatch clause to `m-eval` to handle `unset!` expressions
- modify the binding abstraction to save old values when mutations occur and support the `unset-binding-value!` operation.

- write `eval-unset!`, which undoes the last `set!` to the variable. It returns an unspecified value (i.e. whatever you want it to). If the variable is unbound, it should generate an unbound variable error.

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. For example, suppose we have a simple special form, called `until` with the following behavior. An `until` expression contains a sequence of subexpressions, the first of which is a test expression. The idea is that one first evaluates the test expression – if it is true, we are done, and can simply return some value, such as the symbol `done`. If the test expression is not true, then we evaluate each of the remaining expressions in order and repeat the process. One way to implement an `until` is to rewrite it in terms of more primitive Scheme expressions. For instance,

```
(until (> x n)
      (write-line x)
      (set! x (+ x 1)))
```

can be rewritten as

```
(let ()
  (define (loop)
    (if (> x n)
        'done
        (begin (write-line x)
                (set! x (+ x 1))
                (loop))))
  (loop))
```

and similarly

```
(until test
      exp1
      exp2
      ...
      expn)
```

can be rewritten as

```
(let ()
  (define (loop)
    (if test
        'done
        (begin exp1
                exp2)))
  (loop))
```

```

      ...
      expn
      (loop)))
(loop))

```

where *test*, *exp1*, *exp2*, ..., *expn* can be arbitrary expressions.

Hence one way to implement the transformation is as a procedure, which takes the `until` expression and returns the transformed expression.

```

(define (until->transformed exp)
  (let ((test (cadr exp))
        (other-exps (caddr exp)))
    (list
     'let
     '()
     (list 'define
           '(loop)
           (list 'if
                 test
                 ''done
                 (cons 'begin
                       (append other-exps (list '(loop))))))
     '(loop))))

```

This is the direct way of creating the transform. You should check it through carefully to see how the use of `list` and `quote` will create an appropriate expression. This form is also a bit cumbersome, because we have to do so much direct list manipulation.

Optional Material: Quasiquote, Unquote, Unquote-splicing

Here is an alternative method called `quasiquote`. Note that this is much hairier - - don't feel you have to use this approach as the version above is perfectly fine! However, you can check out the Scheme reference manual for more details on `quasiquote`.

```

(define (until->transformed exp)
  (let ((test (cadr exp))
        (other-exps (caddr exp)))
    `(let ()
       (define (loop)
         (if ,test
             'done
             (begin ,@other-exps
                   (loop))))
      (loop))))

```

As this example illustrates, the special syntax characters backquote (```), comma (`,`), and at-sign (`@`) are extremely useful in writing syntactic transformations.

Placing a backquote before an expression is similar to placing an ordinary quote before the expression, except that any subexpression preceded by a comma is evaluated. If a subexpression is preceded by comma at-sign, it is evaluated (and must produce a list) and the list is appended into the result being formed. For example:

```
(define x '(1 2 3))
(define y '(4 5 6))

`(a b c ,x ,@y 7 8)
;Value: (a b c (1 2 3) 4 5 6 7 8)
```

Note that that the evaluated subexpressions of a backquote form can be actual expressions, not just symbols. Thus `until->transformed` can also be defined as

```
(define (until->transformed exp)
  `(let ()
     (define (loop)
       (if ,(cadr exp)
           'done
           (begin ,@(caddr exp)
                  (loop))))
     (loop)))
```

Adding new derived expressions to Scheme

Early in the semester, we introduced the idea of "syntactic sugar," that is, the notion that some of the special forms in our language can be expressed as simple syntactic transformations of other language elements. Examples are `cond`, which can be implemented in terms of `if`; and `let`, which can be implemented in terms of `lambda`. Such expressions are also called **derived expressions**. Your job in this part will be to design and implement a new derived expression for Scheme.

Section 4.1.2 of the textbook demonstrates how to implement `cond` as a derived expression: to evaluate a `cond` expression, we transform it to an equivalent `if` expression using the procedure `cond->if`; and then we evaluate the transformed expression. `Let` can also be implemented as a derived expression, as explained in exercise 4.6 on page 375.

Computer Exercise 6: Let's desugar for a `while`.

Implement a syntactic transformation to convert a `do...while` loop expression from Exercise 3 into a simpler expression. Then change your evaluator so that

when the special form `do...while` is recognized, the evaluator desugars this expression into the simpler form and evaluates that.

Remember that in a syntactic transformation, you are converting one list structure into another (which will subsequently be evaluated). Thus, you should start with a simple `do...while` example, and write down what the equivalent simpler expression would be. You can probably use the same syntactic selectors as in Exercise 3, although the evaluation clause in `m-eval` will be different.

Hint: one correct solution to this problem involves desugaring a `do...while` loop into an expression having `do...while` as a sub-expression.

Computer Exercise 7: 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? lst)` is false, then the expression `(length lst)` 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 PS9, 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`.) You may read about this behavior in more detail in the Scheme reference manual.

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. In Exercise 9 we give you another way to fix this problem.)

Computer Exercise 8: Desugaring `case`.

We have used the `case` expression in a number of recent projects. Remember its form:

```
(case message
  ((TYPE) `do-something-if-message-is-type)
  ((FOO) `do-something-else-if-message-is-foo)
  (else `do-something-further-if-nothing-matches)
```

Remember the behavior of a `case`. In this example, the value of `message` would be compared against the symbol `type`. If they are `eq?`, then the expressions in that clause are evaluated, and the value of the last one is returned as the value of the whole expression. If not, we move on to the next expression and repeat the process. If an expression has the keyword `else` as its first subexpression, we automatically evaluate the remaining subexpressions in this clause.

Extend `m-eval` to handle `case` expressions, by desugaring them into some other form, such as a `cond`. Be sure to create some appropriate syntax procedures for extracting parts of a `case` expression, as well as building the procedure to

convert such an expression into a `cond`. Show examples of your procedure converting `case` expressions into `cond` expressions. Also add this to your evaluator and show examples of expressions being correctly evaluated.

Computer Exercise 9: Name Capture and Static Analysis

You'll note that the implementation sketched above for the `until` special form has a flaw, namely that if we try to use a symbol named `loop` in an `until` loop, we get into trouble. For example:

```
(define loop 3.14159)
(define (many-circles n)
  (let ((x 1))
    (until (> x n)
           (display (* loop x x))
           (set! x (+ x 1)))))
```

would transform to

```
(define loop 3.14159)
(define (many-circles n)
  (let ((x 1))
    (let ()
      (define (loop)
        (if (> x n)
            'done
            (begin
              (display (* loop x x))
              (set! x (+ x 1))
              (loop))))
      (loop))))
```

which would give an error when we try to multiply `loop` as a numeric value. This general problem is called 'name capture.' One method to avoid this is to analyze the expression we are desugaring, and instead of calling the helper function `loop` we will give it a name that is not used already in the expression we're desugaring. Thus we will need a procedure `(names-used-in exp)` which will analyze an expression and give us back a list of all of the symbol names which are referenced in that expression. For example

```
(names-used-in '(until (> x n) (display (* loop x x)) (set! x (+ x 1)))) => (> x n display * loop)
```

Having analyzed the expression to determine the names that are used, we need to compute a symbol not occurring in the list of used names. Such a symbol is usually called *fresh*. Here is one way to compute a fresh symbol :

```

(define (fresh-symbol free)
; list<symbol> -> symbol
; computes a new symbol not occurring in free
  (fold-right symbol-append 'unused free))

```

The fixed code is below, with changes shown in bold:

```

(define (until->transformed exp)
  (let* ((test (cadr exp))
        (other-exps (caddr exp))
        (names-occurring (names-used-in exp))
        (loop-name (fresh-symbol names-occurring)))
    (list
     'let
     '()
     (list 'define
           (list loop-name)
           (list 'if
                 test
                 'done
                 (cons 'begin
                       (append other-exps (list '(loop)))))))
     (list loop-name))))

```

The procedure `names-used-in` computes all of the free variables in `exp`, returning a list of symbols. The procedure `fresh-symbol` then computes a new symbol not occurring in the free variables list, and that symbol is used instead of the symbol `loop`.

We want you to fill in the definition of the procedure `names-used-in`, which computes the variables used in a particular expression. A skeleton of the procedure is provided in the code, and you will need to add some extra cases.

Give some examples that directly show your implementation of `names-used-in` works. Also, come up with a test case for which one of your original desugarings (for `and`, `or`, `case`, or `do...while`) fails because of name capture. Then, change the appropriate desugaring so that it uses `names-used-in` and `fresh-symbol` to avoid name capture, and show that the problem is fixed by this change.

Exercise 10: Avoiding Name Capture (Optional Bonus Problem)

As mentioned in problem 7, there is a desugaring for `or` which completely avoids name capture. Write up a procedure to perform this desugaring, using `quote`, `quasiquote`, `unquote`, and `unquote-splicing` where appropriate. A correct solution should be short, around 6 lines of code.

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.