MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.037—Structure and Interpretation of Computer Programs
IAP 2018

Project 3

Release date: 18 January, 2018
Due date: 25 January, 2018 at 1900h

Background

Code to load for this project: A link to the system code file `eval.scm` is provided 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 SICP. 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 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 written in Scheme, 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.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.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 level 1

(+ 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 DrRacket 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 this does not re-initialize the environment, so any definitions you have made should still be available.

To quit out of the new evaluator, simply evaluate the expression `**quit**` or click the yellow "`eof`" button. This will return you to the underlying Scheme evaluator and environment.

Making Changes to the Evaluator

In this assignment, it will generally be easier to modify `eval.scm` directly, rather than writing your code in a separate file.

Most of your changes will be small, however: new procedures, or changes to small existing procedures. To make it easier for your TA to see where you changed `eval.scm`, put new code and small changed procedures at the *end* of `eval.scm`, marked prominently with the question for which the change was made, e.g. `;; ==== QUESTION 1 ====`. Put test results for each question in the same place, at the end of the file.

When you have to make a change to a large procedure, such as m-eval or the primitive procedure list, then you can make that change in-place without moving or copying the large procedure, but make sure to mark the change prominently with a comment (e.g. `;; ==== QUESTION 1 ====`).

Question 1: Exploring m-eval

Start by loading `eval.scm` 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 `list, cadr, cddr, newline, printf, 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 by pressing DrRacket's "Run" button.

Mark your changes to the evaluator with prominent comments, and turn in a demonstration of your extended evaluator working correctly.

Question 2: Adding a special form directly

Let's look at how we can extend the evaluator to be able to handle the `and` special form.

One attempt is to type in a definition for an `and` procedure at the M-Eval prompt. However, this will not work. **Show an example of why `and` doesn't behave properly.** What about `and` makes it different from the other procedures you've written in this class?

Write a new procedure, `eval-and`, and supporting procedures such as `and?` and `and-clauses`. This code should directly implement `and` as a special form. **Insert a clause in `m-eval`'s `cond` expression to test for an `and` expression and invoke your new `eval-and`.**

Show that your new `and` expression works by coming up with suitable test cases. Make sure you have non-trivial test cases, such as `(and (< x 5) (= 6 6) (pair? '()))`. Include the results in your submission.

Question 3: Adding a special form via transformer

As we saw in the case of the handling of `cond`, sometimes it is more convenient to rewrite an expression in terms of other expressions that the evaluator can already handle. Take for instance a new special form, `until`, used in the following example:

```
(define (countup n)
  (let ((x 0))
    (until (> x n)
           (printf "~s~n" x)
           (set! x (+ x 1)))))
```

The idea is that `until` checks to see if the first sub-expression is true. If it is, the `until` returns immediately (with an undefined value). Otherwise, it evaluates all the remaining sub-expressions in order, and then repeats the process from the start. As shown in the example above, it is intended to be used with set! or some other form of mutation to implement a simple loop.

This can be generalized to:

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

where `test`, `exp1`, ..., `expn` are expressions. Thus, one way to implement an until is to rewrite it in terms of more primitive Scheme expressions. For instance, in the above example,

```
(until (> x n)
       (printf "~s~n" x)
       (set! x (+ x 1)))
```

can be rewritten as

```
(let ()
  (define (loop)
    (if (> x n)
        #t
        (begin (printf "~s~n" x)
               (set! x (+ x 1))
               (loop))))
  (loop))
```

and thus

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

can be rewritten as

```
(let ()
  (define (loop)
    (if test
        #t
        (begin exp1
               exp2
                ...
               expn
               (loop))))
  (loop))
```

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

Write a procedure `until->transformed` which takes an `until` expression and returns the transformed expression. Your procedure should not call `m-eval`! It is just producing a new, simpler Scheme expression that does the same thing as the original. **Add a clause to `m-eval` to identify `until` expressions, and evaluate the transformed version.** You might find the `quasiquote` special form useful, although it is not required to solve this problem.

Question 4: Undoing assignments

In our standard evaluator, if we `set!` a variable, we lose its previous value. We would like to change this behavior, so that we keep track of the all of the values the variable has had over time.

We want to introduce a new **special form**, `unset!`. Evaluating `(unset! var)` in some environment should have the following behavior:

- if `var` has never been defined, the result should be an error.
- if the variable `var` has been previously defined with the `define` special form, or by being a formal parameter to a function, its value is reset to the previous value that the variable had.
- calling `unset!` on a variable `var` more times than it has been modified by `set!` is not an error; it simply does nothing.

For example:

```
(define x 5)
x ;; => 5

(set! x 10)
(set! x 11)
(set! x 12)
x ;; => 12
(unset! x)
x ;; => 11     (unset! undoes one of the set!'s)
(unset! x)
x ;; => 10     (unset! undoes the previous set!)
(unset! x)
x ;; => 5      (and again)
(unset! x)
x ;; => 5      (unset! more times than set! is fine)

(define x 15)
(unset! x)
x ;; => 15     (most recent value defined to x)
```

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

- Change the binding abstraction so that it stores both the current value of a variable and all of its previous values, and modify `define-variable!` and/or `set-variable-value!` so that they do the appropriate thing when a variable is defined or set.
- Add a new special form called `unset!`. Evaluating this special form should change the binding of 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 `unset!` expressions.

Implement this change and demonstrate your new evaluator showing this new behavior. Make sure to mark your change to `m-eval` with a prominent comment (like `;===QUESTION 4===`), and put new procedures, small changed procedures, and test cases at the end of `eval.scm`.

Question 5: Making environments first-class

Next, we'd like to make it possible to inspect the frames and environments of the evalutator from inside the evaluator. Specifically, we'd like to be able to ask for the environment of a procedure, and get back a new *environment* type which we can inspect to look up particular symbols in that environment. We could use this as follows:

```
(define (make-counter)
   (let ((n 0))
      (lambda ()
         (set! n (+ n 1))
         n)))
(define c (make-counter))
(c)   ;; => 1
(c)   ;; => 2
(env-value 'n (procedure-env c)) ;; => 2
```

Extend your evaluator to support the `(current-env)` and `(procedure-env proc)` special forms, which return an environment object representing the current environment or stored procedure environment respectively.

Now that we have first-order environments, we'll want to add ways to inspect them. First, **add an `env-variables` primitive procedure** which takes an environment and returns a list of the symbols bound in the top-most frame of the environment. Next, **add an `env-parent` primitive procedure** which takes an environment and returns the parent environment of it, removing the top-most frame. Finally, **add an `env-value` primitive procedure** which takes a symbol and an environment, and looks up the symbol in that environment; it should return `#f` if there is no such symbol defined in the environment.

As always, show examples of your new functionality in action.

Optional: Ensure that the objects returned by `(current-env)` and `(procedure-env proc)` cannot be mutated using `set-car` and `set-cdr` in a way which would actually mutate the evaluator's environment model. The easiest way to do this is to return a Scheme datatype for your m-eval *environment* which cannot be inspected or modified inside m-eval, simply due to a lack of primitives inside m-eval to deal with it. For this purpose, you may find the `box` primitive useful.

Question 6: m-evals all of the way down

The grand finale is to realize the name of the project – the *meta-circular* evaluator. We can prove that the evaluator we have written is sufficiently general by loading a complex program into it. It just so happens that we have such a program readily available – the evaluator itself!

Happily, the syntax that m-eval parses is a pretty close subset of the Scheme that we used to write the evaluator. However, there are still a number of things that you will need to adjust, either by simplifying the Scheme code that the evaluator is written in, or by extending the syntax that m-eval supports, in order to be able to run m-eval inside itself.

We have provided a `(load-meval-defs)` procedure which passes the definitions from the file you are working from into m-eval, one at a time. That is, it is as if you ran `(driver-loop)` and typed `(require r5rs)`[1], followed by `(define first car)`, then `(define second cadr)`, and so forth. Because `(load-meval-defs)` code reads the version of the file that is stored on disk, be sure to save each time before you run `(load-meval-defs)`, or it may read a too-old version.

In order for this to work, you will need to add several more primitive procedures to the list that m-eval supports. **Add the appropriate primitive procedures until `(load-meval-defs)` completes successfully.** You may need to go back and add a few more later as you exercise the rest of the code which m-eval is written in. If you used quasiquote in answering any of the above problems,

[1] Don't worry about having to implement `(require r5rs)` – we've taken care of implementing some of the logistical detritus for you already.

you may need to replace it with other, equivalent constructs (quasiquote is not simple to implement in m-eval).

Once you have successfully loaded your meta-circular evaluator into itself, you should fire it up to ensure that it works:

```
> (load-meval-defs)
loaded
> (driver-loop)


;;; M-Eval input level 1
(driver-loop)


;;; M-Eval input level 2
(+ 17 42)

;;; M-Eval value:
59


;;; M-Eval input level 2
**quit**

;;; M-Eval value:
meval-done


;;; M-Eval input level 1
**quit**
meval-done
>
```

At each level, evaluation gets successively slower. To help determine how much slower, we have added Scheme's `time` special form to `m-eval`. **At each level (in Scheme, one level of m-eval, and two levels of m-eval), define a `fib` function which computes Fibonacci numbers, and compute `(time (fib 8))`.** How long does each level take (read the "real time" output)?

If you are *extremely* patient, you can nest the evaluator one more time, by running `(load-meval-defs)` at the "M-Eval input level 1" prompt – which loads the definitions into the second level – and calling `(driver-loop)` in the first *and* second depths. At three levels deep, how long does even simple math take?