MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 2007

Project 5

Release date: 30th April, 2007
Due date: 11th May, 2007, at 6pm

Background

Code to load for this project: A link to the system code file eval.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 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:

(+ 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.

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, display, 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:

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

or you will need to re-evaluate your file and start fresh (which is probably the less confusing option).

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

Question 2: Undoing assignments

In our standard evaluator, if we `set!` a variable, we lose its original value. We would like to change this behavior, so that we keep track of the original binding of a variable which was given when the variable was defined.

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

- if the variable `var` has been previously defined with the `define` special form, its value is reset to the most recent value set with `define`.
- if the variable `var` is a parameter of a lambda expression, then its value is reset to its original value when the lambda expression was applied.
- if `var` has never been defined, the result should be an error.

For example:

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

(set! x 10)
(set! x 11)
(set! x 12)
x ;; => 12
(reset! x)
x ;; => 5     (reset! undoes all the set!'s)

(define x 15)
(reset! 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 its original value, 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 `reset!`. Evaluating this special form should change the binding of the variable to its original value. (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. Make sure to mark your change to `m-eval` with a prominent comment (like `;===QUESTION 2===`), and put new procedures, small changed procedures, and test cases at the end of eval.scm.

Advice

The remaining questions of this assignment introduce *advice* to our programming language. Advice provides a way to change what procedures do without changing the definition of the procedure itself; instead, advice code is wrapped around a procedure, running before the procedure, after the procedure, or both, in order to change its behavior.

This is a simple form of what is known as *aspect-oriented programming*, which is a technique for creating abstractions around code that needs to be sprinkled across large parts of a system. Tracing is a good example of this kind of cross-cutting code. If you're trying to debug a large system, for example, you might want to have a print statement at the beginning of every procedure to see what's going on, but you don't want those print statements cluttering up every procedure definition. With advice, you can separate the tracing code from the procedure definitions, making the code simpler and clearer, and allowing you to turn tracing on and off without having to edit the original procedures.

Question 3: before-advice

Provide a new special form, `before`, that allows a programmer to attach advice to a procedure so that the advice runs each time the procedure is applied, just before the procedure application. The advice is represented by a procedure that takes the same arguments as the procedure it's advising. For example:

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

(before (fact n)
  (printf "fact called on ~a ~n" n))
```

This code creates a procedure `fact` and attaches some advice to it that displays a message every time `fact` is applied. If you run `(fact 3)`, you should see:

```
fact called on 3
fact called on 2
fact called on 1
fact called on 0
6  ;; (the final value of (fact 3))
```

Before-advice is useful for tracing, logging, performance profiling, and error checking.

Advice should be attached to a procedure by changing the binding of the procedure name (e.g. `fact`) so that it points to the advice procedure, which in turn points to the original `fact` procedure. Thus, if `fact` is ever redefined to a new procedure, any advice that was previously attached to `fact` should be thrown away. The `reset!` special form created in the previous problem should also throw away all advice attached to a procedure.

Note that the procedure, in this case `fact`, doesn't necessarily have to be in the global environment. The `before` special form should look up `fact` in the environment in which it is evaluated, and change its binding in the environment frame where `fact` is found.

It should be possible to attach advice to both compound procedures and primitive procedures. Multiple pieces of advice can be attached to a procedure; they should all run before the original procedure does, in an arbitrary (unspecified) order.

The precise syntax of the `before` special form is `(before (`*`procedure-name arguments ...`*`) `*`body..`*`)`. Advice should have the same number of arguments as the procedure bound to *name*. The *body* part may consist of zero or more expressions, like the body of a lambda expression.

Advice should be lexically scoped; in other words, the body of the `before` special form should be evaluated in the environment in which the `before` special form was evaluated, not in the environment where `fact` was evaluated or `fact` was applied.

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

- Define a new kind of procedure object, analogous to primitive and compound procedures, which represents advice.
- Add the `before` special form to m-eval, so that advice can be attached to procedures by changing the binding of a procedure name.
- Change m-apply so that it runs advice before applying a procedure.

Note: don't implement before-advice by translating it to a lambda expression, because this will make it harder to do later parts of the project.

Implement this change and demonstrate your new evaluator showing this new behavior. Be sure to mark your changes with a prominent comment.

Question 4: around-advice

Provide another kind of advice, around-advice. Unlike before-advice which only runs *before* a procedure is applied, around-advice runs *instead* of a procedure application. Around-advice has access to the arguments that the procedure was called with, and it can choose whether or not to

actually perform the procedure application. It can also examine the procedure's return value, and possibly change it.

Here's an example of around-advice for the fact procedure, that extends factorial into negative numbers:

```
(around (fact n)
  (if (< n 0)
      (* (fact (- n)) (if (even? n) 1 -1))
      (fact n)))
```

Notice that the around-advice must actually call the original fact procedure in order for the original application that triggered this advice to happen. In the environment in which the body of the `around` is evaluated, `fact` should be bound to the original fact procedure. (In general, however, when there are multiple pieces of advice for `fact`, `fact` needs to be bound here to something that runs the remaining advice as well as the original fact procedure.)

Like before-advice, around-advice is useful for security checks (i.e., does the caller have permission to call fact?), tracing, and performance profiling. It can also be used to add new behavior to built-in procedures. For example, if you had created an abstraction for representing complex numbers, you could use the following around-advice to generalize the builtin procedure `+` to handle complex numbers as well as real numbers:

```
(around (+ x y)
  (if (or (complex? x) (complex? y))
      (+complex (value-to-complex x) (value-to-complex y))
      (+ x y))))
```

The same procedure may have both before-advice and around-advice, and multiple pieces of each kind of advice. Advice may run in arbitrary (unspecified) order, but multiple around-advice on the same procedure should be *nested*. For example:

```
(define (square x) (* x x))
(around (square x) (+ (square x) 1))
(around (square x) (+ (square x) 1))
(square 5) ; => 27 (because both pieces of around-advice run)
```

Again, the order of execution of multiple pieces of advice is unspecified. It may be that all the before-advice runs first, and then all the around-advice. Or it may be that around-advice and before-advice are intermingled, so that around-advice that chooses not to call down to its attached procedure may actually block before-advice from running. Your implementation can behave arbitrarily, as long as it runs before-advice *before* the original procedure, and around-advice *around* it.

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

- Define a new kind of advice object which represents around-advice.
- Add the `around` special form to the evaluator.

- Extend m-apply to call around-advice, binding the name of the advised procedure in the new environment. Be careful here – it's easy to get into infinite loops, with around-advice repeatedly retriggering itself.

Implement this change and demonstrate your new evaluator showing this new behavior. Mark your changes prominently.

Question 5: after-advice

A third kind of advice to support is `after` advice, which runs *after* its attached procedure has returned a value. After-advice has access not only to the parameters of the procedure, but also to its return value, so the after-advice actually takes one more parameter than the attached procedure. For example:

```
(after (fact n result)
  (printf "fact ~a returned ~a ~n" n result))
```

We want you to implement this special form differently, however. Rather than extending your advice storage and m-apply to handle after-advice, you should instead *desugar* the `after` special form into an instance of the `around` special form, and then use your existing mechanism to evaluate that `around` special form. You can see examples of this kind of translation in the way that `cond` and `let` are handled by m-eval.

Before-, around-, and after-advice can all be mixed on a single procedure, and the advice runs in arbitrary order relative to each other, subject to the basic requirement that before-advice runs before the original procedure, after-advice runs after it, and around-advice runs around it.

Note that the value returned by the after-advice itself is ignored, so after-advice is only useful for side-effects.

The precise syntax of the `after` special form is `(after (`*`procedure-name arguments ... result-name`*`) body..)`.

Be sure to show examples of your code working on test cases.

Question 6: Constraints

It's useful to be able to constrain advice so that it only runs in certain contexts. For example, suppose we're interested in how many times the primitive `+` operator is used in a computation of a Fibonacci number. Here's how we might do it:

```
(define (fib n)
  (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))

(define count 0)

(before (+ (in fib) x y)
  (set! count (+ count 1)))
```

Notice that the before-advice has a new bit of syntax in it: `(in fib)` specifies that the advice should be called only for applications of `+` that appear in the body of `fib`. This is a *lexical* constraint, because it constrains the advice to a particular region of the source program. Applications of `+` outside of `fib` will not run the advice – in particular, the application `(+ count 1)` in the advice body itself won't trigger the advice, which is necessary to prevent the advice from repeatedly calling itself in an infinite regress.

Also, note that the `in` constraint is *lexical*, not dynamic. If `fib` calls another procedure that in turn calls `+`, then the advice will be triggered only if the other procedure is also defined inside `fib`. For example:

```
(define (f)
  (define (g) (+ 2 3))
  (+ (g) (h)))

(define (h)
  (+ 5 6))

(before (+ (in f) x y)
  (printf "~a + ~a ~n" x y))
```

Here, the before-advice should be called for the applications of `+` in `f` and `g` (since they are lexically inside `f`), but not for the application of `+` in `h`. Thus, applying `(f)` should print:

```
2 + 3
5 + 11
16   ; (the final result)
```

Implement the `in` lexical constraint for all forms of advice (before, around, and after). The constraint should be optional, so you should continue to support advice that isn't constrained (both forms are found in the example above).

To do this, you will have to make several changes to the evaluator:

- Extend the syntax of `before`, `around`, and `after` special forms to detect and handle the `in` constraint.
- Extend your advice procedure abstraction to store an optional constraint.
- Extend m-apply so that it tests the constraint before applying advice. You will have to decide how to determine whether the application is found in the body of another procedure. One reasonable way to do this is to modify the environment so that each frame records which procedure created it. Then, m-apply should search through the environment looking for a frame created by the constraint procedure.

Be sure to show examples of your code working on test cases, and mark your changes prominently. Again, add new procedures, small changed procedures, and test cases to the end of eval.scm, but make changes to large procedures in-place with a prominent comment indicating that the change was due to Question 6.

Submission

Hand in a single file, a copy of eval.scm with your changes incorporated in it. Mark each change prominently with a comment showing the question number that change addresses, and put new code or small changed procedures at the end of the file. Include comments explaining your code, and at the end of the file, 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.