MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1998

# Problem Set 8

## The Evaluator

- Issued: Tuesday, November 10, 1998
- Holiday Reminder: No recitations on Wednesday, November 11, 1998
- Tutorial preparation for: Week of November 16, 1998
- Written solutions due: Friday, November 20, 1998
- Reading: Read Section 4.3 before lecture on November 12, 1998 and Section 5.1 before lecture on November 17, 1998.
- Drop Date Reminder: Wednesday, November 18, 1998

This problem set requires very little actual programming. In order to do it, however, you will need to have a good understanding of Sections 4.1 and 4.2 of the textbook. If you have lost the habit of planning your work carefully before coming to lab, this would be a good week to change your working style—otherwise you are likely to waste a great deal of time in the lab.

## Evaluator Code

The code for this problem set includes these files:

- `syntax.scm` contains the procedures that define the syntax of expressions, as described in section 4.1.2.
- `meval.scm` is basically the metacircular evaluator described in section 4.1.1 of the notes. In order to avoid confusing the `eval` and `apply` of this evaluator with the `eval` and `apply` of the underlying Scheme system, we have renamed these procedures `meval` and `mapply`.
- `evdata.scm` contains the procedures that define the evaluator's data structures, as in section 4.1.3. We have set up the initial environment so that the following names are bound to their values in the underlying Scheme: `car, cdr, cons, null?, write-line`. These are the only primitives that have been implemented. You will be installing some more primitives as part of this problem set.
- `lex-dyn.scm` contains a partial implementation of an evaluator that can perform both lexical and dynamic scoping. You will be using this code for Lab Exercise 6.

## Tutorial Exercises

These tutorial exercises are meant to help you gain some understanding of the evaluator so that you are familiar with the code before you go to lab.

**Tutorial Exercise 1:** Browse through the code files. How would you extend the initial global environment to have bindings for `+ - * / = < > 1+ list pair? symbol? eq?`. How would you extend the global environment so that `nil` would have a binding?

**Tutorial Exercise 2:** Do Exercise 4.14 in the textbook which looks at extending the evaluator to handle `map`.

**Tutorial Exercise 3:** Do Exercise 4.1 in the textbook which will help you understand how flexible the evaluator is for handling different orders of argument evaluation.

**Tutorial Exercise 4:** Scoping rules dictate how values for free variables are found. With lexical scoping (which is what you have seen so far), they are looked up in the environment in which the procedure was created. With dynamic scoping however, they are looked up in the caller's environment.

It may be easier to illustrate this using environment diagrams. With the rules you have seen so far, when a procedure is applied, you create a frame and then you hang this new frame off of the frame in which this procedure was created. If dynamic scoping rules are in play, then this new frame hangs off of the environment in which the procedure is being applied.

What do the following expressions evaluate to if lexical scoping were used? What if dynamic scoping were used? Draw environment diagrams to explain how your answers arise.

```
(let ((y 1))
  (let (( f (lambda (y) (lambda (x) (+ x y))) ))
    ((f 20) 300)))

(let ((y 1))
  (let (( f (lambda (x) (+ x y)) ))
    (let ((y 20))
      (f 300))))
```

## Using the Evaluator

Here is some helpful information for using the evalator:

- When you ``M-x load-problem-set: 8'' in Edwin, this will load the files `syntax.scm`, `evdata.scm`,`meval.scm` and `lex-dyn.scm`. Evaluating the code in these files and then evaluating `(driver-loop)` in the `*scheme*` buffer will start the read-eval-print loop for the meta-circular evaluator with a freshly initialized global environment.

- In order to help you avoid confusion, we've arranged it so that the driver loop will print input and output prompts. For example,

  ```
  ;;; M-Eval input:
  (+ 3 4)
  ;;; M-Eval value:
  7
  ```

  shows an interaction with the `meval` evaluator. To evaluate an expression, you type the expression into the `*scheme*` buffer at the `;;; M-Eval input:` prompt, and press `ctrl-x ctrl-e`.

- You should keep in a separate file any procedure definitions you want to install in an evaluator. If your Edwin Scheme buffer is running the read-eval-print loop of an evaluator, you can then visit this definitions file and type `M-o` to enter the definitions into the evaluator.

- The evaluator you are working with does not include any error system. If you hit an error you will bounce back into ordinary Scheme. You can restart the evaluator, with its global environment still intact, by evaluating `(driver-loop)`. Evaluating `(init)` will also do this, but `(init)` will also re-initialize the global environment, so you will lose any definitions you have made. At any time, you can break out of the evaluator and get back to the underlying Scheme by typing `ctrl-C ctrl-C`.

- It can be instructive to trace `meval` and/or `mapply`. (You may need to do this while debugging your code for this assignment.)

- Since environments are generally 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. This was done by

  ```
  (set! *unparser-list-depth-limit* 7)
  (set! *unparser-list-breadth-limit* 10)
  ```

  at the end of the file `evdata.scm`. You may want to alter the values of these limits to vary how much list structure will be printed as output.

## Lab exercises

As usual, for all the lab exercises below, you should turn in listings of any procedures you define in your solutions, as well as sample evaluations demonstrating their correct behavior.

You may find it helpful to keep your code, including any modifications you make to the code we have provided, in a separate file. After you have loaded the problem set code, you can then evaluate select portions of code from your answer file to redefine a current definition.

**Lab Exercise 1: Getting Acquainted** Actually extend the primitives that are bound in the initial global environment with those from Tutorial Exercise 1. Start the evaluator by typing `(init)` and evaluate a few simple expressions and definitions. It's a good idea to make an intentional error and practice restarting the read-eval-print loop (both with and with-out wiping the environment clean). Turn in a listing of the procedures that were changed and an excerpt of this practice session.

(If for any reason you need other primitives defined, you now know how to include them in the evaluator.)

**Lab Exercise 2: AND Extension** Let's look at how we can extend the evaluator to be able to handle the `AND` special form.

(a) One attempt is to type in a definition for `AND` at the evaluator prompt in the runtime environment of the simulation. 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?

The next two approaches involve modifications to the evaluator `meval` code.

(b) You've already seen how `cond` can be implemented as a derived `if` expression in the evaluator code. Implement `AND` as a derived expression.

(c) Instead of repackaging the components of `AND` as a derived expression of another expression that you already know how to evaluate, you can implement `AND` behavior by handling it directly as a special form (which is what you would have to do if there were no way for `AND` to be written as a derived expression). Show how to implement `AND` in this manner.

**Lab Exercise 3: `LET` Special Form** Let's have some more practice with derived expressions by extending the evaluator so that it can handle `let`.

(a) Do Exercises 4.6 in the book which asks you to extend the evaluator to handle `let`.

(b) Do Exercises 4.7 in the book which asks you to extend the evaluator to handle `let*`.

**Lab Exercise 4: A Loop Construct** Consider the `for` special form: `(for <var> <initial> <pred> <next> <body>)`

Here's what happens when this expression is evaluated:

1. The initial value of variable `<var>` is `<initial>`.
2. If `<pred>` is false, then the loop terminates and return the symbol `'ok` as the value of the `for` expression.
3. If `<pred>` is true, then `<body>` is evaluated.
4. The next value of `<var>` is then determined by evaluating `<next>`. Loop back to Step 2.

For example: `(for x 1 (<= x 10) (+ 1 x) (write-line x))` would print the numbers from 1 to 10.

Why is `for` a special form? Add the `for` construct as a special form to the evaluator.

In the next exercise, you will be implementing dynamic scoping. You may want to put your definitions in a different file because you will want to use the evaluator that you have at this point (which has incorporated your answer from Lab Exercises 1-4) for Lab Exercise 6.

**Lab Exercise 5: Dynamic Scoping** Modify the evaluator so that it follows dynamic scoping rules rather than lexical scoping rules.

Turn in a listing that shows that your modification correctly performs dynamic scoping instead of lexical scoping. If you've successfully augmented your system to handle `let`, then you can use the `let` expressions from Tutorial Exercise 4 to test whether dynamic scoping works.

**Lab Exercise 6: Using Lexical Scoping with Dynamic Lookup** This exercise explores the possibility of lexical and dynamic scoping co-existing in the same system. Recall that when a procedure is applied, a new environment is created and an expression is evaluated in this new environment. Scoping rules specified how environments are linked together so that this new environment is linked to either the environment in which the procedure was created (a lexical link) or the caller's environment in which the procedure was applied (a dynamic link). In this manner, scoping rules basically told you where to look for the value of a free variable. Imagine a hybrid environment structure where each environment had not one but two links – both a dynamic link `dyn-env` and a lexical link `lex-env`, and furthermore, the programmer had the ability to specify that a free variable should be looked up in a dynamically scoped manner instead of the usual lexically scoped way.

We must be able to do two things:

1. declare that a certain variable is a dynamic variable – i.e. its value, if not present in the current frame, should be looked up in a dynamically scoped manner.
2. define a dynamic variable (i.e. declare that it is dynamic and supply a value).

We do this by creating a special form called `dynamic`. If `dynamic` is provided with a variable name *in addition to* a variable value, then this indicates that a new dynamically scoped variable is being created and defined. If `dynamic` is provided with a variable name only, then this indicates that this variable is a dynamic one and should be looked up in a dynamically scoped manner (by looking in dynamically enclosing environments).

To support dynamic variable declarations and dynamic scoping as an option in addition to the usual lexical scoping rules, we expand the notion of an environment. Specifically, it contains:

- the current FRAME
- the surrounding lexical environment, LEX-ENV
- the surrounding dynamic environment, DYN-ENV
- a list of declared dynamic variable names, DYNAMICS.

Note that dynamic variables and lexical variables can live in the same frame, but in order for a dynamic variable to be accessed (when an expression is evaluated with respect to this environment) the variable must also be present in the list DYNAMICS (i.e. declared to be a dynamic variable).

(We will also require `dynamic` statements, like internal `defines`, to appear at the start of a procedure body. If the user does not adhere to this, then anything goes and you cannot guarantee what the outcome will be. In your implementation below though, you do not have to check for this, for simplicity, assume that `dynamic` statements are correctly placed, although in practice you would have to handle this.)

Here's an example:

```
(define (foo x y)
  (define z 100)
  (dynamic *w* 50)    ;; definition (and declaration)
  (bar (+ x y z)))

(define (bar b)
  (gorp (* 2 b)))

(define (gorp c)
  (dynamic *w*)       ;; declaration
  (+ c *w*))

(foo 1 2)
=> 256
```

These three procedures are defined in the global environment. Notice that in `foo`, `*w*` and a value are supplied to `dynamic` and so a dynamic binding is created and in `gorp`, this dynamic variable is then accessed.

Let's look at what happens when we evaluate `(foo 1 2)`:

- `(foo 1 2)`: Here, a new frame E1 is created with both `lex-env` and `dyn-env` linked to the global environment. Within E1, `z` is defined and bound to 100, and `*w*` is defined and bound to 50. In addition, `*w*` is declared to be a dynamic variable.
- `(bar 103)`: This leads to the creation of frame E2 which `lex-env` pointing to the lexical global (because that is where `bar` was created), and `dyn-env` linked to E1 (the caller's environment). The expression `(gorp 206)` is next evaluated in E2.
- `(gorp 206)`: Now, E3 is created and this also hangs off of the lexical global environment via `lex-env`, however, `dyn-env` points to E2. When the `(dynamic *w*)` line is encountered, the value of `*w*` is sought, first in the current frame E3 and then by following `dyn-env` links until it is finally found in E1.

Your goal now is to implement this behavior in the evaluator. Use the evaluator that you ended up with before doing Lab Exercise 5.

To start you off, we've provided some code for you in `lex-dyn.scm` including a modified version of `meval` and `mapply`. You need to complete the definitions of several procedures in this file:

(a) Complete the definition of the syntax procedures: `dynamic?`, `dynamic-declaration?`, `dynamic-definition?`, `dynamic-variable` and `dynamic-value`.

(b) Study the new environment structure. Complete the definitions for the selectors: `enclosing-environment`, `enclosing-dyn-env` and `dynamics-in-env`, and also the definition for `add-dynamic-in-env` which adds a variable to the DYNAMICS list.

(c) When looking up a variable, we check if it's on the list of DYNAMICS. If it is, then we look it up in a dynamically scoped manner (we look it up in the current frame, if it's not there, we trace `dyn-env` links to the dynamically enclosing environment). If it is not on the list of DYNAMICS, we check the current frame for the variable and if it is not found in the current, we follow `lex-env` links to the lexically enclosing environment.

Complete the definition of `lookup-dynamic-loop`. (If you find a need to, you may modify the code for `lookup-variable-value` and `lookup-lexical-loop` and/or provide any extra helping procedures so long as the procedures are consistent with their specifications.)

Turn in a listing of all the procedures that have been modified or newly defined.

(d) Another operation that requires traversing the environment structure is `set-variable-value!`. Now that you have had experience with `lookup`, complete the definitions of `set-variable-value!`, `set-lexical-variable-loop` and `set-dynamic-variable-loop` so that you `set!` a variable in a manner that is similar to how you would `lookup` a variable.

Turn in a listing of all the procedures that have been modified or newly defined.

(e) Turn in a listing with well chosen examples as test cases that clearly show that your code and the resulting evaluator correctly handle both `dynamic` and `lexical` scoping.

(f) Now that you have understand how lexical scoping and dynamic scoping work, what situations might dynamic scoping be good for? What are some advantages? Are there any risks or disadvantages?

Turn in answers to the following questions along with your answers to the questions in the problem set:

1. About how much time did you spend on this homework assignment? (Reading and preparing the assignment plus computer work.)
2. Which scheme system(s) did you use to do this assignment (for example: 6.001 lab, your own NT machine, your own Win95 machine, your own Linux machine)?
3. We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).
    - If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the text (such as some of the archives), please indicate your consultants' names and your references. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.
    - Otherwise, write "I worked alone using only the reference materials," and sign your statement.