MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.184—Zombies Drink Caffeinated 6.001
IAP, 2011

Project 2, Part 2

Release date: 11th January, 2011
Due date: 20st January, 2011, at 1900h

Background

We noted in Part 1 of this project that the meta-circular evaluator does not include an error system. As the evaluator is currently written, errors encountered by m-eval or m-apply will cause the evaluator to exit back to the DrRacket REPL, and will only sometimes produce a useful error message.

Furthermore, our meta-circular evaluator has no debugging system. A debugger is a tool that allows the user to see more closely what a given program is doing by allowing him to pause and resume execution, and to examine the program state at any time.

In this part of the project, we will add a debugging environment that allows users to see how and explore why their programs encountered an error.

In true Scheme spirit, this debugging environment will, itself, be Scheme. We will do this by basing our debugger on our m-eval REPL from Part 1, with additional procedures for interacting with and manipulating the program state. Many of the modifications have already been made, but you will have to implement some of them yourself.

Like in Part 1, the questions in Part 2 don't generally involve a lot of code. Rather, they require you to think carefully about where to make small changes or additions. The instructions in "Making Changes to the Evaluator" from Part 1 apply to Part 2, as well.

Dynamic Scoping

Because a debugger inherently involves changing the execution flow of a program, our debugging system will makes heavy use of continuations and dynamic binding. This section will briefly review dynamic scoping and `fluid-let`.

Recall from lecture that Scheme uses *lexical* scoping (sometimes also called *static* scoping). A "binding" is a correspondence between a name and a value, and "lexical" means "according to the program text" (instead of its runtime behavior). So, when *lexical* binding is used, the value of an evaluated name is looked up in the environments that enclose the name in the program text.

In contrast, when *dynamic* binding is used, the value of an evaluated name is looked up in the most recent environment.

For example, in a language with lexical scoping, free variables in the body of a lambda expression will be looked up in the environment where the lambda expression was *evaluated*, whereas in a language with dynamic scoping, those variables will be looked up in the environment where the procedure is *applied*.

The dynamic binding primitive we are going to use is called `fluid-let`. It works like regular `let` except that the variables are bound to the given values dynamically within the body of `fluid-let` instead of lexically. For example, the following scheme program produces a value of `5`:

```
(define value 2)
(define (get-the-value) value)

(fluid-let ((value 5))
  (get-the-value)) ;; => 5
```

Note that you can't use `fluid-let` with variables that have not been defined in the lexical scope of the `fluid-let`. For example, we could not have omitted the `(define value 2)` in the above program. This is because `fluid-let` works by temporarily binding new values to existing names, and then restoring the original values later.

Another important aspect of `fluid-let` is that it plays nicely with continuations. If you jump out of the body of a `fluid-let` using a continuation, the original value of bound variables will be restored, and if you jump into the body of a `fluid-let` the new value will be installed. For example:

```
(let ((return-value '())
      (cont #f)
      (times-through 0)
      (x 2))
  (define (do-append)
    (set! return-value (append return-value (list x))))
  (do-append)
  (fluid-let ((x 5))
             (call/cc (lambda (k)
                        (set! cont k)))
             (set! times-through (+ times-through 1))
             (do-append)
             (set! x 7))
  (do-append)
  (if (= times-through 1)
      (cont 0))
  return-value)
;; => {2 5 2 7 2}
```

Work through the above example carefully and convince yourself that you understand it.

Printing

A few questions in this project ask you to print or display something. You may find the procedures `display`, `newline`, `pretty-display`, and `printf`[1] useful. The `pretty-display` procedure is like `display`, except that it formats complex structures nicely.

A value displayed is not to be confused with a value returned. The REPL prints the values of expressions, so it can sometimes be hard to see the distinction. Using these definitions

```
(define (return-num)
  5)
```

[1] Racket `printf` format strings are described under `fprintf` in the Racket reference manual on this page: `http://docs.racket-lang.org/reference/Writing.html`

```
(define (display-num)
  (display 5)
  (newline))

(define (return-string)
  "world")

(define (display-string)
  (display "world")
  (newline))
```

try evaluating each of these expressions and observe the output and behavior:

- `(return-num)`
- `(display-num)`
- `(+ 1 (return-num))`
- `(+ 1 (display-num))`
- `(return-string)`
- `(display-string)`
- `(string-append "hello " (return-string))`
- `(string-append "hello " (display-string))`

Changes to the Evaluator

We have provided you with a new version of the evaluator. It can be found in `eval-debugger.scm`, linked off the course web page. Note that you do not have to add your changes from Part 1 to the new code, but you can if you wish to.

The new version has several important changes. First, because we will interact with our debugger using Scheme, we have added another REPL to be used with debugging. It is invoked via `debugger-loop` and described in Question 1. Second, in addition to the `primitive-procedures` list, there is now a `debugger-procedures` list which contains the names and values for procedures that we will make available to the user while debugging. It has the same structure as `primitive-procedures`. Third, primitive procedures exposed in m-eval and m-eval itself will now start the debugger upon encountering an error. This is described more fully in Question 1.

Finally, we have added code for keeping track of the program *history*. The history keeps track of what expressions are currently being evaluated. The history is a stack of *sub-problems*. The top sub-problem on the history stack is the current expression under evaluation. The sub-problem right below the top is the expression that requires the value of the current expression to proceed.

For example, suppose we are evaluating the program `(+ 2 (* 3 4))` and we are currently evaluating the expression `3`. Then, the history stack will look like this:

Index	Expression
0 (top)	`3`
1	`(* 3 4)`
2	`(+ 2 (* 3 4))`

Notice that `4` doesn't appear as its own entry in the history. That's because it has either already been evaluated (right before `3`), or it will be evaluated after `3`, depending on the order that elements of a combination are evaluated (which is unspecified by Scheme). `2`, `*`, and `+` don't appear for similar reasons.

Each sub-problem will be a tagged list of three elements. In addition to the expression for that history item, each sub-problem will contain the environment that the expression is being evaluated in and the continuation for the expression's evaluation.

The history is maintained right at the top of `m-eval` using `fluid-let`.

The Debugger

Question 1: Dream within a Dream

First, using your evaluator from Part 1, try evaluating a Scheme expression which should result in an error, such as `(car 5)`. Now, load up the code for Part 2, and try running it again. **What's different?**

The short explanation is that our evaluator will now invoke `start-debugger` whenever there is an error in something the user entered at the meta-circular evaluator prompt.

Essentially, we've wrapped primitive procedures so that errors produced by the underlying DrRacket interpreter when evaluating user code are caught by our own interpreter[2], which will then invoke a procedure called `start-debugger`. We also make sure the meta-circular evaluator calls `m-error` instead of `error` when it finds an error in user code (such as the attempted use of an unbound variable) because `m-error` will similarly invoke `start-debugger` instead of exiting.

Note that errors caused by bugs in the meta-circular evaluator will not get caught, and will trigger an exit as usual.

As stated in the message you should have received when evaluating erroneous expressions using the Part 2 code, the `start-debugger` procedure is not yet implemented.

One of the things `start-debugger` will do is call `debugger-loop`. Try starting the debugger manually by running `(debugger-loop the-global-environment)`. You should see something like this:

```
;;; M-Eval Debugger (level 0)
;;; Current expression: <none>
;;; Input:
```

This is identical to a regular m-eval REPL except for the appearance of the prompt; that is to say, you should be able to evaluate expressions like normal. The debugger displays some additional information in its prompt. The "Current expression:" line will show you what was being evaluated when the debugger was invoked. Since you invoked the debugger manually, it displays "<none>."

[2]If you're interested in the details, take a look at the `wrap-primitive-proc` procedure.

The "level N" in parentheses shows how many nested debuggers have been invoked. This is to allow us to debug errors that occur while already in the debugger.

You can exit the debugger by evaluating `**quit**`, just like normal. Also note that when you quit debugger level 2, you should return to debugger level 1.

Unlike `driver-loop`, `debugger-loop` takes an environment parameter. This is the environment that the REPL will execute in. This means, for example, that variables defined at the REPL will be bound in the environment passed to `debugger-loop`.

Which environment should `start-debugger` pass to `debugger-loop`? If it passes the environment of the top-most sub-problem, then the debugger will execute in the environment of the expression that caused the error. This means we can examine the cause of errors simply by evaluating the names of variables we suspect might have the wrong values! We can also manipulate variables in our program using the familiar `define` and mutation functions such as `set!`.

Write the `start-debugger` procedure. It should invoke `debugger-loop` and increment `debug-level` correctly so that the debugger prompt tells you how far down the rabbit hole you are. Show that errors executed inside of the m-eval REPL now start the debugger, and that the debugger is running in the environment of the erroneous sub-problem. Also demonstrate nested debugging REPLs. When you exit a nested debugging REPL to a lower level, does the level displayed in the prompt decrease?

To test that the debugger ends up in the correct environment, you can try running the following program:

```
(define (foo x)
  (car x))

(foo 5) ;; <crash>

;; from the debugger REPL
x ;; should be 5
```

Question 2: Augmented Reality

At this point, the debugger is already quite useful for finding many common bugs. However, there is more functionality we would like to add, and most of it will be accessed via procedure calls.

The problem is that we only want these procedures to be accessible while the user is in the debugger and not while in the regular m-eval REPL. If we use `extend-environment` to add the debugger procedures to the environment we passed to `debugger-loop` in Question 1, then we lose the ability to manipulate the environment of the top-most sub-problem; variables `define`d in the debugger would be bound in the debugger's environment, not the environment of the top sub-problem.

The solution we will use is to create a new environment that contains the same frames as the top sub-problem's environment plus a new frame that will contain all the debugger procedures. The new frame will be placed as the parent of the global environment.

Write a procedure called `make-debugger-environment` that takes a base environment and creates a new environment which is the same except that it has one more frame that contains the debugger procedures. Demonstrate that it works. Finally, use `make-debugger-environment` in `start-debugger` to make the debugger procedures available in the debugger REPL. Show that you can run debugger procedures (we have

provided a couple for you) and that modifications to the environment affect the program being debugged. For the last part, here is something you can try:

```
(driver-loop)
(define x 5)
(car x) ;; causes an error

;; now in the debugger
(set! x 10)

;; exit the debugger
**quit**

x ;; should evaluate to 10
```

Question 3: The Eye of Providence

When debugging, it is often useful to inspect the current values of variables when an error occurs. We can already do that manually now by evaluating the names of individual variables. However, it can be nice to look at all bound names at once.

The procedure `examine-env` should print out the current environment. **Why should it not just return the current environment and let the REPL take care of printing?**

Write `examine-env` and add it to `debugger-procedures` so it can be used from the debugger. Demonstrate that it can be used within the debugger (including within lambda bodies), but not in the regular m-eval REPL.

Question 4: Hindsight

In addition to knowing the immediate expression that caused an error, it would be useful to know how we got there. For example, if you run the following program:

```
(define y '(1 2 3))
(define (foo x)
  (bar (+ x (car y))))
(define (bar y)
  (cons (car y) '()))
(foo 5)
```

your debugger should currently print

```
Caught error: mcar: expects argument of type <mutable-pair>; given 6

;;; M-Eval Debugger (level 1)
;;; Current expression: (car y)
;;; Input:
```

However, this output doesn't tell us which `(car y)` in the program actually caused the error.

A display of the sub-problem history is called a *backtrace*[3]. **Write a procedure called `backtrace` that prints the current history and add it to `debugger-procedures`.** Its output should look something like this:

```
Backtrace:
Sub-problem 0: (backtrace)
Sub-problem 1: (car y)
Sub-problem 2: (cons (car y) '())
Sub-problem 3: (bar (+ x (car y)))
Sub-problem 4: (foo 5)
```

You can make the output look however you want, but **it should display the sub-problem number**, as it will be useful in the next problem.

Question 5: Operator, I need an exit

At present, once we've entered the debugger, we can't leave except by evaluating `**quit**`. In this problem, we'll add the debugging procedure `return`. This powerful debugger procedure will allow us to return arbitrary values for arbitrary sub-problems. It takes two arguments: a sub-problem number and a value. When invoked, `return` causes execution to continue from the given sub-problem as if that sub-problem's expression had evaluated to the given value.

Here is an example usage:

```
;;; M-Eval input:
(cons 1 (car 5))
Caught error: mcar: expects argument of type <mutable-pair>; given 5

;;; M-Eval Debugger (level 1)
;;; Current expression: (car 5)
;;; Input:
(backtrace)
Backtrace:
Sub-problem 0: (backtrace)
Sub-problem 1: (car 5)
Sub-problem 2: (cons 1 (car 5))

;;; M-Eval Debugger (level 1) value:
#<void>


;;; M-Eval Debugger (level 1)
;;; Current expression: (car 5)
;;; Input:
(return 1 '(2 3))
```

[3]In imperative languages, backtraces often only show the *function calls* leading up to the current point in the program.

```
;;; M-Eval value:
{1 2 3}
```

In this example, the user got an error from trying to take the `car` of an integer. Knowing that the result should have been the list `(2 3)`, he uses `return` to cause the erroneous call to `car` to return that value, instead. The program then continues with the call to `cons`.

Write `return` and add it to debugger-procedures. Show it working by returning from the debugger to m-eval. Also show returning from one debugger level to a lower one. Does your debugger prompt show the correct level after using `return` to escape from one debugger level to another? Why? If not, fix the problem.

Question 6: Barriers to Entry

So far, the only way we have been able to invoke the debugger is by causing an error. However, sometimes it's useful to be able to stop execution of known-buggy code to examine the program state before we actually hit an error. The point at which you would like execution to stop is called a *breakpoint*.

We will implement breakpoints with a new special form called `break`. It has the form `(break` *expression*`)`. When the interpreter evaluates a break expression, it should start the debugger before executing the contained expression. Once the user has finished his debugging session, he can invoke the `continue` procedure, which will resume execution of the program, starting with the expression inside the break. For example:

```
;;; M-Eval input:
(begin (display 1)
       (break (display 2))
       (display 3))
1

;;; M-Eval Debugger (level 1)
;;; Current expression: (break (display 2))
;;; Input:
(continue)
23
;;; M-Eval value:
#<void>
```

It should be an error to invoke `continue` when the debugger was not started via `break`. The reason is that the only other way to get into the debugger is via an error, and you can't continue execution once an error has occurred.

Write the `break` special form and the `continue` procedure and add it to `debugger-procedures`. Demonstrate that evaluating a break expression while the debugger is running because of a previous break behaves as expected.

Question 7 (Optional): To Boldly Go...

Do something cool! If you need some help coming up with ideas, here are a few suggestions. Note that their difficulties vary widely! You can extend the debugger from Part 2 or the vanilla evaluator from Part 1.

- Add *conditional* breakpoints. That is, make the `break` special form take an optional additional argument that determines whether the debugger should be invoked or not. If the extra expression returns `#t`, the debugger should be invoked. Otherwise, the program should keep going.
- Implement a new debugger procedure that lets the user start a debugging REPL that executes in the environment of any sub-problem currently in the history.
- Allow the user to *step* the program that is being debugged. When the user steps the program, the evaluator executes exactly one sub-problem and then stops, returning control to the debugger. If you're feeling even more adventurous, you can implement *backwards stepping*. That is, allow the user go back to the sub-problem immediately before the current one. It should undo assignments, definitions, and `reset!`s as it steps backwards. A debugger that can run a program both forwards and backwards is sometimes called a *time-traveling debugger*. Note that you may need to make the evaluator use continuation-passing-style to make forward and backward stepping work.
- Implement `call/cc` inside the meta-circular evaluator. These continuations should still interact well with the debugging system.
- Add a more full-featured exception handling system.
- Collect statistics (such as number of lambdas applied, number of environment bindings, etc.) while the evaluator is running and allow the user to access them from the debugger.
- Add typed expressions, as assigned in Project 2, Part 2 from 2009. You can find a copy of that assignment linked off the course web page.