CSCI 301, Lab # 7

Fall 2024

Goal: This is the fifth in a series of labs that will build an interpreter for Scheme. In this lab we will add the `letrec` special form.

Submission: This lab builds on Lab 6. You will extend the functionality your Lab 6 program and submit `eval.rkt` to Canvas. As usual, unit tests are provided. As in prior labs, please make sure your file has a block comment at the top with your name, etc., and each function has a comment describing its purpose, arguments, and return value.

Unit tests: Your program must pass the unit tests found in the file `lab7-test.rkt`. Place this file in the same folder as your program, and run it; there should be no output.

Letrec creates closures that include their own definitions: Consider a typical application of `letrec`:

```
(letrec ((plus (lambda (a b) (if (= a 0) b (+ 1 (plus (- a 1) b)))))
         (even? (lambda (n) (if (= n 0) true (odd? (- n 1)))))
         (odd? (lambda (n) (if (= n 0) false (even? (- n 1))))))
  (even? (plus 4 5)))
```

`plus` is a straightforward recursive function. `even?` and `odd?` are mutually recursive functions, each one requires the other.

If we use `let` instead of `letrec`, we will evaluate the `lambda` forms in the current environment, and none of the three functions will be defined in that environment. Each of the closures will contain a pointer to an environment in which the recursive functions are *not* defined. Thus, we cannot simply use `let`.

We want the closures to close over an environment in which `plus`, `even?` and `odd?` *are* defined. To do this, we will follow this strategy.

1. To evaluate a `letrec` special form, we first run through the variable-value pairs in the `letrec` expresion as if it was a simple `let`. In other words, we go ahead and create the closures with the *wrong* environment. We will fix this later.

2. Anything else in a `letrec` is also handled in the `let` fashion, for example

   ```
   (let ((a 2))
      (letrec ((x (+ a a)))
         (+ x x)))
   ```

   will just return 8. You can reuse your old `let` code from previous labs for this part.

3. In the course of evaluating a `let` expression, you created a mini-environment, and appended that to the current environment, to get a new environment. We will need pointers to all three of these in what follows. For this writeup, I'm going to call them the `OldEnvironment`, the `MiniEnvironment`, and the `NewEnvironment`. They are illustrated as follows:

   ```
   ((x 5) (y 10) ...)                                  <= OldEnvironment
   ((plus ...) (even? ...) (odd? ...))                 <= MiniEnvironment
   ((plus ...) (even? ...) (odd? ...) (x 5) (y 10) ...) <= NewEnvironment
   ```

4. At this point, the closures in `MiniEnvironment` contain pointers to `OldEnvironment`. We need to change these to point to `NewEnvironment`.

5. If there are any closures in `OldEnvironment`, however, they are already correct, so we don't want to change them!

6. So, we need to loop through just the variable-value pairs in `MiniEnvironment`. If any variables are bound to *closures*, we change the *saved* environment pointer inside the closure to point to `NewEnvironment`.

   Make sure you loop through only the variable-value pairs in `MiniEnviroment`. Note that `NewEnvironment` includes both `MiniEnvironment` and `OldEnvironment`. So we *don't* want to loop through all the closures in `NewEnvironment`.

7. Since we need to change a data structure that already exists, this is definitely *not* functional style programming. In fact, lists in **Racket** are immutable! So we *cannot* use lists any more to represent closures.

   Look up the documentation in **Racket** on mutable lists. You'll find procedures such as `mcar`, `mcdr`, and `mcons`, which handle mutable lists just the way `car`, `cdr` and `cons` handle immutable lists.

   But you will also find procedures such as `set-mcar!` and `set-mcdr!` for changing existing lists into new ones.

   Using mutable lists allows us to change the implementation of closures so that we can change the environment inside:

Mutable Closures

```
(define closure
  (lambda (vars body env)
    (mcons 'closure (mcons env (mcons vars body)))))
(define closure?
  (lambda (clos) (and (mpair? clos) (eq? (mcar clos) 'closure))))
(define closure-env
  (lambda (clos) (mcar (mcdr clos))))
(define closure-vars
  (lambda (clos) (mcar (mcdr (mcdr clos)))))
(define closure-body
  (lambda (clos) (mcdr (mcdr (mcdr clos)))))
(define set-closure-env!
  (lambda (clos new-env) (set-mcar! (mcdr clos) new-env)))
```

If you take the previous lab and replace the old closure implementation with this one, everything should work as before. Now aren't you glad you respected the interface in the last lab?

If your previous lab doesn't work with this new implementation of closures, fix it!

8. After we replace all the closures in `MiniEnvironment` with pointers to `NewEnvironment`, we can now evaluate the body of the `letrec` form, using `NewEnvironment`. Yay! Recursive functions!

9. Check the unit test file for some tricky examples!

   Can you believe you've written an interpreter for such a complex language?