CSCI 301, Lab # 8

Spring 2018

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

Due: Your program, named `lab08.rkt`, must be submitted to Canvas before midnight, Monday, June 4.

Unit tests: At a minimum, your program must pass the unit tests found in the file `lab08-test.rkt`. Place this file in the same folder as your program, and run it; there should be no output. Include your unit tests in your submission.

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 evaluate the `lambda` forms in the current environment, none of the three functions will be defined in that environment (or else they will have the *wrong* definitions).

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. When we evaluate the `letrec` special form, we do so in the context of some environment. Let's call this the `OldEnv`.
2. We now create a dummy environment (which can be an empty list), to use in the next step. Let's call this `DummyEnv`.
3. We now create closures in the `letrec` form using `DummyEnv`. In other words, the *saved* environment in each closure is `DummyEnv`.
4. Now create a *new* environment where the symbols in the `letrec` form (in this case, `plus`, `even?` and `odd?`) are bound to these closures.

   Do not add anything to this environment except what the variable-value combinations from the `letrec` form!

   Let's call this environment the `LetrecEnv`.
5. We now have three environments we're dealing with: `OldEnv`, `DummyEnv`, and `LetrecEnv`. Make sure you know what each of these is.
6. Create a fourth environment by appending `LetrecEnv` to the front of `OldEnv`. Let's call this one `NewEnv`.

7. Now go through the closures in `LetrecEnv` and *change* the *saved* environment in each closure to `NewEnv`. Make sure you only go through the newly created closures in `LetrecEnv`, and not through all the closures in `OldEnv` or `NewEnv`. (Note that `NewEnv` includes `OldEnv`, but `LetrecEnv` does not.)

   Consult the **Racket** documentation to see how to change a field of a structure. You will have to change the structure definition to make a mutable field:

```
(struct closure (vars body (env #:mutable)))
```

8. We can now evaluate the body of the `letrec` form, using `NewEnv`!

Some tricky examples.

```
1  (letrec ((f
2             (lambda (x) (if (= 0 x) 1 (* x (f (- x 1)))))) ; f from line 1
3     (f                                                     ; f from line 1
4        (let ((f
5                (lambda (x) (* 3 x))))
6           (let ((f
7                   (lambda (x) (f (f x)))))                ; f from line 4
8              (f 2)))))                                    ; f from line 6
```

```
1  (letrec ((f
2             (lambda (x) (if (= 0 x) 1 (+ x (f (- x 1)))))) ; f from line 1
3     (f                                                     ; f from line 1
4        (let ((f
5                (lambda (x) (f (+ x 2)))))                 ; f from line 1
6           (f                                              ; f from line 4
7              (let ((f
8                       (lambda (x) (f (+ x 3)))))          ; f from line 4
9                (f 3))))))                                 ; f from line 6
```