CSCI 301, Lab # 8

Winter 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, March 12.

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 a 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) (= 1 1) (odd? (- n 1)))))
           (odd? (lambda (n) (if (= n 0) (= 1 2) (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. Use a dummy environment (any one will do, even an empty one), and create closures using this dummy environment. In other words, the saved environment in the closure is the dummy environment.
2. Now create a new environment where the symbols (in this case, `plus`, `even?` and `odd?`) are bound to these closures. This environment *will* include the definitions of the three symbols, but those definitions will be wrong (the closures will store an incorrect environment). Let's call this environment `NewEnv`.
3. Now go through the closures you've just added to the environment, and *change* the environment stored in each closure to `NewEnv`. Make sure you only go through the newly created closures in the `letrec` form, and not through *all* closures in the environment.

Mutable data structures. Unfortunately, lists in Racket are not mutable. You cannot change, or assign new values to `car`s and `cdr`s. So, we will change the representation of closures from `list`s to `struct`s. The creation and use of a simple struct is illustrated below:

```
(struct closure (vars expr (env #:mutable)))
(define cl1 (closure '(a b) '(+ a b) '((x 1) (y 2))))
(define print-closure
  (lambda (cl)
    (display (list 'closure (closure-vars cl) (closure-expr cl) (closure-env cl)))))
(set-closure-env! cl1 NewEnv)
```

Here we create a `closure` struct with three fields, `vars`, `expr`, and `env`. The `print-closure` routine illustrates how to access each of these fields, and the `set-closure-env!` routine shows how to change the `env` field.

Some tricky examples.

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

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