CSCI 301, Lab # 5

Fall 2024

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

Submission: This lab builds on Lab 4. You will extend the functionality your Lab 4 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 `lab5-test.rkt`. Place this file in the same folder as your program, and run it; there should be no output.

Let: The special form `let` has the following syntax, with a typical example shown at right. It consists of a list beginning with the symbol `let`, then a list of symbol-value lists, and finally a single expression. You will expand your `evaluate-special-form` function to handle this case, and also add code to the `special-form?` boolean to recognize a `let` form. No other changes need be made to your interpreter.

```
(let
  ((sym1 exp1)
   (sym2 exp2)
   ...       )
  expr)
```

```
(let
  ((x (+ 2 2))
   (y x)
   (z (* 3 3)))
  (+ a x y z))
```

To evaluate this form, in an environment `e1`, we evaluate all the forms, `exp1`, `exp2`, ... in the environment `e1`. Note that this means `y` will get the value `x` had in `e1`, not 4. Let's say that in `e1` `x` had the value 10 and `a` had the value 20.

We now have a list of variables, `(x y z)`, and a list of values, `(4 10 9)`. We make a new environment by adding these variables and their values to the *front* of `e1`. Notice that adding to the front is key: the local bindings will now be found before prior definitions when searching the environment, creating the behavior where a local variable "shadows" one that is defined in an outer scope.

The single expression at the end of a `let` form is now evaluated in this *new, extended* environment. Thus, the final value will be `(+ a x y z) => (+ 20 4 10 9) => 43`. Make sure you understand this example before proceeding.

It is also important to understand that this new, extended environment, is *only* used to evaluate the `expr` embedded in the `let`

form. After the `let` form is evaluated, you go back to using the old environment. For example:

```
(let ((x 10))
     (+ (let ((x 20)) (+ x x)) x))
=>  50
```

In this example, we bind `x` to 10, then create a local environment in which `x` is bound to 20, in this new environment we evaluate `(+ x x)` to get 40, and then, outside of the new environment we evaluate `x` again, getting 10, which is added to 40 to get 50.

A slightly trickier example is this:

```
(let ((x 10))
   (+ (let ((x (+ x x))) (+ x x)) x))
```

What do you think this will evaluate to? Enter it into the Racket interpreter to see if you really understand. Your interpreter should get the same result.

In your implementation, all this tricky scoping will be handled by the fact that the extended environment in the `let` form is *only* used to evaluate the included `expr`. It should not be visible outside of handling the `let` form.