

CSCI 301, Lab # 7

Spring 2018

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

Due: Your program, named `lab07.rkt`, must be submitted to Canvas before midnight, Monday, May 28.

Unit tests: At a minimum, your program must pass the unit tests found in the file `lab07-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.

Lambda creates a closure: If you consider a simple application of a `lambda` form, it's a lot like `let`, that you did last week:

```
((lambda (x y) (+ x y)) 5 7)    <=>    (let ((x 5) (y 7)) (+ x y))
```

In each case, the list of variables, `(x y)`, is bound to the list of values `(5 7)`, this is used to extend the environment, and then the expression, `(+ x y)`, is evaluated in the extended environment. If this were the only use of `lambda`, then it would just be a different syntax for `let`.

However, the `lambda` is actually more powerful. A `lambda` can be created in one environment, and applied in another. Consider the following code:

```
(let ((a 10)
      (f (let ((a 20)) (lambda (x) (+ x a))))))
(f 30)
```

Is the answer `30 + 20` or `30 + 10`? You can check it out with Racket, but hopefully you can see that it *should* be `30 + 20`. When the `lambda` form was *evaluated*, `a` was bound to 20. When the resulting function, bound to `f`, was *applied*, `a` was 10.

The solution to this is that a `lambda` form creates a special data structure called a *closure*. The closure is what is bound to `f`, in the above example. A closure consists of three things:

1. A list of symbols (the variables to be bound when it is called).
2. An expression to be evaluated.
3. A stored environment, the environment in which the closure was *created*.

The first two are easy to understand. For example, in the `lambda` form: `(lambda (x y) (+ x y))`, the list of symbols is `(x y)` and the expression is `(+ x y)`. The third thing is the environment that was in effect when the closure was created. In other words, when the `lambda` form was *evaluated*.

We will represent closures as *structures* containing the three items. We declare such a structure in **Racket** like this:

```
(struct closure (vars body env))
```

Note that structures do not print nicely in **Racket**, but we can make a utility to see them like this:

```
(define closure->list
  (lambda (cl)
    (if (not (closure? cl))
        (error "Not a closure:" cl)
        (list 'closure
              (closure-vars cl)
              (closure-body cl)
              (closure-env cl)))))
```

For example, if the environment `e1` is

```
((x . 5) (y . 8) (z . 10))
```

then the following closures would look like this:

```
(closure->list (evaluate '(lambda (x) (+ x y)) e1))
=> (closure (x) (+ x y) ((x . 5) (y . 8) (z . 10)))
(closure->list (evaluate '(lambda (a b c) (cons a (list b c))) e1))
=> (closure (a b c) (cons a (list b c)) ((x . 5) (y . 8) (z . 10)))
(closure->list (evaluate '(let ((x 10)) (lambda (foo) (+ foo foo)) e1))
=> (closure (foo) (+ foo foo) ((x . 10) (x . 5) (y . 8) (z . 10))))
```

Note that the closure remembers the environment in which it was created, even if that environment was a special environment, for example, like the one created by a `let` form.

Applying lambda to some arguments: Now that we know how to evaluate a `lambda` form, we need to know how to *apply* a lambda form to some arguments. For example, consider the form

```
((lambda (x y) (+ x y)) 10 20)
```

This is *not* a `lambda` form. It is the *application* of the `lambda` form `(lambda (x y) (+ x y))` to the arguments, `(10 20)`. If we follow our rules for evaluating things that are not special forms, we would evaluate each of the items in the list (the `lambda` form, the 10 and the 20), getting this list (printing the closure as a list):

((lambda (x y) (+ x y))	10	20)
	↓	↓	↓	
((closure (x y) (+ x y) ((x . 5) (y . 8) (z . 10)))	10	20)

And now we have to apply the closure to the list of arguments. Up to now, we've only been applying Racket built-in functions to their arguments, using the Racket `apply` function. However, Racket doesn't understand our closures. We have to figure out how to apply them ourselves.

So, we will add a new `apply-function` procedure to our interpreter. When evaluating a normal (non-special-form) list, we evaluate each item in the list, and then call `apply-function` with just two arguments: the `car` and the `cdr` of our evaluated list of items, the function and its arguments.

`apply-function` will look at its first argument. If it is a `procedure?`, then it calls the Racket built-in `apply` function. If it is a `closure?`, it calls `apply-closure`. Otherwise it should throw an error, reporting an unknown function type.

`apply-closure` takes two arguments, a closure and a list of values. The closure has three components: the variables, the expression, and a saved environment. This procedure extends the *saved* environment by appending the variables and their values to the front (just like `let` did), and then evaluates the expression of the special form in this environment.

Note that `let` is not the only form that introduces a local environment, now `lambda` does, too. Consider the following code:

```
(let ((f (lambda (a) (lambda (b) (+ a b)))))
  (let ((g (f 10))
        (h (f 20)))
    (list (g 100) (h 100))))
```

Here, the result of applying `f` to 10 creates a closure in an environment in which `a` is bound to 10. When we apply `f` to 20, we create a closure in an environment in which `a` is bound to 20. What should be the result of the call? Does your interpreter get that result?

Note also that we can't define recursive functions directly using `let`. What would happen if we tried this?

```
(let ((f
      (lambda (n)
        (if (= n 0)
            1
            (* n (f (- n 1)))))))
  (f 5))
```

Try this in the **Racket** interpreter, and also in your interpreter. In our next (and final) lab, we will define the `letrec` special form to remedy this.

However, we don't really need to wait for that to define recursive functions. We just have to be a little more clever. For example, what would the following do in **Racket**? What does it do in your interpreter?

```
(let ((f
      (lambda (f n)
        (if (= n 0)
            1
            (* n (f f (- n 1)))))))
  (f f 5))
```