Evaluating and Calling Lambda Expressions

First, some review.

How do we parse an application such as (+ 2 3)??

Answer: we parse an application such as (+ 2 3) into an app-exp that has two fields

A. The parsed function part. In (+ 2 3) + is the function part. In all applications the function part is the car of the input.
B. The list of parsed arguments.

Next, how do we evaluate an app-exp?

Answer: We evaluate (app-exp parsed-proc list-of-parsed-args) by calling apply-proc with the evaluated procedure part and a list of evaluated arguments.

The first part of Lab 7 deals with implementing lambda expressions. Much of the work is already done:

A. First of all, parse a lambda expression such as (lambda (x y z) body) into a lambda-exp structure (that is a new tree) with two fields: the parameter list (x y z) and the parsed body. Note that the parameter list is not parsed.

B. We need to evaluate a lambda-exp. What could a lambda expression possibly evaluate to?

Everybody knows that a lambda expression evaluates to a closure. So we need yet another data type. This should have constructor

new-closure

recognizer

closure?

and getters for each of the three fields:

parameter list

parsed body

closure-environment

The first two of these come directly from the lambda-exp. The closure environment is the argument env from the eval-exp procedures.

So parsing and evaluating lambda expressions is easy and should take you about 5 minutes to implement.

We don't have to do anything about parsing calls to lambda expressions; that is what our app-exp already does.

We still need to think about evaluating calls to lambda expressions.

We said that all applications are evaluated by calling apply-proc with the evaluated procedure and the list of evaluated arguments.

Here is what apply-proc should look like after Lab 6:

```
(define apply-proc (lambda (p args)
        (cond
                [(prim-proc? p) (apply-primitive-proc .....)]
                [else (error ....)])))
```

```
(define apply-proc (lambda (p args)
        (cond
                [(prim-proc? p) (apply-primitive-proc .....)]
                [else (error ....)])))
```

To extend this to apply to calls of lambda expressions we add a middle line:

```
                [(closure? p) (.....)]
```

Remember that apply-proc is called with the argumengts already evaluated. To handle the application, we recursively call eval-exp on the body (which is one of the parts of the closure) in the environments we get by extending the closure environment with bindings of the closure symbols to the arguments.

This is just one line of code (maybe a longish line) and you have all of the pieces. If you find yourself wanting to pass the environment env from eval-exp to apply-proc there is something wrong; you shouln't need to do that.

Example: ((lambda (x y) (+ x y)) 3 5)

We parse this into an app-exp, where the procedure part
is (lambda-exp (x y) (app-exp (var-ref +) ((var-ref x) (var-ref y)))
and the arg list is ((lit-exp 3) (lit-exp 5))

As with all applications, we evaluate it in environment env by calling apply-proc with the evaluated procedure and the evaluated arguments.

The procedure evaluates to

```
(closure (x y)
         (app-exp (var-ref +) ((var-ref x) (var-ref y) ))
         env)
```

Example continued ...

The args, of course, evaluate to (3 5)

The line of code we added to apply-proc says we do this application by calling eval-exp on (app-exp (var-ref +) ((var-ref x) (var-ref y)))
in the environment (extended-env (x y) (3 5) env)

Naturally, eval-exp gives us 8 as the value of this.

Another example: (let ([f (lambda (x) (* 2 x))]) (f 6))

This parses to a let-exp where the binding symbols are (f),
the parsed binding values are
((lambda-exp (x) (app-exp (var-ref *) ((lit-exp 2) (var-ref x)))))
and the parsed body is (app-exp (var-ref f) ((lit-exp 6)))

We evaluate the let-exp in environment env by calling eval-exp on the parsed body in the environment we get by extending env with a binding of f to the value of that lambda expression.

Second example, continued.

That lambda expression evaluates to a closure with parameters (x), body (app-exp (var-ref *) ((lit-exp 2) (var-ref x)))
and environment env. This closure gets bound to f in the environment in which we evaluate the let-body.

So finally, we evaluate the let-body (app-exp (var-ref f) ((lit-exp 6)))
We evaluate the function part by looking up f in the environment and getting the closure above. We call that closure with argument 6 by evaluating the closure body, (app-exp (var-ref *) ((lit-exp 2) (var-ref x))) in an environment were x is bound to 6. This, of course, gives us 12.