Thinking about the Interpreter Project

After class I will post Lab 6, the first of 3 labs that make up the interpreter project. Lab 6 just deals with environments.

In the overall project you will write 2 primary functions:

(parse exp) creates a tree structure that represents the expression *exp*.

(eval-exp tree env) evaluates the given tree structure within the given environment and returns its value.

Environments are used over and over in eval-exp to look up the value bound to a variable.

There are two functionalities we need with environments. One is that we need to be able to look up the value bound to a symbol in an environment. We want the most recent binding. For example

```
        (let ([x 3])
                (let ([x 4])
                        (+ x 5)))
```

should return 9, since the innermost binding of x is to 4, not 3.

The second functionality for environments is that we need to be able to extend an environment, producing a new environment (rather than modifying the old one). For example

```
(let ([x 3])
    (begin
        (let ([x 4])
            (+ x 1))
        (+ x 6)))
```

This evaluates to 9. The outermost let extends the top-level environment with a binding of x to 3; let's call this extended environment E1. The innermost let extends E1 with a binding of x to 4, to make environment E2, but the last line (+ x 6) is evaluated in E1, not in E2.

There are lots of options for structuring an environment. To choose one it helps to think about when we need to extend our environment.

There are only 2 places where an environment is extended.

One of these is a function *call*. To evaluate

(proc exp1 exp2 exp 3)

proc should evaluate into a closure with 3 parts: the procedure's parameter list, its body, and the environment in which it was created. The three arguments exp1 exp2 and exp3 are evaluated and the closure environment is extended with bindings of the closure parameters to the argument values.

In other words, in this situation we get a list of parameters, perhaps (x y z), and a list of values, perhaps (3 6 9), and we want to extend the old environment with bindings of each parameter to the corresponding value: x to 3, y to 6, z to 9.

The other situation where we extend the environment is when we evaluate a let expression. Consider the expression

```
        (let ([x (+ 3 4)] [y 5] [z (- 6 6)])   body )
```

It is easy to find the binding list; that is just cadr of the whole expression. The symbols being bound are the first elements of the binding list: (map car binding_list). The binding expressions are (map cadr binding_list); we need to evaluate them in some environment env:

```
        (map (lambda (x) (eval-exp x env) (map cadr binding_list)))
```

Again, we can easily get a list of symbols and a list of values being bound to them.

So it seems that every time we want to extend the environment we have a list of symbols, such as (x y z) and a list of values to bind to them, such as (3 6 9).

This gives us an easy way to make an environment datatype. We will represent an empty environment with the list ('empty-env) and an extended environment with the list

```
        ('extended-env symbols values old-env)
```

Of course, if we try to lookup the value of x in an empty environment, we know it isn't there (so we throw an error).

If we try to lookup the value of x in an extended environment where we have a list syms of symbols and a list vals of values and an old environment, we compare x to the first symbol; if they are equal we return the first value, otherwise we recurse on the cdrs of the syms list and vals list. If we get down to the syms and vals lists both being empty, we recursively look up x in the old environment. Eventually we either find x or get to an empty environment, where we throw an error.

Lab 5 asks you to implement the environment datatype (which includes the extended-env procedure for extending an environment), and a lookup procedure (lookup environment sym) that returns the value bound to *sym* in the environment..

This is the shortest lab of the semester. It will be formally due after Fall Break, but if you get it done before Break you can start on the more interesting parts of the interpreter right after Break.