Conditional and Let Expressions

Before handling conditional expressions we need to talk about Boolean values. The Boolean values in Scheme are #t and #f. The Boolean values in MiniScheme are True and False. #t and #f are not part of MiniScheme. If you evaluate (equals? 5 (+ 2 3)) you should get answer True, not #t.

To make this happen you should extend the environment once again to bind symbols True and False to values True and False. If you want to know if a condition is true you should evaluate it and ask if its value is 'True.

Other than a bunch of primitive procedues, the thing you need to implement for part D of Lab 6 is the if-expression.

Our grammar now looks like this:

```
EXP ::- NUMBER        ; parse into lit-exp
        | SYMBOL      ; parse into var-ref
        | (if  EXP  EXP  EXP); parse into an if-exp
        | (EXP EXP*)  ; parse into app-exp
```

Of course, this means we need to build a new data type if-exp with constructor *new-if-exp*, recognizer *if-exp?*, and 3 getters: one for the condition, one for the true branch and one for the false branch.

The parser looks like this:

```
(define parse (lambda (input)
                (cond
                    [(number? input) (new-lit-exp input)]
                    [(symbol? input) (new-var-ref input)]
                    [(not (pair?  input)) (error 'parse "Bad syntax ~s" input)]
                    [(eq? (car input) 'if)  (new-if-exp ......)]
                    [else (new-app-exp ..............)])))
```

Think about what a typical if-expression looks like. parse's argument *input* will be something like '(if (< x 1) (+ x 100) x)

The condition of this expression is (< x 1); that is (cadr input).
The true branch is (+ x 100); t hat is (caddr input).
The false branch is x, which is (cadddr input).

The parser just grabs these three pieces of the input, parses them, and sticks them in an if-exp datatype.

eval-exp starts working on an if-exp by evaluating the condition (which is parsed and ready to evaluate). Remember that your interpreter is written in Scheme and you can take advantage of that. Your code for this line of (eval-exp tree env) might say something like

```
(let([cond (eval-exp (if-cond tree) env)])
        (if (or (eq? cond 'False) (= cond 0)).....))
```

(note: in this code I used *if-cond* as the getter for the if-exp data type.

Lab6 says that if the condition evaluates to False or 0 we take the false-branch of the expression; otherwise we take t he true-branch. To "take a branch" just return the result of calling eval-exp on the appropriate parsed expression taken from the if-exp tree.

Note that it is very important that you not implement the if-expression by evaluating all three parts and then returning the appropriate one depending on whether the condition is True or False. If you do the latter recursion can't work; all recursions would be infinite.

The last part of Lab 6 involves implementing let-expressions. That is easier than you might think.

Think of what a let-expression looks like. Perhaps

```
(let ([A 3][B 5]) (* A B))
```

To evaluate this we need to extend the environment with bindings of A to 3 and B to 5, and then evaluate (* A B) in this new environment.

Our extended-env new environment constructor wants a list of binding symbols, a list of binding values and the old environment being extended.

Our parser doesn't know anything about environments, but it can handle the rest. We need a let-exp datatype that stores 3 fields:

a) the binding symbols

b) the parsed binding values (the parser parses anything it can)

c) the parsed body

Suppose input is the expression '(let ([A 3][B 5]) (* A B)).
The binding list is ([A 3] [B 5]). We get this as the second element (cadr) of the input. We need the list of binding symbols, (A B). This is the first element of each pair, (map car (cadr input)). Note that we just need the list of symbols; we don't parse them.

Next, we need the parsed list of binding values. The binding value expressions are derived just like the symbols: (map cadr (cadr input)). We need to parse them. That just means mapping parse onto (map cadr (cadr input))

Finally, we need to parse the let body. This is the third element, or caddr, of the input.

Altogether, to parse a let-expression we build a let-exp with

a) the list of (unparsed) binding symbols
b) the list of (parsed) binding values
c) the parsed body.

For (eval-exp tree env) when tree is a let-exp, we recursively call eval-exp on the parsed body, in the environment we get from extending env with the binding symbols bound to the evaluated binding values. We have all of this ready to go, except that we have parsed binding values instead of evaluated binding values. If *pv* is any one of those parsed values we could evaluate it by calling (eval-exp pv env). We have a whole list of parsed values. How could we possibly call a function on each entry of a list and make a list of the results??? It seems like something we ought to be able to do .....