Free and Bound Variables

If we didn't have to deal with state, an interpreter for Scheme expressions would be easy.

The idea is this:
to evaluate ((lambda (x) body) e) we recursively evaluate e to get value a, then evaluate body with variable x replaced by a.

To evaluate ((let ([x e]) body) we again recursively evaluate e to get value a, then evaluate body with variable x replaced by a.

We won't take this all the way here, but we will write a few of the Scheme functions we will need to do this formally.

First, we need the notions of *free* and *bound* variables.

We say that variable x occurs free in expression E if

- E is the variable x
- E has the form (lambda (params) E'), x is not one of the params, and x occurs free in E'.
- E has the form (let (binding list) E'), x is not one of the variables from the binding list, and x occurs free in E'
- E has the form (E_1 E_2 ... E_k) and x is free in any of the E_i.

We can write a procedure that tests for this:

```
(define occurs-free?
        (lambda (x E)
                (cond
                        [(null? E) #f]
                        [(atom? E) (eq? x E)]
                        [(eq? (car E) 'lambda)
                                (and (not (member x (bindings E)))
                                        (occurs-free? x (body E)))]
                         [(eq? (car E) 'let)
                                (and (not (member x (let-bindings E)))
                                        (occurs-free? x (body E)))]
                        [else (or (occurs-free? x (car E))
                                (occurs-free? x (cdr E)))])))
```

Here **bindings, body** and **let-bindings** are simple functions that pull apart the elements of a lambda-expression or a let-expression.

We say that variable x occurs bound in an expression E of

- E has the form (lambda (params) E') and x is one of the params and x occurs free in E'
- E has the form (lambda (params) E') and x occurs bound in E'
- E has the form (let (bindings) E') and either x is one of the params and x occurs free in E' or else x occurs bound in E'
- E has the form (E_1 E_2 ... E_k) and x occurs bound in any of the E_i.

We can write this in Scheme as well:

```
(define occurs-bound?
        (lambda (x E)
                (cond
                        [(null? E) #f]
                        [(atom? E) #f]
                        [(eq? (car E) 'lambda)
                                (or  (and (member x (bindings E))
                                          (occurs-free? x (body E)))
                                     (occurs-bound? x (body E)))]
                        [(eq? (car E) 'let)
                                (or (and (member x (let-bindings E))
                                         (occurs-free? x (body E)))
                                    (occurs-bound? x (body E)))]
                        [else (or (occurs-bound? x (car E))
                                  (occurs-bound? x (cdr E)))])))
```

For example

(occurs-free? 'x '(lambda (y) (+ x y))) returns #t

(occurs-bound? 'y '(lambda (y) (+ x y)) returns #t

The idea behind a substitution interpreter is to evaluate the expression (let ([x a] [y b] body) by replacing the free occurrences of x and y in body with a and b, and evaluating the result. We do the same thing with
((lambda (x y) body) a b)

The following function does the substitution

First we handle the case of just one variable:

```
(define substitute-for-free (lambda (a x E)
    (cond
        [(null? E) null]
        [(atom? E) (if (eq? E x) a E)]
        [(eq? (car E) 'lambda )
            (if (occurs-free? x E)
                (list 'lambda (bindings E) (substitute-for-free a x
                                                        (body E)))
                E)]
       [(eq? (car E) 'let)
            (if (occurs-free x E)
                (list 'let (bindings E)  (substitute-for-free a x
                                                        (body E)))
                E)]
      [else (cons (substitute-for-free a x (car E))
                  (substitute-for-free a x (cdr E)))])))
```

Then we do this for extended binding lists:

```
(define substitute-many-for-free
    (lambda (values syms E)
        (cond
            [(null? values) E]
            [else (substitute-for-free (car values) (car syms)
                        (substitute-many-for-free (cdr values) (cdr syms) E))])))
```

For example
(substitute-many-for-free (2 5) (x y) (* (+ x 3) y) returns (* (+ 2 3) 5)

The procedure names here are long, but this is easy coding. If we had a good way to represent procedures we could easily extend this into a full interpreter of the functional parts of Scheme. But then there is set! .....