Compare these very similar functions:

```
(define index
     (let ([inc (lambda (x) (if (= -1 x) -1 (+ 1 x)))])
          (lambda (a lat)
               (cond
                         [(null? lat) -1]
                         [(eq? a (car lat)) 0]
                         [else (inc (index a (cdr lat)))]))))

(define index2  (lambda (a lat)
     (let ([inc (lambda (x) (if (= -1 x) -1 (+ 1 x)))])
               (cond
                         [(null? lat) -1]
                         [(eq? a (car lat)) 0]
                         [else (inc (index2 a (cdr lat)))]))))
```

index and index2 are both recursive procedures. The closure environment for index contains a binding for procedure inc. The closure environment for index2 is the top-level. Each time we call index2 its body, containing the let-binding for inc, has to be evaluated. This doesn't happen when we call index, so index is more efficient than index2.

index and index2 are both recursive functions that use a let-block to define a non-recursive helper function inc.

You can't create recursive functions within let bindings:

The following doesn't work:

```
(let ([f (lambda (x) (if (= x 1) 1 (* x (f (- x 1)))))])
  (f 3))
```

just gives you an error message saying f is unbound.
It is easy to see why: the let expression evaluates its bindings in the current (top-level in this case) environment. This is the environment for the closure bound to f. When we evaluate the body of f in the extended environment (extended with a binding for x) we can't do the recursive call because f isn't defined in this extended environment.

There is a second version of let that handles this:

```
(letrec (bindings) body)
```

works just like (let (bindings) body) only the binding expressions are evaluated in an environment that includes the binding symbols, so recursion works.

Here is another problem with let, and another variation to solve it.

The following code makes sense, but doesn't work:

```
(let ([x 3] [y x]) y)
```

If x is bound to 3 and y is bound to the value of x, y should also be bound to 3. However, let evaluates its bindings in an environment that doesn't include the binding symbols, so we get an error on the second binding [y x].

let* solves this problem by taking the bindings one at a time:

```
          (let* ([sym1 exp1][sym2 exp2] ...) body)
```

is equivalent to

```
          (let ([sym1 exp1])
                    (let ([sym2 exp2])
                              ....


                              body)))))
```

In other words, each binding is evaluated in an environment that includes all of the previous bindings.

```
(let* ([x 3][y x]) y)
```

is equivalent to

```
(let ([x  3] )
      (let ([y x])
            y))
```

which evaluates to 3.

Note that the let-expression is unnecessary. Consider the following example:

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

This is completely equivalent to

```
( (lambda (x y) (+ x y)) 3 4)
```

To evaluate either expression we create a new environment, which is the current environment extended to have bindings of x to 3 and y to 4, and evaluate the expression (+ x y) in this environment.

In fact, any let-expression

```
(let
        ([x1 exp1]
         [x2  exp2]
         [x3  exp3]
                ...
        [xn  expn])
    body)
```

is equivalent to

```
((lambda (x1 x2 ... xn) body) exp1 exp2 ... expn)
```

When we write an interpreter for Scheme, one option will be to translate let expressions into the corresponding lambda expressions and use our interpreter for the latter.