More About Scheme:
closures, environments, set! and let

There is a global environment that binds symbols to values.

The expression
        (define foo bar)
creates (or changes) the binding of symbol foo to the value of bar.

Consider this example:
        (define f (lambda (x) (+ x 1)))
        (define g (lambda (x) (f (* x 5))))
        (g 2) => 11

        (define f (lambda (x) 23)
        Now (g 2) => 23

Moral: Be careful how you define things. (define foo bar)
can change things other than foo.

Procedures in Scheme are "first-class values". This means we can use procedures like any other data value -- they can be arguments to calls, or the return values of other procedures.   To understand this we need to be very explicit about how procedure calls work in Scheme.

When we evaluate a list such as (A B C) each of the elements is evaluated in the current environment, the first element should evaluate to a procedure, and that procedure is called with the values of the other elements of the list. For the call a new environment is created in which the parameters of the procedure are bound to the values of the arguments, and the body of the procedure is evaluated in this environment.

Example:
	(define foo (lambda (x y) (+ (* x 2) y)))
	(define bar (lambda (x) (foo 3 x)))
	Evaluate (bar 7) in the top-level environment.

	The top-level environment has symbols foo and bar bound
	to the values of their procedures.

	We evaluate bar and get #<procedure bar>; we evaluate 7
	and get 7. So we call #<procedure bar> with value 7. A new
	environment is created with x, the parameter of bar, bound
	to 7. We evaluate the body of bar: (foo 3 x) in this
	environment. Another environment is created with
	bindings for the parameters of foo: x is bound to 3 and y is
	bound to 7. We evaluate (+ (* x 2) y) in this environment
	and get 13.

To make this possible, procedures need to carry their environments with them. The value of a procedure consists of three things:

a) The parameter list of the procedure.
b) The environment in which the procedure was created.
c) The body of the procedure.

This is called a *closure*. When we apply the procedure to arguments, the environment of its closure is extended to include bindings of the parameters to the values of the arguments, and the body is evaluated within this extended environment.

Here is another example:

(define fo (lambda (x) (lambda (y) (+ x y))))
Evaluate ((fo 3) 4) in the top-level environment.

The top-level environment binds fo to its closure. The environment for #<procedure fo> is the top-level. The first thing we do is evaluate (fo 3). This extends fo's environment (the top level) with a binding for x to 3. We then evaluate fo's body: (lambda (y) (+ x y)) in this environment. This evaluates to a closure with parameter list (y), environment {x->3}, and body (+ x y). So that is the value of (fo 3). We then apply this closure to the argument 4. To do this we extend the closure's environment to include a binding of its parameter y to 4: the new environment is {x->3, y-> 4}. We evaluate (+ x y) in this environment and get 7.

Note the similarity and differences between the following functions:

(define f (lambda (x y) (+ x y)))
(define fo (lambda (x) (lambda (y) (+ x y))))

They are similar, since (f 3 4) is 7 and ((fo 3) 4) is 7.
But they are also different since f is a function of two variables and foo is a function of 1.   You can think of (fo 3) as the result of freezing the first argument of f as x=3, leaving a function of just variable y.   This is called *currying* f (named after Haskell Curry, a mathematician at Penn State from 1929 to 1965 who studied such things. Mr. Curry, by the way, was so important to the foundations of programming languages that long after his death the Haskell language was named after him).

Here are some new Scheme expressions: begin, set! and let.

(begin e1 e2 e3 e4 ... en)
This evaluates each of the expressions, returning the result of the last one.

(begin (+ 3 4) (* 5 6)) => 30

Of course, this is a bit silly until we get expressions that have side-effects.

(set! x e)   (pronounced "set bang")
This changes the binding of symbol x in the current
environment to the value of e. Symbol x must already be
bound in the environment.

This gives us side-effects:

(define x 0)
(begin (set! x 5) (* x 3) => 15

set! causes all sorts of problems that we will talk about later. As much as possible we will program functionally -- without set!

```
(let
        ([sym1  exp1]
         [sym2  exp2]
         [sym3  exp3]
            ...
         [symn  expn])
   body)
```

This evaluates the binding values exp1, exp2, etc. in the current environment. A new environment is created extending the current environment with bindings for each of the symbols sym1, sym2, ... to the value of its expression. The value of the let expression is the value of its body evaluated within this new environment. The body may be either a single expression or a sequence of expressions. In the latter case each of these is evaluated and the value of the last of them is returned.

Examples:
```
(let ([x 3] [y 4]) (+ x y))   => 7


(let
    ([f  (lambda (x) (+ x 1))]
     [g  (lambda (x) (* 2 x))] )
   (f (g 3) )  => 7

(define sumSquares
      (let ([sq (lambda (x) (* x x))])
           (lambda (lat)
               (cond
                    [(null? lat) 0]
                    [else (+ (sq (car lat)) (sumSquares (cdr lat)))]))))
```

Compare these very similar functions. Both produce the correct sum of the squares of the numbers in lat. If lat is (1 2 3) both return 14.

```
(define sumSquares
      (let ([sq (lambda (x) (* x x))])
          (lambda (lat)
              (cond
                  [(null? lat) 0]
                  [else (+ (sq (car lat)) (sumSquares (cdr lat)))]))))

(define sumSquares2 (lambda (lat)
      (let ([sq (lambda (x) (* x x))])
              (cond
                  [(null? lat) 0]
                  [else (+ (sq (car lat)) (sumSquares2 (cdr lat)))]))))
```

sumSquares and sumSquares2 are both recursive procedures. The closure environment for sumSquares contains a binding for procedure sq. The closure environment for sumSquares2 is the top-level. Each time we call sumSquares2 its body, containing the let-binding for sq, has to be evaluated. This doesn't happen when we call sumSquares, so sumSquares is more efficient than sumSquares2.