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

sumSquares is a recursive function that uses a let-block to define a non-recursive helper function sq.

You can't create recursive functions within let bindings:
The following doesn't work:

(let ([f (lambda (x) (if (= x 0) 1 (* x (f (- x 1)))))])
  (f 3))

This 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. There is a requirement that it must be possible to evaluate the binding values without knowing the values of the binding variables. This is not a problem, since we usually use a letrec expression to bind a recursive procedure to a symbol. The value of the procedure is a closure; we don't need the value of the symbol it is bound to until the procedure is called.
Here is another problem with let, and another variation to solve this problem:

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

$$(\text{let } ([x \ 3] \ [y \ x]) \ y)$$

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 all of its bindings in an environment that doesn't include the binding symbols, so we get an error on the second binding $[y \ x]$. 
Note that letrec doesn't help here:

(letrec ([x 3] [y x]) y)

requires us to know the value of x in order to add the binding [y x] to the environment.

To be honest, if you type (letrec ([x 3][y x] y) into Dr. Racket it prints 3, as you would expect. But a slightly more complicated example:

(letrec ([x 3] [y x]) (begin (set! x 47) y)) crashes.
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:

\[
\text{(let ([x 3] [y 4])}
\]
\[
\hspace{1cm} (+ x y))
\]

This is completely equivalent to

\[
\text{( (lambda (x y) (+ x y)) 3 4)}
\]

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
(\[x1 exp1] [x2 exp2] [x3 exp3] ...
[\[x n \ exp n\])
body)

is equivalent to
((lambda (x1 x2 ... x n) body) exp1 exp2 ... exp n)

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.