Control and Recursion
A *control structure* is a programming structure that determines a particular sequence of operations.

In Java if-statements, while-loops and function calls are examples of control structures.

In Scheme if and begin expressions as well as application expressions represent control structures.
Whether or not a language definition specifies the order of execution of statements, any particular implementation of the language will force certain sequences of operations because the language is executed on sequential hardware.

For example, Scheme does not determine the order of evaluation of the arguments to a function. We can see the order of evaluation through games like this:

```scheme
( (lambda (a b c) a)
    (begin (display "a") 1)
    (begin (display "b") 2)
    (begin (display "c") 3))
```
One very interesting control structure is the procedure call, particularly with regard to recursive functions.

Consider

```scheme
(define f
  (lambda (x)
    (let ([y 23] [z 45])
      (g (+ x y))))

(define g
  (lambda (x)
    (let ([y 0] [z 2])
      (h x y z))))

(define h
  (lambda (a b c) (+ a b c)))
```
Suppose we call (f 3).
(f 3) calls (g 26 )
which in turn calls h(26 0 2)
which returns 28

Each function's environment must be available during its execution. Whether this is done through environment lists or on the runtime stack, all of the bindings must be available.
Here is what the stack looks like during the call to h:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h's c</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>h's b</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h's a</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>g's z</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>g's y</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>g's x</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>f's z</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>f's y</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>f's x</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

If instead of this we pass an environment around, these bindings are constructed there.
Either way, the environment grows as we call functions from within functions.

There is a natural bound on how deeply nested we call functions within functions since each function needs its own code. To nest 1000 function calls without recursion we would need 1000 functions.
However, with recursive function calls there is no natural limit. The environment with recursive functions grows with each successive call. We can stack up 10,000 calls with just one recursive function.

For example, consider

```lisp
(define fact (lambda (x)
    (if (= 0 x) 1
        (let ([a (fact (- x 1))])
            (* x a)))))
```
If we call (f 5) the stack looks like

- x for (fact 0) 0
- x for (fact 1) 1
- a for (fact 1)
- x for (fact 2) 2
- a for (fact 2)
- x for (fact 3) 3
- a for (fact 3)
- x for (fact 4) 4
- a for (fact 4)
- x for (fact 5) 5
- a for (fact 5)

If we call (f 10) the stack will be twice as large; if we call (f 1000) it will be much, much larger.
Now consider a new definition of a factorial function:

(define facta (lambda (n)
             (fact-acc n 1)))

(define fact-acc (lambda (n acc)
            (if (= 0 n)
                acc
                (fact-acc (- n 1) (* n acc)))))
Now imagine the stack as we compute \((\text{facta } 5)\):

\[
\begin{align*}
n &\text{ for } \text{fact-acc}(0, 120) & 0 \\
\text{acc} &\text{ for } \text{fact-acc}(0, 120) & 120 \\
n &\text{ for } \text{fact-acc}(1, 120) & 1 \\
\text{acc} &\text{ for } \text{fact-acc}(1, 120) & 120 \\
n &\text{ for } \text{fact-acc}(2, 60) & 2 \\
\text{acc} &\text{ for } \text{fact-acc}(2, 60) & 60 \\
n &\text{ for } \text{fact-acc}(3, 20) & 3 \\
\text{acc} &\text{ for } \text{fact-acc}(3, 20) & 20 \\
n &\text{ for } \text{fact-acc}(4, 5) & 4 \\
\text{acc} &\text{ for } \text{fact-acc}(4, 5) & 5 \\
n &\text{ for } \text{fact-acc}(5, 1) & 5 \\
\text{acc} &\text{ for } \text{fact-acc}(5, 1) & 1 \\
n &\text{ for } \text{facta}(5) & 5
\end{align*}
\]
At first glance this looks the same as the stack for fact, but examine the code for fact-acc more carefully:

```
(define fact-acc (lambda (n acc)
    (if (= 0 n)
        acc
        (fact-acc (- n 1) (* n acc))))
```

The very last thing it does is to recurse. This means it can't make any use of its environment during or after the recursion and we don't need to keep the environment around for the recursive calls. All that we need are

n for fact-acc
acc for fact-acc
What is the difference between fact and fact-acc? fact-acc is *tail recursive*. A tail-recursive function is one that recurses as its last step. The alternative to tail recursion is *deep-recursion*.

A *properly tail recursive* implementation of a language is one that does not stack tail-recursive calls -- the current environment is modified instead of stacking new environment. When this happens the tail recursion can be replaced by a loop.

One of the main design feature of the Scheme language was to make it properly tail recursive -- all tail recursions execute like loops.