Introducing Scheme
A Scheme expression (S-exp) is either an *atom* or a *list*.

An atom can be

- A symbol
- A number
- #t or #f (Boolean values)
- A string (which we will seldom use)

A list is either

- null, which represents the empty list
- or a pair consisting of a head and a tail. The head must be an S-exp (so it could be an atom or a list) and the tail must be a list.
The Scheme interpreter evaluates Scheme expressions.

• The value of an atom: a number, #t, #f, or a string, is the atom itself.
• The value of a symbol is the value bound to it.
• The value of a null list is null.
• The value of a non-null list depends on the head of the list. If the head is one of a specific set of symbols: define, lambda, let, letrec, set!, etc, then the list represents a special form. Each special form has its own way of being evaluated.

(More on the next slide)
• If the non-null list is not a special form its values is the result of calling the head of the list as a procedure with the tail of the list as its arguments. For example, the value of
  
  \((+ 3 4)\)
  
is 7.
• Note that we can't evaluate the list \((1 2 3)\) because it is not a special form and the head of this list, 1, is not a procedure.
The quote ' is used to prevent evaluation.

'(1 2 3) evaluates to the list (1 2 3)
Basic procedures for working with lists:

- **car** (Contents of the Address part of a Register on an old IBM 704) (This is pronounced "car", like an automobile)
- **cdr** (Contents of the Decrement part of a Register on that 704) (pronounced "could - er"; rhymes with "should stir")
- **cons** (Construct)

By the way, the IBM 704, introduced in 1954, was the first commercial computer with floating-point hardware. Transistors were just being invented; the 704 was a vacuum tube computer.
Here are the Scheme meanings of the list procedures:

- (cons a b) creates a new pair, where a is the head and b is the tail. If b is a list, this makes a new list.

  \( \text{(cons a b) creates a cons-box } \begin{array}{c} a \\ b \end{array} \) \( \text{Note that (1 2) is the same as (list 1 2) and as (cons 1 (cons 2 null))} \)

- (car x) is an error unless x is a pair, in which case (car x) is the head of x. So (car (cons 'a 'b)) is 'a.

- (cdr x) is an error unless x is a cons-box, in which case (cdr x) is the tail of x. So (cdr (cons 'a 'b)) is 'b.
Procedures always evaluate their arguments before performing their actions:

(car (1 2 3)) is an error because the argument (1 2 3) can't be evaluated.

(car '(1 2 3)) performs the car procedure on the value of the argument, which is the list (1 2 3). The head of this list is 1, so (car '(1 2 3)) => 1.
Examples

• (cons 3 null) => (3)
• (cons 2 (cons 3 null)) => (2 3)
• (cons '(1 2) '(3 4)) => ((1 2) 3 4)
• (car '((1 2) (3 4 5 6))) => (1 2)
• (cdr '(1 2 3)) => (2 3)
cadr is shorthand for "car of the cdr"

\[
\text{(cadr '(1 2 3 4)) => (car (cdr '(1 2 3 4)))}
\]
\[
\Rightarrow (\text{car '(2 3 4))}
\]
\[
\Rightarrow 2
\]

In other words (cadr x) is the second element of list.

Similarly there are procedures caddr, cadddr, etc.
define changes the global environment by binding a value to a symbol.

e.g. (define Beatles '(john paul george ringo))
    (car Beatles) => 'john

The most common things to define are procedures.
Procedures are created with lambda-expressions, following the pattern
(lambda (parameter-list) body)

e.g.
(lambda (x) (+ x 5))
as in  ( (lambda (x) (+ x 5)) 6)  => 11

or
(lambda () 23)

For example

(define f (lambda (x y) (+ (* x y) 1)))
(define square (lambda (x) (* x x)))
The expression \( \text{lambda (parameters) body} \) evaluates to a \textit{closure}. A closure consists of three parts:

a) the parameter list  
b) the body as an un-evaluated expression  
c) the environment at the time the lambda expression is evaluated.

Suppose the lambda expression has parameters \((x \ y)\) and it is called with arguments \((a \ b)\). This call is evaluated by extending the closure’s environment with a binding of variable \(x\) to value \(a\), and a binding of \(y\) to value \(b\), and then evaluating the closure’s body in this new environment.
Note that define binds symbols to values, not to expressions.

(define f (lambda (x) (+ x 1)))
(define bob (f 0))
(define f (lambda (x) (* x x)))

bob => 1, not 0
There are two *conditional* expressions:

\[(\text{if } <\text{test}> \ <\text{test-true exp}> \ <\text{test-false exp}>)\]

and

\[(\text{cond}

\[\text{[test1 exp1]}

\[\text{[test2 exp2]}

\[\text{...} \ ]\]

\text{You can use the symbol *else* for the final test.}

Note that the square bracket is just an alternative parenthesis.

E.g.

\[(\text{if } (< 1 2) 3 4) \Rightarrow 3\]

\[(\text{cond } [(< 2 1) 3] [(< 5 6) 4] [\text{else 5}]) \Rightarrow 4\]
E.g.

(if (< 1 2) 3 4) => 3

(cond
[(< 2 1) 3]
[(< 5 6) 4]
[else 5]) => 4
We can put all of this together to get our first interesting procedure:

```
(define f (lambda (x)
  (cond
    [(= x 1) 1]
    [else (* x (f (- x 1)))])))
```