The following are informal notes that I will be using as reference material for my whiteboard lecture. They are not intended to be comprehensive or even to be understandable in isolation. However, I am making them available so that you don't need to write everything down while I am giving the lecture. The primary and authoritative resource for today's material is the textbook, sections 15.5.8-11 and 6.9.

[Finish basics from last time. Then...]

1. Functions as first-class objects, and lambda expressions

Functions can be used as variables, parameters etc. (this is often stated as "functions are first-class objects")

```
(define (eval-at-5 f) (f 5))

(eval-at-5 add1)
```

Challenge: create a function called "increasing" that accepts a single function f as a parameter, and returns true if f(0) < f(1) < f(2), otherwise returns false.

Solution:

```
(define (increasing f)
  (and (> (f 1) (f 0))
       (> (f 2) (f 1))))
```

Test it:

```
     (increasing add1)
```

Challenge: we have seen how to use a function as a parameter. How can we use a function as a return type? For example, try to define a function "add-something", which takes a single integer parameter x, and returns a function whose effect is that y->y+x.

Solution: We can't do it without using lambda expressions.

Here is the solution:

```
(define (add-something x)
  (lambda (y) (+ y x)))
```

test it:

```
((add-something 10) 5)
(define add10 (add-something 10))
(add10 3)
```

2. Lists

Lists are the most commonly used data types in scheme. They are created with the "list" function:

```
(list 2 4 6 8)
```

`car` returns the first element, `cdr` returns the remainder:

```
(car (list 2 4 6 8))
(cdr (list 2 4 6 8))
```

Challenge: what is `(car (cdr (list 2 4 6 8)))` -- work it out without entering it!!

[Mention etymology briefly]

Can combine these up to a ridiculous number of a's and d's, e.g.

```
(cadddr (list 2 4 6 8))
```

Add element to the start of a list using cons (for "construct"):

```
(cons 5 (10 15 20))
```

In some Lisp dialects the empty list is represented by `()`, but in our dialect it is either `empty` or `null` or `'()` . The single-quote will be explained soon

Lists can be nested:

```
(list (list 60 70) 4 5 (list 20 30) 6 7)
```

challenge:

- what is `(car (list (list 60 70) 4 5 (list 20 30) 6 7))`?
- what is `(cdddr (list (list 60 70) 4 5 (list 20 30) 6 7))`?

A list containing the empty list is not the same thing as the empty list:

```
(equal? (list empty) empty)
```

3. quote

To use an expression as data, without evaluating it, use `(quote )` -- and this can be abbreviated as a single quote: `'`

```
(quote a)
'a

(quote (a b c))
'(a b c)
```

4. let

`let` allows you to define something similar to local variables

[Java style]
```
int x = 5;
int y = 2;
return x + y;
```

[Scheme style]
```
(let ((x 5)
      (y 2))
  (+ x y))
```

`letrec` is a recursive version of `let` which permits later expressions to depend on earlier ones. (The only disadvantage is that it's a little less efficient.)

[Java style]
```
int x = 5;
int y = 2;
int x2 = x * x;
int y2 = y * y;
return x2 + y2;
```

[Scheme style]
challenge: do it yourself

Of course, you can define functions within `let` and `letrec` too:

Challenge: what is the output of the following (without typing it in)?:
```
(letrec ((x 5)
         (y 2)
         (x2 (* x x))
         (y2 (* y y))
         (f (lambda (v w) (+ v w))))
  (f x2 y2))
```

5. loops

These don't exist. Seriously -- no "while" or "for". Use recursion instead.

6. recursion

[Java style]

```
int start = 0;
int stop = 5;
for(int i = start; i < stop; i++){
      System.out.println(i);
}
```

[Scheme style]

```
(define (print-numbers start stop)
  (if (not (= start stop))
      (begin
        (display start)
        (newline)
        (print-numbers (add1 start) stop))
      (newline)))

(print-numbers 0 5)
```

A less silly example:

[Java style]

```
int start = 0;
int stop = 5;
int sum(int start, int stop) {
    int total = 0;
    for(int i = start; i < stop; i++){
      total += i;
      }
    return total;
}
```

[Scheme style]

```
(define (sum start stop)
  (letrec
      ((sum-helper
        (lambda (total start stop)
          (if (not (= start stop))
              (sum-helper (+ total start) (add1 start) stop)
              total))))
    (sum-helper 0 start stop)))

(sum 0 5)
```