Fractions
Let's take a look at how Scheme can be used to implement data structures. For our first example we will implement an easy datatype -- Fractions.

First, how should we represent a fraction, such as 3/4?
An obvious solution is to use the pair (3 4) to represent 3/4.

This leads to some easy definitions:

```
(define make-rat (lambda (num denom)
   (list num denom)))

(define num (lambda (r)
   (car r))

(define denom (lambda (r)
   (cadr r))

(define rat+ (lambda (r1 r2)
   (make-rat (+ (* (num r1) (denom r2)) (* (num r2) (denom r1)))
         (* (denom r1) (denom r2)))))
```
This works but if you add 1/2 and 1/2 this says the answer is (4 4), which we would write as the fraction 4/4.

A better solution is to improve our make-rat procedure, so it reduces the fraction "to lowest terms":

```
(define make-rat (lambda (a b)
    (let ([g (gcd a b)]) ; gcd is a standard Scheme procedure
        (list (/ a g) (/ b g)))))
```

Now the result of
(rat+ (make-rat 1 2) (make-rat 1 2)) is (1 1)
It is easy to go from here to a full implementation of fractions, with +, -, *, / operators.

See the file fractions.rkt

One thing to notice here is the print-rat procedure:

```
(define print-rat (lambda (r)
    (printf "~/s/~/s" (num r) (denom r))))
```

This is analogous to print "~%d ~%d
" %(num(r), denom(r)) in Python

or printf( "%d %d
", num(r), num(r)) in Java.

The first argument to printf is a format string; the remaining arguments give values for the ~s placeholders.
Using the pair \((a \ b)\) to represent the fraction \(a/b\) is an obvious choice, but not the only choice. Here is another way to represent fractions:

\[
\begin{align*}
\text{(define make-rat} & \quad \text{(lambda} (\text{a} \ \text{b}) \\
& \quad \text{(let} ([g (gcd a b)]) \\
& \quad \quad \text{(lambda} (s) \\
& \quad \quad \quad \text{(cond} \\
& \quad \quad \quad \quad \text{[(eq? s 'num) (/ a g)]} \\
& \quad \quad \quad \quad \text{[(eq? s 'denom) (/ b g)]} \\
& \quad \quad \quad \quad \text{[else 'error]}))))
\end{align*}
\]

\[
\text{(define num (lambda(r) (r 'num)))} \\
\text{(define denom (lambda(r) (r 'denom)))}
\]