Numbers

On Monday we saw a set of Scheme procedures that implemented a Fraction datatype:

```
(define make-rat
  (lambda (a b)
    (let ([g (gcd a b)])
      (list (/ a g) (/ b g)))))

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

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

plus definitions of operations rat+, rat-, etc.

We did an alternative implementation where fractions were represented by lambda-expressions that returned the numerator or denominator when given the right argument.

We will now combine these into one system so that fractions might be created as either lists or procedures and used interchangeably regardless of how they were created.

The trick to this is to provide a clue as to how an object was created so that our system will know how to work with it. As we have seen before, all that is necessary to do arithmetic with fractions is knowing their numerators and denominators.

We will go with the following conventions. Naturally, many other conventions would also be reasonable:

a) For datatypes that are stored in lists, we will use the first element of the list as a type descriptor.
b) For datatypes that are stored in procedures, we will expect the procedure to return a type descriptor if we give it the argument 'type.

With these conventions we could define both of our implementations of fractions:

```
(define make-rat1
    (lambda (a b)
        (let ([g (gcd a b)])
            (list 'rat1 (/ a g) (/ b g)))))
(define make-rat2
    (lambda (a b)
        (let ([g (gcd a b)])
            (lambda (s)
                (cond
                    [(eq? s 'type) 'rat2]
                    [(eq? s 'num) (/ a g)]
                    [(eq? s 'denom) (/ b g)]
                    [else 'badFraction]))))))
```

The numerator procedure just needs to take these into account. To show the generality here, we will add in a third type -- the standard integer type.

```
(define num
   (lambda (r)
      (cond
         [(number? r) r]
         [(pair? r) (if (eq? (car r) 'rat1) (cadr r) 'UnknownType)]
         [else (if (eq? (r 'type) 'rat2) (r 'num) 'UnknownType)])))
```

If we create a few fractions:

```
(define f1 (make-rat1 2 3))
(define f2 (make-rat2 3 4))
```

We can then work with our arithmetic operators without worring about how the fractions were created:

```
(print-rat (rat* 3  (rat- f2 f1)))     prints 1/4
```