Fold

Think about our functions that recurse on lats.

```
(define member
        (lambda (a lat)
                (cond
                        [(null? lat) #f]
                        [(equal? a (car lat)) #t]
                        [else (member a (cdr lat))])))
```

```
(define sum
    (lambda (lat)
        (cond
            [(null? lat) 0]
            [else (+ (car lat) (sum (cdr lat)))])))
```

Most functions that recurse on lats have a base case where we do something with the empty list, do something with the car of the lat, and recurse on the cdr of the lat

We can abstract this with a function called foldr that takes three arguments: a lat, what to return when the lat is null, and a function of two variables: the car of the lat and the result of recursing on the cdr:

```
(define foldr
    (lambda (f base-value lat)
        (cond
            [(null? lat) base-value]
            [else (f   (car lat)
                       (foldr  f base-value (cdr lat)))])))
```

Here f is a function of two arguments. You can think of these arguments as "the car of the list" and "the result of recursing on the cdr of the list".

For example the member function is

```
(define member (lambda (a lat)
   (foldr  (lambda (x y)
              (if (equal? x a) #t y))
           #f
           lat)))
```

We can sum a list of numbers with fold:

```
(define sum
    (lambda (lat)
        (foldr + 0 lat)))
```

It is inefficient to have a recursive function with 3 arguments, two of which don't change in the recursive call. Here is a better version of foldr:

```
(define foldr
    (lambda (f base-value lat)
        (letrec
            ([helper (lambda (ls)
                (cond
                    [(null? ls) base-value]
                    [else (f (car ls)
                                (helper (cdr ls)))]))])
            (helper lat)))
```

We can disentangle the way foldr works on a small list. (foldr f base null) returns base

(foldr f base '(c)) returns (f c base)

(foldr f base '(b c)) returns (f b (f c base))

(foldr f base '(a b c)) returns (f a (f b (f c base)))

If you imagine f as being an arithmetic operator, such as +, -, or times, (foldr f op base '(a b c)) gives

(op a (op b (op c base)))

In standard infix notation this is

a op b op c op base

associated from the right. For this reason, many people call this "fold-right". Dr. Racket calls it "foldr".

For example, (foldr - 0 '(1 2 3 4)) gives

(1 - (2 - (3 - (4-0)))

which is the same as 1 - 2 + 3 - 4, or -2

Foldr-ing - over a vector gives the alternating sum of its elements. Foldr-ing + over a vector gives the sum of its elements. Foldr-ing * gives the product of the elements and foldr-ing / gives the alternating quotient and product:

(foldr / 1 '(1 2 3 4)) is 1 / 2 * 3 / 4, or 3/8

Of course, you can guess that if there is a foldr there is also a foldl. This is defined differently by different communities. The standard Scheme definition of "fold-left" just associates from the left, so

(fold-left - 0 '(1 2 3 4)) gives

0 - 1 - 2 - 3 - 4 (associated from the left)

or -10

racket follows Haskell and some other languages in defining foldl differently.

(foldl op base '(a b c)) is

(op c (op b (op a base)))

This version of foldl starts with the left end of the list, forms the operation between the car of the list and the base value, then the operation between the next entry of the list and this value, and so forth.

So we can define this as follows:

```
(define foldl (lambda (f base lat)
        (cond
                [(null? lat) base]
                [else (foldl f (f (car lat) base) (cdr lat))])))
```

This is tail recursion with the base variable used as an accumulator.

Of course, foldl is already a part of racket so you don't need to define it yourself.

You can think of the function f as taking two variables, where the first variable is the car of the list and the second variable is the result of everything you have seen so far.

Perhaps the clearest example of the differences between foldr and foldl is the following:

(foldr cons null '(a b c d)) returns (a b c d)

(foldl cons null '(a b c d)) return (d c b a)

That may be the most confusing way you can imagine to reverse a lat.