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 fold 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 fold
  (lambda (recur base-value lat)
    (cond
      [(null? lat) base-value]
      [else (recur
              (car lat)
              (fold recur base-value (cdr lat))))]))
Here recur is a function of two arguments
For example, the member function is

```
(define member (lambda (a lat)
    (fold (lambda (x y)
        (if (equal? x a) #t y))
    #f
    #f
    lat)))
```
We can sum a list of numbers with fold:
(define sum
  (lambda (lat)
    (fold + 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 fold:
(define fold
  (lambda (recur base-value lat)
    (letrec
      ([helper (lambda (ls)
                   (cond
                     [(null? ls) base-value]
                     [else (recur (car ls)
                                  (helper (cdr ls)))]
                   )]
       )
       (helper lat))))