```
(define (map func lst)
  (if (null? lst) '()
    (cons (func (car lst)) (map func (cdr lst)))))

(define (double x) (* x 2))
(map double '(1 2 3)) => '(2 4 6)

(map (lambda (x) (* x 2)) '(1 2 3)) => '(2 4 6)

(map car '((1 2 3) (4 5) (6) (7 8 9))) => '(1 4 6 7)

(define (scale factor lst)
  (map (lambda (x) (* x factor)) lst)

(scale 2 '(1 2 3)) => '(2 4 6)

(map (lambda (x) (+ x 1)) '(1 2 3)) => ?

(map (lambda (x) (cons x '()) '(1 2 3)) => ?

(map (lambda (x) (list x)) '(1 2 3)) => ?

(map (lambda (x)
  (if (> x 0) (* x 2) (* x 3))) '(1 -2 -3 4))

(define (filter func lst)
  (cond ((null? lst) '()
    ((func lst) (cons (func (car lst)) (filter func (cdr lst))))
    (#t (filter func (cdr lst)))))

(filter odd? '(1 2 3)) => '(1 3)

(define (keep-odds lst) (filter odd? lst)

(filter (lambda (x) (> x 0)) '(-1 2 -3 4)) => '(2 4)

(filter (lambda (x) (= 1 (remainder x 2))) '(1 2 3)) => ?

(define (keep-divisible factor lst)
  (filter _______ lst))

(filter (lambda (lst) (even? (car lst)))
          '((1 2 3) (4 5) (6 7))) => ?

(filter (lambda (lst) (even? (car lst)))
          '((1 2 3) (4 5) (6 7) ())) => ?

(define (keep-longer-than n lst)
  (filter _______ lst))



(define (square x) (expt x 2))
(define (cube x) (expt x 3))

; a function that returns a function
(define (to-the-power exponent)
        (lambda (x) (expt x exponent)))

; how to use this new function:
(define square-new (to-the-power 2))
(define cube-new (to-the-power 3))

; another example
; first, the old way:
(define (add3 num) (+ 3 num))
(define (add17 num) (+ 17 num))
; and the new way:
(define (create-add-function inc)
  (lambda (num) (+ inc num)))
(define add3-new (create-add-function 3))
(define add17-new (create-add-function 17))

; a function that takes two functions and returns a function
(define (compose f g)
  (lambda (x) (f (g x))))
(define second (compose car cdr))
```

```
(define third (compose car (compose cdr cdr)))
;(map third '((2013 5 6) (2012 1 8) (2000 7 7)))

; a function that takes a function and returns a "safe" version of the same function
(define (make-safe func)
  (lambda (lst)
    (if (or (not (list? lst))
            (null? lst))
        "No can do!"
        (func lst))))

(define (divisible n)
        (lambda (x) (= 0 (remainder x n))))

(define (make-quad-polynomial a b c)
   (lambda (x) (+ (* a x x) (* b x) c)))

; Building to foldr
; these three functions are all similar
(define (length lst)
  (if (null? lst) 0
    (+ 1 (length (cdr lst)))))

(define (sum-list lst)
  (if (null? lst) 0
    (+ (car lst) (sum-list (cdr lst)))))

;(define (map func lst)
;  (if (null? lst) '()
;    (cons (func (car lst)) (map func (cdr lst)))))

; resulting in foldr!
(define (foldr combine base lst)
  (if (null? lst) base
    (combine (car lst) (foldr combine base (cdr lst)))))

(define (sum-list-new lst)
        (foldr + 0 lst))

(define (length-new lst)
  (foldr (lambda (elt cdr-len) (+ 1 cdr-len)) 0 lst))

(define (my-map func lst)
  (foldr (lambda (car cdr) (cons (func car) cdr)) '() lst))


#|
- Write a function make-gt which takes a number n and returns a function of a single
argument x that evalutes to #t if n > x.

  Ex: (define gt5 (make-gt 5))
      (filter gt5 '(2 4 6 8)) ==> '(6 8)

- Write a function called make-polynomial that takes a list of coefficients and returns
a function that represents the equivalent polynomial.
  Ex: (make-polynomial '(2 1)) returns the polynomial 2x + 1, or rather it
          returns the function (lambda (x) (+ (* 2 x) 1))

  Ex: (make-polynomial '(2 4 -8)) returns the polynomial 2x^2 + 4x - 8
    This should work for a list of any length.  A polynomial for a list of length
    zero is the zero polynomial f(x) = 0.

  You should be able to do something like
  (map (make-polynomial '(2 1)) '(1 2 3)) ==> '(3 5 7)

- Write filter and count (from the previous lecture) in terms of foldr.

- Write max in terms of foldr.  You may assume all the numbers you will be comparing
are positive.
|#
```