```
#lang racket
; Programming Languages
; Lecture on First Class Functions

(define (sum a b)
  (if (> a b) 0
      (+ a (sum (+ a 1) b))))

(define (sum-squares a b)
  (if (> a b) 0
      (+ (expt a 2) (sum-squares (+ a 1) b))))

(define (sum-abs a b)
  (if (> a b) 0
      (+ (abs a)
         (sum-abs (+ a 1) b))))

; Abstract away the thing you do before summing; result
; is this function:
(define (sum-any func a b)
  (if (> a b)
      0
      (+ (func a)
         (sum-any func (+ a 1) b))))

(define (silly-function x) (/ (* x x) 2))
(sum-any silly-function 1 10)

; Better to use an anonymous function, because we'll
; probably never need the silly function again:
(sum-any (lambda (x) (/ (* x x) 2)) 1 10)

(define (do-n-times func n x)
  (if (= n 0) x (do-n-times func (- n 1) (func x))))

(define (get-nth lst n) (car (do-n-times cdr n lst)))

; raise x to the y power
(define (power x y)
  (do-n-times (lambda (a) (* x a)) y 1))

(define (mystery lst)
  (if (null? lst) '()
      (cons (car lst) (mystery (cdr lst)))))

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

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

#|
Write a function called scale that takes a list of numbers and a number,
  and returns a new list with all the numbers scaled by the number supplied.
        Hint: use map and an anonymous function.

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

Write a function called keep-odds that takes a list of numbers and returns a new list
  with all only the numbers that are odd retained.

        Hint: There's a built-in function (odd? p) that returns #t if p is odd.  Use filter
        and odd?.

        Ex: (keep-odds '(1 2 3)) ==> '(1 3)
```

```
Write a function called keep-greater-than that takes a number x and an addition list of
        numbers.  This function returns a new list only the numbers that greater than x
        retained.

        Ex: (keep-greater-than 3 '(3 5 2 7 4)) ==> '(5 7 4)

Write a higher-order function called find that takes a function and a list.
  The function argument must be a function of a single value and return #t or #f (aka
  a predicate).  Find should return the first element in the list that satisfies
  the predicate (makes the function return true).

  Do not use filter for this, even though it would be very simple.  (That algorithm
  would be rather inefficient if the list were very long but the first item in the list
  satisfied the predicate, because filter would still have to filter all the remaining
  elements.)

  Ex: (find odd? '(2 4 3 5)) ==> 3
  Ex: (find (lambda (x) (< x 0)) '(3 2 1 0 -1 -2)) ==> -1
  Ex: (find (lambda (lst) (even? (car lst))) '((1 2 3) (4 5 6) (7 8 9))) ==> '(4 5 6)

Write a higher-order function called greatest that takes a predicate and a list.  The
  predicate must be a comparison predicate of two arguments (e.g., < or >).  greatest
  returns the element in the list that the comparison operator says is #t for all the
  elements in the list when compared against each other.

  Ex: (greatest > '(1 3 2 4 6 5)) ==> 6
  Ex: (greatest < '(1 3 2 4 6 5)) ==> 1
  Ex: (greatest before? '((2012 4 2) (2012 1 3) (2012 10 10))) ==> '(2012 1 3)

Write a higher-order function called count that takes a predicate and a list, and returns
        the number of items in the list that satisfy the predicate.

        Write this in two ways: first using filter, then again not using filter.

Write two higher-order predicate functions called any and all.  Each takes a predicate
  and a list as arguments.  Any returns #t if any of the items in the list satisfy the
  predicate.  All returns #t if all the elements satisfy the predicate.

  Ex: (any odd? '(1 2 3 4)) ==> #t
  Ex: (any odd? '(2 4 6 8)) ==> #f
  Ex: (all even? '(2 4 6 8)) ==> #t
  Ex: (all even? '(1 2 3 4)) ==> #f

  Write these first from scratch without using filter.  Then write them by using filter.
  Then write them using count.  Which way(s) are more efficient (in terms of time) than othe
rs?
|#
```