Exercise

Write a function `(normalize L)` that consumes a `(listof Num)`, and returns the list containing each value in `L` divided by the sum of the values in `L`. **Compute the sum only once.**

```
(normalize (list 4 2 14)) => (list 0.2 0.1 0.7)
```

Exercise

Write a function `vector2D+` that consumes two `Posn` and does *vector addition*.
(That is, the new x is the sum of the x values, and the new y is the sum of the y values.)

```
;; (vector2D+ v1 v2) return the vector sum of v1 and v2.
;; vector2D+: Posn Posn -> Posn
;; Example:
(check-expect (vector2D+ (make-posn 2 3) (make-posn 5 8)) (make-posn 7 11))
```

Exercise

Write `(discard-bad L lo hi)`. It consumes a `(listof Num)` and two `Num`. It returns the list of all values in `L` that are between `lo` and `hi`, inclusive.

```
(discard-bad (list 12 5 20 2 10 22) 10 20) => (list 12 20 10)
```

Exercise

Complete `count-sheep`.

```
;; (count-sheep L) return the number of 'sheep in L.
;; count-sheep: (listof Any) -> Nat
;; Example:
(check-expect (count-sheep (list 6 'sheep 'ram 3.14 'sheep 'ox)) 2)
```

Exercise

Using `cond` and `map`, write a function `neg-odd` that consumes a `(listof Nat)`. The function returns a `(listof Int)` where all odd numbers are negative, and all even numbers positive.

```
(neg-odd (list 2 5 8 11 14 17)) => (list 2 -5 8 -11 14 -17)
```

Exercise

Write a function `(times-row n len)` that returns the `n`th row of the times table. This should be a list of length `len`. Write you function in the form `(map ... (range 1 (+ len 1) 1))`.

```
(times-row 3 4) => (list 3 6 9 12)
(times-row 5 3) => (list 5 10 15)
```

Exercise

Complete `join-names`.

```
;; (join-names G S) Make a list of full names from G and S.
;; join-names: (listof Str) (listof Str) -> (listof Str)
;; Example:
(check-expect (join-names gnames snames)
              (list "Joseph Hagey" "Burt Matthews" "Douglas Wright"
                    "James Downey" "David Johnston"))
```

Exercise

Complete `tree-sum`.

```
;; (tree-sum tree) return the sum of all keys in tree.
;; tree-sum: SSTree -> Num
;; Example:
(check-expect (tree-sum tree12) 48)
```

Exercise

Using `lambda` and `filter` but no [named] helper functions, write a function that consumes a `(listof Str)` and returns a list containing all the strings that have a length of 4.

```
(keep4 (list "There's" "no" "fate" "but" "what" "we" "make" "for" "ourselves"))
=> (list "fate" "what" "make")
```

Exercise

Write a function (find-ldict key dict) that consumes a Nat and a LDict.
The function returns the value in dict associated with the key. You may assume key appears exactly once in dict.

```
(check-expect (find-ldict 6938 student-dict) (list "Al Gore" "government"))
```

Exercise

Complete pfd-lcm.

```
;; (pfd-lcm L1 L2) return the lcm of p1 and p2.
;; pfd-lcm: PFD PFD -> PFD
;; Example:
(check-expect (pfd-lcm (list 2) (list 2)) (list 2))
(check-expect (pfd-lcm (list 2 2 3) (list 2 3 3 5)) (list 2 2 3 3 5))
```

Exercise

Complete dot-product.

```
;; A Vector is a (listof Num).

;; (dot-produce u v) return the dot product of u and v.
;; dot-product: Vector Vector -> Num
;; Requires: u and v have the same length.
;; Example:
(check-expect (dot-product (list 2 3 5) (list 7 11 13)) 112)
```

Exercise

Write a function that consumes a (listof Str), where each Str is a person's name, and returns a list containing a greeting for each person.

```
(greet-each (list "Ali" "Carlos" "Sai")) => (list "Hi Ali!" "Hi Carlos!" "Hi Sai!")
```

Exercise

Complete the function total-value that consumes an Inventory and returns the amount of money we would get if we sell out of item.

```
;; (total-value item) return cost of all our item.
;; total-value: Inventory -> Num
;; Example:
(check-expect (total-value (make-inventory "rice" 5.50 6)) 33.00)
```

Exercise

Write a function (raise-price dollars item) that consumes a Num and a Inventory and returns the Inventory that results from increasing the price of item by dollars.

```
;; (raise-price dollars item) return item with price increased by dollars.
;; raise-price: Num Inventory -> Inventory
;; Example:
(check-expect (raise-price 0.49 (make-inventory "rice" 5.50 6))
              (make-inventory "rice" 5.99 6))
```

Exercise

Write purpose, contract, examples, and tests for:

(1) The absolute value function

Exercise

Complete eval-binexp so it can handle '+ and '*.

```
;; (eval-binexp expr) return the value of expr.
;; eval-binexp: BinExp -> Num
;; Examples:
(check-expect (eval-binexp (make-binode '* 7 6)) 42)
(check-expect (eval-binexp (make-binode '* 7 (make-binode '+ 4 2))) 42)
```

Exercise

Write a function (sentence->list S) that consumes a Sentence and returns a (listof Str) containing the words in S.

```
(check-expect (sentence->list catS) (list "the" "cat" "ate"))
```

Exercise

Complete count-leaves.

```
;; (count-leaves tree) return the number of leaves in tree.
;; count-leaves: SSTree -> Nat
;; Example:
(check-expect (count-leaves tree12) 2)
```

Exercise

Complete insert.

```
;; (insert item L) Add item to L so L remains sorted in increasing order.
;; insert: Int (listof Int) -> (listof Int)
;; Requires: L is sorted in increasing order.
;; Examples:
(check-expect (insert 6 (list 7 42)) (list 6 7 42))
(check-expect (insert 81 (list 3 9 27)) (list 3 9 27 81))
(check-expect (insert 5 (list 2 3 7)) (list 2 3 5 7))
```

Exercise

The factorial function, $n!$, returns the product of the numbers from 1 to n. For example, $4! = 1 \times 2 \times 3 \times 4 = 24$.

Write a function (factorial n) that returns $n!$.

```
(factorial 5) => 120
(factorial 1) => 1
```

Exercise

Write a recursive function (sum-between n b) than consumes two Nat, with n $\geq$ b, and returns the sum of all Nat between b and n.

```
(sum-between 5 3) => (+ 5 4 3) => 12
```

Exercise

Using lambda and map, but no [named] helper functions, write a function that consumes a (listof Num) and returns a list containing the cube of each Num. (x^3)

Exercise

Complete join-names.

```
;; (join-names G S) Make a list of full names from G and S.
;; join-names: (listof Str) (listof Str) -> (listof Str)
;; Example:
(check-expect (join-names gnames snames)
              (list "Joseph Hagey" "Burt Matthews" "Douglas Wright"
                    "James Downey" "David Johnston"))
```

Exercise

Create a function (even-mean-minus-odd-mean L) that returns the mean of the even values in L minus the mean of the odd values.

Include a local helper function (mean M) that consumes a (listof Int) and returns the mean of the values in M. Do not create any additional helper functions.

```
(even-mean-minus-odd-mean (list 16 14 5 1)) => 12
```

Exercise

```
(define x 4)
(define (f x) (* x x))
(f 3)
```

Exercise

Copy this code and see how it behaves:

```
;; (portions L) divide each value in L by sum of L.
;; portions: (listof Num) -> (listof Num)
(define (portions L)
  (cond [(empty? L) '()]
        [else (cons (/ (first L) (sum L))
                    (portions (rest L)))]))
```

Exercise

Complete `count-leaves`.

```
;; count-leaves: GenTree -> Nat
;; Examples:
(check-expect (count-leaves (make-gnode 'wut (list "foo" "bar" "baz"))) 3)
(check-expect (count-leaves
               (make-gnode '+
                           (list 2 3 (make-gnode '* (list 6 7 42))))) 5)
```

Exercise

Perform a trace of

```
(and (= 3 3) (> 7 4) (< 7 4) (> 0 (/ 3 0)))
```

Exercise

Use recursion to complete `append-lists`.

```
;; (append-lists L1 L2) form a list of the items in L1 then L2, in order.
;; append-lists: (listof Any) (listof Any) -> (listof Any)
;; Example:
(check-expect (append-lists (list 3 7 4) (list 6 8)) (list 3 7 4 6 8))
```

Exercise

Change `ponder` so `muck-after-str` also changes every value that immediately follows the word `"SQUARE"` be the square of that number.
E.g. `(muck-after-str (list 5 7 "SQUARE" 4 3)) => (list 5 7 16 3)`

Exercise

Write a function that consumes a `(listof Num)` and returns a list with each number doubled.
The following function works. Rewrite it using `foldr`, without using `map`.

```
(define (double n) (* n 2))

(define (double-each L) (map double L))
```

Exercise

Write a function `(distances xs ys)` that consumes two lists: the first contains x values, and the second contains y values. The output is a list containing the distance of each point (x, y) from $(0, 0)$.

```
(distances (list 3 0 2) (list 4 7 2)) => (list 5 7 #i2.828427)
```

(Since $(3, 4)$ is at distance 5; $(0, 7)$ is at distance 7; and $(2, 2)$ is at distance $\sqrt{8} \approx 2.828427$.)

Exercise

Write a function `remove-second` that consumes a list of length at least two, and returns the same list with the second item removed.

```
(remove-second (list 2 4 6 0 1)) => (list 2 6 0 1)
```

xercis

Write a function `myfun` that allows `use-foldr` to do something.

Exercise

Write a function (times-table len) that returns the n×n times table.
Use times-row as a helper function.

```
(timestable 5) =>
  (list (list  1  2  3  4  5)
        (list  2  4  6  8 10)
        (list  3  6  9 12 15)
        (list  4  8 12 16 20)
        (list  5 10 15 20 25))
```

Exercise

Write a function that returns the number of odd numbers in a (listof Nat).
Hint: read the documentation on remainder.
Can you do this using map and foldr? Just using foldr?

Exercise

Given that use-foldr consumes a (listof Nat):

```
(define (use-foldr L) (foldr myfun "some-str" L))
```

(1) What is the contract for myfun ?
(2) What is the contract for use-foldr ?

Exercise

Write a full design recipe for a function distance which computes the distance between $(0, 0)$ and a given point (x, y).
Include **purpose**, **contract**, **examples**, **implementation**, and **tests**.

Exercise

Write a function acronymize that consumes a (listof Str), where each Str is of length at least 1, and returns a Str containing the first letter of each item in the list.

```
(acronymize (list "Portable" "Network" "Graphics")) => "PNG"
(acronymize (list "GNU's" "Not" "UNIX")) => "GNU"
```

Exercise

Write a function (non-decreasing L) that consumes a (listof Num), and returns a (listof Num) containing only those values at least as big as all the values that came before.
For example,

```
(non-decreasing (list 2 3 1 6 8 6 4 8 1 9))
=> (list 2 3 6 8 8 9)
```

Exercise

Complete factorize. It may be helpful to consider the count-up template for recursion on a Nat, starting at 2.

Exercise

Write a function prod that returns the product of a (listof Num).

```
(prod (list 2 2 3 5)) => 60
```

Exercise

Complete countdown-to using recursion.

```
;; (countdown-to n b) return a list of Int from n down to b.
;; countdown-to: Int Int -> (listof Int)
;; Examples:
(check-expect (countdown-to 2 0) (cons 2 (cons 1 (cons 0 '()))))
(check-expect (countdown-to 5 2) (list 5 4 3 2))
```

Exercise

Write a function (sum-odds-or-evens L) that consumes a (listof Int). If there are more evens than odds, the function returns the sum of the evens. Otherwise, it returns the sum of the odds.
Use local, but do not use L more than *twice* (in map, filter, foldr, or otherwise).

```
(sum-odds-or-evens (list 1 3 5 20 30)) => 9
```

Exercise

Write a recursive function `sum` that consumes a `(listof Int)` and returns the sum of all the values in the list.

```
(sum (list 6 7 42)) => 55
```

That is, use recursion to duplicate the following function:

```
(define (sum L) (foldr + 0 L))
```

Exercise

Complete `expand-bst`.

```
;; (expand-bst L tree) add all items in L to tree, adding the last first.
;; expand-bst: (listof Association) BST -> BST
;; Example:
(check-expect
 (expand-bst (list (make-association 4 "four")) '())
 (make-node 4 "four" '() '()))
(check-expect
 (expand-bst (list (make-association 2 "two")
                   (make-association 6 "six")
                   (make-association 4 "four")) '())
 (make-node 4 "four"
            (make-node 2 "two" '() '()) (make-node 6 "six" '() '())))
```

Exercise

Write a function `drop-e` that converts a `Str` to a `(listof Char)`, replaces each `#\e` with a `#\*`, and converts it back to a `Str`.

```
(drop-e "hello world, how are you?") => "h*llo world, how ar* you?"
```

Exercise

Write a function `times-square` that consumes a `(listof Nat)` and returns the product of all the perfect squares $(1, 4, 9, 16, 25, \dots)$ in the list.

```
(times-square (list 1 25 5 4 1 7)) => (* 1 25 4 1) => 100
```

Exercise

Use `define` to create a function `(add-twice a b)` that returns $a + 2b$.

```
(add-twice 3 5) => 13
```

Exercise

Complete `dict-add`.

```
(define-struct node (key val left right))

;; A binary search tree (BST) is either
;; * '() or
;; * (make-node Nat Any BST BST)...
(define-struct association (key val))
;; An Association is a (make-association Nat Any)

;; (dict-add newassoc tree) return tree with newassoc added.
;; dict-add: Association BST -> BST
;; Examples:
(check-expect (dict-add (make-association 4 "four") '())
              (make-node 4 "four" '() '()))
(check-expect
 (dict-add (make-association 6 "six")
           (dict-add (make-association 2 "two")
                     (dict-add (make-association 4 "four") '())))
 (make-node 4 "four" (make-node 2 "two" '() '())
            (make-node 6 "six" '() '())))
```

Exercise

Write a function that returns the average (mean) of a non-empty `(listof Num)`.

```
(average (list 2 4 9)) => 5
(average (list 4 5 6 6) => 5.25)
```

Recall that `(length L)` returns the number of values in `L`.

Exercise

Following the template, complete `depth`.

```
;; (depth tree) return the max distance from the root to a leaf of tree.
;; depth: LLT -> Nat
;; Examples:
(check-expect (depth (list 6 7)) 1)
(check-expect (depth (list 2 (list 3 (list 5)))) 3)
```

Exercise

Write a function `(sum-square-difference n)` that consumes a `Nat` and returns the difference between the square of the sum of numbers from 0 to n, and the sum of the squares of those numbers.

$$\texttt{(sum-square-difference 3) => (- }\underbrace{\texttt{(sqr (+ 0 1 2 3))}}_{\text{square of the sum}}\quad\underbrace{\texttt{(+ 0 1 4 9)}}_{\text{sum of the squares}}\quad\texttt{) => 22}$$

Exercise

Complete `dict-add`.

```
;; (dict-add d k v) return a new dictionary containing all values in d,
;; and new value (make-asc k v). Keep data sorted by key.
;; If key is already in d, replace its value.
;; dict-add: Dict Nat Any -> Dict
;; Example:
(check-expect
 (dict-add student-dict
           7587
           (make-student "George W Bush" "business"))
 (list (make-asc 6938 (make-student "Al Gore" "government"))
       (make-asc 7334 (make-student "Bill Gates" "appliedmath"))
       (make-asc 7587 (make-student "George W Bush" "business"))
       (make-asc 8838 (make-student "Barack Obama" "law"))))
```

Exercise

Write a function `(countdown-by top step)` that returns a list of `Nat` so the first is `top`, the next is `step` less, and so on, until the next one would be zero or less.

```
(countdown-by 15 3) => (list 15 12 9 6 3)
(countdown-by 14 3) => (list 14 11 8 5 2)
```

Exercise

Write a function `(add-total L)` that consumes a `(listof Num)`, and adds the total of the values in `L` to each value in `L`.

```
(add-total (list 2 3 5 10)) => (list 22 23 25 30)
```

Exercise

Use `filter` to write a function that keeps all items which are a `(list a b c)` containing a Pythagorean triple $a < b < c : a^2 + b^2 = c^2$

```
(check-expect
 (pythagoreans
      (list (list 1 2 3) (list 3 4 5) (list 5 12 13) (list 4 5 6)))
 (list (list 3 4 5) (list 5 12 13)))
```

Exercise

Change `ponder` so `muck-after-str` also removes every value that immediately follows the word `"POP"`.
E.g. `(muck-after-str (list 5 7 "POP" 4 3)) => (list 5 7 3)`

Exercise

Perform a trace of

```
(or (< 7 4) (= 3 3) (> 7 4) (> 0 (/ 3 0)))
```

Exercise

Use `foldr` to write a function that behaves like `filter`.

```
(my-filter odd? (list 4 5 9 6)) => (list 5 9)
```

Exercise

Read the documentation on `string-length`.
Write a function that returns the total length of all the values in a `(listof Str)`.

```
(total-length (list "hello" "how" "r" "u?")) => 11
```

Exercise

Complete `n-th-item`.

```
;; (n-th-item L n) return the n-th item in L, where (first L) is the 0th.
;; n-th-item: (listof Any) Nat -> Any
;; Example:
(check-expect (n-th-item (list 3 7 31 2047 8191) 0) 3)
(check-expect (n-th-item (list 3 7 31 2047 8191) 3) 2047)
```

Exercise

Make the word `"ADD"` add up the two values that come after it.

```
(muck-after-str (list 5 7 "ADD" 7 3 5)) => (list 5 7 10 5)
```

Exercise

Complete the function `(admission after5? age)` that returns the admission cost.

```
;; admission: Bool Nat -> Num
```

Exercise

Write a function `(multiply-each L n)`. It consumes a `(listof Num)` and a `Num`, and returns the list containing all the values in `L`, each multiplied by `n`.

```
(multiply-each (list 2 3 5) 4) => (list 8 12 20)
```

Exercise

Write a recursive function `list-max` that consumes a nonempty `(listof Int)` and returns the largest value in the list.

Exercise

Using `foldr`, write a function `(keep-evens L)` that returns the list containing all the even values in `L`.
That is, it acts like `(filter even? L)`.

```
(keep-evens (list 1 2 3 4 5 6)) => (list 2 4 6)
```

Exercise

Using recursion, create a function (and necessary helper functions) to create the times tables up to a given value. For example,

```
(times-tables 4) => (list (list 0 0 0 0)
                          (list 0 1 2 3)
                          (list 0 2 4 6)
                          (list 0 3 6 9))
```

Exercise

Write a recursive function `(step-sqr-sum-between lo hi step)`, that returns the sum of squares of the numbers starting at `lo` and ending before `hi`, spaced by `step`.
That is, duplicate the following function:

```
(define (step-sqr-sum-between lo hi step)
  (foldr + 0 (map sqr (range lo hi step))))
```

Exercise

Write a function that consumes a `Num`, and returns

- `"big"` if $80 < x \leq 100$,
- `"small"` if $0 < x \leq 80$,
- `"invalid"` otherwise.

Exercise

Complete the function `list-cubes`.

```
;; (list-cubes n) return the list of cubes from 1*1*1 to n*n*n.
;; list-cubes: Nat -> (listof Nat)
;; Examples:
(check-expect (list-cubes 4) (list 1 8 27 64))
```

Exercise

```
(define y 3)
(define (g x) (+ x y))
(g 5)
```

Exercise

Write a recursive function `keep-evens` that consumes a `(listof Int)` and returns the list of even values. That is, use recursion to duplicate the following function:

```
(define (keep-evens L) (filter even? L))
```

Exercise

Use recursion to complete the function `list-cubes`.

```
;; (list-cubes b t) return the list of cubes from b*b*b to t*t*t.
;; list-cubes: Nat Nat -> (listof Nat)
;; Examples:
(check-expect (list-cubes 2 5) (list 8 27 64 125))
```

Exercise

Write a recursive function `divide-each` that allows `portions` to achieve its purpose.

```
;; (portions L) divide each value in L by sum of L.
;; portions: (listof Num) -> (listof Num)
;; Examples:
(check-expect (portions (list 1 1 2)) (list 0.25 0.25 0.5))
(check-expect (portions (list 6 1 3)) (list 0.6 0.1 0.3))

(define (portions L)
  (divide-each L (sum L)))
```

Exercise

Complete `tree-search`. Clever bit: only search `left` or `right`, not both.

```
;; (tree-search tree item) return #true if item is in tree.
;; tree-search: SSTree Num -> Bool
;; Example:
(check-expect (tree-search tree12 10) #true)
(check-expect (tree-search tree12 7) #false)
```

Exercise

Use `foldr` to write a function `(add-n-each n L)` that adds `n` to each value in `L`.

```
(add-n-each 7 (list 2 4 8)) => (list 9 11 15)
```

Exercise

Trace the program: (`(sqrt n)` computes $\sqrt{n}$ and `(sqr n)` computes n^2)

```
(define (disc a b c) (sqrt (- (sqr b) (* 4 (* a c)))))
(define (proot a b c) (/ (+ (- 0 b) (disc a b c)) (* 2 a)))
(proot 1 3 2)
```

Exercise

Write a function `(absdiff a b)` that consumes two `(listof Int)` and returns a `(listof Nat)` containing the absolute value of the difference between corresponding values.

```
(absdiff (list 1 3 5 7) (list 7 3 6 1)) => (list 6 0 1 6)
```

Exercise

Experiment with `fold-sub`. Describe how it behaves, and why.

```
(define (fold-sub L) (foldr - 0 L))
(fold-sub (list 6 5 2)) => ?
```

Exercise

What is wrong with each of the following?

- `(* (5) 3)`
- `(+ (* 2 4)`
- `(5 * 14)`
- `(* + 3 5 2)`
- `(/ 25 0)`

Exercise

```
(define z 3)
(define (h z) (+ z z))
(h 7)
```

Exercise

Write a recursive function `(sum-to n)` that consumes a `Nat` and returns the sum of all `Nat` between `0` and `n`.

```
(sum-to 4) => (+ 4 3 2 1 0) => 10
```

Exercise

Trace the program:

```
(+ (remainder (- 10 2) (quotient 10 3)) (* 2 3))
```

Exercise

Using `lambda` just once and `foldr` just once, and no [named] helper functions, write a function that consumes a `(listof Int)` and returns the sum of all the even values.

```
(sum-evens (list 2 3 4 5)) => 6
```

Exercise

Use `filter` to write a function that consumes a `(listof Num)` and keeps only values between 10 and 30, inclusive.

```
(keep-inrange (list -5 10.1 12 7 30 3 19 6.5 42)) => (list 10.1 12 30 19)
```

Exercise

Complete `flatten`. Hint: use the `append` function.

```
;; (flatten tree) return the list of leaves in tree.
;; flatten: LLT -> (listof Num)
;; Examples:
(check-expect (flatten (list 1 (list 2 3) 4)) (list 1 2 3 4))
(check-expect (flatten (list 1 (list 2 (list 3 4)))) (list 1 2 3 4))
```

xercis

Read about stacks, and be amazed.

Exercise

Complete `enumerate-words`.

```
;; (enumerate-words L) format the values in L with their index, like:
;; 1. first item
;; 2. second item
;; 3. third item
;; enumerate-words: (listof Str) -> (listof Str)
;; Examples:
(check-expect (enumerate-words (list "Mercury" "Venus" "Earth" "Mars"
                                     "Jupiter" "Saturn" "Uranus" "Neptune"))
              (list "1. Mercury" "2. Venus" "3. Earth" "4. Mars"
                    "5. Jupiter" "6. Saturn" "7. Uranus" "8. Neptune"))
```

Exercise

Write a Racket function corresponding to

$$g(x, y) = x\sqrt{x} + y^2$$

(`(sqrt n)` computes $\sqrt{n}$ and `(sqr n)` computes n^2.)

Exercise

Rewrite `insertion-sort` to use recursion instead of `foldr`.
(You will still use `insert`.)

```
;; (insertion-sort L) return a copy of L, sorted in increasing order.
(define (insertion-sort L)
  (foldr insert '() L))
```

Exercise

Using recursion, write a function `(add-first-each L)` that consumes a `(listof Int)` and adds to each value in the list the first in the list.

```
(add-first-each (list 3 2 7 6 5)) => (list 6 5 10 9 8)
```

Exercise

Write a recursive function `vector-add` that adds two vectors.

```
(vector-add (list 3 5) (list 7 11)) => (list 10 16)
(vector-add (list 3 5 1 3) (list 2 2 9 3)) => (list 5 7 10 6)
```

Exercise

Complete `dict-find`. You may assume `key` appears at most once in `dict`.

```
;; (dict-find d key) return value associated with key in d.
;;      If key is not in d, return #false.
;; dict-find: Dict Nat -> Any
;; Examples:
(check-expect (dict-find student-dict 7334)
              (make-student "Bill Gates" "appliedmath"))
(check-expect (dict-find student-dict 9999) #false)
```

Exercise

Digital signals are often recorded as values between 0 and 255, but we often prefer to work with numbers between 0 and 1.
Write a function `(squash-range L)` that consumes a `(listof Nat)`, and returns a `(listof Num)` so numbers on the interval $[0, 255]$ are scaled to the interval $[0, 1]$.

```
(squash-range (list 0 204 255)) => (list 0 0.8 1)
```

Exercise

Write a function `(collatz-next sk)` that consumes a `Nat` representing an item in a Collatz sequence, and returns the next item in the sequence.

```
(collatz-next 3) => 10
(collatz-next 12) => 6
```

Exercise

Write a function (at-index L) that consumes a (listof Int) and returns all the values in L so item i is at location i.
For example,

```
(at-index (list 0 6 2 3 5 6 0 7)) => (list 0 2 3 7)
; . . . . . . . 0 1 2 3 4 5 6 7
```

Exercise

Write a function that consumes a (listof Num) and returns the list containing just the values which are greater than or equal to the average (mean) value in the list.

Exercise

Write a function count-at that consumes a Str and counts the number of times #\a or #\t appear in it.

```
count-at("A cat sat on a mat") => 7
```

Exercise

```
(define (huh? huh?) (+ huh? 2))
(huh? 4)
```

Exercise

Complete list=?

```
;; (list=? a b) return true iff a and b are equal.
;; list=?: (listof Any) (listof Any) -> Bool
;; Examples:
(check-expect (list=? (list 6 7 42) (list 6 7 42)) true)
```

Exercise

Complete countdown using recursion. (Hint: use cons.)

```
;; (countdown n) return a list of the natural numbers from n down to 0.
;; countdown: Nat -> (listof Nat)
;; Examples:
(check-expect (countdown 3) (cons 3 (cons 2 (cons 1 (cons 0 '())))))
(check-expect (countdown 5) (list 5 4 3 2 1 0))
```

Exercise

Complete sorted?.

```
;; (sorted? L) return #true if every value in L is >= the one before.
;; sorted? (listof Int) -> Bool
;; Examples:
(check-expect (sorted? (list 42)) #true)
(check-expect (sorted? (list 2 3 3 5 7)) #true)
(check-expect (sorted? (list 2 3 5 3 7)) #false)
```

What is the base case?

Exercise

Using foldr, write a function (keep-multiples n L) that returns the list containing all the values in L that are multiples of n.
That is, it acts like (filter (lambda (x) (= 0 (remainder x n))) L).

```
(keep-multiples 3 (list 1 2 3 4 5 6 7)) => (list 3 6)
```

Consider the function add-index:

```
;; (add-index L) to each item in L, add the distance from the front of L.
;; add-index: (listof Num) -> (listof Num)
;; Examples:
(check-expect (add-index (list 0 0 0)) (list 0 1 2))
(check-expect (add-index (list 2 3 5 7 11)) (list 2 4 7 10 15))
```

Exercise

Complete `merge`.

```
;; (merge L1 L2) return the list of all items in L1 and L2, in order.
;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: L1 is sorted; L2 is sorted.
;; Example:
(check-expect (merge (list 2 3 7) (list 4 6 8 9)) (list 2 3 4 6 7 8 9))
```

Exercise

Write the helper function `(ponder new-item answer)` that allows `muck-after-str` to work.

```
(muck-after-str (list 2 7 "X" 3 5)) => (list 2 7 6 5)
```

Exercise

Given these definitions:

```
(define foo 4)
(define (bar a b) (+ a a b))
```

What is the value of this expression?

```
(* foo (bar 5 (/ 8 foo)))
```

Exercise

Use `foldr` to write a function that behaves like `map`.

```
(my-map sqr (list 4 5 3)) => (list 16 25 9)
```

Exercise

Write `(squash-bad lo hi L)`. It consumes two `Num` and a `(listof Num)`. Values in `L` that are greater that `hi` become `hi`; less that `lo` become `lo`.

```
(squash-bad 10 20 (list 12 5 20 2 10 22))) => (list 12 10 20 10 10 20)
```