CS 135 Winter 2017

Tutorial 6: Types of Recursion
Goals of this tutorial

You should be able to...

- understand difference between structural and accumulative recursion.
- recognize generative recursion.
Review: Structural vs. Accumulative

In (pure) structural recursion, all arguments to the recursive call or calls are either unchanged, or *one step* closer to a base case.

In accumulative recursion, arguments are the same as above, plus one or more accumulators, or arguments containing partial answers. The accumulatively recursive function is a helper function, and a wrapper function sets the initial value of the accumulator(s).
Group Problem - Structural Recursion

Write a structurally recursive function called `member-count` that consumes a `(listof Any)` and an item of `Any` type. It should produce the number of times that item appears in the `(listof Any)`.

```scheme
;;; (member-count loa item) Counts the number of times item appears in the loa
;;; member-count: (listof Any) Any → Nat
;;; Examples
(check-expect (member-count empty 5) 0)
(check-expect (member-count (list 'I 'knew 'you 'were 'trouble 'when 'you 'walked 'in 'trouble 'trouble 'trouble) 'trouble) 4)
;;; Tests
(check-expect (member-count (list 'yellow 'green 'purple 'pineapple) 'help) 0)
```
Group Problem - Accumulative Recursion

Write a function called acc-member-count that produces identical values to member-count but uses accumulative recursion instead. You should define a helper function that consumes more parameters than acc-member-count that will do most of your calculations.

;; (acc-member-count loa item) Counts the number of times item appears in the loa
;; acc-member-count: (listof Any) Any  →  Nat
;; Examples
(check-expect (acc-member-count empty 5) 0)
(check-expect (acc-member-count (list 'I 'knew 'you 'were 'trouble 'when 'you
  'walked 'in 'trouble 'trouble 'trouble)
  'trouble) 4)
;; Tests
(check-expect (acc-member-count (list 'yellow 'green 'purple 'pineapple) 'help) 0)
Group Problem - Closest

Write a function closest that consumes a non-empty list of Posns and produces the Posn in the list that is closest to the origin, (make-posn 0 0). If two Posns are equally close to the origin, produce the one that occurred first. Your solution should use accumulative recursion. Include a Purpose and Contract for closest, as well as for any helper functions you write.

;; Examples
(check-expect (closest (list (make-posn 1 1)))
  (make-posn 1 1))
(check-expect (closest (list (make-posn 2 3)
                           (make-posn −3 1)
                           (make-posn 0 −4)))
  (make-posn −3 1))
Group Problem - anagram?

An anagram is a rearrangement (permutation) of the letters of a word. For example, “ate” is an anagram of “tea”, and “foster” is an anagram of “forest”. Write a predicate function called `anagram?` that determines if two strings are anagrams of each other, producing `true` if they are. You may assume that the input strings contain only lower-case letters (i.e., no spaces, punctuation, or upper-case letters).
Generative Recursion - list-gcd

Write a function list-gcd that consumes a list of natural numbers of size at least two and produces a gcd of all numbers in the list. Include a Purpose and Contract for list-gcd, as well as for any helper functions you write.

;; Examples
(check-expect (list-gcd (list 2 4 8 16)) 2)
(check-expect (list-gcd (list 27 36)) 9)