CS 135 Fall 2015
Tutorial 5: Accumulative Recursion and Nested Lists

Goals of this tutorial

You should be able to...

- write functions that process two lists at different rates.
- understand and process two-dimensional data represented by nested lists.
- identify and implement both of pure structural recursion and accumulative recursion.

Review: Processing two lists at different rates

If the two lists lst1, lst2 being consumed are of different lengths, all four possibilities for their being empty/nonempty are possible:

```
(and (empty? lst1) (empty? lst2))
(and (empty? lst1) (cons? lst2))
(and (cons? lst1) (empty? lst2))
(and (cons? lst1) (cons? lst2))
```

Exactly one of these is true, but all must be tested in the template.

Group Problem - Separate Words

Write a function called separate-words that consumes a (listof Char) which contains all non-whitespace characters of a paragraph, and a (listof Nat) representing the length of each word in the paragraph. It should produce a Str generated by inserting a space (#\space) to separate every pair of words in the paragraph.

Note that if the paragraph is empty your program should produce an empty string. If the (listof Nat) is empty your program should produce the whole paragraph in string form. Otherwise you can assume that the sum of lengths of all words is equal to the number of characters in the paragraph.

The contract and examples are given below.

Clicker Question - Nested Lists

```
(cons (cons 5 (cons 4 empty))
      (cons (cons 3 empty)
            (cons (cons 2 (cons 5 empty))
                  (cons 5 (cons 4 empty)))))
```

Which of the following lists is equivalent to the one above?

A (list 5 4 3 2 5 5 4)
B (list (list 5 4 3) (list 2 5) 5 4)
C (list (list 5 4) (list 3) (list 2 5) 5 4)
D (list (list 5) (list 4) (list 3) (list 2) (list 5) (list 5) (list 4))
E (list (list 5 4) (list 3) (list 2 5) (list 5 4))

Clicker Question - Parsing Nested Lists

```
(define lonum (list (list 5) (list 4 3) (list 2) 1))
```

Which of the following would produce a value of 3?

A (rest (first (rest (first lonum))))
B (first (rest (rest lonum)))
C (first (rest (rest (rest lonum))))
D (rest (rest (first (rest lonum))))
E (first (rest (first (rest lonum))))

Group Problem - Remove Firsts

Write a function called remove-firsts that consumes a (listof (listof Any)). It should produce a new (listof (listof Any)) with the first item removed from each of the sublists. Note that if a list in the (listof (listof Any)) is empty, then no item should be removed from it. Do not use helper functions. Include the contract, examples and tests.

For testing you can include the following test:

```
(check-expect (remove-firsts '((1 2 3) (2 3) (1) ()))
              '((2 3) (3) () ()))
```


Group Problem - Replace CS

Write a function called replace-cs that consumes a (listof (listof Char)). It should produce a new (listof (listof (anyof Nat Char))) with every occurrence of the upper-case characters C and S replaced by its corresponding ASCII value.

The purpose and contract for built-in functions char=? (assume it takes in exactly two parameters) and char→integer are given below, you may want to use them in your solution.

Including contract and examples is optional.

Group Problem - Replace CS

```
;; (char=? x y) determines whether the characters x and y are equal.
;; char=?: Char Char → Bool


;; (char→ integer x) lookups and produces the number that
;; corresponds to the given character in the ASCII table.
;; char→ integer: Char → Int
```

Review: Structural vs. Accumulative

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

In **accumulative recursion**, parameters are the same as above, plus one or more accumulators, or parameters 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 Sym) and a Sym. It should produce the number of times that Sym appears in the (listof Sym).

```
;; (member-count losym item) Counts the number of times item appears in the losym
;; member-count: (listof Sym) Sym → Nat
;; Examples
(check-expect (member-count empty 'help) 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 losym item) Counts the number of times item appears in the losym
;; acc-member-count: (listof Sym) Sym → Nat
;; Examples
(check-expect (acc-member-count empty 'help) 0)
(check-expect (acc-member-count
                (list 'I 'knew 'you 'were 'trouble 'when 'you
                      'walked 'in 'trouble 'trouble 'trouble)
                'trouble) 4)
```

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))
```