CS 135 Winter 2019
Tutorial 10: Abstract List Functions and Generative Recursion

More Problems - Abstract List Functions
You may omit the design recipe for the following list of exercises.

Write a function called `factors` that consumes a positive integer, `n`, and produces a list of its factors from 1 to `n` inclusive. Do not use explicit recursion.

(factors 30) ⇒ (list 1 2 3 5 6 10 15 30)

Write a function called `my-min` that consumes a non-empty list of numbers and produces the minimum number in the list. Do not use explicit recursion and do not use the built-in function `min`.

(my-min (list 1 2 3 2.3 5.5)) ⇒ −2.3

Write a function called `even` that consumes a list of natural numbers and counts the even numbers in it.

(even (list 1 78 4 5 6)) ⇒ 3

Foldl
Recall that `foldl` is the built-in function that abstracts accumulative recursion on lists. The effect of the application `(foldl f b (list x1 x2 .... xn))` is to compute the value of the expression

(f x1 (....(f x2 (f x1 b))...)...)
Group Problem: my-append
Next, write the function my-append that consumes two lists and appends them into one single list. Of the built in functions, you may use only cons, foldr and foldl. Try writing one solution using foldl and one solution using foldr. You may not use explicit recursion. Hint: you may want to reverse the first list for the foldl solution.

Clicker Question - Maximum of a list
Which of the following expressions can be used to find the maximum value of a list of numbers lon?
A (foldr max empty lon)
B (foldr max (first lon) (rest lon))
C (foldr (lambda (x y) (> x y)) empty lon)
D (foldr (lambda (x y) (> x y)) true lon)
E (foldr (lambda (x y) (> x y)) (first lon) (rest lon))

Group Problem: flatten
;; A nested list (Nest-List) is one of:
;; empty
;; (cons Any Nest-List)
;; (cons Nest-List Nest-List)

Next, write the function flatten that consumes a Nest-List and flattens them into one list. Of the built in functions, you may only use foldr, append and cons.
(flatten (list (list 1 2 3) (list 4 5 6) (list 7 8 9))) ⇒ (list 1 2 3 4 5 6 7 8 9)
(flatten (list (list (list 1) (list 2) 3) (list 4 5 6) (list 7 8 9)))
⇒ (list 1 2 3 4 5 6 7 8 9)
Group Problem: increasing-lists

Without using explicit recursion, write a function called increasing-lists that consumes a positive integer n and produces a list of n lists of natural numbers, where the $i^{th}$ list contains the first $i + 1$ natural numbers.

\[
\begin{align*}
(\text{increasing-lists } 1) & \Rightarrow \'((0)) \\
(\text{increasing-lists } 4) & \Rightarrow \\
\'((0)) & \\
(0 & 1) \\
(0 & 1 2) \\
(0 & 1 2 3))
\end{align*}
\]

Group Problem - ones-on-diagonal

In Tutorial 05, we wrote a function called ones-on-diagonal that produced a table as a \textit{(listof (listof Nat))} that contains 1s on the main diagonal and 0s everywhere else. We originally wrote it is using pure structural recursion, with multiple helper functions. This time we will write the same function but without explicit recursion and without any helper functions.

\[
\begin{align*}
(\text{ones-on-diagonal } 0) & \Rightarrow \text{empty} \\
(\text{ones-on-diagonal } 1) & \Rightarrow (\text{list (list 1)}) \\
(\text{ones-on-diagonal } 3) & \Rightarrow \\
\text{list (list 1 0 0)} & \\
\text{list 0 1 0) & } \\
\text{list 0 0 1)) & 
\end{align*}
\]

Clicker Question - Choosing ALFs

Consider the purpose and contract of some function, 3-in-a-row. Which abstract list function would be most useful when implementing 3-in-a-row?

\[
\begin{align*}
(3\text{-a-row desired lochar}) & \text{ determines if at least 3 consecutive occurrences of the consumed character, desired, appear in the list lochar} \\
3\text{-a-row: Char (listof Char)} & \rightarrow \text{Bool}
\end{align*}
\]

A foldr
B map
C filter
D build-list
Clicker Question: Functional Abstraction
Consider the following definition of mystery. Which of the following is the most appropriate contract for mystery, such that any function application obeying this contract will not produce an error?

```scheme
(define mystery (lambda (x y)
  (cond
    [(= (+ x 3) (- y 4)) (* x y)]
    [else (lambda (a b c)
              (<= (abs (- (string-length a)
                         (string-length b))) c)))]))
```

A ;; mystery: Num Num → (Num → (Str Str Num → Bool))
B ;; mystery: Num Num → (anyof Num Bool)
C ;; mystery: Num Num → (Str Str Num → (anyof Num Bool))
D ;; mystery: Num Num → (anyof Num (Str Str Num → Bool))
E mystery is a constant. It does not have a contract.

Generative Recursion: Bisection Method
In this question, we will approximate a root of a function using the bisection method. The process is described as follows:

1. Given two values left and right, determine the midpoint mid.
2. Given a function \( f \), compute the function value at the midpoint, \( f(mid) \).
3. If \( f(mid) \) is sufficiently small within a given tolerance of zero, simply produce the value of mid.
4. Otherwise, examine the sign of \( f(mid) \), and in the next recursive call, replace either left or right with mid, such that the range of the interval is halved, and there is a zero crossing within the new interval.

Generative Recursion: Bisection Method
Write a function \( \text{find-root} \) that consumes a function \( f \), two numbers left and right, and a positive number tolerance. The function will produce a root of the function \( f \) using the bisection method. You may assume the following:

- \( f \) is a continuous function
- \( left \leq right \)
- either \( f(left) \leq 0 \leq f(right) \) or \( f(right) \leq 0 \leq f(left) \)

\( (\text{find-root} \ (\lambda (x) \ (- \ (sqr x) 5)) \ 1 \ 3 \ 0.1) \Rightarrow 2.25 \)
(Optional) Group Problem - arbitrary-partition

Write a function `arbitrary-partition` that consumes a list called `lst`, and a list of positive integers called `sizes`. The numbers in `sizes` must add up to the length of `lst`. `arbitrary-partition` splits `lst` into sublists of lengths specified by `sizes`. Note this question is very similar to the `string->strlines` function that we skipped in lecture.

- `(arbitrary-partition '() '(0 0 0)) ⇒ '(() () ()`
- `(arbitrary-partition '(a b c) '(3)) ⇒ '((a b c))`
- `(arbitrary-partition '(a b c d e f g) '(3 1 2 1)) ⇒ '((a b c) (d) (e f) (g))`