Reminders

- Assignment 08 is due Monday, November 26 at 9:00pm
- Assignment 09 is due Monday, December 3 at 9:00pm

Foldl

Recall that foldl is the built-in function that abstracts accumulative recursion on lists. The effect of the application \((\text{foldl } f \ b \ (\text{list } x_1 \ x_2 \ldots \ x_n))\) is to compute the value of the expression

\[(f \ x_n \ (\ldots(f \ x_2 \ (f \ x_1 \ b)\ldots)))\]

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.
Group Problem: flatten
Next, write the function flatten that consumes a list of lists 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: flatten
Next, write flatten using foldl.

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))
Generative Recursion: Binary Search

In this question, we will approximate a function, \( f \), using the bisection method, or binary search.

The process is described as follows:

1. Given two values \( \text{left} \) and \( \text{right} \), determine the midpoint \( \text{mid} \).
2. Given a function \( f \), compute the function value at the midpoint, \( f(\text{mid}) \).
3. If \( f(\text{mid}) \) is sufficiently close to zero, simply produce the value of \( \text{mid} \).
4. Otherwise, examine the sign of \( f(\text{mid}) \), and in the next recursive call, replace either \( \text{left} \) or \( \text{right} \) with \( \text{mid} \), such that the range of the interval is halved, and there is a zero crossing within the new interval. Note we assume such a zero exists from the input restrictions.

Write a function \( \text{find-root} \) that consumes a function \( f \), two numbers \( \text{left} \) and \( \text{right} \), and a positive number \( \text{tolerance} \), so that a number would be considered a root as long as its value evaluated with \( f \) is within the tolerance. The function will produce a root of the function \( f \) using the binary search method. You may assume the following:

- \( f \) is a continuous function
- \( \text{left} < = \text{right} \)
- either \( f(\text{left}) < = 0 < = f(\text{right}) \) or \( f(\text{left}) > = 0 > = f(\text{right}) \)

Note from the input restrictions and the intermediate value theorem, there must exists a root of \( f \) between \( \text{left} \) and \( \text{right} \) (see the diagram in the next slide).

\[
\text{find-root (lambda (x) (- (sqr x) 5)) 1 3 0.1)} \Rightarrow 2.25 \\
\text{find-root (lambda (x) (- (sqr x) 5)) 0 300 0.001)} \Rightarrow 2.236175537109375
\]

Consider the image shown above, where \( a \) is our left and \( b \) is our right. The function shows that in the case that \( f(\text{right}) > = f(\text{left}) \), and our function \( f \) is continuous, it must pass through \( y=0 \) for some \( x=c \) and our goal is to approximate \( c \) to a small enough tolerance. If \( f(\text{left}) > = f(\text{right}) \), the case would be similar and we would also have a root.
Clicker Question - count-vertices
How many vertices are there in the following graph?

'((A (C F))
 (B ())
 (C (B))
 (D (A E))
 (E ())
 (F (B C G))
 (G (D)))

A 4
B 6
C 7
D 8

Clicker Question - Cycles in Explicit Graphs
Is there a cycle in this adjacency list representation of an explicit graph?
Note: the easiest way to see cycles is by drawing the graph

'((A (C F))
 (B ())
 (C (B))
 (D (A B E))
 (E ())
 (F (B C G))
 (G (D)))

A Yes
B No

Group Problem - node-exists?
Write a function node-exists? that consumes a graph and a symbol and produces whether the input symbol is a node in the graph.

;; A Node is a Sym
;; A Graph is a (listof (list Node (listof Node)))
(define graph-a
 '((A (C F))
  (B ())
  (C (B))
  (D (A B E))
  (E ())
  (F (B C G))
  (G (D)))

(node-exists? graph-a 'A) ⇒ true
(node-exists? graph-a 'Cat) ⇒ false
Group Problem - node-exists?
Next, write node-exists? without explicit recursion. You may try to write one solution using filter and another solution using foldr:

```scheme
;; A Node is a Sym
;; A Graph is a (listof (list Node (listof Node)))
(define graph-a
  '('((A (C F))
       (B ())
       (C (B))
       (D (A B E))
       (E ())
       (F (B C G))
       (G (D))))

(node-exists? graph-a 'A) ⇒ true
(node-exists? graph-a 'Cat) ⇒ false
```

Group Problem (optional) - ones-on-diagonal
In Tutorial 05, we wrote a function called ones-on-diagonal that produced a table as a (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.

```scheme
(ones-on-diagonal 0) ⇒ empty
(ones-on-diagonal 1) ⇒ (list (list 1))
(ones-on-diagonal 3) ⇒
  (list (list 1 0 0)
        (list 0 1 0)
        (list 0 0 1)))
```

Review: Graph Terminology
In a directed graph, an edge is an ordered pair of nodes (also called vertices), which we can represent by an arrow from one node to another.

A sequence of nodes \( v_1, v_2, \ldots, v_k \) is a path of length \( k - 1 \) if \( (v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k) \) are all edges.

If \( v_1 = v_k \), this is called a cycle.

Directed graphs without cycles are called DAGs (directed acyclic graphs).
Review: Adjacency List Representation
We can represent a graph as a list of pairs, each pair consisting of a symbol (the node's name) and a list of symbols (the names of the node's out-neighbours).
This is called the adjacency list representation.

;; A Node is a Sym

;; A Graph is a (listof (list Node (listof Node)))