Goals of this tutorial

You should be able to...

- understand the difference between structural and accumulative recursion.
- write functions using accumulative recursion.
- recognize generative recursion.
- perform structural recursion on binary trees

Clicker Question - Types of Recursion

```scheme
;; sum-pairs: (listof Num) → (listof Num)
;; requires: lon has an even number of elements
;; Examples:
(check-expect (sum-pairs empty) empty)
(check-expect (sum-pairs '(1 2 3 4)) '(3 7))
(define (sum-pairs lon)
  (cond
   [(empty? lon) empty]
   [else (cons (+ (first lon) (second lon))
              (sum-pairs (rest (rest lon))))]))
```

What kind of recursion is this?

A Pure structural
B Generative
C Accumulative
D This is not recursion??
Group Problem - Generative Recursion
Rewrite sum-pairs to not use generative recursion.

;;; sum-pairs: (listof Num) → (listof Num)
;;; requires: lon has an even number of elements
;;; Examples:
;;; (check-expect (sum-pairs empty) empty)
;;; (check-expect (sum-pairs '(1 2 3 4)) '(3 7))

(define (sum-pairs lon)
  (cond
    [(empty? lon) empty]
    [else (cons (+ (first lon) (second lon))
               (sum-pairs (rest (rest lon))))]))

Group Problem - Encode-msg
Recall the Data Definition for a Decryptor from last assignment:

;;; A Decryptor is a (list Nat Nat Nat)

Also recall the bonus question from last assignment: write a function
encode-msg that consumes two strings and produces the Decryptor needed
to hide the second string in the first one. You may assume that a valid
Decryptor exists and the second string has length 3 and does not contain *.
Use accumulative recursion. Of the built-in list functions, you may use only
cons, first, second, third, rest, empty?, cons?, and list. You may use
string->list and list->string but other string functions are banned. Use
accumulative or generative recursion.

Here are some examples:

(encode-msg "abcdefg" "abc") ⇒ (list 0 0 0)
(encode-msg "abcdefg" "beg") ⇒ (list 1 2 1)

Review: CS 135 Data Progression
Here is a visual representation of the kind of data studied in CS 135.
Review: CS 135 Data Progression

- One can think of a value as a type of tree where each element has 0 children. For a simple manipulation of such a value, no recursion is necessary.

- A list can be thought of as a tree where each element has only 1 child, and so the template has 1 recursive call for a function that consumes a list.

- Following this pattern, for a binary tree, each element (or node) has 2 children, so the template contains 2 recursive calls.

Group Problem: exists-in?

Recall the definition of a Node and Binary Tree from Lecture:

```
(define-struct node (key left right))
;; A Node is a (make-node Num BT BT)

;; A binary tree (BT) is one of:
;; * empty
;; * Node
```

Write a function called `exists-in?` that consumes a number and a binary tree, and determines whether or not the number exists in the binary tree as a key.

Group Problem: replace-val/bt

Recall:

```
(define-struct node (key left right))
;; A Node is a (make-node Num BT BT)

;; A binary tree (BT) is one of:
;; * empty
;; * Node
```

Write a function called `replace-val/bt` that consumes a binary tree and 2 strings called `old-key` and `new-key`. It replaces every occurrence of `old-key` in the tree as a key with value `new-key`.

Group Problem: max-gap

Recall:

\[
\text{(define-struct node (key left right))}
\]

\[\text{;; A Node is a (make-node Nat BST BST)}\]
\[\text{;; requires: key > every key in left BST} \]
\[\text{;; key < every key in right BST} \]
\[\text{;; A BST is one of:} \]
\[\text{;; * empty} \]
\[\text{;; * Node} \]

Write a function called max-gap that consumes a non-empty binary search tree and produces the maximum difference between any 2 nodes in the BST.

---

Structural vs. Accumulative

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

In \text{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).

If a parameter is used to produce the answer in the base case, then that parameter is probably an \textit{accumulator}.

---

Structural vs. Generative

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

In \text{generative recursion}, arguments to the recursive call may be calculated without any restrictions.

If an argument to the recursive call is changing so that it is closer to a base case, always refer to its \textit{data definition} to check that it is \textit{exactly one step} closer to its base case.