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
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).

If a parameter is used to produce the answer in the base case, then that parameter is probably an accumulator.
Group Problem: Structural Recursion

Write a function called `sum-lon` that consumes a `(listof Num)` and produces the sum of the numbers in the list. This function should use pure structural recursion.
Group Problem: Accumulative Recursion

Write a function called `sum-lon2` that consumes a `(listof Num)` and produces the sum of the numbers in the list. This function should use **accumulative** recursion.
Review: Structural vs. Generative

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

In 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 data definition to check that it is exactly one step closer to its base case.
;; 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  I don’t know
;; average-lon: (listof Num) Num Nat → Num
;; requires: lon is non-empty or n > 0
;; Examples:
(check-expect (average-lon '(1) 0 0) 1)
(check-expect (average-lon '(1 2 3) 0 0) 2)

(define (average-lon lon sum n)
  (cond
   [(empty? lon) (/ sum n)]
   [else (average-lon (rest lon)
                        (+ (first lon) sum)
                        (add1 n))])))

What kind of recursion is this?
A Pure structural
B Generative
C Accumulative
D I don’t know
Review: CS 135 Data Progression

Here is a visual representation of the kind of data studied in CS 135.

- children: 0 ➔ children: 1 ➔ children: 2 ➔ children: x
  - value ➔ list ➔ binary tree ➔ unbounded tree

Each element:
- value
- child1 (rest)

We've studied these

Now we're studying these

CS135 Fall 2017  Tutorial 6: Types of Recursion and Trees
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: replace-val/list

Write a function called replace-val/list that consumes a list of strings and 2 strings called old-str and new-str. It replaces every occurrence of old-str in the list with new-str.
Group Problem: replace-val/bt

(define-struct node (key val left right))
;; A Node is a (make-node Num Str 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-str and new-str. It replaces every occurrence of old-str in the tree as a value, with new-str.
Group Problem: exists-in?

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