We now want to be able to store information in more complicated ways.

It turns out that the structure of a tree has massively many applications in computer science. By using trees well we can make our code faster, often dramatically so.

You will see trees used in many contexts in future courses. Here we will discuss only a few key highlights.
The expression \((2 \times 6) + (5 \times 2))/(5 - 3)\) can be represented as a tree:
This phylogenetic tree tracks the evolution of COVID-19 in the first four months of the recent pandemic.

Image: nextstrain.org/ncov
A tree is a set of nodes and edges where an edge connects two distinct nodes.

A tree has three requirements:

- One node is identified as the root.
- Every node $c$ other than the root is connected by an edge to some other node $p$. $p$ is called the parent and $c$ is called the child.
- A tree is connected: for every node $n$ other than the root, the parent of $n$ or the parent of the parent of $n$ or the parent of the parent ... of $n$ will be the root.
Other useful terms:

- **leaves**: nodes with no children
- **internal nodes**: nodes that have children
- **labels**: data attached to a node
- **ancestors** of node \( n \): \( n \) itself, the parent of \( n \), the parent of the parent of \( n \), etc. up to the root
- **descendents** of \( n \): all the nodes that have \( n \) as an ancestor
- **subtree** rooted at \( n \): \( n \) and all of its descendents
Characteristics of trees

- Number of children of internal nodes:
  - exactly two
  - at most two
  - any number
- Labels:
  - on all nodes
  - just on leaves
- Order of children (matters or not)
- Tree structure (from data or for convenience)
A **binary tree** is a tree with at most two children for each node.

Binary trees are a fundamental part of computer science, independent of what language you use.

Binary arithmetic expression trees and evolution trees are both examples of binary trees.

We’ll start with the simplest possible binary tree. It could be used to store a set of natural numbers.
> Drawing binary trees

Note: We will consistently use $\texttt{Nat}$s in our binary trees, but we could use symbols, strings, structures, ...
(define-struct node (key left right))
;; A Node is a (make-node Nat BT BT)

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

The node’s label is called “key” in anticipation of using binary trees to implement dictionaries.

What is the template?
Aside: Tips for building templates

Follow the data definition. For each part of the data definition, if it

- is a defined data type, apply that type’s template.
- says “one of” or is mixed data, include a cond to distinguish the cases.
- is compound data (a structure), extract each of the fields, in order.
- is a list, extract the first and rest of the list.

Add elipses (…) around each of the above.

Apply the above recursively.
check your template! How did you do?
;; (sum-keys bt) sums the keys in the binary tree bt.
;; Examples
(check-expect (sum-keys empty) 0)
(check-expect (sum-keys (make-node 10 empty empty)) 10)
(check-expect (sum-keys (make-node 10

  (make-node 5 empty empty)

  empty)) 15)

;; sum-keys: BT -> Nat
(define (sum-keys bt)
Consider this sample tree:

```
(define test-tree
  (make-node 5
    (make-node 1
      (make-node 3 empty empty)
      (make-node 6 empty empty))
    (make-node 1
      (make-node 1
        (make-node 4 empty empty)
        (make-node 6 empty empty))
      (make-node 3 empty empty))))
```

Exercise

Write a function `(count-nodes tree target)` that consumes a BT and a Nat. It counts how many times `target` appears in `tree`.

- `(check-expect (count-nodes test-tree 1) 3)`
- `(check-expect (count-nodes test-tree 6) 2)`
- `(check-expect (count-nodes test-tree 7) 0)`
> Example: Increment keys

```
;; (increment tree) adds 1 to each key in the tree.
;; increment: BT -> BT
(define (increment tree)
  (cond
    [(empty? tree) empty]
    [else (make-node (add1 (node-key tree))
                       (increment (node-left tree))
                       (increment (node-right tree))))])
```
Write \((\text{count-leaves } t)\) which consumes a binary tree and produces the number of leaf nodes.

Write \((\text{count-evens } t)\) which consumes a binary tree and produces the number of nodes with an even key.

Write \((\text{reverse-tree } t)\) which consumes a binary tree and produces a new tree that has the same keys as \(t\) but every node has the left and right subtrees reversed (the left subtree becomes the right; the right subtree becomes the left).
We are now ready to search our binary tree for a given key. It will produce **true** if the key is in the tree and **false** otherwise.

Our strategy:

- See if the root node contains the key we’re looking for. If so, produce **true**.
- Otherwise, recursively search in the left subtree and in the right subtree. If either recursive search finds the key, produce **true**. Otherwise, produce **false**.
Now we can fill in our BT template to write our search function:

;; (search k tree) produces true if k is in tree; false otherwise.
;; search: Nat BT -> Bool
(define (search-bt k tree)
    (cond [(empty? tree) false]
          [ (= k (node-key tree)) true]
          [else (or (search-bt k (node-left tree))
                     (search-bt k (node-right tree)))]))

Is this more efficient than searching a list?
Correct the `contains?` function, above. Keep to the spirit of the existing function and use a Boolean expression with no helper functions.
Write a function, `search-bt-path`, that searches for an item in the tree. As before, it will return `false` if the item is not found. However, if it is found `search-bt-path` will return a list of the symbols `'left` and `'right` indicating the path from the root to the item.

If the tree contains a duplicate, produce the path to the left-most item.

The path from 6 to 9 is `(list 'right 'right 'left)`. 
(define test-tree
  (make-node 6 (make-node 3 ...)
    (make-node 10 ...))))

(check-expect (search-bt-path-v1 0 empty) false)
(check-expect (search-bt-path-v1 6 test-tree) empty)
(check-expect (search-bt-path-v1 3 test-tree) (list 'left))
(check-expect (search-bt-path-v1 9 test-tree) (list 'right 'right 'left))
(check-expect (search-bt-path-v1 0 test-tree) false)
;; search-bt-path-v1: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path-v1 k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) empty]
    [(list? (search-bt-path-v1 k (node-left tree)))
     (cons 'left (search-bt-path-v1 k (node-left tree)))]
    [(list? (search-bt-path-v1 k (node-right tree)))
     (cons 'right (search-bt-path-v1 k (node-right tree)))]
    [else false]]

Double calls to search-bt-path. Uggh!
;; search-bt-path-v2: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path-v2 k tree)
  (cond
   
   [(empty? tree) false]
   [ (= k (node-key tree))(empty)]
   [else (choose-path-v2 (search-bt-path-v2 k (node-left tree))
                           (search-bt-path-v2 k (node-right tree)))]))

(define (choose-path-v2 left-path right-path)
  (cond
   [(list? left-path) (cons 'left left-path)]
   [(list? right-path) (cons 'right right-path)]
   [else false])))
We will now make one change that can make searching much more efficient. This change will create a tree structure known as a binary search tree (BST).

For any given collection of keys, there is more than one possible tree.

How the keys are placed in a tree can improve the running time of searching the tree when compared to searching the same items in a list.
A Binary Search Tree (BST) is one of:
- empty
- a Node

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

A Node is a (make-node Nat BST BST)

Requires: key > every key in left BST
key < every key in right BST

The BST **ordering property**:

- key is greater than every key in left.
- key is less than every key in right.

Note: the ordering property holds in every subtree
(make-node 5
  (make-node 1 empty empty)
  (make-node 7
    (make-node 6 empty empty)
    (make-node 14 empty empty))))
There can be several BSTs holding a particular set of keys.
Main advantage: for certain computations, one of the recursive function applications in the template can always be avoided.

This is more efficient (sometimes considerably so).

In the following slides, we will demonstrate this advantage for searching and adding.

We will write the code for searching, and briefly sketch adding, leaving you to write the Racket code.
How do we search for a key $n$ in a BST? We reason using the data definition of BST.

- If the BST is empty, then $n$ is not in the BST.
- If the BST is of the form \((\text{make-node } k \text{ left right})\), and $k$ equals $n$, then we have found it.
- Otherwise it might be in either the left or right subtree.
  - If $n < k$, then $n$ must be in left if it is present at all, and we only need to recursively search in left.
  - If $n > k$, then $n$ must be in right if it is present at all, and we only need to recursively search in right.

Either way, we save one recursive function application.
(define (search-bst n t)
  (cond [(empty? t) false]
        [(= n (node-key t)) true]
        [(< n (node-key t)) (search-bst n (node-left t))]
        [(> n (node-key t)) (search-bst n (node-right t))])))
Write a function \( \text{count-smaller} \ (n \ t) \) that consumes a \( \text{Nat} \) and a \( \text{BST} \), and returns the number of keys in \( t \) that are less than \( n \).

\[
\text{define example}
\begin{align*}
&\text{(make-node } 5 \\
&\quad \text{(make-node } 1 \text{ empty empty)} \\
&\quad \text{(make-node } 7 \\
&\qquad \text{(make-node } 6 \text{ empty empty)} \\
&\qquad \text{(make-node } 14 \text{ empty empty})\text{))}
\end{align*}
\]

\[
\begin{align*}
\text{(check-expect} \ (\text{count-smaller} \ 8 \ \text{example}) \ 4) \\
\text{(check-expect} \ (\text{count-smaller} \ 1 \ \text{example}) \ 0) \\
\text{(check-expect} \ (\text{count-smaller} \ 100 \ \text{example}) \ 5)
\end{align*}
\]

For efficiency: don’t always search both children!
How do we add a new key, \( n \), to a BST \( t \)?

Reasoning from the data definition for a BST:

- If \( t \) is \texttt{empty}, then the result is a BST with only one node containing \( n \).

- If \( t \) is of the form \texttt{(make-node k left right)} and \( k = n \), the key is already in the tree and we can simply produce \( t \).

- Otherwise, \( n \) must go in either the \texttt{left} or \texttt{right} subtree.
  - If \( n < k \), then the new key must be added to \texttt{left}.
  - If \( n > k \), then the new key must be added to \texttt{right}.

Again, we need only make one recursive function application.
Imagine a function to convert a list of keys into a **BST**. Let’s call it `build-bst-from-list`.

How can we do this?

Since we are consuming a list, we reason using the data definition of a list:

- If the list is empty, the **BST** is empty.
- If the list is of the form `(cons k lst)`, we add the key `k` to the **BST** created from the list `lst`. The first key in the list is inserted **last**.

It is also possible to write a function that inserts keys in the opposite order.
If the BST has all left subtrees empty, it looks and behaves like a sorted list, and the advantage is lost.

In later courses, you will see ways to keep a BST “balanced” so that “most” nodes have nonempty left and right children. We will also cover better ways to analyze the efficiency of algorithms and operations on data structures.
Practise problems on binary search trees:

- Write a function, `bst-min`, which consumes a non-empty BST and produces the minimum value in the tree.
- Write a function, `bst-max`, which consumes a non-empty BST and produces the maximum value in the tree.
- Write a function, `bst-add`, which consumes a BST, `t`, and a new key, `n`. It produces a BST with all of the keys found in `t` as well as `n`. If `n` happens to already exist in `t` it produces a BST just like `t`.
- Write a function, `bst-from-list`, which consumes a (listof Nat) and produces a BST containing those same values. Use simple recursion.
- Write a function, `bst-from-list/acc`, which consumes a (listof Nat) and produces a BST containing those same values. Use accumulative recursion.
- Rewrite `search-bst` without a cond (using a single Boolean expression).
So far nodes have been `(define-struct node (key left right))`. We can **augment** the node with additional data:

`(define-struct node (key val left right)).`

- The name `val` is arbitrary – choose any name you like.
- The type of `val` is also arbitrary: could be a number, string, structure, etc.
- You could augment with multiple values.
- The set of keys remains unique.
- The tree could have duplicate values.
An augmented BST can serve as a dictionary that can perform significantly better than an association list.

Recall from Module 08 that a dictionary stores a set of (key, value) pairs, with at most one occurrence of any key. A dictionary supports lookup, add, and remove operations.

We implemented dictionaries using an association list, a list of two-element lists. Search could be inefficient for large lists.

We need to modify node to include the value associated with the key. search needs to return the associated value, if found.
(define-struct node (key val left right))
;; A binary search tree dictionary (BSTD) is either:
;; * empty
;; * (make-node Nat Str BSTD BSTD)

;; (search-bst-dict k t) produces the value associated with k
;; if k is in t; false otherwise.
;; search-bst-dict: Nat BSTD -> (anyof Str false)
(define (search-bst-dict k t)
  (cond[(empty? t) false]
       [ (= k (node-key t)) (node-val t)]
       [(< k (node-key t)) (search-bst-dict k (node-left t))]
       [(> k (node-key t)) (search-bst-dict k (node-right t))])))
(define test-tree (make-node 5 "Susan"
    (make-node 1 "Juan" empty empty)
    (make-node 14 "David"
        (make-node 6 "Lucy" empty empty)
        empty)))

(check-expect (search-bst-dict 5 empty) #false)
(check-expect (search-bst-dict 5 test-tree) "Susan")
(check-expect (search-bst-dict 6 test-tree) "Lucy")
(check-expect (search-bst-dict 2 test-tree) #false)
The expression \(((2 \times 6) + (5 \times 2))/(5 - 3)\) can be represented as a **binary expression tree:**
Representing binary arithmetic expressions

Internal nodes each have exactly two children.

Leaves have number labels.

Internal nodes have symbol labels.

We care about the order of children.

The structure of the tree is dictated by the expression.
A binary arithmetic expression (BinExp) is one of:

* a Num
* a BINode

(define-struct binode (op left right))

A Binary arithmetic expression Internal Node (BINode)

is a (make-binode (anyof '⁺ '⁻ '⁺' '⁻') BinExp BinExp)

Some examples of binary arithmetic expressions:

5
(make-binode '⁺ 2 6)
(make-binode '⁻ 2 (make-binode '⁺ 5 3))
A more complex example

(make-binode '/
    (make-binode '+ (make-binode '* 2 6)
        (make-binode '* 5 2))
    (make-binode '- 5 3))
;; binexp-template: BinExp -> Any
(define (binexp-template ex)
  (cond [(number? ex) (... ex ...)]
       [(binode? ex) (binode-template ex)]))

;; binode-template: BINode -> Any
(define (binode-template node)
  (... (binode-op node) ...
       (binexp-template (binode-left node)) ...
       (binexp-template (binode-right node)) ...))
(check-expect (eval 5) 5)
(check-expect (eval (make-binode '+ 2 5)) 7)
(check-expect (eval (make-binode '/' (make-binode '-' 10 2) (make-binode '+' 2 2))) 2)

(define (eval ex)
  (cond [(number? ex) ex]
        [(binode? ex) (eval-binode ex)]))
;; (eval-binode node) evaluates the expression represented by node.
;; eval-binode BINode -> Num

(define (eval-binode node)
  (cond [(symbol=? '*(binode-op node))
          (* (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '/(binode-op node))
          (/ (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '+/ (binode-op node))
          (+ (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '-/ (binode-op node))
          (- (eval (binode-left node)) (eval (binode-right node)))])))
(define (eval ex)
  (cond [(number? ex) ex]
        [(binode? ex) (eval-binode (binode-op ex)
                                 (eval (binode-left ex))
                                 (eval (binode-right ex)))]))

(define (eval-binode op left-val right-val)
  (cond [(symbol=? op '* ) (* left-val right-val)]
        [(symbol=? op '/ ) (/ left-val right-val)]
        [(symbol=? op '+ ) (+ left-val right-val)]
        [(symbol=? op '- ) (- left-val right-val)]))
Binary trees can be used for a large variety of application areas.

One limitation is the restriction on the number of children.

How might we represent a node that can have up to three children?

What if there can be any number of children?

Trees with an arbitrary number of children (subtrees) in each node are called **general trees**.

Our example of a general tree will be arithmetic expressions.
For binary arithmetic expressions, we formed binary trees.

Racket expressions using the functions $+$ and $\ast$ can have an unbounded number of arguments. For example,

```
(+ (* 4 2)
  3
  (+ 5 1 2)
  2)
```

For simplicity, we will restrict the operations to $+$ and $\ast$. 
> Example tree

\[(+ (* 4 2)) 3
  (+ 5 1 2)
  2)\]
For a binary arithmetic expression, we defined a structure with three fields: the operation, the first argument, and the second argument.

For a general arithmetic expression, we define a structure with two fields: the operation and a list of arguments (which is a list of arithmetic expressions).
Representing general trees

;; An Arithmetic Expression (AExp) is one of:
;; * Num
;; * OpNode

(define-struct opnode (op args))

;; An OpNode (operator node) is a
;; (make-opnode (anyof '∗ '⁺) (listof AExp)).

AExp is defined using OpNode and OpNode is defined using AExp. This will, eventually, lead to mutual recursion.
Examples of arithmetic expressions

3

(make-opnode ' * (list 3 4 5))

(make-opnode ' + (list (make-opnode ' * (list 4 2))
                      3
                      (make-opnode ' + (list 5 1 2))
                      2))
We wish to write a function `eval` to transform a `AExp` to the corresponding `Num`.

Consider the data definition of our `AExp`:

- In the base case, we have a `Num`. As before, this evaluates to itself. We’ll have:
  
  ```scheme
  (cond [(number? ex) ex]
  ```

- Otherwise, the `args` field contains a `(listof AExp)`.

A list... didn’t we have an easy way to deal with a list of things?
Remember map

If our expression contains a (listof AExp), to determine the value of the expression, we need to determine the value of each item in this list.

Each item in the list is a AExp. And how do we find the value of an (AExp? With eval.

Let’s look at this first at a high level. Each item in the list, we need to convert to a corresponding Num. We can use map:

```
*  
4 2 ↓
  8

3 ↓
  3

+  
5 1 2 ↓
  8
  2
```

```
2
```
Evaluating general arithmetic expressions with `map` and `foldr`.

Since we have a `(listof AExp)`, process each value in the list. With what? With the function that consumes an `AExp`. That is, the function we are writing right now. Something like:

```scheme
;; aexp-template: AExp -> ...
(define (aexp-template ex)
  (cond [(number? ex) (... ex)]
        [else (... (opnode-op ex)
                  (map aexp-template (opnode-args ex))))]))
```
Evaluating general arithmetic expressions with `map` and `foldr`.

Now the call to `map` gives us a `(listof Num)`. We know how to combine these: use `foldr`.

If we only need to add values, this code suffices:

```scheme
;; eval: AExp -> Num
(define (eval ex)
  (cond [(number? ex) ex]
        [(symbol=? '+ (opnode-op ex))
          (foldr + 0 (map eval (opnode-args ex)))]))
```

**Exercise**

Extend `eval` so it can also deal with expressions that contain nodes where the `op` is `'*'`.

**Exercise**

Further extend `eval` so to deal with `'-'` and `'/'.

Note that `(- a b c d ...)` ⇒ `(- a (+ b c d ...))` and `(/ a b c d ...)` ⇒ `(/ a (* b c d ...))`. 
This general arithmetic expression is an example of a *general tree*.

We can make a more general tree by separately defining

- a recursive tree type
- a leaf type.

Most generally,

```scheme
(define-struct gnode (label children))
;; a generic tree node (GNode)
;; is a (make-gnode Any (listof GenTree))

;; a generic Tree (GenTree) is either:
;; a GNode (recursive type) or
;; possibly something else (leaf type).
```
(define-struct gnode (label children))

;; a generic tree node (GNode) is a (make-gentree Any (listof GenTree))

;; a generic Tree (GenTree) is one of:
;; * a GNode (recursive type) or
;; * possibly something else (leaf type).

(define (gentree-function gt)
  (cond [(not (gnode? gt)) ; This is a leaf.
         (... gt)] ; Do something with a leaf.
        [else ; This is not a leaf, so it's a
         ( ; (make-gentree Any (listof GNode)). Work with the list!
           ... (gnode-label gt) ... ; Do something with the label.
           (foldr
            ... ; Function to combine answers.
            ... ; Base case for the combining function.
            (map ; Using this function, process each child.
                 gentree-function (gnode-children gt))))])))
Recalling the following data definition:

```
(define-struct gnode (label children))
;; a generic tree node (GNode)
;; is a (make-gnode Any (listof GenTree))
```

;; a generic Tree (GenTree) is either:
;;  a GNode (recursive type) or
;;  possibly something else (leaf type).

---

**Exercise**

Complete `count-leaves`.  
;; (count-leaves T) return the number of leaves in T.  
;; count-leaves: GenTree -> Nat  
;; Examples:

```
(check-expect (count-leaves (make-gnode 'wut (list "foo" "bar" "baz"))) 3)
(check-expect (count-leaves (make-gnode '+ (list 2 3 (make-gnode '*' (list 6 7 42))))) 5)
```
a Sentence is a GenTree where:
  each label is a Sym
  each leaf is a Str.

(define catS (make-gnode
  'S (list
      (make-gnode
        'NP (list
            (make-gnode 'D (list "the"))
            (make-gnode 'N (list "cat"))))
      (make-gnode
        'V (list "ate")))))

Exercise
Write a function (sentence->list S) that consumes a Sentence and returns a (listof Str) containing the words in S, in order.

(check-expect (sentence->list catS) (list "the" "cat" "ate"))
Now that we have some experience working with trees at a high level, let’s take another look at them using only recursion. We start by again evaluating an AExp.

Notice we have 2 data definitions, each referring to the other:

```scheme
;;; An Arithmetic Expression (AExp) is one of:
;;;  * Num
;;;  * OpNode
```

```scheme
(define-struct opnode (op args))
;;; An OpNode (operator node) is a
;;; (make-opnode (anyof '* '+) (listof AExp)).
```

It is possible, but unwise, to treat both of these in one recursive function.

Instead, we will write two recursive functions: one for each data definition.
;; (eval exp) evaluates the arithmetic expression exp.
;; Examples:
(check-expect (eval 5) 5)
(check-expect (eval (make-opnode '+ (list 1 2 3 4))) 10)
(check-expect (eval (make-opnode '* (list 2 3 4))) 24)
(check-expect (eval (make-opnode '+ (list 1

    (make-opnode '* (list 2 3))

3))) 10)

;; eval: AExp -> Num
(define (eval exp)
;; (eval exp) evaluates the arithmetic expression exp.
;; Examples:
(check-expect (eval 5) 5)
(check-expect (eval (make-opnode '+ (list 1 2 3 4))) 10)
(check-expect (eval (make-opnode '* (list 2 3 4))) 24)
(check-expect (eval (make-opnode '+ (list 1
        (make-opnode '* (list 2 3))
        3))) 10)

;; eval: AExp -> Num
(define (eval exp)
  (cond [(number? exp) exp]
        [(opnode? exp) (apply-op (opnode-op exp)
                                   (opnode-args exp))])))
(apply-op op args) applies the arithmetic operator op to args.

Examples:
(check-expect (apply-op '+ (list 1 2 3 4)) 10)
(check-expect (apply-op '* (list 2 3 4)) 24)
(check-expect (apply-op '+ (list 1 (make-opnode '* (list 2 3)))) 7)
(check-expect (apply-op '+ empty) 0)
(check-expect (apply-op '* empty) 1)
;; apply-op: (anyof '+ '* ) (listof AExp) -> Num
(define (apply-op op args)
  (cond [[(empty? args) (cond [[(symbol=? op '+) 0]
                                 [[(symbol=? op '* ) 1]]
                              [(symbol=? op '+) (+ (eval (first args))
                                         (apply-op op (rest args)))]
                              [(symbol=? op '* ) (* (eval (first args))
                                         (apply-op op (rest args)))]]))
Mutual recursion arises when complex relationships among data result in cross references between data definitions.

The number of data definitions can be greater than two.

Structures and lists may also be used.

In each case:

- create templates from the data definitions and
- create one function for each template.
We can generalize from allowing only two arithmetic operations and numbers to allowing arbitrary functions and variables.

In effect, we have the beginnings of a Racket interpreter.

But beyond this, the type of processing we have done on arithmetic expressions can be applied to tagged hierarchical data, of which a Racket expression is just one example.

Organized text and Web pages provide other examples.
(list 'chapter
   (list 'section
      (list 'paragraph "This is the first sentence."
           "This is the second sentence.")
      (list 'paragraph "We can continue in this manner.")
   (list 'section ...)
   ...
)
Representing a web page

(list 'webpage
  (list 'title "CS 135: Designing Functional Programs")
  (list 'paragraph "Let me tell you about"
    (list 'bold "something important.")
    "Enjoy the course!")
  (list 'horizontal-line)
  (list 'paragraph "(Last modified yesterday.)")
)

Note that these examples match our general tree pattern very closely: some items are Str, and others are lists:

1. The first of each list is always a Sym: the label.
2. The rest of each list is a listof something: the stuff inside.

We might say:

;; a LTree is one of:
;; * a Str
;; * a (cons Sym (listof LTree))
Exercise
Using \texttt{map} and \texttt{foldr}, but without any helper functions, write a function 
\texttt{(count-label label tree)}.
It consumes a \texttt{LTree} and a \texttt{Sym}, and counts the number of times \texttt{label} appears in \texttt{tree}.

Remember: if \texttt{tree} is a list,

- \texttt{(first tree)} is the label
- \texttt{(rest tree)} is the list of children.

(Hint: use \texttt{lambda} to pass \texttt{tag} to the recursive call.)

Use mutual recursion to rewrite \texttt{count-label} without using \texttt{map} or \texttt{foldr}.

Remember: write 2 functions:

1. one that consumes a \texttt{LTree}, with 2 cases;
2. one that consumes a \texttt{(listof LTree)}, with the 2 cases of the \texttt{(listof X)} template.
We have discussed flat lists (no nesting):

\[\text{list} \ 'a\ 1\ "hello"\ 'x\]

and lists of lists (one level of nesting):

\[\text{list} (\text{list} 1\ "a")\ (\text{list} 2\ "b")\ (\text{list} 3\ "c")\]

We now consider **nested lists** (arbitrary nesting):

\[\text{list}\ (\text{list}\ 1\ (\text{list}\ 2\ 3))\ 4\ (\text{list}\ 5\ (\text{list}\ 6\ 7\ 8)\ 9\ \text{empty})\]

This is a further simplification of the LTree. We no longer have the label: just the structure and the leaves. Nevertheless, they are often a useful way to store information.
It is often useful to visualize a nested list as a tree, in which the leaves correspond to the elements of the list, and the internal nodes indicate the nesting:
Data definition for nested lists

Sample nested lists:

- empty
- (list (list 1 2) 3 (list 4 empty))
- (list 1 2 3)
- (list 1 (list 2 3) 4)

Observations:

- A nested list might be empty
- The first item of a non-empty nested list is either:
  - a nested list
  - a single item (a number, not a list)
- The rest of a non-empty nested list is a nested list

;: A nested list of numbers (Nest-List-Num) is one of:
;: * empty
;: * (cons Nest-List-Num Nest-List-Num)
;: * (cons Num Nest-List-Num)

This can be generalized to generic types: (nested-listof X).
The template follows from the data definition.

;; nest-lst-template: (nested-listof X) -> Any
(define (nest-lst-template lst)
  (cond [[(empty? lst) ...]
          [(list? (first lst)) (... (nest-lst-template (first lst)) ... (nest-lst-template (rest lst)) ...)]
          [else (... (first lst) ...
                     (nest-lst-template (rest lst)) ...)]]
> The function count-items

;; (count-items nl) counts the number of items in nl.

;; Examples:

(check-expect (count-items empty) 0)
(check-expect (count-items (list (list 10 20) 30)) 3)
(check-expect (count-items (list (list 10 20) empty "thirty")) 3)

;; count-items: (nested-listof X) -> Nat

(define (count-items lst)
  (cond [(empty? lst) 0]
        [(list? (first lst)) (+ (count-items (first lst))
                                  (count-items (rest lst)))]
        [else (+ 1 (count-items (rest lst)))]))
Condensed trace of count-items

(count-items (list (list 10 20) 30))
⇒ (+ (count-items (list 10 20)) (count-items (list 30)))
⇒ (+ (+ 1 (count-items (list 20))) (count-items (list 30)))
⇒ (+ (+ 1 (+ 1 (count-items empty))) (count-items (list 30)))
⇒ (+ (+ 1 (+ 1 0)) (count-items (list 30)))
⇒ (+ (+ 1 1) (count-items (list 30)))
⇒ (+ 2 (count-items (list 30)))
⇒ (+ 2 (+ 1 (count-items empty)))
⇒ (+ 2 (+ 1 0)) ⇒ (+ 2 1) ⇒ 3
`flatten` produces a flat list from a nested list.

`; (flatten lst) produces a flat list with all the elements of lst.

Examples:
```
(check-expect (flatten (list 1 2 3)) (list 1 2 3))
(check-expect (flatten (list (list 1 2 3) (list 'a 'b 'c))) (list 1 2 3 'a 'b 'c))
(check-expect (flatten (list (list 1 2 3) empty (list 'a (list 'b 'c)))) (list 1 2 3 'a 'b 'c))
```

`; flatten: (nested-listof X) -> (listof X)

We make use of the built-in Racket function `append`.

```
(append (list 1 2) (list 3 4)) ⇒ (list 1 2 3 4)
```
> Flattening a nested list

;; (flatten lst) produces a flat list with all the elements of lst.

;; Examples:
(check-expect (flatten (list 1 2 3)) (list 1 2 3))
(check-expect (flatten (list (list 1 2 3) (list 'a 'b 'c))) (list 1 2 3 'a 'b 'c))
(check-expect (flatten (list (list 1 2 3) empty (list 'a (list 'b 'c))))
  (list 1 2 3 'a 'b 'c))

;; flatten: (nested-listof X) -> (listof X)
(define (flatten lst)
  (cond
    [(empty? lst) empty]
    [(list? (first lst)) (append (flatten (first lst))
                                (flatten (rest lst)))]
    [else (cons (first lst)
                 (flatten (rest lst)))]))
(flatten (list (list 10 20) 30))
⇒ (append (flatten (list 10 20)) (flatten (list 30)))
⇒ (append (cons 10 (flatten (list 20))) (flatten (list 30)))
⇒ (append (cons 10 (cons 20 (flatten empty))) (flatten (list 30)))
⇒ (append (cons 10 (cons 20 empty)) (flatten (list 30)))
⇒ (append (cons 10 (cons 20 empty)) (cons 30 (flatten empty)))
⇒ (append (cons 10 (cons 20 empty)) (cons 30 empty))
⇒ (cons 10 (cons 20 (cons 30 empty)))
Following the template, write a function that adds up all the values in a
(nested-listof Num).
(check-expect (nest-lst-sum (list 1 1)) 2)
(check-expect (nest-lst-sum (list 1 2 (list 3 4) empty 7
    (list (list 1 4) 1))) 23)

Following the template, write a function that calculates the maximum depth of a
(nested-listof X).
(check-expect (nl-max-depth empty) 0)
(check-expect (nl-max-depth (list empty)) 1)
(check-expect (nl-max-depth (list 1)) 1)
(check-expect (nl-max-depth (list 1 (list 1 2))) 2)
You should be familiar with tree terminology.

You should understand the data definitions for binary trees, binary search trees, evolutionary trees, binary arithmetic expressions, general arithmetic expressions, and nested lists.

You should understand how the templates are derived from those definitions, and how to use the templates to write functions that consume those types of data.

You should understand the definition of a binary search tree and its ordering property.

You should be able to write functions which consume binary search trees, including those sketched (but not developed fully) in lecture.
Goals of this module (2/2)

- You should be able to develop and use templates for other binary trees, not necessarily presented in lecture.
- You should understand the idea of mutual recursion for both examples given in lecture and new ones that might be introduced in lab, assignments, or exams.
- You should be able to develop templates from mutually recursive data definitions, and to write functions using the templates.
In this module we added the following to our toolbox:

foldl foldr lambda map range

These are the functions and special forms currently in our toolbox: