If you have not already, make sure you

- Read *How to Design Programs* Sections 14, 15, 16.
We are going to discuss how to represent mathematical expressions, such as

\[((2 \times 6) + (5 \times 2)) \div (5 - 3)\]

There are three things we need to represent here:

- Each number we can store as a `Num`.
- The brackets, which represent structure, are the main topic of this module.
- The operators (\(\times\), +, etc.).

To represent an operator we could use a `Str`: "\(*\)", "\(+\)", etc. But we will not. Instead, we will use a new data type for symbols: `Sym`. 
Symbols

A Sym is a new type of data. It starts with a single quote ', followed by its name; the name must be a valid identifier.
Examples: 'blue, 'spades, '*', 'x22.

The only computations possible on Sym is equality comparison and the symbol? type predicate.

(define mysymbol 'blue)
(equal? mysymbol 42) => #false
(equal? mysymbol "blue") => #false
(equal? mysymbol 'blue) => #true
(equal? mysymbol 'red) => #false
(symbol? '$@) => #true
(symbol? "the artist formerly known as Prince") => #false
Symbols

Everything possible with \texttt{Sym} could be done using \texttt{Str}.

But code using \texttt{Sym} may be clearer. Since there are no functions to do anything with a \texttt{Sym}, the reader can be sure the values is either exactly one \texttt{Sym}, or exactly another.

When there is a small, fixed number of values to consider, and the only needed computation is equality, consider using a \texttt{Sym}.
A little practice with Sym

Complete count-sheep.

;;; (count-sheep L) return the number of 'sheep in L.
;;; count-sheep: (listof Any) -> Nat
;;; Example:
(check-expect (count-sheep (list 6 'sheep 'ram 3.14 'sheep 'ox)) 2)

Checking if a value is a Sym and checking if a Sym is a particular Sym is all you can do with them.
Operators such as $+,-,\times,$ and $\div$ take two arguments, so we call them “binary operators”. We have worked with these ourselves for years, but now we can get our computers to work with them for us. Consider:

$$((2 \times 6) + (5 \times 2)) \div (5 - 3)$$

We can split this expression into two expressions, and then recursively split them!
c is a sibling of d, and d is a sibling of c. e, f, and g are siblings of each other.
Important features:

- Every **leaf** is a number
- Every **internal node** is an operator, and it operates on its **children**.

To represent the operators, we will use `Sym`:

```
;; an Operator is (anyof '+' '-' '*' '/')
```

One good approach for describing the tree: use a structure for each internal node.

```
(define-struct binode (op arg1 arg2))
;; a binary arithmetic expression internal node (BINode)
;; is a (make-binode Operator BinExp BinExp)
```

```
;; A binary arithmetic expression (BinExp) is either:
;; a Num or
;; a BINode
```

```
(make-binode '* 7 6)  ⇔  7 × 6
(make-binode '* 7 (make-binode '+ 2 4))  ⇔  7 × (2 + 4)
```
Evaluating binary arithmetic expressions

\[(\text{make-binode '*/7 ((make-binode '/+2 4))})\]

Evaluation works just like in tracing: evaluate arguments, recursively. Then apply the appropriate function to the two arguments. A number evaluates to itself.

**Exercise**

Complete `eval-binexp` so it can handle '+ and '*.

```scheme
;; (eval-binexp expr) return the value of expr.
;; eval-binexp: BinExp -> Num
;; Examples:
(check-expect (eval-binexp (make-binode '*/7 6)) 42)
(check-expect (eval-binexp (make-binode '*/7 ((make-binode '/+4 2)))) 42)
```

**Ex.**

For completeness, extend `eval-binexp` so it also handles '-' and '/.'
In order to develop a template which works generally, I will use a generic tree definition:

```
(define-struct bintree (label child1 child2))
;; a BinTree is a (make-bintree Any Any Any)
;; Requires: child1 and child2 are each either
;; a BinTree or
;; a leaf. Define the leaf separately!
```

Following this, many binary tree functions can be created starting from the following template:

```
(define (my-bintree-fun T)
  (cond [(... T) ...] ; Some base case.
        [else (... (bintree-label T) ...
                           ... (my-bintree-template (bintree-child1 T)) ...
                           ... (my-bintree-template (bintree-child2 T)) ... )]])
```
I can store information in a tree very efficiently. Suppose we design as follows: a tree stores a value as its label. It stores all smaller values in a tree, which is its left child. It stores all larger values in a tree, which is its right child.

(define-struct snode (key left right))
;; a SNode is a (make-snode Num SSTree SSTree)

;; a simple search tree (SSTree) is either
;; * () or
;; * a SNode, where keys in left are less than key, and in right greater.

(define tree12
 (make-snode 12
  (make-snode 10
   ()
   (make-snode 11 '() '()))
  (make-snode 15 '() '())))
(define-struct snode (key left right))
;; a SNode is a (make-snode Num SSTree SSTree)

;; a simple search tree (SSTree) is either
;; * '() or
;; * a SNode, where keys in left are less than key, and
in right greater.

(define tree12
  (make-snode 12
    (make-snode 10
      '()
      (make-snode 11 '() '()))
    (make-snode 15 '() '())))

Exercise
Complete count-leaves.
;; (count-leaves tree) return the number of leaves in tree.
;; count-leaves: SSTree -> Nat
;; Example:
(check-expect (count-leaves tree12) 2)
(define-struct snode (key left right))
;; a SNode is a (make-snode Num SSTree SSTree)

;; a simple search tree (SSTree) is either
;;  * '() or
;;  * a SNode, where keys in left are less than key, and
;;    in right greater.

(define tree12
  (make-snode 12
    (make-snode 10 '()
      (make-snode 11 '() '()))
    (make-snode 15 '() '())))

Exercise
Complete tree-sum.
;; (tree-sum tree) return the sum of all keys in tree.
;; tree-sum: SSTree -> Num
;; Example:
(check-expect (tree-sum tree12) 48)
(define-struct snode (key left right))

;; a SNode is a (make-snode Num SSTree SSTree)

;; a simple search tree (SSTree) is either
;; * () or
;; * a SNode, where keys in left are less than key, and in right greater.

(define tree12 (make-snode 12
                  (make-snode 10 ()
                  (make-snode 11 () ()
                  (make-snode 15 () ()
                  ())))))

Exercise

Complete tree-search. Clever bit: only search left or right, not both.

;; (tree-search tree item) return #true if item is in tree.
;; tree-search: SSTree Num -> Bool
;; Example:
(check-expect (tree-search tree12 10) #true)
(check-expect (tree-search tree12 7) #false)
A binary search tree will be fast to search only if it is balanced, meaning both children are of approximately the same size.

A completely balanced tree:

```
        4
       / \
      2   6
     / \ / \ 
    1  3 5  7
```

A completely unbalanced tree:

```
        1
       / \
      2   7
     /   / 
    3   6   
   /  /  /  
  5  6  7  4
```

It is thus important to keep such trees “reasonably well” balanced. This is an interesting but complex topic! We are not going to discuss how to keep trees balanced here, but it’s something to think about.
Now we can find items in a binary tree faster than if we simply stored them in a list. Earlier as dictionaries we stored a list of key-value pairs:

```scheme
(define student-dict
  (list (list 6938 (make-student "Al Gore" "government"))
       (list 7334 (make-student "Bill Gates" "appliedmath"))
       (list 7524 (make-student "Ben Bernanke" "economics")))
```

By extending our tree data structure a tiny bit, we can store a value (val) as well as a key, to make a fast dictionary!

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

;; A binary search tree (BST) is either
;; * () or
;; * (make-node Nat Any BST BST)...
;; which satisfies the ordering property recursively:
;; * every key in left is less than key
;; * every key in right is greater than key
(define-struct node (key val left right))
(define-struct student (name programme))

(define student-bst
  (make-node 7524
    (make-student "Ben Bernanke" "economics")
    (make-node 6938
      (make-student "Al Gore" "government")
      ()
    )
  )
(key: 7524
val: Bernanke...)

(make-node 7334
  (make-student "Bill Gates" "appliedmath")
  ()
)(key: 6938
val: Gore...)

(make-node 8535
  (make-student "Conan O'Brien" "history")
  ()
)(key: 8535
val: O'Brien...)

(make-node 8838
  (make-student "Barack Obama" "law")
  ()
)(key: 8838
val: Obama...)

For compactness, we usually draw only the key on our trees. The val could be any type.
Dictionary lookup

Our lookup code is almost exactly the same as tree-search in a SSTree. The are two small differences: the structure has an extra field; and when we find the key, we return the corresponding val, instead of #true.

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

;;; (dict-search dict item) return correct val if item is in dict.
;;; dict-search: BST Num -> Any
;;; Example:
(check-expect (dict-search student-bst 6938) (make-student "Al Gore" "government"))
(check-expect (dict-search student-bst 9805) #false)

(define (dict-search dict item)
  (cond [(empty? dict) #false]
        [(= item (node-key dict)) (node-val dict)]
        [(< item (node-key dict)) (dict-search (node-left dict) item)]
        [(> item (node-key dict)) (dict-search (node-right dict) item)]))
Creating a BST

It’s easier to create a list of key-value pairs than a BST. So let’s discuss how to convert a list to a BST.

First consider: how can we add one key-value pair to a BST?

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

\[
\text{;; A binary search tree (BST) is either}
\text{;; * '() or}
\text{;; * (make-node Nat Any BST BST)}\ldots
\text{;; which satisfies the ordering property recursively:}
\text{;; * every key in left is less than key}
\text{;; * every key in right is greater than key}
\]
Creating a BST

- If the BST is empty, return a single node which represents the key-value pair. (Base case.)
- Otherwise, add the key to the left or right child, as appropriate.

How to do this? We need to use the template for a function that consumes and returns a make-node, and modify it a little.

```scheme
;;; mynode-template: BST -> Any
(define (mynode-template tree)
  (make-node (node-key tree)
             (node-val tree)
             (node-left tree)
             (node-right tree)))
```

Most of this we don’t change; we make a copy of the same node. Recursively add to the (node-left tree) or (node-right tree).
Complete dict-add.

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

;; A binary search tree (BST) is either
;; * () or
;; * (make-node Nat Any BST BST)...
(define-struct association (key val))
;; An Association is a (make-association Nat Any)

;; (dict-add newassoc tree) return tree with newassoc added.
;; dict-add: Association BST -> BST
;; Examples:
(check-expect (dict-add (make-association 4 "four") '())
  (make-node 4 "four" () '()))

(check-expect
  (dict-add (make-association 6 "six")
    (dict-add (make-association 2 "two")
      (dict-add (make-association 4 "four") '())))
  (make-node 4 "four" (make-node 2 "two" () '())
    (make-node 6 "six" () '()))
Creating a BST

These ideas will let us add one item to a tree. How can we extend that to add a list of items? Idea: start with an empty tree, '()', and use foldr or recursion. Expand the BST using dict-add for each item in the list.

Complete expand-bst.

```scheme
;; (expand-bst L tree) add all items in L to tree, adding the last first.
;; expand-bst: (listof (list Num Any)) BST -> BST
;; Example:
(check-expect
 (expand-bst (list (make-association 4 "four")) '())
(make-node 4 "four" '() '()))
(check-expect
 (expand-bst (list (make-association 2 "two")
                 (make-association 6 "six")
                 (make-association 4 "four")) '())
(make-node 4 "four"
          (make-node 2 "two" '() '())
          (make-node 6 "six" '() '())))
```

Exercise
Arithmetic Expressions

Recall with binary trees we could represent an expression containing binary operators (which have exactly 2 arguments), such as

$$((2 \times 6) + (5 \times 2)) \div (5 - 3)$$

But in Racket we aren’t required to have just 2 arguments. How could we represent

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

We can make a new structure where each node can have any number of children, instead of just 2.

Here we still label each node; the only difference is the number of children.
Representing binary vs general arithmetic expressions

Previously, our data definitions were as follows:

;; an Operator is (anyof '+ ' '- ' '* ' '/')

(define-struct binode (op arg1 arg2))
;; a binary arithmetic expression internal node (BINode)
;; is a (make-binode Operator BinExp BinExp)

;; A binary arithmetic expression (BinExp) is either:
;; a Num or
;; a BINode

Now we want all same, except: instead of arg1 and arg2, we will use a list:

(define-struct ainode (op args))
;; an arithmetic expression internal node (AINode)
;; is a (make-ainode Operator (listof AExp))

;; An arithmetic expression (AExp) is either:
;; a Num or
;; a AINode
Representing expressions

So given the data definition:

(define-struct ainode (op args))

;; an arithmetic expression internal node (AINode)
;; is a (make-ainode Operator (listof AExp))

;; An arithmetic expression (AExp) is either:
;; a Num or
;; a AINode

How can we represent an expression such as (+ (* 4 2) 3 (+ 5 1 2) 2) ?

(make-ainode '+ (list (make-ainode '* (list 4 2))
  3
  (make-ainode '+ (list 5 1 2))
  2))
Interpreting expressions

When we evaluated binary trees we used the following function:

```scheme
;; (eval-binexp expr) return the value of expr.
;; eval-binexp: BinExp -> Num
(define (eval-binexp expr)
  (cond [(number? expr) expr] ; numbers evaluate to themselves
        [(equal? (binode-op expr) '*)
          (* (eval-binexp (binode-arg1 expr))
             (eval-binexp (binode-arg2 expr)))]
        [(equal? (binode-op expr) '+)
          (+ (eval-binexp (binode-arg1 expr))
             (eval-binexp (binode-arg2 expr)))]))
```

All that is different now is that we have a list of values, `args`, instead of just `arg1` and `arg2`.

We will now complete a function to evaluate an arithmetic expression:

```scheme
;; (eval-ainexp expr) return the value of expr.
;; eval-ainexp: AExp -> Num
;; Examples:
(check-expect (eval-ainexp (make-ainode '*(list 2 3 5))) 30)
```
Evaluating Expressions

Our code for binary expressions is almost perfect. Instead of evaluating exactly two items, 
(binode-arg1 expr) and (binode-arg2 expr), then combining them with * or +, we have this list. 
We need to replace the code:

```
(+ (eval-binexp (binode-arg1 expr))
   (eval-binexp (binode-arg2 expr))))
```

We will evaluate each argument in the list. We could do this recursively. Or use map!
```
(map eval-ainexp (ainode-args expr)))
```

This takes a list of expressions, evaluates each one, and returns the resulting list.

Now we need to combine them. Again, we could do this recursively. But we don’t need to! 
Use foldr instead:
```
(foldr + 0
       (map eval-ainexp (ainode-args expr)))
```
Evaluating Expressions

The whole code then becomes:

;; (eval-ainexp expr) return the value of expr.
;; eval-ainexp: AExp -> Num
;; Examples:
(check-expect (eval-ainexp (make-ainode '*' (list 2 3 5))) 30)

(define (eval-ainexp expr)
  (cond
   [(number? expr) expr] ; numbers evaluate to themselves
   [(equal? (ainode-op expr) '*)
     (foldr * 1
               (map eval-ainexp (ainode-args expr)))]
   [(equal? (ainode-op expr) '+)
     (foldr + 0
               (map eval-ainexp (ainode-args expr)))]
   ; subtraction and division are trickier. Only one is included here:
   [(equal? (ainode-op expr) '-)
     (- (eval-ainexp (first (ainode-args expr)))
         (foldr + 0 (map eval-ainexp (rest (ainode-args expr))))))])}
Recalling the following data definition:

```
(define-struct ainode (op args))
;; an arithmetic expression internal node (AINode)
;; is a (make-ainode Operator (listof AExp))

;; An arithmetic expression (AExp) is either:
;;   a Num or
;;   a AINode
```

Exercise

Complete `count-leaves`.

```
;; (count-leaves expr) return the number of leaves in expr.
;; count-leaves: AExp -> Nat
;; Examples:
(check-expect (count-leaves
               (make-ainode
                '+ (list 2 3 (make-ainode
                               '* (list 6 7 42)))))) 5)
```
Mutual Recursion

When one function calls a second, and the second calls the first, it is called **mutual recursion**. Our approach to writing such code is unchanged: write a template for each datatype, based on the data definition; then build functions following the template. Consider:

;; an EvenNat is either
;; 0 or
;; (add1 x) where x is an OddNat.

;; an OddNat is a Nat that is not even.

;; (is-even? n) return #true if n is an EvenNat, otherwise #false.
;; is-even?: Nat -> Bool

(define (is-even? n)
  (cond [(= n 0) #true]
        [else (is-odd? (sub1 n))]))

(define (is-odd? n)
  (not (is-even? n)))
Mutual Recursion

(define-struct ainode (op args))
;; an arithmetic expression internal node (AINode)
;; is a (make-ainode Operator (listof AExp))

;; An arithmetic expression (AExp) is either:
;;   a Num or
;;   a AINode

We have defined:

1. AINode, which is defined in terms of (listof AExp)
2. (listof AExp), which is defined in terms of AINode.

If we wrote a recursive function evaluate-and-add to work through the (listof AExp), the function would itself call eval-ainexp. But eval-ainexp would call evaluate-and-add. This is a mutually-recursive definition.

It is easy to get lost thinking about mutually-recursive code. To help make sense of it: focus on one recursive definition, and make the code for it work. Then consider the other recursive definition.
Leaf Labelled Trees

Earlier we worked with trees which had labels on all the nodes. Each node consisted of a (make-node Label (listof Node)); the Label indicating something about the node (in our example, the operator, '+' or '*').

We can simplify trees even further by not labelling internal nodes.

;; a leaf-labelled tree (LLT) is either
;; a Num or
;; a non-empty (listof LLT).

(list 2)    (list 5 7)    (list 2 (list 5 7) 3)    (list 2 (list 3 (list 5 7 11)))
a leaf-labelled tree (LLT) is either
   a Num or
   a non-empty (listof LLT).

There are two cases to consider.

1. If the LLT is a Num, it is a base case. Simply return the answer.

2. Otherwise, the LLT is a (listof LLT). Treat it like any other list! Solve the problem with higher order functions, or using recursion.
   Your function may run itself recursively, using map, and summarize the results using foldr.

;; my-LLT-fun: LLT -> Any
(define (my-LLT-fun L)
  (cond [(number? L) ;; this is a leaf.
         (... L ...)]
        [else ;; this is not a leaf, so it's a (listof LLT).
         (... (foldr ... (map my-LLT-fun L) ...))])))
Working on Leaf Labelled Trees

;; my-LLT-fun: LLT -> Any
(define (my-LLT-fun L)
  (cond [(number? L) ;; this is a leaf.
         (... L ...)]
        [else ;; this is not a leaf, so it's a (listof LLT).
         (... (foldr ... (map my-LLT-fun L) ...))])

Ex.
Following the template, write a function to count the leaves of a LLT.

Exercise
Following the template, complete depth.

;; (depth tree) return the max distance from the root to a leaf of tree.
;; depth: LLT -> Nat
;; Examples:
(check-expect (depth (list 6 7)) 1)
(check-expect (depth (list 2 (list 3 (list 5)))) 3)
Sometimes we want to extract the leaves from a leaf-labelled tree. For example:

```
; (flatten tree) return the list of leaves in tree.
;; flatten: LLT -> (listof Num)
;; Examples:
(check-expect (flatten (list 1 (list 2 3 4)) (list 1 2 3 4)))
(check-expect (flatten (list 1 (list 2 (list 3 4)))) (list 1 2 3 4))
```
Module summary

Be able to work with general trees and leaf-labelled trees.
Write programs that use recursion on trees, and either recursion or higher order functions on lists within the trees.
Create templates from data definitions. Use data definitions and templates to guide design of functions, both recursive and non-recursive.