Module 9: Trees

Mathematical Expressions

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.

Exercise

First create a constant: `(define mysymbol 'blue)`
Then see what each of these expressions evaluates to:

```
(equal? mysymbol 42)
(equal? mysymbol "blue")
(equal? mysymbol 'blue)
(equal? mysymbol 'red)
(symbol? '*@)
(symbol? "the artist formerly known as Prince")
```

Symbols

Everything possible with `Sym` could be done using `Str`.

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

Exercise

Complete count-sheep.

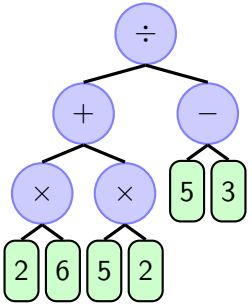
```
;; (count-sheep L) return the number of 'sheep in L.
;; count-sheep: (listof Any) -> Nat
;; Example:
(check-expect (count-sheep (list 6 'sheep 'ram 3.14 'sheep 'ox)) 2)
(check-expect (count-sheep (list 1 2 3)) 0)
(check-expect (count-sheep (list 'sheep 2 3 'sheep 'sheep 'sheep)) 4)
```

Checking if a value is a Sym and checking if a Sym is a particular Sym is *all* you can do with them.

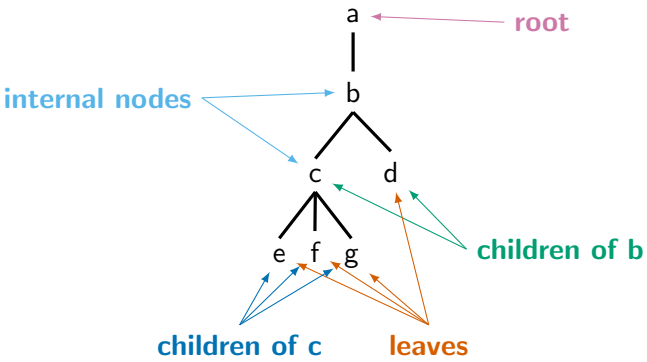
Operators such as $+$, $-$, $\times$, and $\div$ take two arguments, so we call them "binary operators". We have worked with these ourselves for years, but now we can get our computers to work with them for us. Consider:

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

We can split this expression into two expressions, and then recursively split them!



Tree Terminology



c is a **sibling** of d, and d is a sibling of c. e, f, and g are siblings of each other.

How can we represent this tree which stores a binary arithmetic expression?

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
```

Some examples:

`(make-binode '* 7 6)` $\longleftrightarrow$ 7×6

`(make-binode '* 7 (make-binode '+ 2 4))` $\longleftrightarrow$ $7 \times (2 + 4)$

Evaluating binary arithmetic expressions

```
(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.

Looking at the data definition, the **base case** is that the `BinExp` is a `Num`. The value of a `Num` is itself; this gives us the base case for our code!

Exercise

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

```
;; (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)
```

It may help to look at template on the next slide for inspiration.

Ex.

For completeness, extend `eval-binexp` so it also handles `'-` and `'/`.

A template for certain binary trees

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 (often from a data definition)
        [else (... (bintree-label T) ...
                   ... (my-bintree-fun (bintree-child1 T)) ...
                   ... (my-bintree-fun (bintree-child2 T)) ... )]))
```

Decoding Binary Trees

I will work with `BinTree`s where each `label` is a `Str`, and each leaf is an `Int`.

Exercise

Here are a few example trees.
Make sure you understand the relationship between each tree and the definition.

```
(define tree0 42)
```

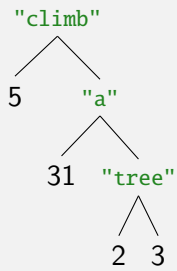
42

```
(define tree1
  (make-bintree "hello" 13 17))
```

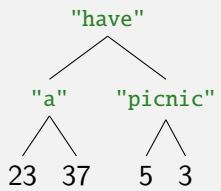

```
(define tree2
  (make-bintree "sunny"
    (make-bintree "day" 7 19)
    3))
```

Decoding Binary Trees

```
(define tree3
  (make-bintree "climb"
     5
     (make-bintree "a"
        31
        (make-bintree "tree"
           2
           3))))
```




```
(define tree4
  (make-bintree "have"
     (make-bintree "a"
        23
        37)
     (make-bintree "picnic"
        5
        3)))
```



Consider `child1` on the left, and `child2` on the right.

Exercise

Write a function `(leftmost-child T)` that consumes a `BinTree` and returns the leftmost leaf.

```
(check-expect (leftmost-child tree0) 42)
(check-expect (leftmost-child tree1) 13)
(check-expect (leftmost-child tree2) 7)
(check-expect (leftmost-child tree3) 5)
(check-expect (leftmost-child tree4) 23)
```

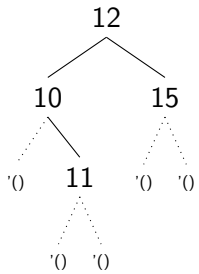
Exercise

Write a function `(join-labels T)` that consumes a `BinTree` and returns the `Str` that consists of the label followed by the labels of the left, then right, child.

```
(check-expect (join-labels tree0) "")
(check-expect (join-labels tree1) "hello")
(check-expect (join-labels tree2) "sunnyday")
(check-expect (join-labels tree3) "climbatree")
(check-expect (join-labels tree4) "haveapicnic")
```

Binary Search Trees

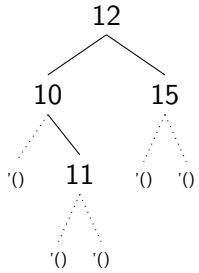
I can store information in a tree very efficiently.
Suppose we design as follows: a tree stores a value as its label. It stores all smaller values in a tree, which is its left child. It stores all larger values in a tree, which is its right child.




```
(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 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)
```

Hint

Given `tree12`, the left child should return 21, and the right child should return 15.
What do we need to stick them together?

Searching in a SSTree

Consider carefully this part of the data definition:

```
;; * a SNode, where keys in left are less than key, and in right greater.
```

If we are looking for a key, there are only four possibilities:

1. The tree is empty, so the key is not there.
2. The key of the tree we're looking at is the key we're seeking. So we found it.
3. The key is greater than the one we seek. The keys in the right side are even bigger. So we don't need to look there; only look in the left side.
4. The key is less than the one we seek. The keys in the left side are even smaller. So we don't need to look there; only look in the right side.

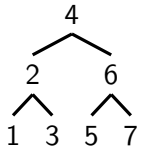
Exercise

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

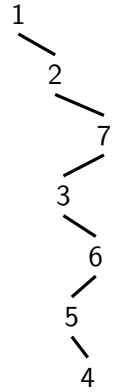
```
;; (tree-search tree item) return #true if item is in tree.
;; tree-search: SSTree Num -> Bool
;; Example:
(check-expect (tree-search tree12 10) #true)
(check-expect (tree-search tree12 7) #false)
```

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:



A completely unbalanced tree:



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.

Extending binary search trees to create better dictionaries

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:

```
(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!

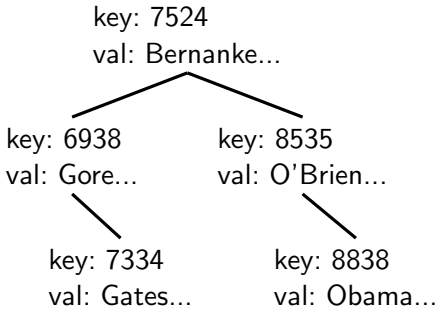
```
(define-struct node (key val left right))

;; 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
```

BST Dictionaries

```
(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")
   '()
   (make-node 7334
    (make-student "Bill Gates" "appliedmath")
    '()
    '()))
  (make-node 8535
   (make-student "Conan O'Brien" "history")
   '()
   (make-node 8838
    (make-student "Barack Obama" "law")
    '() '()))))
```



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)]))
```

Ex. Trace this code and make sure you understand it.

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?

```
(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
```

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.

```
;; 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.
Either the `(node-left tree)` or the `(node-right tree)` will be modified by a recursive call. (Two separate recursive cases!)

Exercise

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.

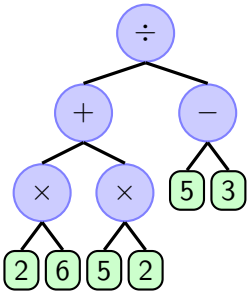
Exercise

Complete `expand-bst`.

```
;; (expand-bst L tree) add all items in L to tree, adding the last first.
;; expand-bst: (listof Association) 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" '() '())))
```

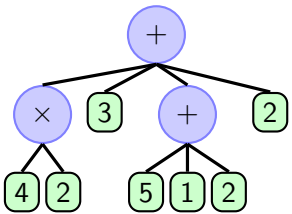
Recall with binary trees we could represent an expression containing binary operators (which have exactly 2 arguments), such as

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



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))
```

When we evaluated binary trees we used the following function:

```
;; (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:

```
;; (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)))]))
```

Think carefully about that this code does: it runs the function recursively on all the (2) children, then adds up the results.
Now instead of having 2 children, we have a list of children.

We need to run a function on each item in the list. We could do this recursively.
Even better: 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)))]))
```

This code makes all the recursive calls, and adds them up.

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)))]))
```

Exercise

Carefully read and test `eval-ainexp`.
Make sure you understand how it works.

General Trees

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,

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

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

General Trees Template

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

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

(define (gentree-function T)
  (cond [(not (gnode? T)) ; This is a leaf.
         (... T)]         ; 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 T) ... ; 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 T)))
          ...)]))
```

Recalling the following data definition:

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

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

Exercise

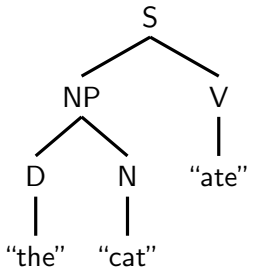
Complete `count-leaves`.

```
;; (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)
```

General Trees

```
;; 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"))
```

Hint

Use the `append` function.

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.

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

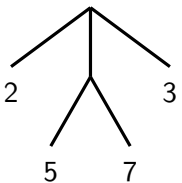
But then we would have only a `(make-node (listof Node))`.

So the structure adds nothing! Just store a `(listof Node)`. Some examples:

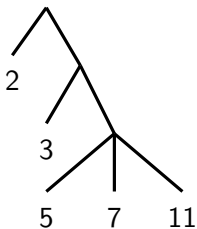
```
;; a leaf-labelled tree (LLT) is either
;;   a Num or
;;   a non-empty (listof LLT).
```

`(list 2) (list 5 7)`

`(list 2 (list 5 7) 3)`



`(list 2 (list 3 (list 5 7 11)))`



Template for Leaf Labelled Trees

```
;; 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) ...))]))
```

```
;; 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. Use the template to write `count-llt` that counts the leaves of a `LLT`.

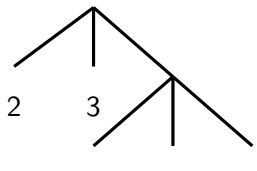
Exercise

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)
```

Flattening a list

Sometimes we want to extract the leaves from a leaf-labelled tree. For example:



$\rightarrow$ `(list 2 3 5 7 11)`

Exercise

Complete `flatten`. Hint: use the **`append`** function.

```
;; (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.

Further Reading: *How to Design Programs* section 19.