Assignment Guidelines.

- This assignment covers material in Module 9.
- Submission details:
  - Solutions to these questions must be placed in files `a09q1.rkt`, `a09q2.rkt`, `a09q3.rkt`, and `a09q4.rkt`, respectively, and must be completed using Racket *Intermediate Student with lambda*.
  - Unless otherwise indicated in the question you may use only the built-in functions and special forms introduced in the lecture slides from CS115 up to and including the modules covered by this assignment.
  - Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters.
  - All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  - Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  - For full style marks, your program must follow the CS115 Style Guide.
  - Be sure to review the Academic Integrity policy on the Assignments page.
  - For the design recipe, helper functions only require a purpose, a contract and an example.
- Restrictions:
  - Read each question carefully for additional restrictions.

> **!** You *may* use recursion or higher order functions (`map`, `filter`, `foldr`). You may use any combination of the tools we have learned so far.

- **The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.**

1. Fibonacci Foolishness. The Fibonacci sequence is a sequence of numbers that can be defined recursively as follows: $f_0 = 0$, $f_1 = 1$, and $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$.

It is easy to write a recursive function to compute elements of the Fibonacci sequence:

```
;; (fib n) Return the n-th term of the Fibonacci sequence.
;; fib: Nat -> Nat

(define (fib n)
  (cond [(= n 0) 0]
        [(= n 1) 1]
        [else
         (+ (fib (- n 1))
            (fib (- n 2)))]))
```

When you run this function, it calls itself, recursively, generating a tree of calls to `fib`, as shown in Figure 1. (Do a trace of `(fib 5)` to see how it works.)

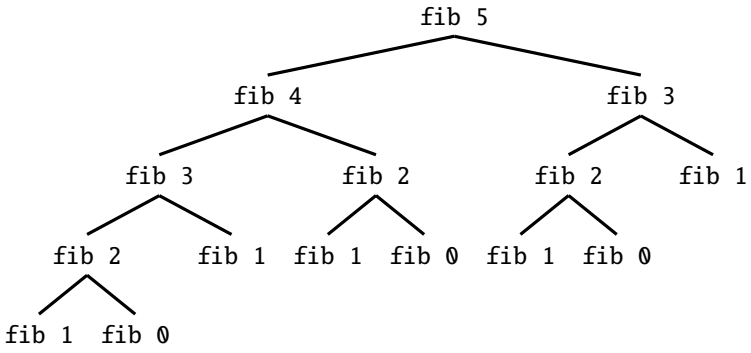


FIGURE 1. Tree of calls from calling `(fib 5)`.

In this exercise you are going to create a tree representing the calls to `fib`. Use the data definition:

```
(define-struct fn (label left right))
;; a FibNode is a (make-fn Str FibTree FibTree)

;; a FibTree is either:
;;   "fib 0", "fib 1", or
;;   a FibNode
```

Exercise

Write a function `(fib-tree n)` that consumes a `Nat` and returns the `FibTree` corresponding to `n`. For example,

```
(check-expect (fib-tree 1) "fib 1")
(check-expect (fib-tree 2)
              (make-fn "fib 2"
                       "fib 1"
                       "fib 0"))
(check-expect (fib-tree 5)
              (make-fn "fib 5"
                       (make-fn "fib 4"
                                (make-fn "fib 3"
                                         (make-fn "fib 2"
                                                  "fib 1"
                                                  "fib 0")
                                         "fib 1")
                                (make-fn "fib 2"
                                         "fib 1"
                                         "fib 0"))
                       (make-fn "fib 3"
                                (make-fn "fib 2"
                                         "fib 1"
                                         "fib 0")
                                "fib 1")))
```

Exercise

Now write a function to walk through the result from `fib-tree`, and compute the corresponding value of the Fibonacci sequence. For example,

```
(check-expect (fib-sum "fib 1") 1)
(check-expect (fib-sum (make-fn "fib 4"
                                (make-fn "fib 3"
                                         (make-fn "fib 2"
                                                  "fib 1"
                                                  "fib 0")
                                         "fib 1")
                                (make-fn "fib 2"
                                         "fib 1"
                                         "fib 0")))
              3)
```

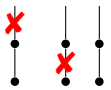
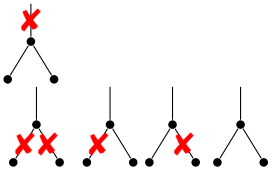
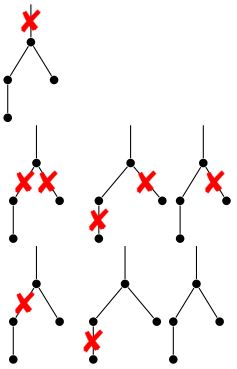
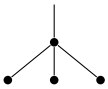
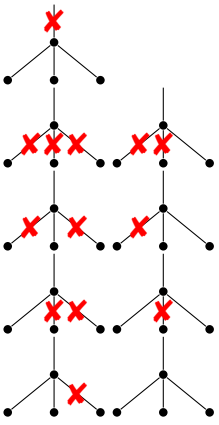
! Don't read the numeric value of the labels. You must traverse the tree and look at the leaves!

2. Counting.

The Rule of Product is a fundamental counting principle, that says that if there are n ways of doing one thing, and m ways of doing another, then there are $n \times m$ ways of doing both.
For example, if an ice cream shop has 3 kinds of cones, and 5 kinds of ice cream, then there are $3 \times 5 = 15$ combinations of cone and ice cream.
You will use the rule of product in this question.

There are certain connections between trees and Feynman Diagrams, which are studied in particle physics. The details we not go into here. But for certain calculations, we need to count how many different ways we can cut a tree, so it is cut at most once between each leaf and the root.

We will include cutting just "above" the root. Here are some examples of trees, and all possible cuts for each tree:

	Tree	Ways to Cut	Count	Explanation
#1			2	Either cut just above the root, or do not.
#2			3	If we cut just above the root, we cannot cut further (1). If we do not cut just above the root, then we have a #1 (2).
#3			5	If we cut just above the root, we cannot cut further (1). If we do not cut just above the root, then the left side is a #1, and the right side is another #1. By the Rule of Product these give $2 \times 2 = (4)$.
#4			7	If we cut just above the root, we cannot cut further (1). If we do not cut just above the root, then the left side is a #2, and the right side is a #1. By the Rule of Product, these give $3 \times 2 = (6)$.
#5			9	If we cut just above the root, we cannot cut further (1). If we do not cut just above the root, then we have three #1. By the Rule of Product, these give $2 \times 2 \times 2 = 8$.

These examples, and a few more, are included in the interface file.

We will use the following data definition:

```
(define-struct ultree (children))
;; A UlTree (unlabelled tree) is a (make-ultree (listof UlTree))
;; note: base case is when children is empty.
```

Exercise Write a function `(count-cuts T)` that consumes a `UlTree` and returns the number of different ways of cutting it so it is cut at most once between each leaf and the root.

3. The Trees They Are a-Changin'. Recall the definition of a leaf-labelled tree:

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

Exercise

Write a function `(llt-add n T)` that consumes a `Num` and a `LLT`.
It returns a `LLT` with the same shape as the one it is given, but where the value of each leaf has been increased by `n`.
For example,

```
;; Examples:
(check-expect (llt-add 1 (list 2 (list 3 (list 5))))
              (list 3 (list 4 (list 6))))
(check-expect (llt-add 2
                       (list 2 (list  5 7) 3))
              (list 4 (list 7 9) 5))
```

4. Converting a `BinExp` to a `Str`. We will use the following definitions:

```
;; an Operator is a Str of length 1.
;; more specifically, (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 Nat or
;;    a BINode
```

Note these are slightly modified from what is discussed in the notes: here an `Operator` is a `Str`, and we consider only expressions containing `Nat` (no decimals). This modification reduces complexity.

Exercise

Write a function `(expand-binexp e)` that consumes a `BinExp` and returns a `Str` that represents it.
Add brackets as follows:

(1) a `BinExp` which is a `Nat` does not have brackets.

```
(check-expect (expand-binexp 42) "42")
```

(2) every other `BinExp` has brackets around it, even if they are not strictly necessary.

```
(check-expect (expand-binexp (make-binode "*" 12 15)) "(12*15)")
(check-expect (expand-binexp (make-binode "+" 1 (make-binode "+" 2 3)))
              "(1+(2+3))")
(check-expect (expand-binexp (make-binode "+" (make-binode "+" 1 2) 3))
              "((1+2)+3)")
```