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. **Constructing a Balanced Tree.** Recall the data definition:

```
(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.
```

Write a function `(build-balanced-tree L)` that consumes a `(listof Num)` that is sorted in increasing order. It returns a SSTree where every value in L exists as a key, and which is balanced. A tree is balanced if the number of nodes in the left side is the same, or at most 1 more, than the number of nodes in the right side.

For example, given `(L 1 2 3 4 5 6 7)`, the root of the tree will be 4, left child will be a SSTree containing `(list 1 2 3)`, and the right child will be a SSTree containing `(list 5 6 7)`.

```
(look-expect (build-balanced-tree (list 1 2 3 4 5 6 7))
(make-snode 4
 (make-snode 2 (make-snode 1 '() '()) (make-snode 3 '() '()))
 (make-snode 6 (make-snode 5 '() '()) (make-snode 7 '() '())))
```

```
(look-expect (build-balanced-tree (list 7 11 13 15))
(make-snode 13
 (make-snode 11 (make-snode 7 '() '()) '())
 (make-snode 15 '() '()))
```

**Hint:** Build a set of recursive helper functions: one that returns the first n values of a list, one that returns the n-th item of a list, and one that returns all the items after the first n.

2. **Tree Map.** Recall the definition of a leaf-labelled tree:

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

Write a function `(llt-map fn T)` that consumes a `Function` and a LLT. It returns a LLT with the same shape as the one it is given, but where each leaf has been modified by fn.

For example,

```
(look-expect (llt-map add1 (list 2 (list 3 (list 5))))
 (list 3 (list 4 (list 6))))
```

```
(look-expect (llt-map (lambda (x) (* x 2)) (list 2 (list 5 7) 3))
 (list 4 (list 10 14) 6))
```

2
3. **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 (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.

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)")
4. **Trie.** A *trie* (pronounced “try”) is a tree data structure that can efficiently store a large number of words. A project at the University of Waterloo used this data structure in putting the Oxford English Dictionary on CD-ROM. (This was a big deal in 1993.)

(In a classic trie, the nodes are empty, and the *edges* are labelled. Since our model of trees labels the nodes, we will continue in this manner, with a somewhat odd description of a trie.)

Each node of the trie stores a prefix, and its children store all the possible continuations of that word. Each path from the root to a leaf creates a word.

For example, consider the following trie. By following the leftmost path, we encounter "z", "a", "ny", so this leaf is the word "zany". Other paths include "z" "e" "ro" => "zero", "z" "i" "p" "m" => "zip" and "z" "i" "p" "s" => "zips".

![Trie Diagram]

We will define a *Trie* as a kind of *GenTree*, as follows:

```scheme
(define-struct trien (label children))
;; a trie-node (TrieN) is a (make-trien Str (listof Trie))

;; a Trie is either:
;;   a TrieN, or
;;   a Str.
```

Write a function `list-words T` that consumes a *Trie* and returns a *(listof Str)* containing all the words in `T`, starting at the leftmost leaf.

For example, `z-trie`, defined in the interface file, describes the trie shown above:

```scheme
(list-words z-trie) =>
(list "zany" "zap"
  "zeal" "zero" " zest"
  "zinc" "zing" " zip" "zips"
  "zone" "zoo" " zoom" "zoos")
```

```scheme
(list-words (make-trien "m" (list "at" "e" "y"))) => (list "mat" "me" "my")
```

**Exercise**

Don’t forget to write your contract!

**Hint**

You may use `append` on this exercise. (That’s also a hint.)