Assignment Guidelines.

- This assignment covers material in Module 9.

Submission details:
- Solutions to these questions must be placed in files `a9q1.rkt`, `a9q2.rkt`, `a9q3.rkt`, and `a9q4.rkt`, respectively, and must be completed using Racket.
- 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. A list of functions described in each module of the lecture slides can be found at [https://www.student.cs.uwaterloo.ca/~cs115/built_in](https://www.student.cs.uwaterloo.ca/~cs115/built_in)
- 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. **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.

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)")`
2. Dictionary Search. This question works with the following data definitions:

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

```
;; A binary word search tree (BWT) is either
;; * () or
;; * (make-node Str Nat BWT BWT)...
;; which satisfies the ordering property recursively:
;; * every key in left is less than key
;; * every key in right is greater than key
```

The following sample data is used for examples:

```
(define dict1
  (make-node "blue" 1
    ()
  (make-node "gold" 99
    (make-node "brown" 20
      (make-node "bright cyan" 59
        ()
        ()
      )
    )
  )
(())
)
```

Exercise

Modify the `dict-search` function from slide 20 of module 9 of the notes, so it works using a `Str` as the key in the dictionary, and a `Nat` as the value.

```
(check-expect (bwt-lookup dict1 "dark gray") 56)
(check-expect (bwt-lookup dict1 "brown") 20)
(check-expect (bwt-lookup dict1 "bright cyan") 59)
(check-expect (bwt-lookup dict1 "gold") 99)
```

Ensure your code only searches one side of the tree or the other, not both.
3. **Tree Structure.** Sometimes we care only about the structure of the tree, not the contents. In this case, we can use any kind of tree, and just ignore the labels on the vertices. But a LLT is nice because then we only need to ignore the leaves.

(a) **Tree Size.** The size of a tree is the number of nodes in it.

Exercise

Write a function \( (\text{tree-size } T) \) that consumes a LLT and returns the number of nodes in the tree.

(check-expect (tree-size 17) 1)
(check-expect (tree-size (list 2 3 5)) 4)
(check-expect (tree-size (list 12 13 (list 15 16))) 6)

(b) **Tree Factorial.** In *The Art of Computer Programming, volume 3* (1972), section 5.1.4, exercise 20, Donald Knuth does a computation involving “the product of the subtree sizes”. This computation is sometimes referred to as the tree factorial.

The **tree factorial** of a tree \( T \) may be defined as the number of nodes in \( T \), multiplied by the product of the tree factorials of all its children.

Exercise

Write a function \( (\text{tree-fact } T) \) that consumes a LLT and returns its tree factorial. For example:

(tree-fact 17) => 1 ; the tree has no children, so just the size
(tree-fact (list 2 3 5)) => 4 ; tree has size 4, each child has 1

(tree-fact (list (list 2 3 5))) => 20
;; size of the root is 5
;; size of the trunk is 4
;; size of each leaf is 1
;; (* 5 4 1 1 1) => 20

(tree-fact (list (list 2 3 (list 5 7)) (list 1 (list 2 2)))) => 3240

The factorial of a natural number \( n \) is the product of the numbers from 1 to \( n \). This may be viewed as a special case of the tree factorial, where the tree is linear. For example,

(tree-factorial (list (list (list 17)))) => 24

Since this tree has 4 nodes (one from each list, plus the leaf), and \( 4! = 24 \).