• Unless otherwise indicated by the question, you may only use 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 on LEARN.
• For this and all subsequent assignments, you are expected to use the design recipe when writing functions from scratch, including helper functions.
• 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.
• Do not send any code files by email to your instructors or tutors. Course staff will not accept it as an assignment submission. Course staff will not debug code emailed to them.
• Read each question carefully for restrictions.
• Test data for all questions will always meet the stated assumptions for consumed values.
• Download the interface file from the course Web page to ensure that all function names are spelled correctly, and that each function has the correct number and order of parameters.
• You may use the purposes, contracts and examples from the provided interface file in your solutions.
• Check MarkUs and your basic test results to ensure that your files were properly submitted. In most cases, solutions that do not pass the basic tests will not receive any correctness marks.
• Any string constant values must exactly match the descriptions in the questions. Any discrepancies in your solutions may lead to a severe loss of correctness marks. Basic tests results will catch many, but not necessarily all of these types of errors.
• Read the course Web page for more information on assignment policies and how to organize and submit your work. Follow the instructions in the Style Guide. Your solutions should be placed in files a9qY.rkt, where Y is a value from 1 to 3.
• Since each file you submit will contain more than one function, it is very important that your code runs. If your code does not run then none of the functions can be tested for correctness.
• Plagiarism: The following applies to all assignments in CS115.
  – Be sure to read the Plagiarism section at: https://www.student.cs.uwaterloo.ca/~cs115/#assignments

Language level: Intermediate Student with Lambda
Coverage: Module 9

This assignment uses the following structure definitions.

(define-struct binode (op arg1 arg2))
;; A Binary arithmetic expression Internal Node (BINode)
;; is a (make-binode (anyof '* '+ '/ '-) BinExp BinExp)

;; A Binary arithmetic expression (BinExp) is one of:
;; * a Num
1. Write a function `binexp->string` that consumes a `BinExp` (`bexp`) and produces a string representing `bexp` in standard mathematical format, along with the result of evaluating `bexp`. The produced string is formatted as follows:
   - Each `BinExp` structure which is not an integer is enclosed in parentheses.
   - The result of the expression is at the end of the string with an equals sign, "=". in between.
   - There are only two white space characters in the produced string; one before and one after the "=" character.

You may assume there is no division by zero in the consumed `BinExp`. We will not test this.

For example:
   - `(binexp->string 5.2) => "5.2 = 5.2"
   - `(binexp->string (make-binode '+ 13 4)) => "(13+4) = 17"
   - Considering the following constant,

```
(define my-bexp1 (make-binode '+
                          (make-binode '* (make-binode '+ 4 1)
                                          (make-binode '+ 5 2))
                          (make-binode '- 6 3)))
```

we have

```
(binexp->string my-bexp1) => "(((4+1)*(5+2))+(6-3)) = 38"
```

2. Write a function `neighbours` that consumes a `BST` (`abst`) and a natural number (`n`). If `n` is one of the keys in `abst`, it may have a `smaller neighbour` and a `larger neighbour`. A `smaller neighbour` is a key in `abst` that is less than `n`, but closer to `n` than all other keys that are less than `n`. A `larger neighbour` is a key in `abst` that is greater than `n`, but closer to `n` than all other keys that are greater than `n`. 

Copyright ©2019
The `neighbours` function produces a list of the two keys that represent the smaller neighbour and the larger neighbour of \( n \), if they exist in `abst`. If the key \( n \) does not exist in `abst` or if one or both of the neighbours of \( n \) do(es) not exist in `abst`, then `neighbours` produces `false`.

For full marks your solution should take advantage of the ordering property of a BST, and only search the necessary parts of `abst`. Your solution must **not** covert `abst` into a list.

For example, let `my-bst-for-neighbours` be the binary search tree below, with only its keys displayed.

```
    8
   / \
  4   12
 /   / \
2   6   10
 /   /   / \
1   5   7   14
```

Then
- \( \text{neighbours my-bst-for-neighbours 8} \) => (list 7 9), and
- \( \text{neighbours my-bst-for-neighbours 14} \) => `false`.

3. Create a function `bst-desc-traverse->list` that consumes a BST (`abst`), and produces a list of all the strings contained in the value fields of the binary search tree nodes. The elements in the produced list must be in descending order of the values from their associated key fields from the binary search tree nodes.

Considering the following constant,

```
(define my-bst-for-traverse
  (make-node 5 "Tony"
    (make-node 1 "Qiang" empty empty)
    (make-node 6 "Judy" empty
      (make-node 14 "Wole" empty empty))))
```

we have

```
(bst-desc-traverse->list my-bst-for-traverse) =>
(list "Wole" "Judy" "Tony" "Qiang").
```