Assignment 08
Due: Wednesday, March 21st, 2018 at 10 a.m.

- Academic Integrity Links
  - https://uwaterloo.ca/academic-integrity/basic-info
  - https://uwaterloo.ca/library/get-assignment-and-research-help/academic-integrity/academic-integrity-tutorial

- For full marks, it is not sufficient to have a correct program. Be sure to follow all the steps of the design recipe. Read the Style Guide carefully to ensure that you are following the proper conventions.

- Your solution must include the definition of constants and helper functions where appropriate.

- If you write a solution that includes a wrapper function, you are only required to include examples and tests for the wrapper function, and not for the function that it wraps.

- You must provide the data definition and template in your solutions only when the question specifically indicates they are required for compound data types described in the question. If you create any additional data types that are beyond the question description, your program file should include a data definition and a template for each additional data type.

- If you include a template in your solution, the template should appear as comments.

- You may want to include defined constants to help reduce the writing for the examples and tests.

- 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 may be found at https://www.student.cs.uwaterloo.ca/~cs115/built_in.

- Use the design recipe when writing functions (and helper functions) from scratch.

- 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.

- Read each question carefully for restrictions.

- Test data for all questions will always meet the stated assumptions for consumed values.

- Do not copy the purpose directly from the assignment description. The purpose should be written in your own words and include references to the parameter names of your 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.

- 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 or symbol values must exactly match the descriptions in the questions. Any discrepancies in your solutions may lead to a severe loss of correctness marks.

- 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 a08qY.rkt, where Y is a value from 1 to 4.

Language level: Beginning Student with List Abbreviations
Coverage: Module 8
The following definitions are required for the assignment.

;; An association (As) is (list Num Str),
;; where
;; * the first item is the key,
;; * the second item is the associated the value.

;; An association list (AL) is one of
;; * empty
;; * (cons As AL)
;; Note: All keys must be distinct.

(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
;; * a BINode

(define-struct node (key val left right))
;; A Node is a (make-node Nat Str BT BT)

;; A binary tree (BT) is one of
;; * empty
;; * (make-node Nat Str BT BT)

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

(define leaf0 (make-node 0 "zero" empty empty))
(define leaf2 (make-node 2 "two" empty empty))
(define leaf4 (make-node 4 "four" empty empty))
(define leaf6 (make-node 6 "six" empty empty))
(define bst1 (make-node 1 "one" leaf0 leaf2))
(define bst5 (make-node 5 "five" leaf4 leaf6))
(define bst3 (make-node 3 "three" bst1 bst5))
(define bst4 (make-node 5 "Five"
    (make-node 4 "Four"
        (make-node 1 "One" (make-node 0 "Zero" empty empty) empty)
        (make-node 2 "Two" empty empty))
    empty)
    (make-node 9 "Nine" empty empty)))

(define non-bst1 (make-node 1 "one" leaf6 empty))
1. Write a Racket function `binexp->string` that consumes a `BinExp` and produces a string representing the `BinExp` in standard mathematical format, along with the result of evaluating the expression. 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 equal sign " = " in between.
- There is only two white space characters in the produced string; one before and one after the " = " character.
- The function may represent a decimal outcome in fraction form.

You may assume there is no division by zero in the consumed `BinExp` (we won’t test that).

For example:
- `(binexp->string 5.2) => "26/5 = 26/5"
- `(binexp->string (make-binode '+ 13 4)) => "(13+4) = 17"

Considering the following constant,

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

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

2. Write a Racket function `equal-children?` that consumes a binary tree, and produces true if the sum of all keys in the left and right subtrees are the same, and false otherwise. The function produces true if the binary tree is empty.

For example:
- `(equal-children? (make-node 1 "One" empty (make-node 2 "Two" empty empty) (make-node 2 "Two" empty empty))) => true
- `(equal-children? (make-node 1 "One" empty (make-node 3 "Three" empty empty))) => false
- `(equal-children? empty) => true

3. Write a Racket function `bst?` that consumes a binary tree `bt` and produces true if the binary tree is a BST, and false otherwise.

For example:
- `(bst? empty) => true
- `(bst? bst3) => true
- `(bst? non-bst1) => false
4. Write a Racket function `bst->al` that consumes a BST and produces an association list with the same set of key-value pairs from the BST. The produced list should be stored in a decreasing order of the numerical keys.

For example:
- `(bst->al (make-node 10 "ten" empty empty)) => (list (list 10 "ten"))`
- `(bst->al bst4) =>
  (list (list 9 "Nine") (list 5 "Five") (list 4 "Four")
  (list 2 "Two") (list 1 "One") (list 0 "Zero"))`

The association list can be created in sorted order directly from the BST. Marks will be deducted if you use any sorting function (built-in or written yourself) to sort the association list.

Do not sort, do not use the built-in reverse function, do not use your own implementation of the reverse function, do not flatten the BST, and do not convert the BST into a list in an intermediate step.