Assignment: 7

Due: Monday, July 13 at 4:00 pm
Language level: Beginning Student with List Abbreviations
Coverage: Modules 1–8

For this and all subsequent assignments, to receive full marks you are required to use the design recipe for every function you write. You may include the examples given in the assignment in your submissions, but they will be ignored by the markers—you must develop a complete suite of examples and tests on your own. For your convenience, an interface file which contains the headers of the required functions is available on the course webpage.

Do not send any code files to course staff; they will not be accepted. Submissions must be made via MarkUs as described on the course webpage. After submission, check your basic test results to ensure your files were properly submitted. Solutions that do not pass the basic tests are unlikely to receive any correctness marks.

Remember, the solutions you submit must be **entirely your own work**.

1. Write a Racket function *binexp→string* which consumes a *BinExp* as described in class and produces a string representation of the binary expression. Parenthesis should be included around every operation and all operations should be written using *infix* notation, e.g.,

   - The string representation of (*make-binode* '* 2 6) is `"(2*6)"`.
   - The string representation of (*make-binode* '+ 2 (*make-binode* '- 5 3)) is `"(2+(5-3))"`.

   When just a number is consumed, *binexp→string* should produce the same output as *number→string*. For example, (*binexp→string* 1) should produce `"1"`. The function *symbol→string* can be used to convert the operation symbols to their string equivalents.

2. Write a Racket function *max-depth* which consumes a *BinExp* as described in class and produces the maximum "depth" of any number in the binary expression, where the depth of a number is the number of parenthesis which surround it in the string representation.

   For example, in `"(2+(5-3))"` the number `"2"` has a depth of 1 and the numbers `"5"` and `"3"` have a depth of 2. Therefore, (*max-depth* (*make-binode* '+ 2 (*make-binode* '- 5 3))) should produce 2.

   Hint: Your code should work directly on binary expressions, i.e., don't convert the binary expression to a string.

3. (a) Write a Racket function *bst?* which consumes a binary tree (a *BT* as described in class) and produces *true* if the tree is a binary search tree (a *BST* as described in class) and *false* otherwise.

For example,

```
(bst? (make-node 5 "Tony"
                   (make-node 1 "Qiang" empty empty)
                   (make-node 6 "Judy"
                                empty
                                (make-node 14 "Wole" empty empty))))
```

should produce *true*, and

```
(bst? (make-node 5 "Tony"
                   (make-node 1 "Qiang" empty empty)
                   (make-node 6 "Judy"
                                (make-node 4 "Wole" empty empty)
                                empty)))
```

should produce *false*.

Hint: Consider using helper functions which determine the maximum and minimum keys in a *BT* or *BST*.

(b) Write a Racket function *add-bst* which consumes an association (a (*list Num Str*) as described in class) and a *BST* and produces a *BST* which is the same as the consumed *BST* except that it includes one additional node which has a key value pair corresponding to the given association. You may assume that the key in the association being added does not appear in the given *BST*.

For example,

```
(add-bst (list 14 "Wole")
         (make-node 5 "Tony"
                      (make-node 1 "Qiang" empty empty)
                      (make-node 6 "Judy" empty empty)))
```

should produce

```
(make-node 5 "Tony"
             (make-node 1 "Qiang" empty empty)
             (make-node 6 "Judy"
                          empty
                          (make-node 14 "Wole" empty empty))) .
```