Assignment Guidelines:

• For this and all subsequent assignments, you are expected to use the design recipe when writing functions from scratch, including helper functions.

• **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. **In addition, your solution must include the definition of constants and helper functions where appropriate.**

• 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.

• 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.

• You may post general assignment questions using the discussion groups on Waterloo LEARN. Choose Connect Discussions. Read the guidelines for posting questions. Do NOT post any code as part of your questions.

• 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.

• 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 4. Submit a file `a9q5.rkt` if you wish to attempt a solution to the bonus question.

**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](https://www.student.cs.uwaterloo.ca/~cs115/assignments)
Assignment 9
Due: December 3rd, 2018
Language Level: Intermediate Student with Lambda
Coverage: Modules 09, 10

Question 1

In this question you will rewrite the solutions to questions from previous assignments and labs, taking advantage of abstract list functions and lambda. For each question, your solution should contain exactly one use of define. Your solution must not contain any explicit recursion, and you should not write any helper functions; you should rely on built-in abstract list functions, where the function argument is a built-in function or a lambda expression. You can use other built-in functions and special forms as needed.

You are welcome to rewrite these functions from scratch, to borrow from your solution to the original problem, or to borrow from the model solution for the past assignment. But you will not receive marks unless your solution is expressed as required above. Because you have already written these functions in the past, you are not required to include design recipes. Of course you are still permitted to include them (and it’s always a good idea to test your solutions!).

(a) Rewrite make-sqr-al (Lab 08, Question 6) using only abstract list functions and lambda.

(b) Rewrite filter-prefix (Assignment 5, Question 3) using only abstract list functions and lambda. Unlike in Assignment 5, here you are permitted to use built-in string functions like string-ci=?, string-upcase, and string-downcase.

(c) Rewrite catalan (Assignment 6, Question 1) using only abstract list functions and lambda.
Question 2

Write a function `remove-leaf` that consumes a binary search tree (BST) and a key (using the BST data definition given in the lecture notes). It produces a new BST in which the node with the given key is removed, if that node is a leaf node of the tree. If the given key is not found in the tree, the function should produce the symbol `not-found`. If the given key is found but the node is not a leaf node, the function should produce `not-leaf`. For example:

```
(remove-leaf empty 5)  \Rightarrow  'not-found

(remove-leaf (make-node 4 "" empty (make-node 5 "" empty empty)) 5)  \Rightarrow  (make-node 4 "" empty empty)

(remove-leaf (make-node 5 "" empty (make-node 8 "" empty empty)) 5)  \Rightarrow  'not-leaf
```

Your solution must make use of the BST property: any recursion performed on the tree should travel down one sub-tree or the other, but never both.

It's possible to solve this problem in a single recursive function. But some people may find it more straightforward to write a non-recursive wrapper function that first checks via a helper function if the key is present and/or is a leaf node, and a second helper function that performs the actual removal if both of those conditions are true.
Question 3

In the Module 09 lecture notes we developed a data definition for BinExp, a binary arithmetic expression. We then developed a function eval that consumes a binary arithmetic expression and produces its simplified numerical value, evaluating all binary operations along the way.

For convenience, let's introduce a few very simple functions to make it easier to construct binary arithmetic expressions:

```racket
(define (e+ a b) (make-binode '+ a b))
(define (e- a b) (make-binode '- a b))
(define (e* a b) (make-binode '* a b))
(define (e/ a b) (make-binode '/ a b))
```

With these, binary arithmetic expressions can be made to look more like ordinary Racket arithmetic, by adding e before every operator. For example, instead of writing

```racket
(make-binode '+ (make-binode '* 3 4) (make-binode '- 6 3))
```

we can just write

```racket
(e+ (e* 3 4) (e- 6 3))
```

(a) Write a function binexp->string that consumes a binary arithmetic expression and produces a string containing the expression as it might appear when formatted in Racket code. Include all necessary parentheses in the result, and add single spaces to separate function names and arguments from each other (but no additional spaces beyond that). For example:

```racket
(binexp->string 5) ⇒ "5"
(binexp->string (e- (e* 5 3) 8)) ⇒ "(- (* 5 3) 8)"
```

You may find the built-in functions number->string and symbol->string to be useful. Don't worry about the precise string representations of fractions or inexact numbers—we will test your solution using only integers. You are welcome to use the provided functions e+, e-, e*, and e/ in your tests (but your produced strings should not contain those function names!).
(b) Write a function `step` that consumes a binary arithmetic expression. Unlike `eval` (discussed in lectures), `step` should produce a new binary arithmetic expression in which exactly one substitution step has been carried out, using the stepper rules for built-in functions as defined at the beginning of the term. If the expression is already fully simplified, `step` should leave it as-is. For example:

```
(step 6) ⇒ 6
(step (e* 4 5)) ⇒ 20
(step (e* (e+ 2 3) (e- 6 4))) ⇒ (make-binode '* 5 (make-binode '-' 6 4))
(step (e- 15 (e* 3 4))) ⇒ (make-binode '-' 15 12)
```

You can assume that the simplification step is legal (i.e., that it doesn't cause an error like a division by zero).

You can write more compact tests by taking further advantage of the four helper functions for constructing binary arithmetic expressions. For example:

```
(check-expect (step (e* (e+ 2 3) (e- 6 4)))
             (e* 5 (e- 6 4)))
```

Hint: the `step` function is similar to the `eval` function from the lecture notes.
Question 4

A mobile is a hanging sculpture, typically made from horizontal rods, vertical wires, and a set of weights. The configuration of weights and rods is chosen so that every part of the mobile is balanced: where a rod is suspended from a wire at a connection point, the downward forces exerted by the weights on either side of that connection point cancel each other out. A simple formula tells us how much downward force a given weight exerts on a rod: just multiply its mass by the horizontal distance to the rod’s connection point. For example, the following mobiles are all balanced:

![Balanced Mobile Diagram]

On the left, a single weight is always balanced. In the middle, we have equal weights (both 2) at equal distances (both 1) from the connection point. On the right, the weight on the left exerts a force of $2 \cdot 4 = 8$, and the weights on the right exert a combined force of $5 \cdot 1 + 1 \cdot 3 = 8$, so everything is balanced.

Of course, a weight can be replaced by a “recursive mobile” as long as that mobile is itself balanced, and its total weight (ignoring rods and wires) adds up to the weight it’s replacing. For example, this mobile (a variation on the right-hand one above) is also balanced:

![Recursive Mobile Diagram]

We introduce mutually-recursive data definitions to describe an arbitrarily nested mobile structure:

```scheme
;; A Mobile is an (anyof Num Rod)
;; A Rod is a (listof (list Num Mobile))
```

A Mobile is either a single weight hanging from a wire, or it’s a horizontal rod hanging from a wire, with recursive mobiles suspended from that rod. A Rod is a list of two-element lists, where each sub-list gives a distance to the rod’s connection point and a recursive mobile hanging at that distance. Here, a negative distance indicates a mobile to the left of the connection point, and a positive distance is to the right of the connection point. A distance of zero would correspond to a recursive mobile directly underneath the connection point. The distances are not required to be sorted.
For example, the four balanced mobiles shown on the previous page could be described via the following Racket expressions:

\[
3
\]

\[
\text{list (list 1 2) (list -1 2))}
\]

\[
\text{list (list 1 5) (list 3 1) (list -4 2))}
\]

\[
\text{list (list -4 (list (list -1 1) (list 1 1))) (list 1 (list (list -2 3) (list 3 2))) (list 3 1))}
\]

Remember to begin by writing a template for a function that consumes a Mobile, and a template for consuming a Rod. These will help you stay on track when solving the problems below.

a) Write a function total-mass that consumes a Mobile and produces a number giving the sum of all the weights in that mobile structure. For example, the total masses of the four mobiles above would be 3, 4, 8, and 8. Remember that in this question you’re just adding up the weights, and ignoring the distances from those weights to connection points.

b) Write a function is-balanced? that consumes a Mobile and produces true if the mobile is in equilibrium, as described above, and false otherwise. That is, every recursive mobile must independently be in equilibrium, and a set of mobiles hanging from a rod must exert balanced downward forces on either side of the rod’s connection point. The four mobiles described above are all balanced; on the other hand, the two shown below are not balanced.

Note the one on the example in particular: although the weights acting on the left and right sides of the topmost rod are both 3, the recursive sub-mobile given by the expression (list (list -1 2) (list 1 1)) is not.
Bonus Question

Suppose we knew the shape of the mobile we wanted to construct (i.e., the tree structure and all the distances of recursive mobiles along their rods), but not the actual weights to use. We could use the following data definition to describe this information:

```scheme
;; A MobileShape is an (anyof 'X Rod)
;; A RodShape is a (listof (list Num MobileShape))
```

In this data definition, all the 'X symbols are locations where we’d like to fill in a weight.

Write a function `solve-balance-problem` that consumes a MobileShape containing $n$ copies of the symbol 'X, and produces a balanced Mobile by replacing every 'X with a natural number between 1 and $n$, where each of these natural numbers is used exactly once. If no solution can be found, produce the value `false`. For example,

```scheme
(solve-balance-problem
 (list (list -3 'X)
       (list -1 'X)
       (list 2 (list (list -2 'X)
                     (list -1 'X)
                     (list 1 'X))))
)
```

Would produce the following balanced mobile:

```scheme
(list (list -3 3)
      (list -1 5)
      (list 2 (list (list -2 1)
                     (list -1 2)
                     (list 1 4)))
)
```

It happens that this puzzle has a unique solution if we require that each of the weights 1, 2, 3, 4, and 5 is used exactly once. Most “balance puzzles” are designed this way, and so we impose that restriction here.

You can find a collection of weight balancing puzzles on Erich Friedman’s website. These puzzles can take a long time to solve computationally. In practice, you should be able to solve a puzzle up to $n = 9$ in under a minute. We won’t test you on larger puzzles.