CSCE 314 Programming Languages – Fall 2015
Hyunyoung Lee
Assignment 3
Assigned on Wednesday, September 16, 2015

Electronic submission to eCampus due at 9:00 a.m., Monday, Sep. 28, 2015
Honor code signed coversheet due at the beginning of class on Monday, Sep. 28, 2015
If you do not turn in a Honor code signed coversheet your work will not be graded.

"On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment."

Carsten Hood	502
Typed or printed name of student	Section (501 or 502)
	922009787
Signature of student	UIN

In this assignment, you will practice functional programming using higher-order functions in Haskell. Below, you will find problem descriptions with specific requirements. Read the descriptions and requirements carefully! There may be significant penalties for not fulfilling the requirements. You will earn total 140 points.

Note 1: This homework set is *individual* homework, not a team-based effort. Discussion of the concept is encouraged, but actual write-up of the solutions must be done individually.

Note 2: Submit electronically exactly one file, namely, *yourLastName-yourFirstName-*a3.hs, and nothing else, on eCampus.tamu.edu.

Note 3: Make sure that the Haskell script (the .hs file) you submit compiles without any error. If your program does not compile, you will likely receive zero points for this assignment.

Note 4: Remember to put the head comment in your file, including your name and *acknowledgements of any help received* in doing this assignment. Again, remember the honor code.

Keep the name and type of each function exactly as given.

Exercise 1: List comprehensions (10 points)
Use list comprehensions in the following two exercises.

(a) A *Pythagorean triad* is a triple (x, y, z) of positive integers such that $x^2 + y^2 = z^2$. Define a function `triads :: Int -> [(Int, Int, Int)]` that for a given

n returns a list of all triads whose elements are at most n. If some (a, b, c) is a Pythagorean triad, so is (b, a, c). The result of `triads` function should only include one of each such pair of triads, the one with $x \leq y$.

(b) A positive integer n is *perfect* if it equals the sum of all of its factors, excluding n itself. For example, 6 is a perfect number because factors of $6 = \{1, 2, 3, 6\}$, and $6 = 1 + 2 + 3$. Define a function `perfect :: [Int]` that returns the infinite list of all *perfect* numbers.

Exercise 2: Recursive functions, higher order functions (20 points)

(a) Implement a *merge sort* that is parametrized with the comparison function. Its name and type should be:

```
mergeSortBy :: (a -> a -> Bool) -> [a] -> [a]
```

[Hint: Use helper functions `mergeBy` and `split`. Function `mergeBy` merges two sorted lists so that the resulting list is also sorted. Function `split` splits a list into two lists whose lengths differ by at most one.]

(b) Implement a merge sort function that sorts a list in an *ascending* order. The name and type of your function should be:

```
mergeSort :: Ord a => [a] -> [a]
```

Exercise 3: Higher order functions (50 points)
Implement the following functions with the help of some of `map`, `filter`, `foldr`, `foldl`, `foldr1`, `foldl1`, and the function composition operator `.` (and use other functions too if you find them necessary). You need to `import Data.Char` if you wish to use the `toUpper` function (see the skeleton code).

(a) (5 points) A function that multiplies all elements of a list. Multiplying the empty list should return 1. (Hint: use `foldr`.)

```
multiply :: [Int] -> Int
```

(b) (5 points) String concatenation function. (Hint: use `foldl`.)

```
concatenate :: [String] -> String
```

(c) (10 points) A function that examines a list of strings, keeps only those whose length is odd, converts them to upper case letters, and concatenates the results to produce a single string.

```
concatenateAndUpcaseOddLengthStrings :: [String] -> String
```

(d) (10 points) Insertion sort.

```
insertionSort :: Ord a => [a] -> [a]
```

Hint: To express `insertionSort` as a fold, you may need a helper function that inserts an element into a sorted list and returns a sorted list. A suitable type for such a function would be `Ord a => a -> [a] -> [a]`.

(e) (10 points) A function that finds a maximal element in a list. Use one of the fold functions.

```
maxElementOfAList :: Ord a => [a] -> a
```

(f) (10 points) A function that filters a list, keeping only the elements that fall within a specified closed interval. For example, `keepInBetween a b` keeps the values belonging to $[a, b]$.

```
keepInBetween :: Int -> Int -> [Int] -> [Int]
```

Exercise 4: Data types, type classes (30 points)
Consider the following data type.

```
data Tree a b = Branch b (Tree a b) (Tree a b)
              | Leaf a
```

(a) (15 points) Make `Tree` an instance of `Show`. Do not use `deriving`; define the instance yourself. Make the output look somewhat nice (e.g., indent nested branches).

(b) (15 points) Implement these functions that traverse the tree in the given order collecting the values from the tree nodes into a list:

```
preorder  :: (a -> c) -> (b -> c) -> Tree a b -> [c]
postorder :: (a -> c) -> (b -> c) -> Tree a b -> [c]
inorder   :: (a -> c) -> (b -> c) -> Tree a b -> [c]
```

The values in the tree cannot be collected to a list as such because the values on the leaves are of a different type than the values on the branching nodes. Thus each of these functions takes two functions as arguments: The first function maps the values stored in the leaves to some common type `c`, and the second function maps the values stored in the branching nodes to type `c`, thus, resulting in a list of type `[c]`.

Exercise 5: A tiny language (30 points)
Let E be a tiny programming language that supports the declaration of arithmetic expressions and equality comparisons on integers. Here is an example program in E:

```
1 + 2 == 5 - ( 3 - 1 )
```

When evaluated, this program should evaluate to the truth value true.
In this exercise, we will not write E programs as strings, but as values of a Haskell data type `E`, that can represent E programs as their abstract syntax trees (ASTs). Given the following data type `E`,

```
data E = IntLit Int
       | BoolLit Bool
       | Plus E E
       | Minus E E
       | Multiplies E E
       | Divides E E
       | Equals E E
         deriving (Eq, Show)
```

The above example program is represented as its AST:

```
program = Equals
            (Plus (IntLit 1) (IntLit 2))
            (Minus
              (IntLit 5)
              (Minus (IntLit 3) (IntLit 1)))
```

Define an evaluator for the language E. Its name and type should be

```
eval :: E -> E
```

The result of `eval` should not contain any operations or comparisons, just a value constructed either with `IntLit` or `BoolLit` constructors. The result of the example program above should be `BoolLit True`.
Note that E allows nonsensical programs, such as `Plus (BoolLit True) (IntLit 1)`. For such programs, the evaluator can abort.