CSCE 314 [Sections 595, 596, 597] Programming Languages – Spring 2024
Hyunyoung Lee
Homework Assignment 2
Assigned on Thursday, February 8, 2024

Electronic submission to Canvas is due at **11:59 p.m., Friday, February 16, 2024**
By electronically submitting this assignment to Canvas by logging in to your account, you are signing electronically on the following Aggie Honor Code:
"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."

In this assignment, you will practice advanced concepts of Haskell such as list comprehensions, recursive functions, and higher-order functions.
You will earn total 100 points. Here are some general instructions.

1. This homework set is an *individual* homework, not a team-based effort. Discussion of the concept is encouraged, but actual write-up of the solutions must be done individually by yourself and your final product – the code as well as the comments and explanations – should never be shared.

2. Read the problem descriptions and requirements carefully! There may be significant penalties for not fulfilling the requirements.

3. **Explain each function definition line-by-line in your own words.** In addition, some problems ask you to explain the working of your function with the given input. Your explanation must be consistent with your definition of the function. Your work will be graded not only on the correctness of your answer, but also on the consistency and clarity with which you express it.

4. Submit electronically exactly one file named *YourFirstName-YourLastName*-hw2.**txt**, and nothing else, on canvas.tamu.edu. Make sure to change your Haskell script file extension as txt before submitting it. Only files with the .txt extension will be accepted to the submission link.

5. Make sure that the Haskell script you submit compiles without any error when compiled using the Glasgow Haskell Compilation System (ghc or ghci) version 8 or above[1].

   If your script does not compile, you will receive very few points (more likely zero) for this assignment. To avoid receiving zero for the entire assignment, if you cannot complete defining a function correctly without compile error, you can set the function definition `undefined`, see the skeleton code provided.

6. Remember to put the head comment in your file, including your name, UIN, and *acknowledgements of any help received* in doing this assignment. Again, remember the Honor Code!

[1]Version 8.10.7 is installed in the servers maintained by the college of engineering (linux2.engr.tamu.edu and compute.engr.tamu.edu), and version 9 is the most recent version.

Below, the exercise problems are from the Haskell Textbook: "Programming in Haskell, 2nd Ed." by Graham Hutton. Some problems are modified (with additional requirements) by the instructor. Please read corresponding textbook chapters and the problem statements carefully, paying attention to the requirements. For example, "using `fun1`, define `fun2`" means that calling `fun1` in the definition of `fun2` you are implementing is required. There may be significant penalties for not fulfilling such requirements. Keep the name and type of each function exactly the same as given in the problem statement and the skeleton code.

Problem 1. (5 points) Put your full name, UIN, and *acknowledgements of any help received* in the head comments in your Haskell script file for this assignment.

Problem 2. $(10 + 5 = 15$ points) Chapter 5, Exercise 9 (modified).
The scalar product function should be able to take not only two lists of the same length n but also those of *different* lengths.

2.1 (10 points) Using the list comprehension with the `zip` prelude function, write the scalar product function.

```
scalarproduct :: [Int] -> [Int] -> Int
```

2.2 (5 points) Explain carefully the step-by-step working of your `scalarproduct` function when it is invoked as follows (`ghci>` means GHCi prompt).

```
ghci> scalarproduct [3,2,4] [5..]
```

Problem 3. (10 points) Chapter 6, Exercise 7.
The `merge` function should be able to take two sorted lists of different lengths as well as those of the same length.

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

Problem 4. $(5 + 10 = 15$ points) Chapter 6, Exercise 8.
Defining function `halve` is required (hint: use `splitAt` prelude function).

4.1 (5 points) `halve :: [a] -> ([a], [a])`

4.2 (10 points) `msort :: Ord a => [a] -> [a]`

Problem 5. $(10 + 10 + 15 = 35$ points)

5.1 (10 points) Define a recursive function `mergeBy` that merges two sorted lists by the given criterion, for example, in an ascending order or in a descending order (so that the resulting list is also sorted). The type signature of `mergeBy` is as follows.

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

Notice the difference from `merge :: Ord a => [a] -> [a] -> [a]` in **Problem 3** (Ch. 6 Exercise 7) that `mergeBy` accepts three arguments, the first of which is a comparison function of type (`a -> a -> Bool`) that determines in which way the list is to be sorted. Such comparison function that returns a Boolean value (True or False) is called a *predicate*.

5.2 (10 points) Using `mergeBy` that you wrote above and `halve` that you wrote in **Problem 4**, define a recursive function `msortBy`. The problem specification stays the same as that for *msort* in **Problem 4** (Ch. 6 Exercise 8), except the additional requirement of the first argument being a predicate. Thus, the type of `msortBy` is:

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

5.3 (15 points) Explain carefully how your `msortBy` works step-by-step when it is applied as below (`ghci>` means GHCi prompt). Expand each recursive call of all functions involved.

```
ghci> msortBy (>) [3,2,1,5,4]
```

Problem 6. $(10 + 10 = 20$ points) Chapter 7, Exercise 9.

6.1 (10 points) Define `altMap`.

```
altMap :: (a -> b) -> (a -> b) -> [a] -> [b]
```

6.2 (10 points) Explain carefully how your `altMap` works step-by-step when it is applied as below (`ghci>` means GHCi prompt).

```
ghci> altMap (‘div‘ 2) (*7) [1..9]
```

Have fun!