CSCE 314 [Sections 500, 501] Programming Languages – Spring 2020

Hyunyoung Lee

Homework Assignment 1

Assigned on Tuesday, January 14, 2020

Electronic submission to eCampus due at 10:00 p.m., Monday, Jan. 27, 2020

By electronically submitting this assignment to eCampus 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 the basics of functional programming in Haskell. You will earn total 100 points. Here are some general instructions.

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

2. 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.

3. 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 and your final product – the code as well as the comments and explanations – should never be shared.

4. Submit electronically exactly one file named *YourLastName*-*YourFirstName*-hw1.hs, and nothing else, on eCampus.tamu.edu.

5. Make sure that the Haskell script (the .hs file) you submit compiles without any error when compiled using the Glasgow Haskell Compilation System (ghc or ghci) version 7 and above[1].

   If your program 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 7 is installed in the departmental servers (linux.cse.tamu.edu and compute.cse.tamu.edu), and version 8 is what you will get if you install the Haskell system in your computer.

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 textbook Chapters 1 thru 6 and the problem statements carefully. 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 comment in your .hs file.

Problem 2. (10 points) The n-th Lucas number ℓ_n is recursively defined as follows: $\ell_0 = 2$ and $\ell_1 = 1$ and $\ell_n = \ell_{n-1} + \ell_{n-2}$ for $n \geq 2$. Write a *recursive* function `lucas` that computes the n-th Lucas number.

```
lucas :: Int -> Int
```

Problem 3. ($5 + 10 = 15$ points) Chapter 1, Exercise 4 (modified).
Study the definition of the function `qsort` given in the text carefully, and try it out with an integer list, for example, `[3,2,3,1,4]`. You will notice that it sorts a list of elements in an ascending order.

3.1 (5 points) Write a recursive function `qsort1` that sorts a list of elements in a *descending* order.

```
qsort1 :: Ord a => [a] -> [a]
```

3.2 (10 points) Write your answer for this question in a block comment following the definition of function `qsort1`. Suppose that `qsort1` is invoked with the input `[3,2,3,1,4]`. How many times is `qsort1` called recursively (i.e., without counting the first invocation of `qsort1 [3,2,3,1,4]`)? Explain step-by-step, in particular, at each level of recursive call, what are the values of `x`, `smaller`, and `larger`? [Hint: Think using a binary tree structure.]

Problem 4. ($10 + 10 = 20$ 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.

4.1 (10 points) Using the list comprehension with the `zip` prelude function, write the scalar product function.

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

4.2 (10 points) Explain carefully step-by-step the working of your `scalarproduct` function when it is invoked as follows.

```
*Main> scalarproduct [1,2,3] [4..]
```

Problem 5. (10 points) Chapter 6, Exercise 7.
Like in the previous problem, 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 6. ($8 + 7 = 15$ points) Chapter 6, Exercise 8.
Defining function `halve` is required (hint: use `splitAt` prelude function).

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

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

Problem 7. (10 points) Write a recursive function `isElem` that returns `True` if a given value is an element of a list or `False` otherwise.

```
isElem :: Eq a => a -> [a] -> Bool
```

Sets are the most fundamental discrete structure on which all other discrete structures are built. In the following problem (and continuing in the next homework set), we are going to implement mathematical sets and their operations using Haskell lists.
A set is an *unordered* collection of elements (objects) without duplicates, whereas a list is an *ordered* collection of elements in which multiplicity of the same element is allowed. We define `Set` as a *type synonym* for lists as follows:

```
type Set a = [a]
```

Even though the two types – `Set a` and `[a]` – are the same to the Haskell compiler, they communicate to the programmer that values of the former are *sets* and those of the latter are lists.

Problem 8. (15 points) Write a recursive function that constructs a set from a list. Constructing a set from a list simply means removing all duplicate values. Use `isElem` from the previous problem in the definition of `toSet`.

```
toSet :: Eq a => [a] -> Set a
```

Skeleton code and modes of running your code: The file `hw1-skeleton.hs` contains "stubs" for all the functions you are going to implement and placeholders for your explanations. The Haskell function bodies are initially `undefined`, a special Haskell value that has all possible types (thus, the skeleton file at least compiles).
In the skeleton file you find a test suite that test-evaluates the functions. Initially, all tests fail, until you provide correct implementation for the Haskell function. The tests are written using the HUnit library. Feel free to add more tests to the test suite.
The skeleton code can be loaded to the interpreter (`> ghci hw1-skeleton.hs`). In the interpreter mode, you can test individual functions one at a time while you are implementing them. Evaluating the function `main` (by `> main` in the interpreter mode) runs all of the tests in the test suite.
Alternatively, you can compile the code and execute it in the terminal mode:
`> ghc hw1-skeleton.hs` and then
`> ./hw1-skeleton`
which has the same effect as when you do `> main` in the interpreter mode.

Have fun!