CSCE 314 Programming Languages – Fall 2015

Hyunyoung Lee

Assignment 2

Assigned on Friday, September 4, 2015

Electronic submission to eCampus due at 23:59, Tuesday, Sep. 15, 2015
Honor code signed coversheet due at the beginning of class on Wednesday, Sep. 16, 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."

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

In this assignment, you will practice the basics of functional programming 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 100 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*-a2.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. To avoid receiving zero for the whole assignment, if you cannot complete defining a function correctly, you can set it `undefined`, see the skeleton code provided.

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. Do not use functions in the standard prelude such as `sum`, `product`, `map`, etc. You can use operators such as $+$, $*$, $==$, $:$, $++$, etc.

Problem 1. (5 points) Write a recursive function `factorial`.

```
factorial ::  Int -> Int
```

Problem 2. (5 points) Write a recursive function `fibonacci` that computes the n-th Fibonacci number.

```
fibonacci ::  Int -> Int
```

Problem 3. (5 points) Write a recursive function `mySum` that sums all the numbers in a list.

```
mySum ::  [Int] -> Int
```

Problem 4. (10 points) Write a recursive function `flatten` that flattens a list of lists to a single list formed by concatenation.

```
flatten ::  [[a]] -> [a]
```

Problem 5. (10 points) Write a recursive function `myLength` that counts the number of elements in a list.

```
myLength ::  [a] -> Int
```

Problem 6. (10 points) Write a recursive function that returns `True` if a given value is an element of a list or `False` otherwise.

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

The following problems are 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. Let us 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 7. (10 points) Set constructor. Write a recursive function that constructs a set. Constructing a set from a list simply means removing all duplicate values. Use `isElement` in the definition.

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

All the remaining functions can assume that their incoming set arguments are indeed sets (i.e, lists that do not contain duplicates), and correspondingly, your implementations must guarantee this property for the sets that are returned as results.

Problem 8. (5 points) Using `myLength` that you have already defined, write a function `size` that returns the number of elements in a set.

```
size ::  Set a -> Int
```

Problem 9. (10 points) Write a recursive function `subset` such that `subset a b` returns `True` if `a` $\subseteq$ `b` or `False` otherwise. Use `isElement` in the definition.

```
subset ::  Eq a => Set a -> Set a -> Bool
```

Problem 10. (10 points) Using `subset` you have already defined, write a function `setEqual` that returns `True` if the two sets contain exactly the same elements, or `False` otherwise.

```
setEqual ::  Eq a => Set a -> Set a -> Bool
```

Problem 11. (20 points) The powerset of set s is a set containing all subsets of s. Write a recursive function `powerset` that returns a set containing all subsets of a given set. Use direct recursion and list comprehension.

```
powerset ::  Set a -> Set (Set a)
```

Skeleton code: The file a2-skeleton.hs contains "stubs" for all the functions you are going to implement. The function bodies are initially `undefined`, a special Haskell value that has all possible types. Also in that file you find a test suite that test-evaluates the functions. Initially, all tests fail – until you provide correct implementation for the 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 a2-skeleton.hs`). Evaluating the function main will run the tests. Alternatively, one can compile the code and execute it:
`> ghc a2-skeleton.hs` and then `> ./a2-skeleton`