CSCE 314 [Sections 595, 596, 597] Programming Languages – Spring 2024

Hyunyoung Lee

Homework Assignment 1

Assigned on Thursday, January 25, 2024

Electronic submission to Canvas due at 11:59 p.m., Wednesday, 2/7/2024

By 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 the basics of functional programming in Haskell. You will earn total 140 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 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*-hw1.**txt**, and nothing else, to the submission link on canvas.tamu.edu. **Make sure to change your Haskell script file extension to 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 and 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 problem in Problem 2 is from the Haskell Textbook: "Programming in Haskell, 2nd Ed." by Graham Hutton. The problem is modified (with additional requirements) by the instructor. Reading textbook Chapters 1 through 6 will be helpful for this homework set. Keep the name and type of each function exactly the same as given in the problem statement and in the skeleton code.

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

Problem 2. ($5 + 10 = 15$ 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, `[5,2,6,9,7]`. You will notice that it sorts a list of elements in an ascending order.

2.1 (5 points) Write a recursive function `qsort1` that sorts a list of elements in a *descending* order.

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

2.2 (10 points) Write your answer for this question in a block comment following the definition of the function `qsort1`. Suppose that `qsort1` is invoked with the input `[5,2,6,9,7]`. How many times is `qsort1` called recursively (i.e., without counting the first invocation of `qsort1 [5,2,6,9,7]`)? 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 as shown in the lecture slides haskell-01-basics, slide# 18 and as explained in the lecture video Video 1.2.]

Problem 3. (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 $n > 1$. Write a *recursive* function `lucas` that computes the n-th Lucas number. Explain your code line-by-line.

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

Problem 4. (10 points) Write a *recursive* function `factorial` that computes the n factorial $n!$ (*without using* the prelude function `product`) with the base case $0! = 1$ by definition. Explain your code line-by-line.

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

Problem 5. ($5 + 10 + 10 = 25$ points) Given an integer n, the semi_factorial of n is recursively defined as follows: semi_factorial(0) = 1 and semi_factorial(1) = 1, and

$$\text{semi\_factorial}(n) = n \times \text{semi\_factorial}(n-2) \quad \text{for } n > 1.$$

5.1 (5 points) Write a *recursive* function `semifactorial` that computes the semi_factorial of n. Explain your code line-by-line.

```
semifactorial :: Int -> Int
```

5.2 (10 points) Write your answer for this question in a block comment following the definition of the function `semifactorial`. Suppose that `semifactorial` is invoked with input `12`. How many times is `semifactorial` called recursively (i.e., without counting the first invocation of `semifactorial 12`)? Explain step-by-step.

5.3 (10 points) Write the function `myfactorial` *using* `semifactorial`. The function `myfactorial` applied to n must result the same value as $n!$. However, you are not allowed to use the `factorial` function but must use the `semifactorial` function in the definition. Only one base case should be used, that is, `myfactorial` with argument 0 returns 1. Explain your reasoning clearly.

Problem 6. (10+15+10 = 35 points) We want to write a program to evaluate a polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

so for a given input x, you would like to know the value $y = p(x)$. We implement this naive algorithm as below.

6.1 (10 points) First, implement the function `term` with two arguments `n` (for the degree of the polynomial) and `x` (for the input value x in $p(x)$) that calculates x^n. The term function is recursively defined as

$$\text{term}(n, x) = \begin{cases} x & \text{for } n = 1, \\ x \times \text{term}(n-1, x) & \text{for } n > 1. \end{cases}$$

Explain your code line-by-line.

```
term :: Num a => Int -> a -> a
```

6.2 (15 points) Now, use the term function you defined above to write a function `polynaive` that takes three arguments:

- `as` a list that contains the coefficients such as $[a_n, a_{n-1}, a_{n-2}, \ldots, a_0]$,
- `n` the degree of the polynomial,
- `x` the input value x in $p(x)$,

and evaluates $p(x)$. Explain your code line-by-line.

```
polynaive :: Num a => [a] -> Int -> a -> a
```

6.3 (10 points) Write your answer for this question in a block comment following the definition of the function `polynaive`. Explain step-by-step of the workings of your functions when `polynaive` is invoked as `polynaive [3,-4,2,7] 3 2` that evaluates

$$p(x) = 3x^3 - 4x^2 + 2x + 7$$

for $x = 2$, where the result must be 19.

Your explanation should show the step-by-step of how your `polynaive` function works given those specific argument values `[3,-4,2,7] 3 2`, like, for example, you did for Problem 2.2 with `qsort1`.

Sets are the most fundamental discrete structure on which all other discrete structures are built. In the following problems, 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 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
```

Problem 8. (10 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
```

All the remaining functions can assume that their incoming set arguments are indeed sets (i.e, lists that do not contain duplicates).

Problem 9. (10 points) A set A is called a *subset* of a set B if and only if all elements of A are also elements of B. We write $A \subseteq B$ to denote that A is a subset of B. Write a *recursive* function `subset` such that `subset a b` returns `True` if `a` $\subseteq$ `b` or `False` otherwise. Use `isElem` in the definition.

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

Problem 10. (10 points) Two sets A and B are *equal* if and only if the two sets contain exactly the same elements. The equality of two sets A and B can be proven by checking if the two sets are subsets of each other, that is, if $A \subseteq B$ and $B \subseteq A$, then $A = B$. Using `subset` you have already defined, write a function `setEqual` that returns `True` if the two sets are equal, or `False` otherwise.

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

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!