Assignment 1

CS 4400 / CS 5400 Programming Languages

Due: Friday, September 27, by midnight.

Submission:

1. Submit one file named `Assignment1.hs` via Blackboard.
2. At the very top, the file should contain a preamble following this template.

```haskell
{- |
Module      :  Assignment1
Description :  Assignment 1 submission for <CS 4400 / CS 5400 (choose one)>.
Copyright   :  (c) <your name>

Maintainer  :  <your email>
-}

module Assignment1 where

-- your code goes here
```

   The rest of the file will contain your solutions to the exercises below.

3. Each top-level function must include a type signature, followed by one or more defining equations.
4. Make sure your file loads into GHCi or can be compiled by GHC without any errors.

Purpose: The purpose of this assignment is to get a bit of practice with Haskell, especially with processing lists and trees. Most of the concepts involved in the exercises here should be familiar to you from prerequisite courses or other parts of the curriculum. However, the Haskell specifics might be new, in particular working with types, polymorphism, typeclasses, and some of the syntax. If something is not clear, you are encouraged:

a) to look at online resources – see the course page for online material; and
b) to ask questions after class, during office hours, or on Piazza.

We also recommend familiarizing yourself with Hoogle – a very handy search engine for Haskell's libraries. It allows searching by name or by type.

Grade: To calculate your grade, we will take the following into account:

a) Does your code compile without errors?
b) Does it follow the above rules?
c) Are functions and constants named as specified? Do they have the correct types?
d) Does your code behave as specified? This will be determined by unit testing.
e) How readable is your code?

Readability will initially play a small role, but will become more important with each further assignment.

Exercises

Recursive Functions

Task 1: Give a definition of Fibonacci as a recursive function named `fibonacci` with the following type signature:

```
fibonacci : Integer -> Integer
```

Tip: Use the factorial example from lecture 1 (in `Lec01.hs`) as guidance, but pay attention to how many base cases you need for `fibonacci`.

Lists and Polymorphism

Haskell lists are defined using the empty list constructor `[]` and the infix cons constructor `_ : _`. Moreover, list values can be constructed by listing values between `[` and `]`, separated by a comma `,`. The following are all list values:

```
[]               -- empty list
[1, 2, 3]        -- list containing the numbers 1, 2 and 3
1 : [2, 3]       -- the same list as above
1 : 2 : 3 : []   -- the same as above
```

Functions over lists can pattern-match on these two constructors. For example, here is how we define the length of a list:

```
length :: [a] -> Integer
length [] = 0
length (x : xs) = 1 + length xs
```

This is a *polymorphic function*: it works on lists with elements of any type, as it does not depend on any specific operations on the element type. This is captured in its type by using the type variable a instead of a particular type.

Note on function types in Haskell In Haskell, function types have the general form:

```
f :: ArgumentType1 -> ArgumentType2 -> ... -> ArgumentTypeN -> ReturnType
```

The last arrow separates argument types from the return type. This means that a unary function which takes an integer and returns a boolean has the type `Integer -> Bool`; a function which takes an integer and a string, returning a boolean `Integer -> String -> Bool`; a function which takes three intgers and returns a boolean `Integer -> Integer -> Integer -> Bool`; and so on. We will talk about the reasons for this notation later on, but, for now, think of the type signature

```
g :: Integer -> Integer -> Integer -> Bool
```

as saying

> The function `g` takes an `Integer` *then* it takes another `Integer` *then* it takes another `Integer` and, finally, returns a `Bool`.

Task 2: Checking if a list contains the given value.

Give a recursive definition for the function:

```
isIntegerElem :: Integer -> [Integer] -> Bool
```

(That is, a function `isIntegerElem` which takes an `Integer` and a list of `Integer`s and returns a `Bool`.)

This function should go through the list and return `True` if it finds the given element or `False` if the list contains no such element. Here are some test cases:

```
isIntegerElem 0 [] == False
isIntegerElem 0 [0] == True
isIntegerElem 1 [3, 2, 4, 1, 4, 1] == True
isIntegerElem 1 [3, 4, 2, 0] == False
```

If you have written your function well, you might notice that it should work for any type that supports equality, not just integers. This means that its type is not as general as possible and we might want to *generalize* it. Haskell provides *typeclasses* as a mechanism to support *ad hoc polymorphism*. For example, the `Eq` typeclass requires the `==` ("equal to") and `/=` ("not equal to") operations. To use equality in our function, we need to add a *typeclass constraint* to the type of the function. For example, if I wish to write a function `foo :: a -> Bool` and I want it to work for any type `a` with equality, I say so in the type:

```
foo :: Eq a => a -> Bool
foo x = ... (x == ... -- foo can use equality on values of type a
```

Task 3: Generalizing the type of `isIntegerElem`.

Adapt `isIntegerElem` into a function `isElem` that works for any type with equality (not just integers) by adding a typeclass constraint to its type. Give the function's type signature and adapt the definition you gave for `isIntegerElem` as appropriate.

Example test cases:

```
isElem "x" ["x"] == True
isElem "x" ["xyz"] == False
isElem True [False] == False
isElem 30 [10, 20, 30, 40, 50] == True
```

Task 4: Write a function `count` which counts occurrences of an element in a list. Its type should be as general as possible and it should return an `Integer`. As a hint, here is a type signature template:

```
count :: {- constraint -} => {- element type -} -> {- list type -} -> Integer
```

You will need to replace the comments with the appropriate types / type constraints.

Example test cases:

```
count 10 [] == 0
count 10 [10] == 1
count 1 [0, 1, 1, 42, 1] == 3
count "x" ["xyz", "zyx", "xxx"] == 0
count "x" ["x", "xxx", "x"] == 2
```

Binary Trees

In Haskell, we can specify a (polymorphic) binary tree as the following datatype:

```haskell
data Tree a = Node (Tree a) a (Tree a)
            | Empty
            deriving (Show, Eq)
```

Here, the constructor `Node` takes 3 arguments: the left subtree, an element of type `a` and the right subtree. The constructor `Empty` represents an empty tree. For example, the tree

```
    3
   / \
  5   6
       \
        8
```

is represented as

```haskell
  Node (Node Empty 5 Empty)
       3
       (Node Empty 6 (Node Empty 8 Empty))
```

Task 5: Copy the above definition of `Tree` into your submission file. Transcribe the following tree into Haskell:

```
             18
            /  \
           /    \
          /      \
         15      20
        /  \     /
      40   50   8
     /  \        \
    30  50        13
```

Use the following template to include the tree in your submission file:

```haskell
tree1 :: Tree Integer
tree1 = {- insert your answer here -}
```

Task 6: Write a function inOrder, which performs an in-order traversal of the tree and returns all the elements as a list. The function should have the following type:

```
inOrder :: Tree a -> [a]
```

Example test cases:

```
inOrder Empty == []
inOrder (Node Empty 4 Empty) == [4]
inOrder (Node (Node (Node Empty 1 Empty) 2 Empty) 3 (Node Empty 4 (Node Empty 5 Empty)))
  == [1, 2, 3, 4, 5]
inOrder tree1 == [30, 40, 50, 15, 50, 18, 8, 13, 20]
```

Hint: You might want to refresh your memory on the ++ operator.