Functional Programming - Exercise 3 - 25 Points

Again, in this exercise the main goal is to recap some important concepts and tradeoffs we discussed in the lecture. The focus is really on deepen the understandig and get ideas for practical programming, so please keep the answers very short.

- **Exercise 1: Composing programs** (10 Points). Function composition is an important mechanism for composing programs in purely functional programs and influenced common techniques such as argument piping and method chaining. We alreay looked at function composition in Haskell. The definition was:

  ```
  compose f g x = f (g x)
  ```

  In Haskell it is also known as the `.` operator motivated by the mathematical ∘ symbol. We used it in the second lecture to solve the line-count problem. It looked like this:

  ```
  -- a pure function, comments start with --
  countLines :: String -> Int
  countLines =
       -- first split by lines, filter by size and return the length of the
  result
       length . filter isMini . lines
           where
               isMini lines = length lines > 10
  ```

  Write a variant of the function composition function in a language of your choice (which supports it, e.g. C#, Java, Typescript, Python, Javascript, C++,..). The implementaiton might differ syntactically from the Haskell variant. Identify the main reasons for the differences.

  HINT: Compose is a higher order function, it takes two functions (f and g) and creates a new function which applies the argument first to `g` and then to `f`.

  (Optional) Add-on for interested readers: can you also create a curried version of the function?

- **Exercise 3: Dealing with errors (10 Points)** We looked at various LINQ/Stream operators and their implementations in the lecture. Particularly interesting are operators which might fail (e.g. Min for example if the sequence is empty). In this task your goal is to refactor a F# implementation of the First Operator to use the Option type instead of exceptions. The full source code of the example is in in the moodle assignment `Lists.fsx`. Note that this is very similar to the Java version of this function "findFirst".

  ```
  // Returns the first element in a sequence that satisfies a specified
  condition.
  let rec first (predicate : 'a -> bool) (xs : list<'a>) : 'a =
      match xs with
      | [] ->
          // an empty list does not have an element which satisfies the predicate
          // throw an exception
          let e = System.InvalidOperationException("No element satisfies the
  condition in predicate.")
          raise e // f# way of saying throw
      | x::xs ->
          if predicate x then
              x
  ```

```
        else
            first predicate xs

let testFirst1 = first (fun x -> x < 10) [1;2] // 1
let testFirst2 = first (fun x -> x < 10) [2;11] // 2
let testFirst3 = first (fun x -> x < 10) [11] //
System.InvalidOperationException: No element satisfies the condition in
predicate.
```

Your task is to write a "safe" variant of first which uses the Option type to encode failure. For example:

```
let testFirst4 = tryFirst (fun x -> x < 10) [1;2] // Some 1
let testFirst5 = tryFirst (fun x -> x < 10) [2;11] // Some 2
let testFirst6 = tryFirst (fun x -> x < 10) [11] // None
```

Answer the questions:

- How does the variant with Option compare to the FirstOrDefault Operator.
- Which version would you prefer (and why) - exception or using optional?

- **Exercise 3: Understanding persistent datastructures (5 Points)** Recently, immutability pops up in many languages and communities. Using immutable variables (e.g. const, final) or just preventing mutable variable assignment is part of a functional programming style. Real power of immutability however becomes really appearent, when using immutable data-structures. In the lecture we looked at an immutable set implementation and talked a lot about advantages of immutable data structures. Review the examples of the lecture and answer those questions (please keep them short!):
  - Each operation keeps the old version of the data-structure intact - how do smart implementations prevent copying the whole data-structure?
  - When would you choose an immutable variant of a datastructure over a mutable version. Good examples for the immutable version is here and the standard dictionary here.

- **Sumbission.** Submit your as condensed as possible - e.g. a single file with all the code/markdown. Please don't put it into a zip folder if possible.