universität innsbruck

Functional Programming	WS 2020	LVA 703025
Exercise Sheet 12, 10 points		Deadline: Wednesday, January 27, 2020, 6am

- Please write all the Haskell code into .hs-files and upload them in OLAT.
- Your .hs-files should be compilable with ghc.
- Don't forget to mark your completed exercises in OLAT.

Exercise 12.1 *Modules* **5 p.**

In this exercise you will implement a small module for *bags* (a collection data type that is also referred to as *finite multisets*). Your implementation should reside in a file `Bag.hs`.
Note: Since the specific class constraints of the functions you will implement in this exercise depend on the actual representation you choose in 1, we just use `...` below.

1. Implement a data type `data Bag a = Bag ...`, replacing `...` by some appropriate representation of multisets. Moreover, modify the export list of your module such that the implementation specifics are hidden from client code (that is, only the type constructor `Bag` is visible, but not the value constructor `Bag`). Additionally, implement a function `fromList :: ... => [a] -> Bag a` that allows client code to obtain a bag from a given list and export it from your module. (1 point)

2. Implement a function `toList :: ... => Bag a -> [a]` that transforms a given multiset into a list of its elements and export it from your module. (1 point)

3. Make your data type `Bag a` an instance of the type classes `Show` and `Eq`. Your `Show` instance should generate output in set notation.

   Examples:
   ```
   Bag.fromList [1,1,2] == Bag.fromList [1,2,1]
   show (Bag.fromList []) == "{}"
   show (Bag.fromList [1,2,1]) == "{1,1,2}"
   ```
   (1 point)

4. Implement a function `count :: ... => Bag a -> a -> Int` that counts the number of occurrences of the given element in the given multiset.

   Examples:
   ```
   Bag.count (Bag.fromList []) 1 == 0
   Bag.count (Bag.fromList [0,1,2,1]) 1 == 2
   ```
   (1 point)

5. Implement *multiset sum* `sum :: ... => Bag a -> Bag a -> Bag a` resulting in a multiset that contains all the elements of both arguments to `sum` and export it from your module.
   For any two multisets `a` and `b` and any element `x` the following equation should hold:

   ```
   Bag.count a x + Bag.count b x == count (Bag.sum a b) x
   ```

   Example:
   ```
   Bag.sum (Bag.fromList [1,2,1]) (Bag.fromList [2,3]) == Bag.fromList [1,1,2,2,3]
   ```
   (1 point)

Exercise 12.2 *Interactive IO* 5 p.

Your goal in this exercise is to write a simple REPL (Read Eval Print Loop) that allows a user to inspect specific lines of a file that is specified on startup. Your implementation should reside in a file `filerepl.hs`.

1. Implement a function `replRead :: IO (Maybe Int)` that reads a line of user input and tries to return it as an integer. If the input cannot be turned into an integer there should be some kind of error message together with the result `Nothing`. (1 point)

   Note: You can use `read :: String -> Int` to parse strings representing a valid number to `Int`. Because `read :: Read a => String -> a` is polymorphic in the return type you may want to specify this type, as in `(read "15" :: Int) == 15`.

   Examples:

   ```
   ghci> replRead
   4                          # user input
   Just 4                     # result
   ghci> replRead
   hello                      # user input
   wrong input, try again ... # error message
   Nothing                    # result
   ```

2. Implement a function `replEval :: String -> Int -> IO (Maybe String)` that, given a filename and an integer `n`, tries to return the `n`th line of the corresponding file as a string. If this is not possible (for example when `n` does not correspond to a correct line number of the given file) there should be some kind of error message together with the result `Nothing`. (1 point)

   Examples:

   ```
   ghci> replEval "inputfile.txt" 1
   Just "..."                 # result: the first line of inputfile.txt as string
   ghci> replEval "inputfile.txt" 1000
   1000 is out of bounds      # error message
   Nothing                    # result
   ```

3. Implement a function `replPrint :: Int -> String -> IO ()` that, given an integer `n` and a string `s`, prints `s` as the `n`th line of a file. (1 point)

   Example:

   ```
   ghci> replPrint 1 "this is the first line"
   line 1: this is the first line
   ```

4. Implement a function `replLoop :: String -> IO ()` that combines the previous three functions into an interactive loop. In every iteration of the loop the user should be asked which line of the file she wants to see. Use `replRead` to read the user input. The loop stops when this results in `Just 0`. Otherwise, we either have the result `Nothing` and directly enter the next loop iteration, or we have the result `Just n` for some integer `n`. At this point we try to retrieve the `n`th line of the given file using `replEval`. If this fails we directly enter the next loop iteration. Otherwise we use `replPrint` to print the `n`th line of the given file and afterwards enter the next loop iteration. (1 point)

   Example:

   ```
   ghci> replLoop "inputfile.txt"
   which line do you want to see (0 to quit)?
   1                          # user input
   line 1: this is the first line
   which line do you want to see (0 to quit)?
   0                          # exit loop
   ```

5. Finally, put everything together into a program `filerepl` that takes a filename as command line argument. When started, it should first print some information about the given file and then enter `replLoop`. (1 point)

   Example:

```
$ ghc --make filerepl.hs   # compile your program using GHC
$ ./filerepl inputfile.txt # invoke your program on input inputfile.txt
inputfile.txt has 42 lines # some information about the given file
which line do you want to see (0 to quit)?
0
```