Functional Programming
(WS14/15)

Assignment 1
(50 tokens)

Hints:

- For this assignment, you can achieve up to 50 tokens. For 10 tokens you get 1 bonus point in the course exam, but only if you would pass the exam without any bonus points.
- It is NOT allowed to work in groups or to provide any solutions to other participants of the course. In case of plagiarism, no points are given.
- Hand in your solution as a **single ZIP-file**. The file name must have the form *firstname-surname-number_of_assignment.zip*, e.g. ***max-power-1.zip***.
- For the programming tasks please use *a single* plain-text-file (suffix "*.hs*") in which you put the source code of *all* programming tasks.
- At the beginning of your Haskell source code file write your name and your matriculation number as a comment, e.g.,

```
-- Max Power (1234567)
```

- Do not hand in code that can't be interpreted by your Haskell Interpreter. Otherwise your solutions can't be considered and you won't receive any tokens.
- In case you do not use the *Glasgow Haskell Compiler* (GHC) in Version 7, mention the name of your interpreter and its version number as a comment at the beginning of your Haskell source code file, e.g.,

```
-- used Hugs 98, Version September 2006
```

- If you implement auxiliary functions (*Hilfsfunktionen*) then explain in a comment what they do and how they work.
- You may – but you needn't – use the functions of the package `Data.Char` of the Haskell API after adding the following line at the beginning of your code:

```
import Data.Char
```

  The use of other packages is NOT allowed.
- Upload the ZIP-file with your solutions to the Virtual Campus course by **Sunday, 14. December 2014, 23:55** (server time).

Task 1. **Lists** : In Haskell, lists are a homogenous data structure. They contain several elements of the same type. This means that we can have a list of integers or a list of characters but we cannot have a list that has some integers and then some characters.

1. **Searching Lists** (10 tokens):

   - Codify a function

     ```
     findPos :: String -> [String] -> Int
     ```

     that takes a string and a list of strings. It returns the position of the given string in the given list where 0 is the first position, 1 the second and so on. If no position can be obtained the function returns -1. The string comparison shall **not** be case sensitive, e.g. the characters `'A'` and `'a'` must be considered to be equal. For example:

     ```
     Main> findPos "hello" ["world", "Simple", "Hello"]
     2
     Main> findPos "hello" []
     -1
     Main> findPos "hello" ["world", "Simple", "Hell"]
     -1
     ```

   - Next, codify a function

     ```
     findCharString :: Char -> [String] -> [String]
     ```

     It returns a list of strings which holds those strings of the input list that contain the given character. Note that the comparison is not case-sensitive, again. For example:

     ```
     Main> findCharStrings 'a' ["world", "Simple", "Hello"]
     []
     Main> findCharStrings 'E' ["world", "Simple", "Hello"]
     ["Simple", "Hello"]
     ```

2. **List manipulation** (10 tokens):

   Codify a function

   ```
   sortStrings :: [String] -> [String]
   ```

   that takes a list of strings and returns this list in alphabetically sorted form. Again, upper case and lower case letters shall not be distinguished! For example:

   ```
   Main> sortStrings ["world", "Simple", "Hello"]
   ["Hello", "Simple", "world"]
   Main> sortStrings ["aa", "ZZ", "AAA"]
   ["aa", "AAA", "ZZ"]
   ```

   You may implement any sorting algorithm you consider to be suitable but explain briefly, in a comment, how it works!

   If you are not able to solve this task at all, implement a dummy function that returns a list as it is given as input in unsorted form.

3. **Name Scores** (5 tokens): For every string you can calculate an *alphabetical value* by adding the positions in the alphabet for each letter that is part of the string. For example, the string `"Bill"` has an alphabetical value of 35 (B=2 + I=9 + L=12 + L=12). All characters which are not letters have a value of 0.

   Calculating *name scores* starts by sorting all strings of a list into alphabetical order. Then, working out the alphabetical value for each string, multiply this value by its alphabetical position in the sorted list to obtain a name score. You may use your functions from the previous tasks!

   Provide a function

   ```
   nameScore :: String -> [String] -> Integer
   ```

   that calculates the name score of a certain element of a **sorted** string list. For example:

   ```
   Main> let myList = ["Franklin", "Dwight", "Andrew", "Bill!", "Abraham"]
   Main> nameScore "Bill!" (sortStrings myList)
   70
   ```

   "Bill!" is at position 2 (starting at 0 again) of the *sorted* list, so that is its alphabetic position. The string "Bill!" has an alphabetical value of 35. The *name score* is obtained by multipling the alphabetic value and its alphabetic position, so it is $2 \times 35 = 70$.

   Note that `nameScore` does **not** sort any lists. You may assume that the list that is given as input is already sorted! You can use `fromIntegral` to transform `Int`s to `Integer`s before applying the multiplication.

   If the input string is not an element of the input list then `-1` shall be returned.

Task 2. The I/O - Monad

Let us consider the data type `IO`. You need it whenever you want to have interaction with the environment, e.g., handling user inputs, operating on files, programming GUIs, etc.

We start with two basic commands. Start your interpreter and type:

```
Prelude> putStrLn "Hello."
Hello.
```

The function `putStrLn` receives a string and prints it to your terminal followed by a return symbol. You can also prevent the return-symbol with `putStr`.

The functions `putStrLn` and `putStr` do not really return a value. They *interact* with the environment. You can display the type with `:t`.

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
```

The brackets `()` indicate that no actual value is returned. By contrast, the function `getLine` returns an IO-String.

```
Prelude> :t getLine
getLine :: IO String
```

Note that here we have the IO-wrapper that indicates that we are leaving the secure, pure functional part of Haskell. Now the program flow depends on the environment which might be dangerous as we lose determinism.

An advantage of the type IO is that you can call several IO-functions in sequence. Try the following program:

```
sayHallo :: IO ()
sayHallo = do
             putStrLn "What is your name?"
             name <- getLine
             putStrLn ("Hallo, " ++ name ++ ".")
```

A sequence of functions is started with the key-word `do`. In the next line, the user is asked to give its name. The input is assigned to the *local variable name* with the `<-` operator. The type of `name` is `String` (not `IO String`). So, simply spoken, the `<-` transforms a value of type "`IO a`" to an equivalent value of type "a".

Make sure that the function calls after the `do` start in the same column. Otherwise you might get into trouble.

1. **Reading from the Console** (5 tokens):

   Now create a function

   ```
   readConsole :: IO ()
   ```

   that receives a number (as string) from a user and prints the digit sum of that number:

   ```
   *Main> readConsole
   Give a Number:
   334
   The digit-sum of 334 is 10.
   ```

   If the given string is empty or cannot be transformed into an integer then this must be recognized by your function:

   ```
   *Main> main
   Give a Number:
   Hello
   Next time, enter a valid number!
   ```

2. **IO with Files**: Read the section about *File I/O* at `https://www.haskell.org/haskellwiki/Tutorials/Programming_Haskell/String_IO#File_IO`.

   - **Reading files** (10 tokens):
     Now codify a function

     ```
     readFileTokens :: FilePath -> IO [String]
     ```

     that reads a file and returns a list of strings with all words from the file. All characters that are not letters must be omitted and only spaces (blanks) are considered as word separators. For example:

     ⇒ Content of test.txt:

     ```
     A simple test!!!Yeah.
     ```

     ```
     Main> readFileTokens "test.txt"
     ["A","simple","testYeah"]
     ```

     Note that the type `FilePath` is simply the same as `String`. You do not need to check whether the file exists and you may assume that there is always exactly one space between two words.

- **Writing files** (3 tokens): File writing is as easy as file reading.
  Now codify a function

  ```
  writeFileTokens :: [String] -> FilePath -> IO ()
  ```

  that writes a list of strings to a file. Each element is be seperated by comma and a space. For example:

  ```
  Main> writeFileTokens ["FileIO","is","fun"] "output.txt"
  ```

  ⇒ Content of output.txt:

  ```
  FileIO, is, fun
  ```

  Note that there is no comma after the last string.

- **Putting all together** (7 tokens): Reusing functions from the other tasks, implement

  ```
  main :: IO ()
  ```

  that reads (unsorted) names from a file, calculates the name score (Task 1.3) of each name and writes pairs of names and the corresponding score to an output file. Both file names (input and output file name) are given by the user during runtime. The names and scores are separated by a comma and a space, while the pairs are separated by return symbols '\n'. You do not need to check whether the input file exists. You will find the file `names.txt` in the VC-course to test your function.

  For example:

  ⇒ Content of test.txt:

  ```
  Franklin Dwight Andrew Bill! Abraham
  ```

  Interpreter:

  ```
  *Main> main
  Please enter the path to the input file:
  test.txt
  Please enter the path to the output file:
  output.txt
  Writing to file...
  ```

  ⇒ Content of output.txt:

  ```
  Franklin, 340
  Dwight, 213
  Andrew, 65
  Bill!, 70
  Abraham, 0
  ```