Programming in Haskell – Homework Assignment 3

UNIZG FER, 2014/2015

Handed out: October 18, 2014. Due: October 24, 2014 at 23:59

Note: Define each function with the exact name and the type specified. You can (and in most cases you should) define each function using a number of simpler functions. Provide a type signature above each function definition and comment the function above the type signature. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message. Problems marked with a star ($\star$) are optional.

1. Implement an `interleave` function that interleaves elements of two lists by sequentially alternating between their elements.
```
interleave :: [a] -> [a] -> [a]
interleave [1,2,3] [4,5,6] ⇒ [1,4,2,5,3,6]
interleave [1,2,3,4] [5,6] ⇒ [1,5,2,6]
interleave [1] [3,4] ⇒ [1,3]
interleave "Longish string containing many letters" "" ⇒ ""
```

2. Implement a `slice` function that extracts a slice from a list. More precisely, given two indices, **i** and **j**, returns a list from the $\mathbf{i}^{th}$ to the $\mathbf{j}^{th}$ element (inclusive). Giving the indices in reverse order should produce the same result.
```
slice :: Int -> Int -> [a] -> [a]
slice 2 6 "Example" ⇒ "ample"
slice 6 2 "Example'' ⇒ "ample"
slice 0 0 [[1,2], [3,4], [5,6]] ⇒ [[1,2]]
slice 0 7 [1,2] ⇒ error "Slice index out of range"
slice (-1) 5 any ⇒ error "Slice index out of range"
```

3. Define `decamel`, a function that converts an identifier from lower or upper camel case format to a regular one.
```
decamel :: String -> String
decamel "random" ⇒ "random"
decamel "wordCount" ⇒ "word count"
decamel "AbstractSingletonProxyFactoryBean"
⇒ "abstract singleton proxy factory bean"
decamel "" ⇒ error "identifier is empty"
decamel "contains Whitespace"
⇒ error "input not in camel case format"
```

4. (a) Implement a function `count` that takes a list of elements belonging to the `Eq` typeclass and a single element and returns the number of occurrences of the element in the list. Implement it using a list comprehension.

```
count :: Eq a => [a] -> a -> Int
count "Tiffany" 'f' ⇒ 2
count "Elephant" 'o' ⇒ 0
count [1..]  2 ⇒ ⊥
```

(b) Implement a function `removeUniques` that takes a list and returns a list where elements occurring only once have been filtered out.

```
removeUniques :: Eq a => [a] -> [a]
removeUniques [1,2,3,2,3,5] ⇒ [2,3,2,3]
removeUniques [2,2,2] ⇒ [2,2,2]
removeUniques [1,2,3] ⇒ []
removeUniques [] ⇒ []
```

5. Define a function `mask` that takes a string and a binary mask, defined below, and preserves a character in the string only when a '1' is present at the same index in the mask. If the binary mask is shorter than the string, repeat it until it is of sufficient length (Hint: `cycle`). You can assume the mask will never contain any characters other than '1' and '0'. An empty mask can be considered the same as an all-zero mask.

   We use the `type` keyword to declare type aliases, telling Haskell that the left hand side type is just another name for an existing right hand side type (e.g., `String` ⇔ `[Char]`).

```
type Mask = String

mask :: String -> Mask -> String
longmask = "10000000100000100000000001010000"
mask "Testing only matching positions" longmask ⇒ "Toast"
mask "Longer string" "1110" ⇒ "Loner trig"
mask "No" "001" ⇒ "" mask "String" "" ⇒ ""
```

6. Steve feels lonely and friendless at times. However, he thought of a perfect solution for his problems and it involves programming in Haskell! He devised an application where people can share their location and find the nearest other person wanting to hang out. He needs your help with coding the last part of his application. Write the function `findFriend` that takes a list of people and their locations and finds the name of the closest one to a given point. Use the types defined below.

```
type Point = (Int, Int)
type Friend = (Point, String)

findFriend :: Point -> [Friend] -> String
findFriend (0,1) [((1,1), "Jane"), ((2,5), "Jim"), ((0, -1), "Tom")]
⇒ "Jane"
findFriend (0, 0) [((0,1), "Peter"), ((1,0), "Mary")] ⇒  either
findFriend (0,0) [] ⇒ error "Nobody exists to be your friend"
```

7. (a) Define a function `mulTable` that returns the multiplication table for numbers 1 to `n`, where `n` is strictly larger than or equal to 1.

```
mulTable :: Int -> [[Int]]
mulTable 4 ⇒
[[1, 2, 3, 4], [2, 4, 6, 4], [3, 6, 9, 12], [4, 8, 12, 16]]
```

```
mulTable 1 ⇒ [[1]]
mulTable (-3) ⇒ error "Given number lesser than 1"
```

(b) Define a function `leftpad` that converts a given element to a string and left-pads it with spaces up to a given length.

```
leftpad :: Show a => Int -> a -> String
leftpad 3 1 ⇒ " 1"
leftpad 3 18 ⇒ " 18"
leftpad 2 78 ⇒ "78"
leftpad 2 102 ⇒ error "102 does not fit into 2 characters"
leftpad (-7) any ⇒ error "Cannot pad to negative length"
```

(c) Define an action (a "function" with side effects) named `prettyTable` that pretty-prints a list of lists (a list of rows of elements) in the form of a correctly-aligned table. Place a single empty space character between all the columns. Make all columns the same width. (Hint: See what `Data.List.intercalate` does.)

```
prettyTable :: Show a => [[a]] -> IO ()
ghci> prettyTable [[1, 2, 3], [18, 12, 293]]
  1   2   3
 18  12 293
ghci> prettyTable [[7, 12, 0]]
 7 12  0
ghci> prettyTable [[1, 17, 170], [300, 400, 5000], [7, 13, 15]]
   1   17  170
 300  400 5000
   7   13   15
```

8. Create a simple utility for working with CSV (comma-separated value) files. These are simple textual files where fields are delimited with a character (usually a comma or a semicolon). We will use the types provided below to define a CSV document and implement basic operations over them. We require that the CSV document is well-formed, i.e., that it contains an equal number of fields per row.

```
type Separator = String
type Document = String
type CSV = [Entry]
type Entry = [Field]
type Field = String
doc = "John;Doe;15\nTom;Sawyer;12\nAnnie;Blake;20"
brokenDoc = "One;Two\nThree;Four;Five"
```

(a) Define a function `parseCSV` that takes a separator and a string representing a CSV document and returns a CSV representation of the document. (Hint: Think about how you can convert a problem of splitting on an arbitrary delimited to a problem of splitting on whitespace with `words`).

```
parseCSV :: String -> Document -> CSV
parseCSV ";" doc ⇒
[["John", "Doe", "15"],
["Tom", "Sawyer", "12"],
["Annie","Blake", "20"]]
```

```
parseCSV "," doc
⇒ error "The character ‘,’ does not occur in the text"
parseCSV brokenDoc ";" ⇒ error "The CSV file is not well-formed"
```

(b) Define a function `showCSV` that takes a separator and a CSV representation of a document and creates a CSV string from it.

```
showCSV :: Separator -> CSV -> Document
csv = parseCSV ";" doc
showCSV ";" doc ⇒ "John;Doe;15\nTom;Sawyer;12\nAnnie;Blake;20"
showCSV ";" [["One"], ["Two", "Three"]]
⇒ error "The CSV file is not well-formed"
```

(c) Define a function `colFields` that takes a CSV document and a field number and returns a list of fields in that column.

```
colFields :: Int -> CSV -> [Field]
colFields 1 csv ⇒ ["Doe", "Sawyer", "Blake"]
colFields 3 csv ⇒ error "There is no column 3 in the CSV document"
```

(d) Parsing CSV without the ability to access CSV files is not very useful. Define an IO function (an action) `readCSV` that takes a file path and a separator and returns the CSV representation of the file (wrapped due to impurity). In the examples below we assume the two strings, `doc` and `brokenDoc`, were written into files `doc.csv` and `brokenDoc.csv`, respectively.

Because reading from a file is an impure IO action, we have to introduce a special "wrapper" type around the returned value, IO. This shows up as an `IO a` type, where `a` is a type variable standing for any type. We have dealt with IO before, just without knowing it – see types of `getLine` or `readFile`. Values returned by `readCSV` can be accessed in a similar way in impure code, using `<-`.

```
readCSV :: Separator -> FilePath -> IO CSV
readCSV ";" "doc.csv" ⇒
[["John", "Doe", "15"],
["Tom", "Sawyer", "12"],
["Annie","Blake", "20"]]
readCSV "," "doc.csv"
⇒ error "The character ‘,’ does not occur in the text"
readCSV ";" "brokenDoc.csv"
⇒ error "The CSV file is not well-formed"
```

(e) Define a function `writeCSV` that takes a separator, a file path, and a CSV document and writes the document into a file.

The return type of `writeCSV` is a special case of IO – we need to wrap an impure action, but do not actually have to return anything when writing. Thus, we introduce `()`, or the unit type, which holds no information (consider it a 0-tuple).

```
writeCSV :: Separator -> FilePath -> CSV -> IO ()
writeCSV ";" "test.csv" [["1"], ["2", "3"]]
⇒ error "The CSV file is not well-formed"
writeCSV ";" "doc.csv" (parseCSV document ";")
```

```
$ cat doc.csv
John,Doe,15
Tom,Sawyer,12
Annie,Blake,20
```

(f) Place the implementation in a separate module `CSVUtils`, in a file called `CSVUtils.hs`. Expose only the five functions and five types from previous subtasks, while any helper functions should remain hidden. Submit the file with the homework assignment.

9. Implement simple versions of the following unix command-line utilities:

   (a) The `wc` (wordcount) utility, printing the number of lines, words and characters within a file. (Note: `FilePath` is a type alias for `String`, defined in the `Prelude`.)

   ```
   wc :: FilePath -> IO ()
   $ echo "Tiny sample file\ncontaining a haiku\nvery impressive" \
   > test.txt
   ghci> wc "test.txt"
   3 8 52
   ```

   (b) The `paste` utility, pairing up lines from two input files and printing them joined with a tab character.

   ```
   paste :: FilePath -> FilePath -> IO ()
   $ echo "not\nisn't\nat all, really." > ps.txt
   ghci> paste "test.txt" "ps.txt"
   Tiny sample file     not
   containing a haiku    isn't
   very impressive     at all, really.
   ```

   (c) The `cut` utility, taking a delimiter, index and a file, then cuts out a portion of each line and writes them out to the standard output. The parts are separated by the given delimiter and the printed part is determined by the given index. Column numbering starts from 1.

   ```
   cut :: String -> Int -> FilePath -> IO ()
   $ echo "1#Marko#99.85\n2#Iva#99.30\n3#Pasko#90.00" > scores.txt
   ghci> cut "#" 2 "scores.txt"
   Marko
   Iva
   Pasko
   ```

Corrections

v1.1 – Fixed a bunch of faulty examples in 8 a), b), d)
v1.2 – Changed examples `String -> Char` for 'o' and 'f' in 4 a)