Programming in Haskell – Homework Assignment 8

UNIZG FER, 2013/2014

Handed out: January 06, 2014. Due: January 13, 2013 at 23:59

Note: Define each function with the exact name and type specified. You can (and in most cases 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, and return the exact error message specified. Problems marked with a star ($\star$) are optional.

1. Define a class of types convertible to the JSON format. Define all the instances necessary to evaluate the following examples.

   ```
   json :: JSON a => a -> String
   json (3::Int) ⇒ "3"
   json (3.2::Double) ⇒ "3.2"
   json True ⇒ "true"
   json False ⇒ "false"
   json "haskell" ⇒ "\"haskell\""
   json [2::Int, 4] ⇒ "[2,4]"
   json [True, False] ⇒ "[true, false]"
   json (words "haskell says") ⇒ "[\"haskell\",\"says\"]"
   json (fromList [("a",1),(\b",2)]::Data.Map.Map String Int)
   ⇒ "{\"a\":1,\"b\":2}"
   ```

   To accomplish this, you'll need to use Haskell extensions giving you the option of using instance declarations more advanced than those allowed by the Haskell98 language standard. To enable these extensions, put the following lines at the very top of your Haskell module:

   ```
   {-# LANGUAGE FlexibleInstances #-}
   {-# LANGUAGE OverlappingInstances #-}
   {-# LANGUAGE TypeSynonymInstances #-}
   ```

   You can read more about what these extensions mean by browsing the GHC type class extensions documentation.

2. A binary tree can be defined in a number of ways. For example, a tree might store the values in the inner node, the leaf nodes, or both. Moreover, an inner node may be defined so that it has to have both children nodes or it may be defined so that one of the child nodes is missing (a `Nothing`). Despite these differences, we'd like to be able to perform some common operations on binary trees. We can use a type class to provide a generic inference to binary trees. Let's define the type class as follows:

```
class BinaryTree t where
  leftTree  :: t a -> Maybe (t a)
  rightTree :: t a -> Maybe (t a)
  rootValue :: t a -> Maybe a
  leaf :: a -> t a
  node :: a -> (t a) -> (t a) -> t a
```

The `leftTree`, `rightTree`, and `rootValue` should return the left tree, the right tree, and the value of a node, respectively, or `Nothing` if these don't exist. Function `leaf` constructs a leaf node that stores a value. For trees that don't store a value in a leaf, the function should just return a value without evaluating the given falue (i.e., for such trees `leaf` $\perp$ should work). Similarly, the node function takes a value, the left and the right subtree, and returns a node. For trees that don't store a value in a node, the first argument should not be evaluated.

(a) Let's define two types of two binary trees:

```
data Tree1 a = Leaf1 a | Node1 a (Tree1 a) (Tree1 a)
data Tree2 a = Leaf2 a | Node2 (Tree2 a) (Tree2 a)
```

For example:

```
t1 :: Tree1 Int
t1 = Node1 1 (Node1 2 (Leaf1 3) (Leaf1 4)) (Node1 5 (Leaf1 6) (Leaf1 7))
t2 :: Tree2 Int
t2 = Node2 (Leaf2 1) (Node2 (Leaf2 2) (Leaf2 3))
```

Now, make `Tree1` and `Tree2` instances of the `BinaryTree` type class.

(b) Define a function to check whether a binary tree is a leaf:

```
isLeaf :: BinaryTree t => t a -> Bool
isLeaf t1 ⇒ False
isLeaf (fromJust $ leftTree t2) ⇒ True
```

(c) Define a function to check whether a binary tree has both children:

```
isBranching :: BinaryTree t => t a -> Bool
isBranching t1 ⇒ True
isBranching $ Leaf1 5 ⇒ False
```

(d) Define a preorder traversal function:

```
preorder :: BinaryTree t => t a -> [a]
preorder t1 ⇒ [1,2,3,4,5,6,7]
preorder t2 ⇒ [1,2,3]
```

(e) Define an inorder traversal function:

```
inorder :: BinaryTree t => t a -> [a]
inorder t1 ⇒ [3,2,4,1,6,5,7]
inorder t2 ⇒ [1,2,3]
```

(f) Define a postorder traversal function:

```
postorder :: BinaryTree t => t a -> [a]
postorder t1 ⇒ [3,4,2,6,7,5,1]
postorder t2 ⇒ [1,2,3]
```

(g) ⋆ Define a function that trims the leaves of a binary tree. Do this by converting every node with two children (i.e., a branching node) into a leaf. The value of the node should be retained when it's converted into a leaf. If the leaves store

values but the nodes don't (as in the case of `Tree2`), then take the value of the left leaf as the value of the new leaf.

```
trimLeaves :: BinaryTree t => t a -> t a
trimLeaves t1 ⇒ Node1 1 (Leaf1 2) (Leaf1 5)
trimLeaves t2 ⇒ Node2 (Leaf2 1) (Leaf2 2)
```

3. Vectors can be implemented in a number of ways. Abstractly speaking, a vector is a list of real numbers, with one number for each dimension. If the vector is dense (i.e., many components are non-zero), then it might make sense to represent it as an array or even as a list. If the vector is sparse, then it makes more sense to represent it using a `Data.Map.Map`, which stores only the values of non-zero dimensions. Regardless of how a vector is implemented, it should support some common functionality. This again means that we can define an API for a vector via a type class. Let's define it like this:

```
class Vector v where
  empty         :: v
  zipWith       :: (Double -> Double -> Double) -> v -> v -> v
  add           :: v -> v -> v
  dot           :: v -> v -> Double
  fromList      :: [Double] -> v
  toList        :: v -> [Double]
  fromAssocList :: [(Int,Double)] -> v
  toAssocList   :: v -> [(Int,Double)]
  nonzeroes     :: v -> [Double]
```

The `zipWith` functions combines the corresponding components of two vectors using the provided combining function. The `add` and `dot` function compute vector addition and scalar product of two vectors. `fromList` and `toList` convert from/to a list of components. Likewise, `fromAssocList`, and `toAssocList` convert from/to a list of `(dimension,component)` pairs. Function `nonzeroes` returns the non-zero components.

   (a) Provide default implementations of `add`, `dot`, and `nonzeroes`. The benefit of this is that `empty`, `zipWith`, and the four converting functions will become the minimum complete definition.

   (b) Define a vector implemented as a list:
   ```
   data ListVector = LV [Double]
   ```
   Make this type an instance of the Vector type class. E.g.:
   ```
   v1 = fromList [1,2,0] :: ListVector
   v2 = fromList [0,0.5,0] :: ListVector
   v1 `add` v2 ⇒ LV [1.0,2.5,0.0]
   v1 `dot` v2 ⇒ 1.0
   v1 `dot` empty ⇒ 0.0
   ```

   (c) Define a sparse vector implementation using `Data.Map`:
   ```
   data SparseVector = SV (Data.Map.Map Int Double)
   v1 = fromList [1,2,0] :: SparseVector
   v2 = fromList [0,0.5,0] :: SparseVector
   v1 `add` v2 ⇒ SV (fromList [(0,1.0),(1,2.5),(2,0.0)])
   ```

```
v1 `dot` v2 ⇒ 1.0
v1 `dot` empty ⇒ 0.0
```

(d) Define the function to compute the norm of the vector:

```
norm :: Vector v => v -> Double
norm v1 ⇒ 2.23606797749979
norm (empty::SparseVector) ⇒ 0
```

(e) Define a function to compute the cosine similarity between two vectors:

```
cosine :: Vector v => v -> v -> Double
cosine v1 v2 ⇒ 0.8944271909999159
cosine v1 empty ⇒ 0
```

4. Often we need to be able to measure the similarity between data. If we have a measure of similarity between two data structures, we can use this measure to search for a data structure among a set of data structures. In other words, given a data structure we're looking for, we can measure how similar this structure is to other structures, and construct a search ranking list based on this. This idea originates in information retrieval, where documents are represented as vectors of words: each word is one dimension, and the component value equals the number this word occurs in the document. To compare two documents, one simply computes the cosine similarity between the corresponding vectors. If the documents are very similar, their vectors will also be similar to each other and the cosine will be close to 1. We'll build on this idea, but we want a mechanism that will work on all data structures, not just textual data. As you can guess, we'll be using type classes to achieve this.

   To be able to compare two data structures as vectors, we first need to convert a data structure into a vector. Let's begin by defining a type class for vectorizable types:

```
class Vectorizable a where
  vectorize :: a -> SparseVector
```

   Note that we're using a sparse vector here from problem 3. This is deliberate, because in general our vectors will live in a highly dimensional space, and we expect many of the dimensions to be zero.

   (a) Define a function that counts the elements in a list (the counts are `Doubles`):

```
counts :: Ord a => [a] -> [(a,Int)]
counts "ubuntu" ⇒ [('b',1),('n',1),('t',1),('u',3)]
```

   (b) Define `[Char]` as an instance of `Vectorizable`.

```
instance Vectorizable [Char] where ...
```

   The vector dimensions are the ASCII codes of the characters, and the values are the number of character occurrences. (Hint: Use `fromAssocList` from problem 3.)

```
vectorize "ubuntu"
⇒ SV (fromList [(98,1.0),(110,1.0),(116,1.0),(117,3.0)])
```

   (c) Define a `Vectorizable` instance for `Tree1 Int` from problem 2. (Hint: Flatten the tree into a list using a tree traversal you've implemented as part of problem 2).

```
vectorize t1
⇒ SV (fromList [(1,1.0),(2,1.0),(3,1.0),(4,1.0),(5,1.0),(6,1.0),(7,1.0)])
```

```
vectorize (Node1 1 (Leaf1 6) (Leaf1 7))
⇒ SV (fromList [(1,1.0),(6,1.0),(7,1.0)])
```

(d) Finally, define a function to search for a given data structure in a list of data structures. First vectorize all data structures, then compute the cosine similarity between the given data structure and the data structures in the list, and then produce a ranking list.

```
t3 = Node1 5 (Leaf1 1) (Leaf1 5) :: Tree1 Int
t4 = Leaf1 5 :: Tree1 Int
search :: Vectorizable a => a -> [a] -> [a]
search t3 [t1,t3,t4]
⇒ [Node1 5 (Leaf1 1) (Leaf1 5),Leaf1 5,Node1 1 (Node1 2 (Leaf1 3)
(Leaf1 4)) (Node1 5 (Leaf1 6) (Leaf1 7))]
search "fox" ["ubuntu","xerox","firefox"] ⇒ ["firefox","xerox","ubuntu"]
```

(e) ⋆ You might be unhappy with the fact that we need to define specific `Vectorizable` instances for each new data type. A more generic approach would be to define the type class as follows:

```
class Vectorizable2 t a where
  vectorize2 :: t a -> SparseVector
```

For this you will need to enable the `MultiParamTypeClasses` and `FlexibleInstances` extensions in GHC (read here how to do that). You can now define a rather generic instance of the `Vectorizable` type class as follows:

```
instance (Foldable t, Ix a, Bounded a) => Vectorizable2 t a where
  vectorize2 = ...
```

This instance will work on type constructors that are foldable and that store values that are both indexable (class `Ix` from Data.Ix) and bounded (class `Bounded`). We need the structure to be foldable so that we can convert it in a list using `Data.Fold.toList`. Furthermore, we need the values to be indexable so that we can convert them to unique vector dimensions. For this, we can use the function index with (`minBound,maxBound`) as arguments. For example:

```
index (minBound,maxBound) (5::Int) ⇒ -9223372036854775803
```

Now, finish the above instance definition. This is what you should get:

```
vectorize2 "ubuntu"
⇒ SV (fromList [(98,1.0),(110,1.0),(116,1.0),(117,3.0)])
vectorize2 t3
⇒ SV (fromList [(-9223372036854775807,1.0),(-9223372036854775803,2.0)])
```

5. In this problem we'll be implementing a text justification function for a monospaced terminal output. The alignment is achieved by inserting blanks and hyphentaing the words. For example, given a text

```
text = "He who controls the past controls the future.  He who controls
the present controls the past."
```

we want to be able to align it like this (to a width of, say, 15 columns):

```
He who controls
the  past cont-
rols  the futu-
re. He  who co-
```

```
ntrols the pre-
sent   controls
the past.
```

This problem is more intricate than it seems, so we're going to approach it in a sophisticated way. We'll design the solution step by step, defining a series of small, well-defined functions as we go, which is considered to be good functional programming style.

We'll represent a string as a list of tokens. We'll refer to a list of tokens as a `Line`. Tokens may be words, hyphenated words, or inserted blanks.

```
type Line = [Token]
data Token = Word String | Blank | HypWord String deriving (Eq,Show)
```

(a) Define a function to convert a string into a line. (You may assume that the input contains no hyphenated words! This, of course, is a gross oversimplification.)

```
string2line :: String -> Line
string2line text ⇒ [Word "He",Word "who",Word "controls", ...]
```

(b) Define a function to convert back from a Line into a string:

```
line2string :: Line -> String
line2string (string2line text) == text ⇒ True
```

Note: A hyphenated word should be displayed with a trailing hyphen. For example: `line2string [Word "He",Word "who",HypWord "cont",Word "rols"] ⇒ "He who cont- rols"`

(c) Define a function to compute the length of a token:

```
tokenLength :: Token -> Int
tokenLength (Word "He") ⇒ 2
tokenLength (HypWord "cont") ⇒ 5
tokenLength (Blank) ⇒ 1
```

(d) Define a function to compute the length of a line:

```
lineLength [Word "He",Word "who",Word "controls"] ⇒ 15
lineLength [Word "He",Word "who",HypWord "con"] ⇒ 11
```

(e) Define a function to break a line so that it's not longer than a given width. The function returns a pair of Lines: first line is the broken-up line, and the second line is its continuation.

```
breakLine :: Int -> Line -> (Line,Line)
breakLine 1 [Word "He",Word "who",Word "controls"]
⇒ ([],[Word "He",Word "who",Word "controls"])
breakLine 2 [Word "He",Word "who",Word "controls"]
⇒ ([Word "He"],[Word "who",Word "controls"]
breakLine 5 [Word "He",Word "who",Word "controls"]
⇒ ([Word "He"],[Word "who",Word "controls"])
breakLine 6 [Word "He",Word "who",Word "controls"]
⇒ ([Word "He",Word "who"],[Word "controls"])
breakLine 100 [Word "He",Word "who",Word "controls"]
⇒ ([Word "He",Word "who",Word "controls"],[])
breakLine _ [] ⇒ ([],[])
```

(f) Define a helper function `merger` that works like this:

```
mergers ["co","nt","ro","ls"]
⇒ [("co","ntrols"),("cont","rols"),("contro","ls")]
mergers ["co","nt"] ⇒ [("co","nt")]
mergers ["co"] ⇒ []
```

(g) To be able to align the lines nicely, we have to able to hyphenate long words. Although there are rules for hyphenation for each language, we will take a simpler approach here and assume that there is a list of words and their proper hyphenation. We'll implement this using a "hyphenation map":

```
type HypMap = Data.Map.Map String [String]
```

For example:

```
enHyp :: HypMap
enHyp = Data.Map.fromList [
  ("controls",["co","nt","ro","ls"]),
  ("future",["fu","tu","re"]),\\
  ("present",["pre","se","nt"])]
```

Now, define a hyphenate function that breaks up a token in all possible ways defined by a hyphenation map. (Hint: use the `mergers` function defined previously.)

```
hyphenate :: HypMap -> Token -> [(Token,Token)]
hyphenate enHyp (Word "controls")
⇒ [(HypWord "co",Word "ntrols"),(HypWord "cont",Word "rols"),(HypWord
"contro",Word "ls")]
hyphenate enHyp (Word "firefox") ⇒ [] -- not in the map
```

Note: If the word has trailing punctuation, you need to remove it first, hyphenate the word, and then concatenate back the punctuation to the second part of the word. For example:

```
hyphenate enHyp (Word "future.")
⇒ [(HypWord "fu",Word "ture."),(HypWord "futu",Word "re.")]
hyphenate enHyp (Word "future...")
⇒ [(HypWord "fu",Word "ture..."),(HypWord "futu",Word "re...")]
```

(h) You can now define a function that breaks a line into different ways, by trying to hyphenate the last word in different ways. The broken-up line should not be longer than a given width.

```
lineBreaks :: HypMap -> Int -> Line -> [(Line,Line)]
lineBreaks enHyp 12 [Word "He",Word "who",Word "controls"]
⇒ [([Word "He",Word "who"],[Word "controls"]),([Word "He",Word "who",HypWord
"co"],[Word "ntrols"]),([Word "He",Word "who",HypWord "cont"],[Word
"rols"])]
lineBreaks enHyp 12 [Word "He"] ⇒ [([Word "He"],[])]
```

(i) Define a helper function `insertions` that does the following:

```
insertions :: a -> [a] -> [[a]]
insertions 'x' "abcd" ⇒ ["xabcd","axbcd","abxcd","abcxd","abcdx"]
```

(j) Using the `insertions` function, define a function that inserts a given number of blanks into the line and returns all possible insertions. Do not insert blanks at the very beginning and the end of the line! (Hint: repeatedly use the `insertions` function defined above.)

```
insertBlanks :: Int -> Line -> [Line]
insertBlanks 2 [Word "He",Word "who",Word "controls"]
⇒ [[Word "He",Blank,Blank,Word "who",Word "controls"],[Word "He",Blank,Word
"who",Blank,Word "controls"],[Word "He",Word "who",Blank,Blank,Word
"controls"]]
```

(k) We want the inserted blanks to be spread out and evenly distributed. Define a function that computes the distances between inserted blanks, counted in number of in-between tokens. You should also take into account the distances to the start and end of line.

```
blankDistances :: Line -> [Int]
blankDistances [Word "He",Blank,Blank,Word "who",Word "controls"] ⇒
[1,0,2]
map blankDistances $ insertBlanks 2 [Word "He",Word "who",Word "controls"]
⇒ [[1,0,2],[1,1,1],[2,0,1]] blankDistances [Word "He"] ⇒ []
```

(l) Define two helper functions to compute the average and the variance of values in a list:

```
avg :: Real a => [a] -> Double
var [1,0,2] ⇒ 0.6666666666666666
var [] ⇒ 0
```

(m) We're now capable of producing a number of candidate line splits, but we now need a mechanism to decide how good each line split is. To this end, we'll be defining a scoring function for a line break. The scoring function will be based on costs, define via a data structure, as follows:

```
data Costs = Costs {
  blankCost :: Double,
  blankProxCost :: Double,
  blankUnevenCost :: Double,
  hypCost :: Double } deriving (Eq,Show)
```

The `blankCost` is the cost of introducing a blank, the `blankProxCost` is the cost of having blanks close to each other, the `blankUnevenCost` is the cost of having blanks spread unevenly, and the `hypCost` is the cost of hyphenating the last word in a line. The default costs are:

```
defaultCosts = Costs {
  blankCost = 1,
  blankProxCost = 1,
  blankUnevenCost = 0.5,
  hypCost = 0.5 }
```

Now, define a function to score a line based on a given cost. The total cost is computed simply as the sum of the individual costs. The `blankProxCost` equals the number of tokens minus the average blank distances if the line has blanks, and zero otherwise. The blank uneven cost is computed simply as the variance of blank distances.

```
lineBadness defaultCosts [Word "He",Word "who",Word "controls"] ⇒
0.0
lineBadness defaultCosts [Word "He",Blank,Word "who",Word "controls"]
⇒ 3.625
lineBadness defaultCosts [Word "He",Blank,Word "who",HypWord "cont"]
```

```
⇒ 4.125
lineBadness defaultCosts [Word "He",Blank,Word "who",Blank,Word "controls"]
⇒ 6.0
lineBadness defaultCosts [Word "He",Blank,Blank,Word "who",Word "controls"]
⇒ 6.33
```

(n) We're getting closer to the solution. Define a function that computes the best line break given the costs, the hyphenation map, and the maximum line width. The best line break is the one that minimized the line badness score. If line break is not possible (because one word is longer than a specified width and cannot be hyphenated), return `Nothing`.

```
bestLineBreak :: Costs -> HypMap -> Int -> Line -> Maybe (Line,Line)
bestLineBreak defaultCosts enHyp 8 [Word "He",Word "who",Word "controls"]
⇒ Just ([Word "He",Blank,Blank,Word "who"],[Word "controls"])
bestLineBreak defaultCosts enHyp 12 [Word "He",Word "who",Word "controls"]
⇒ Just ([Word "He",Word "who",HypWord "cont"],[Word "rols"])
bestLineBreak defaultCosts enHyp 1 [Word "He",Word "who",Word "controls"]
⇒ Nothing
```

(o) Finally, define a function that justifies the line. You need to apply `bestLineBreak` iterativelly on each line. If a word gets hyphenated, you need to add the rest of it to the next line before you apply bestLineBreak again. Repeat that until reaching the last line, but do not justify the last line (justifiying the last line would look ugly). If any of the lines cannot be broken up, then return the line as it is.

```
justifyLine :: Costs -> HypMap -> Int -> Line -> [Line]
justifyLine defaultCosts enHyp 8 (string2line text)
⇒ [[Word "He",Blank,Blank,Word "who"],[Word "controls"],[Word "the",Word
"past"],[Word "controls"],[Word "the"],...]
```

(p) As a finishing touch, define a function that takes in a string and returns a list of strings justified to a specified width. If justification is not possible, return the original string.

```
justifyText defaultCosts enHyp 8 text ⇒ ["He who","controls","the
past",...]
```

Typing in

```
putStr . unlines . justifyText (defaultCosts) enHyp 15 text
```

should give you exactly the example we started with.

You can now play around with different parameter settings (e.g., changing the different costs) to see how this affects the output. If you try a larger width, you'll notice that the program slows down. This is because our implementation is a bit naive and considers all possible splits. As the width of the line increases, the number of possible splits increases exponentially. This could be remedied by limiting the number of splits to consider using a suitable heuristics. There are a few other things that could be improved as well, such as allowing hyphenated words as input, calculating the costs of hyphenation based on where the word is hyphenated, redefining `justifyLine` so that it returns a `Maybe [Line]`, etc. We encourage you to try to implement some of these extensions.

Corrections

```
v2:  1: add language extensions
    4d: add type signature
     2: fix pre/in/postorder examples for t2
    2g: fix bad definition of branching node
     3: add explicit signatures, replace Weight with Double
   all: replace M with Data.Map
    5g: add multichar punctuation example

                                  (many thanks to Luka!)
```