universität innsbruck

Functional Programming	WS 2023/2024	LVA 703025
Exercise Sheet 8, 10 points		Deadline: Tuesday, December 05, 2023, 8pm

- Mark your completed exercises in the OLAT course of the PS.
- You can use a template .hs file that is provided on the proseminar page.
- Upload your modified .hs file in OLAT.
- Your .hs file must be compilable with ghci.

Exercise 1 *Vectors and Matrices via Lists* **5 p.**

In this exercise, we will consider a list-based implementation of vectors and matrices:

```
type Vector a = [a]
type Matrix a = [[a]]
```

A vector $\vec{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$ of dimension $n > 0$ is represented as list $[v_1, \ldots, v_n]$, and a matrix $A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$ with dimensions $m > 0$ and $n > 0$ is represented as a list of lists $[[a_{11}, \ldots, a_{1n}], \ldots [a_{m1}, \ldots, a_{mn}]]$.

In the template file `template_08.hs` we have implemented five standard vector and matrix operations. These implementations are correct as their definition mimics the mathematical definition very closely. However, they are slow because of their usage of list indices.

Your task is to provide more efficient implementations of the five operations defined below. Explicit recursion is usually not required, except for the `transpose` function – instead, use higher-order functions on lists whenever appropriate. You may always assume that the dimensions of vectors or matrices fit together.

1. The negation of a vector $\vec{v}$ is defined as the vector $\begin{pmatrix} -v_1 \\ \vdots \\ -v_n \end{pmatrix}$. (0.5 points)
2. The addition of two vectors $\vec{v}$ and $\vec{w}$ of the same dimension n is defined as the vector $\begin{pmatrix} v_1 + w_1 \\ \vdots \\ v_n + w_n \end{pmatrix}$. (1 point)
3. The scalar product of two vectors $\vec{v}$ and vector $\vec{w}$ of dimension n is defined as $\sum_{1 \leq i \leq n} v_i \cdot w_i$. (1 point)
4. The transposed matrix of A is the matrix $\begin{pmatrix} a_{11} & \cdots & a_{m1} \\ \vdots & \ddots & \vdots \\ a_{1n} & \cdots & a_{mn} \end{pmatrix}$. (1 point)
5. The matrix product of two matrices A and B is defined as the matrix C where c_{ij} is the scalar product of row i of A and column j of B. (1.5 points)

Solution 1

```haskell
type Vector a = [a]
type Matrix a = [[a]]

-- Task 1.1
negateVec :: Num a => Vector a -> Vector a
negateVec = map negate

-- Task 1.2
vecAdd :: Num a => Vector a -> Vector a -> Vector a
vecAdd = zipWith (+)

-- Task 1.3
scalarProduct :: Num a => Vector a -> Vector a -> a
scalarProduct v w = sum (zipWith (*) v w)

-- Task 1.4
transpose :: Matrix a -> Matrix a
transpose ([] : _) = []
transpose rows = map head rows : transpose (map tail rows)

-- tricky alternative
-- transpose a = foldr (zipWith (:)) (replicate (length a) []) a

-- Task 1.5
matMult :: Num a => Matrix a -> Matrix a -> Matrix a
matMult a b = let tb = transpose b in [[r `scalarProduct` c | c <- tb] | r <- a]
```

Exercise 2 *Fold Functions* **5 p.**

1. What is the difference between `foldr` and `foldl`? Evaluate the expressions `foldr (-) 0 [1..4]` and `foldl (-) 0 [1..4]` in GHCI and explain your results. (0.5 points)

2. The file `template_08.hs` contains a recursive implementation of the insertion sort[1] sorting algorithm. This algorithm works by iteratively inserting elements at the correct position into an already sorted list. Implement insertion sort using `foldr`. (1 point)

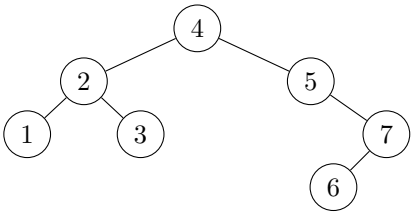
   Hint: The functions `span` or `takeWhile` and `dropWhile` might be useful.
   Example: `insertionSortFold [3,1,2,1,6,12] == [1,1,2,3,6,12]`

3. In this question, we will implement and use a fold function for a binary tree type defined as follows.

   ```haskell
   data Tree a = Leaf | Node (Tree a) a (Tree a)

   exampleTree = let n x = Node Leaf x Leaf in
     Node (Node (n 1) 2 (n 3)) 4 (Node Leaf 5 (Node (n 6) 7 Leaf))
   ```

   Here, `exampleTree` represents the following tree:

   (i) Implement a function `foldt :: (b -> a -> b -> b) -> b -> Tree a -> b` where `foldt f e t` replaces each `Node` in `t` by `f` and each `Leaf` in `t` by `e`. (1 point)
   Example: `foldt (\ accL x accR -> accL + x + accR) 0 exampleTree == 28`

[1] https://en.wikipedia.org/wiki/Insertion_sort

(ii) Implement the following two functions on trees using `foldt`: (0.5 points)

- `height` should compute the height of a tree, e.g., `height exampleTree == 4`
- `flatten` should flatten a tree into a list depth-first from left to right, e.g., `flatten exampleTree == [1..7]`

(iii) Implement the following two functions on trees using `foldt`: (1 point)

- `mirror` should swap the left and right subtree of each node.
- `mapTree` should apply a function to each value in a tree but preserve the tree shape. For instance, `mapTree (* 2) exampleTree` is the tree with the same structure as `exampleTree` but where every number is doubled.

(iv) Implement a function `showTree :: Show a => Tree a -> String` using `foldt` that shows each node of a tree in a separate line, such that `putStr $ showTree $ exampleTree` results in the following output:

```
      1
    /
  2
    \
      3
/
4
 \
   5
    \
        6
      /
    7
```

Here, every level of the tree is indented by three additional spaces and if a node has a non-empty left- or right subtree, then "/" or "\" should be output above or below the node. (1 point)

Solution 2

```haskell
-- Task 2.1
{-
Both foldr and foldl apply a function recursively to a list of elements
starting from an initial value. The function foldr applies the function
recursively from right to left, while foldl applies it from left to right.
Therefore:
foldr (-) 0 [1..4] = 1-(2-(3-(4-0))) = -2
foldl (-) 0 [1..4] = (((0-1)-2)-3)-4 = -10
-}

-- Task 2.2
insertionSortFold :: Ord a => [a] -> [a]
insertionSortFold = foldr (\e acc -> let (xs, ys) = span (<= e) acc in xs ++ [e] ++ ys) []

-- Task 2.3
data Tree a = Leaf | Node (Tree a) a (Tree a) deriving Show

foldt :: (b -> a -> b -> b) -> b -> Tree a -> b
foldt f e Leaf = e
foldt f e (Node l x r) = f (foldt f e l) x (foldt f e r)

height :: Tree a -> Int
height = foldt (\ l _ r -> 1 + max l r) 0

flatten :: Tree a -> [a]
flatten = foldt (\ l x r -> l ++ [x] ++ r) []

mirror :: Tree a -> Tree a
mirror = foldt (\ l x r -> Node r x l) Leaf

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f = foldt (\ l x r -> Node l (f x) r) Leaf

showTree :: Show a => Tree a -> String
showTree = unlines . foldt (\ l x r ->
   map ("   " ++) l ++
   [" /" | not (null l)] ++
   [show x] ++
   [" \\" | not (null r)] ++
   map ("   " ++) r) []
```