FUNCTIONAL PROGRAMMING MT2018

SHEET 4

GABRIEL MOISE

6.8

```
data Tree a = Fork (Tree a) a (Tree a) | Empty


insert :: Ord a => a -> Tree [a] -> Tree [a]
insert x Empty = Fork Empty [x] Empty
insert x (Fork left as right)
  | x < (as !! 0) = Fork (insert x left) as right
  | x == (as !! 0) = Fork left (x:as) right
  | otherwise = Fork left as (insert x right)
```

For insert, we first treat the limit case when we insert a value into an Empty tree, which returns Fork Empty [x] Empty, then we treat the 3 cases:

I. If x is smaller than one element of the list from the current node (I chose the first element because we know it always exists in a non-empty node and because we know that all elements from the list are the same, as we only used lists to avoid repetitions), then we continue with the procedure recursively in the left branch
II. If x is equal to the elements of the list from the node, we simply add x to the node
III. Otherwise, x will be greater than the elements of the list from the node, so we continue with the procedure recursively in the right branch.

```
flatten :: Tree [a] -> [a]
flatten Empty = []
flatten (Fork left xs right) = flatten left ++ xs ++ flatten right
```

First, we treat the case when we want to flatten an Empty tree: it returns the empty list as it adds nothing to our list. Now, in general, if we want to flatten a specific node, with a branch on the left (the left tree) and one to the right (the right tree), we do that recursively, by concatenating the flattening of the left tree, the list from the node and the flattening of the right tree.

```
bsort :: Ord a => [a] -> [a]
bsort = flatten . foldr insert Empty
```

The bsort function is flattening the list obtained from the binary search tree, which is obtained with foldr, by applying insert to every element of the argument of bsort (the list to be sorted).

7.1

```
addElement :: [a] -> [[a]] -> [[a]]
addElement xs yss = foldr (\x acc -> map (x:) yss ++ acc) [] xs
```

This function creates the list of all lists that could be obtained by concatenating one element from the list xs with one list from the list of lists yss.

```
cp :: [[a]] -> [[a]]
cp xss = foldr (\xs acc -> addElement xs acc) [[]] xss
```

Now, we apply addElement xs for every current element from acc, in order to create the result (at first each list from acc has 0 elements, then there are only lists with 1 element, then with 2 and so on).

7.2

```
cols :: [[a]] -> [[a]]
cols xss = foldr (\xs acc -> zipWith (:) xs acc) start xss
   where start = if (length xss == 0) then []
                          else take (maximum [length xs | xs <- xss]) (cycle [[]])
```

This function "zips with" (:) each row to the result kept in acc, which is at start the set of a number of empty lists equal to the maximum length of all the rows of xss, because that's the number of lists we will form with our cols function. So we basically form the lists by adding each element of each row to a list corresponding the column in which it is in the transpose of xss. The definition of start takes care of the case where we have cols [], as we will have start = [] and the function will return []. This gives cols the form of a foldr as foldr f start [] always returns the start value.

8.1

```
rjustify :: Int -> String -> String
rjustify x s
  | x < (length s) = drop (length s - x) s
  | otherwise = take (x-length s) (repeat ' ') ++ s
```

Here, if we need to write a word of more letters than x, we only return the **last** x letters of our word.

```
rjustify1 :: Int -> String -> String
rjustify1 x s
  | x < (length s) = take x s ++ "\n" ++ rjustify1 x (drop x s)
  | otherwise = take (x-length s) (repeat ' ') ++ s
```

Here, we create a function which in the case that x is less than the length of our word, we form the first line out of the first x letters of the word, and then we continue with the procedure recursively on the next line and with the rest of the word. However, the "\n" needs to be interpreted as a new line, so we use our function for the rjustify' function, which prints the result in the desired form.

```
rjustify' :: Int -> String -> IO ()
rjustify' x s = putStr (rjustify1 x s ++ "\n")

ljustify :: Int -> String -> String
ljustify x s
  | x < (length s) = take x s
  | otherwise = s ++ take (x-length s) (repeat ' ')
```

Now, we do the same thing as for rjustify, but if we are in the case that the word has more letters than x, we return the **first** x letters of the word as a result.

```
ljustify1 :: Int -> String -> String
ljustify1 x s
  | x < (length s) = take x s ++ "\n" ++ ljustify1 x (drop x s)
  | otherwise = s ++ take (x-length s) (repeat ' ')

ljustify' :: Int -> String -> IO ()
ljustify' x s = putStr (ljustify1 x s ++ "\n")
```

Here, the things are very similar to the rjustify1 and rjustify', the only difference from rjustify1 is the "otherwise" condition, which aligns the remaining part of the word to the left.

8.2

type Matrix a = [[a]]

1.

```
scale :: Num a => a -> Matrix a -> Matrix a
scale c = map(map (c*))
```

We apply the function map (c*) to each row of the matrix, which means that we apply the function (c*) to every element of the matrix, so basically we multiply every element of the matrix with the scalar c.

2.

```
dot :: Num a => [a] -> [a] -> a
dot xs ys = foldr (\(x,y) acc-> x*y+acc) 0 (zip xs ys)
```

Here, we used foldr to the list of pairs formed by the elements of the two initial lists (vectors) xs and ys, to calculate the products of the elements of the lists and add them to acc.

3.

```
add :: Num a => Matrix a -> Matrix a -> Matrix a
add = zipWith(zipWith (+))
```

Here we apply the zipWith (+) function to every "pair" of corresponding rows of the two matrices, so we apply the (+) function to every "pair" of corresponding elements of the two matrices, and therefore we simply add the matrices. Here, the term "pair" is used just to emphasize the correspondence of the elements/rows, and not as a pair of type (_,_) as we do not use them in this definition.

4.

```
mul:: Num a => Matrix a -> Matrix a -> Matrix a
mul xss yss = [map (dot xs) (cols yss) | xs <- xss]
```

We apply the function dot xs to each line of the transpose of yss, which is cols yss, meaning that for each element of the resulting matrix, let's say from row i and column j, we dot product the i^{th} row from xss with the j^{th} row from cols yss (or the j^{th} column from yss).

5.

```
maxList :: Show a => [a] -> Int
maxList = maximum.map (length.show)
```

This function creates the list of values which are needed as parameters for the list of words in order to justify each word correctly in each column. So, we will justify each word according to the biggest number (in length) that appears on that specific column. (length.show) is the function that, given a number (in our case, but in general the function needs a variable that is from the Show typeclass), returns the length of the String representation of the number, for example (length.show) 123 is 3 as the length of "123" is 3. We apply this function to each element from a row in order to get the lengths that we need and then we apply maximum to the list of lengths, in order to get the greatest length on that row.

```
rjustList :: Show a => [a] -> [String]
rjustList xs = map (rjustify (maxList xs)) (map show xs)
```

This function, given a list (row in our case) of elements (values from the matrix in our case), returns the list of words which are rjustified according to the maximum length of the elements from the list. We apply the rjustify (maxList xs) function to all the elements from xs, which are turned into Strings in map show xs, so the result is a list of Strings which are rjustified.

```
table :: Show a => Matrix a -> String

table = unlines.map unwords.cols.map rjustList.cols
```

The initial function was table xss = unlines(map unwords(cols(map rjustList(cols xss)))), but I rewrote it with function composition as it seems more natural and easy to read and comprehend.

First of all, we apply (map rjustList.cols) to the matrix, which means that we apply rjustList to each row of the transposed matrix (which is the column of the matrix, and the transpose here is cols), so we basically justify it according to the biggest element (in length) from that row. Then we apply cols to the matrix, to get it back to the original form, then we (map unwords) it, so we apply unwords to every row in order to create a list of Strings, one String for each row of the matrix, and then we apply unlines to put a "\n" between them, so when we call (putStr.table) of a matrix, we get the desired result printed each row on a line.