[20241003] INFOFP - Functioneel programmeren - 1 - USP

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur:	2 uur en 30 minuten
Aantal vragen:	4

[20241003] INFOFP - Functioneel programmeren - 1 - USP

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 4

1 This question will be about typing on an old fashioned phone keyboard, as pictured below

a. [3pt] Assume that you have a function

```
pressKeysChar :: Char -> (Int, Int)
```

that tells us which key to press and how many times to press it if we want to enter a character on a phone keyboard. For example,

```
> pressKeysChar 'e'
(3, 2)
> pressKeysChar 'z'
(9, 4)
```

Use 'pressKeysChar' to implement a function

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

that sends a String to the list of keys we should press to type the String on a phone keyboard. For example,

```
> pressKeys "quokka"
[7,7,8,8,6,6,6,5,5,5,5,2]
```

3 pt. **a.**

From now on, we will consider a more efficient way of typing called T9, in which you press each key just once to get a character and later choose from the different words that a number sequence can represent.
You are given the following function

```
letterCombs :: Int -> [Char]

letterCombs 2 = "abc"
letterCombs 3 = "def"
letterCombs 4 = "ghi"
letterCombs 5 = "jkl"
letterCombs 6 = "mno"
letterCombs 7 = "pqrs"
letterCombs 8 = "tuv"
letterCombs 9 = "wxyz"
```

that sends a key from the phone keyboard to the letters that it could represent.

b. [2pt] Assume that you have a function

```
cartesianProd :: [[a]] -> [[a]]
```

that takes a list of lists and produces the cartesian product: 'cartesianProd xs' consists of all those lists 'as' such that, for all 'n', 'as !! n' is an element of 'xs !! n'. For example

```
> cartesianProd [[1,2],[3,4,5], [6]]
[[1,3,6],[1,4,6],[1,5,6],[2,3,6],[2,4,6],[2,5,6]]
```

Use 'cartesianProd' to implement the function

```
allLetterCombs :: [Int] -> [[Char]]
```

that sends a list of numbers to the list of strings that it could represent. For example ,

```
> allLetterCombs [4,6]
["gm","gn","go","hm","hn","ho","im","in","io"]
```

Implement this function without using direct recursion.

`allLetterCombs =` **b.** .... (2 pt.)

c. [2pt] Assume that you have a function

```
subsets :: [a] -> [[a]]
```

that takes a list 'xs' without duplicates and returns the list that contains all, possibly non-contiguous ("niet aaneengesloten"), sublists of elements of 'xs', respecting the order in the original list. For example,

```
> subsets [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

Without using direct recursion, use subsets to define a function

```
typoLetterCombs :: [Int] -> [[Char]]
```

that takes a list of numbers and returns the list of all of the strings that someone intended to type in case the person typing might have accidentally inserted unintended numbers into the list. You do not need to worry about duplicates.

`typoLetterCombs =` **c.** .... (2 pt.)

For example

```
> typoLetterCombs [4,6]
["gm","g","m","","gn","g","n","","go","g","o","","hm","h","m","","hn","h","
n","","ho","h","o","","im","i","m","","in","i","n","","io","i","o",""]
```

d. [4pt] Using direct recursion, now implement 'cartesianProd'.

```
cartesianProd [] =  d. ........................................ (1 pt.)
cartesianProd (as : ass) = e. .................................. (3 pt.)
```

e. [4pt] Implement 'subsets' using a fold and without using direct recursion:

```
subsets = foldr f e where
  e = f. .................................................. (1 pt.)

  f a acc = g. ............................................ (3 pt.)
```

You may return the sublists in any order you desire. The solution must not contain duplicate subsets.

2 Please determine the types of the following expressions or show that they are ill-typed. Please write down all reasoning steps, as they are at least as important as the final answers.
You may use that

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```

a. [6pt]

```
scanl map
```

6 pt. **a.**

b. [5pt]

```
map scanl
```

5 pt. **b.**

3 Consider the following type modeling binary trees:

```
data Tree a = Leaf a
            | Node (Tree a) a (Tree a)
            deriving (Show,Eq)
```

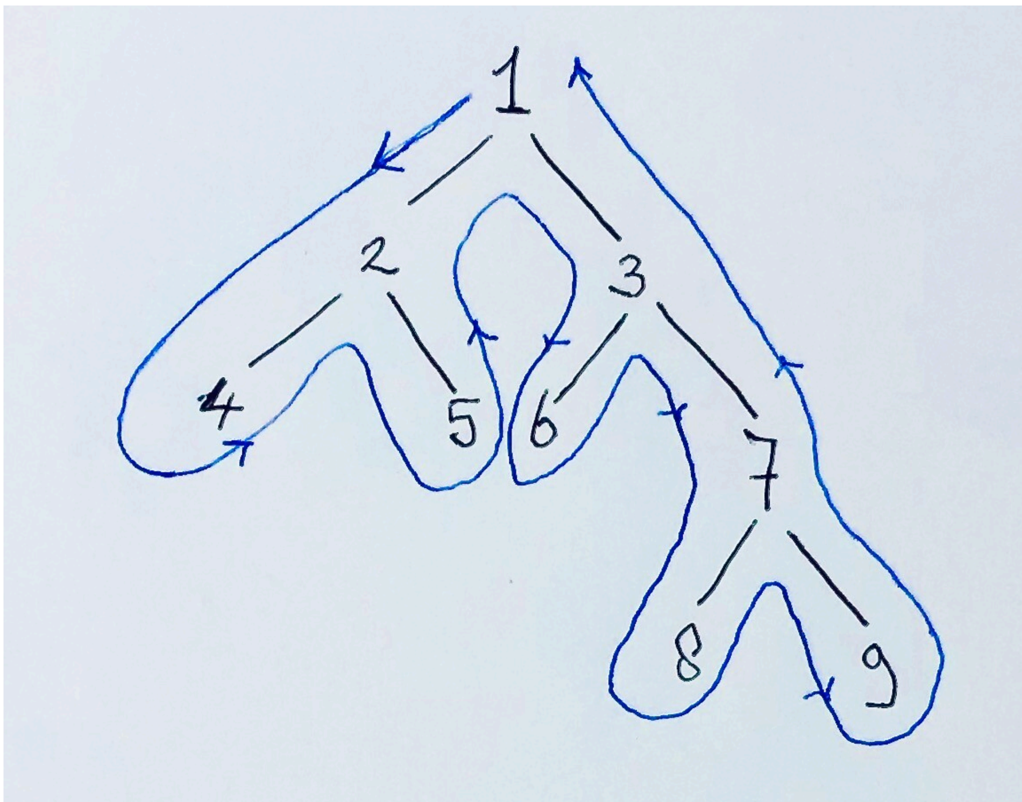
An Euler tour of a tree, is a walk "around the tree" in which we write down all edges that we visit. See the figure below for an example. Every edge of the tree occurs exactly twice in such an Euler tour (once from the parent towards the child, and once from the child back up to its parent.)
We can model such an Euler tour using the following data type:

```
data Chain a = Single a -- a chain with a single vertex and no edges.
             | Multiple [(a,a)] -- we assume the list of edges is
*non-empty*
                 deriving (Show,Eq)
```

For example, for the tree

```
myTree :: Tree Int
myTree = Node (Node (Leaf 4) 2 (Leaf 5)) 1 (Node (Leaf 6) 3 (Node (Leaf 8)
7 (Leaf 9)))
```

The blue curve below indicates an Euler tour



that is represented as

```
Multiple
```

```
[(1,2),(2,4),(4,2),(2,5),(5,2),(2,1),(1,3),(3,6),(6,3),(3,7),(7,8),(8,7),(7
,9),(9,7),(7,3),(3,1)]
```

as a value of type 'Chain Int'.

a. [3pt] Write a function

```
eulerEdges :: Tree a -> Int
```

that, given a tree, computes the number of edges in an Euler tour. Use direct recursion. You **cannot** use the function 'eulerTour' defined below.

```
eulerEdges (Leaf a) = a. ........................................(1 pt.)
eulerEdges (Node l a r) = b. ....................................(2 pt.)
```

b. [5pt] Consider the type class

```
class Semigroup t where
  (<>) :: t -> t -> t
```

Make 'Chain' an instance of 'Semigroup' such that 't1 <> t2' computes the concatenation of two chains of edges, where we ensure to add an edge from the last node of t1 to the first node of t2. Recall that in 'Multiple xs', you may assume that 'xs' is non-empty.

5 pt. **c.**

c. [5pt] Use this 'Semigroup' instance to write a function

```
eulerTour :: Tree a -> Chain a
```

that constructs such an Euler tour.

```
eulerTour (Leaf x) = d. .........................................(1 pt.)
eulerTour (Node l x r) = e. .....................................(4 pt.)
```

d. [5pt] Write a function

```
makeUnique :: Tree a -> Tree (a,Int)
```

that makes sure every value stored in the tree is unique by labeling the nodes in a post order. (i.e. the root should have the maximum label value in its subtree).
You may want to use the function

```
label :: Tree (a, Int) -> Int
label (Leaf (_, n))      = n
label (Node _ (_, n) _) = n
```

Hint: first define a helper function

```
help :: Int -> Tree a -> Tree (a,Int)
```

5 pt. **f.**

e. [5 pt] Given an Euler tour, we can (try to), reconstruct the original tree. Complete the following definition of a function

```
fromEulerTour :: Eq a => Chain a -> Tree a
```

that, given an Euler tour reconstructs the original tree it was built on. You can assume the labels/values stored in the nodes of the tree are unique.

```
fromEulerTour (Single p)       = g.  ..............................(1 pt.)
fromEulerTour (Multiple ((p,x):edges)) =
  let removeFirstAndLast ys = tail $ init ys
      (ls,zs) = break (\e' -> e' == (x,p)) edges
      (_,y) : rs = removeFirstAndLast zs
      left  = case ls of [] -> h.  ..............................(1 pt.)
                         _ -> i.  ..............................(1 pt.)
      right = case rs of [] -> j.  ..............................(1 pt.)
                         _ -> k.  ..............................(1 pt.)
  in Node left p right
```

This code uses the function

```
break :: (a -> Bool) -> [a] -> ([a],[a])
```

that, given a predicate 'p' and a list 'xs', splits 'xs' into two lists '(as,bs)' such that 'as' is the longest prefix of 'xs' such that 'p x' is 'False' for all 'x' in 'xs' and 'bs' is the rest of the list 'xs'. For example,

```
> break even [1,1,1,3,5,2,13,42,17]
([1,1,1,3,5],[2,13,42,17])
```

f. [5 pt] Using foldr, implement the function 'break' as described above.

4 pt. **l.**

4
7 pt.

[7pt] For each of the following expressions, please indicate whether it is correct that they evaluate to the list

```
[1,2,3,4,5]
```

You will receive 1pt for each question correctly answered, -1pt for each wrong answer, and 0pt for each question answered with "Don't know".

<table>
<tr><th></th><th></th><th>Correct
A</th><th>Incorrect
B</th><th>Don't know
C</th></tr>
<tr><td>filter (\x -> 5 > x) [1..21]</td><td>1</td><td></td><td></td><td></td></tr>
<tr><td>[f | f <- [1..10], g <- [1..10], f <= 5]</td><td>2</td><td></td><td></td><td></td></tr>
<tr><td>[d `div` 2 | d <- [1..10], (d + 1) `div` 2 == d `div` 2]</td><td>3</td><td></td><td></td><td></td></tr>
<tr><td>map (2+) (filter (>= -1) [-5 .. 3])</td><td>4</td><td></td><td></td><td></td></tr>
<tr><td>[c + 1 | c <- [1..10], c < 4]</td><td>5</td><td></td><td></td><td></td></tr>
<tr><td>map (+1) [b `div` 2 | b <- [1..10], b `mod` 2 == 1]</td><td>6</td><td></td><td></td><td></td></tr>
<tr><td>[a | a <- [1..10], a < 5]</td><td>7</td><td></td><td></td><td></td></tr>
</table>