[20241107] INFOFP - Functioneel programmeren - 1 - USP

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur:	2 uur en 30 minuten
Aantal vragen:	5

[20241107] INFOFP - Functioneel programmeren - 1 - USP

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 5

1 In this question,we will build a simple ``autocomplete'' feature like you may have in your favourite text editor.
Consider the following datatype of (weighted) tries

```
data Trie w = Step w [(Char, Trie w)] deriving Show
```

which are essentially trees in which the edges are labeled with a 'Char' and in which the nodes are labeled with a value of type 'w'.
We can use these to store sets of strings with weights that represent how common they are relative to other words in the set. For example

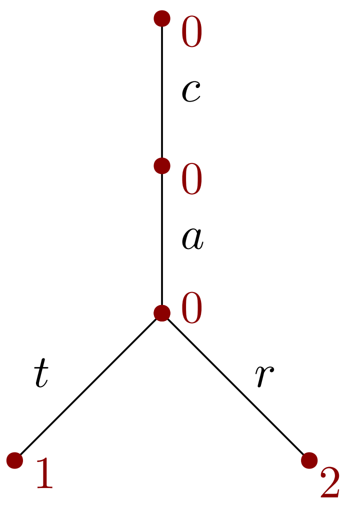
```
exampleTrie1 :: Num w => Trie w
exampleTrie1 = Step 0
 [('c', Step 0
   [('a', Step 0
     [('t', Step 1 []),
      ('r', Step 2 [])])])]
```

stores the strings "cat" and "car" where "car" has twice the weight of "cat".

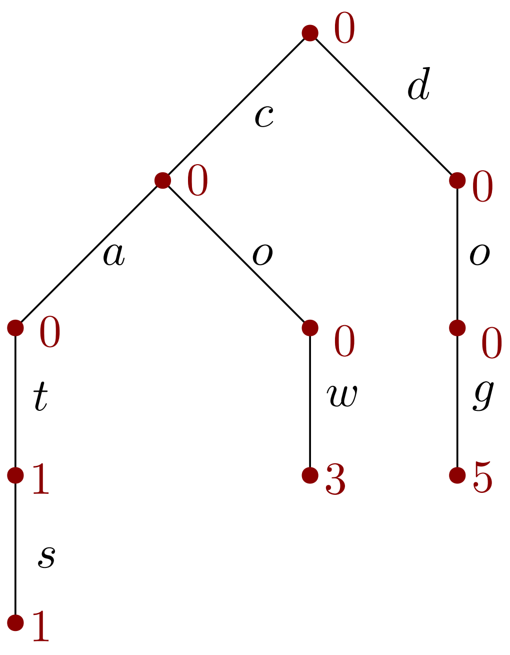
```
exampleTrie2 :: Num w => Trie w
exampleTrie2 = Step 0
 [('c', Step 0
   [('a', Step 0
     [('t', Step 1
       [('s', Step 1 [])])]),
    ('o', Step 0
       [('w', Step 3 [])])]),
  ('d', Step 0
     [('o', Step 0
       [('g', Step 5 [])])])]
```

stores "cat", "cats", "cow", "dog" with weights 1, 1, 3, and 5, respectively. We think of the strings with weight zero (such as "ca" or "") as not being a part of the set. The figure below illustrates these two tries; the nodes and their values are shown in red, whereas the edge labels are shown in black.

exampleTrie1



exampleTrie2



a. [3pt] Implement a function

```
sumWeights :: Num w => Trie w -> w
```

that takes the sum of all the weights in the Trie

```
sumWeights (Step w subTries) = a. ................................ (3 pt.)
```

b. [3pt] Complete the following code to make Trie an instance of Functor:

```
instance Functor Trie where
  fmap = mapTrie

mapTrie                      :: b. ................................ (1 pt.)
mapTrie f (Step w subTries) = c. ................................ (2 pt.)
```

c. [2pt] Without using direct recursion, write a function

```
normalize :: Trie Float -> Trie Float

normalize trie = d. ............................................. (2 pt.)
```

that normalizes the weights in the Trie such that they sum up to 1 (as would be required if we want to interpret them as probabilities; we assume that they are all positive already).

For example

```
> normalize exampleTrie2
Step 0.0
  [('c',Step 0.0
    [('a',Step 0.0
      [('t',Step 0.1
        [('s',Step 0.1 [])])]),
     ('o',Step 0.0
      [('w',Step 0.3 [])])]),
   ('d',Step 0.0
    [('o',Step 0.0
      [('g',Step 0.5 [])])])]
```

d. [4pt] Complete the following implementation of the function

```
flatten :: (Eq w, Num w) => Trie w -> [(String, w)]
flatten = filter f . flatten'
  where
    flatten' (Step w chs) = e.  .............(1 pt.)  : (concat (map g chs))
    f              = f.  .........................................(1 pt.)
    g (c,subTrie) = g.  ............................................(2 pt.)
```

that computes all the strings (**with non-zero weight**) paired with their weight that are present in a Trie. For example

```
> flatten exampleTrie2
[("cat", 1), ("cats", 1), ("cow", 3), ("dog", 5)]
```

e. [4pt] Please implement a function

```
autocomplete :: (Eq w, Num w) => String -> Trie w -> [(String, w)]
```

that computes all sequences and their (non-zero) weight that have a given sequence as a prefix. For example,

```
> autocomplete "ca" exampleTrie1
[("t",1),("r",2)]

 > autocomplete "d" exampleTrie2
[("og",5)]

 > autocomplete "cat" exampleTrie2
[("",1),("s",1)]

> autocomplete "sheep" exampleTrie2
[]
```

You may use the function 'flatten'.
Hint: remember that we have a function

```
lookup :: Eq k => k -> [(k,v)] -> Maybe v
```

4 pt. **h.**

2 In this question, we ask you to use some higher order functions to understand iteration in functional languages and we look at a specification for iteration.

a. [3pt] We can encode traditional imperative while-loops in Haskell using a construct

```
while :: c -> (c -> Bool) -> (c -> c) -> c
```

such that

```
> while init pred step
```

initialises a value of type 'c' to 'init', and, as long as the predicate 'pred' is true for this value of type 'c', it repeatedly transforms that value by applying 'step', until we get a final value of type 'c' as a result.

For example, we can use it to implement Euclid's algorithm for calculating a greatest common diviser as follows

```
gcd1 :: Int -> Int -> Int
gcd1 a b = fst $ while (a, b) (not . (== 0) . snd) step
 where
 step (x, y) = (y, x `mod` y)
```

Please implement while:

```
 while init pred step = a.  ........................................(3 pt.)
```

b. [3pt] Recall that

```
data Either a b = Left a | Right b
```

It is more common to work instead with (equivalent) iteration constructs, instead of 'while':

```
iter :: (a -> Either a b) -> a -> b
iter f a = let Right b = while (Left a) isLeft (step f) in b where
  isLeft (Right _) = False
  isLeft (Left _) = True
  step f (Left a) = f a
```

Please implement the Fibonacci function

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)
```

in terms of 'iter' (without using direct recursion).
Hint: you may want to keep `fib (n - 2)`, `fib (n - 1)` and a counter in a tuple as you iterate. That is, you may want to instantiate iter with a = (Int, Int, Int).

```
fib n = iter fibStep b.  ..................................(1 pt.)  where
  fibStep (fn, fsn, 0) = c.  ..................................(1 pt.)
  fibStep (fn, fsn, count) = d.  ................................(1 pt.)
```

c. [4pt] Please write down a specification

```
specIter :: Eq b => (a -> Either a b) -> a -> Bool
```

for 'iter' that says that iterating 'f' from 'a' is the same as applying 'f' to 'a' once and then either iterating from an updated starting value or returning the final result.

4 pt. **e.**

3 In this question, we test your knowledge of equational reasoning about functional programs.

Using the following definitions

```
const :: a -> b -> a
const a _ = a                           -- a

flip :: (a -> b -> c) -> (b -> a -> c)
flip k a b = k b a                      -- b

(.) :: (b -> c) -> (a -> b) -> (a -> c)
(g . f) x = g (f x)                     -- c

id :: a -> a
id a = a                                -- d


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

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Leaf = Leaf                   -- e
mapTree f (Node l v r) =
  Node (mapTree f l) (f v) (mapTree f r) -- f
```

please prove the following claims, where you justify every reasoning step by marking it with the letter (a - f) of some definition or by marking it as I.H. to refer to an induction hypothesis. Please clearly state any induction hypotheses you use.

a. [6pt] Claim:

```
 flip (.) (flip const g) = id
```

Hint: Observe that the type of this equation is

```
(a -> b) -> (a -> b)
```

6 pt. **a.**

b. [8pt] Claim:

```
mapTree id t = t
```

for any 't :: Tree a'

8 pt. **b.**

4 In this question, we test your knowledge of lazy evaluation.

a. [5pt] Indicate, for each of the following expressions what their WHNF is. If the expression is already in WHNF, please copy the original expression. If the expression crashes in its evaluation to WHNF, please write "undefined".

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

a. .... (1 pt.)

where

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
map (+1) [1, undefined, 3]
```

b. .... (1 pt.)

```
foldr (\b acc -> b || acc) False [False, undefined, True]
```

c. .... (2 pt.)

Hint: remember that (||) is strict in its first but not in its second argument.

```
foldl (\acc b -> b || acc) False [False, undefined, True]
```

d. .... (1 pt.)

```
seq (5, head []) 42
```

e. .... (1 pt.)

b. [2pt] Using the function

```
(||) :: Bool -> Bool -> Bool
```

and without pattern matching, please implement a function

```
strictOr :: Bool -> Bool -> Bool
strictOr b1 b2 = f.  ............................................(2 pt.)
```

that calculates the logical disjunction (OR) and is strict in both arguments.

5 In this question, we ask you to write some code that uses IO types and monads.
Recall that we have the following functions for input and output, in Haskell:

```
putStr :: String -> IO ()
getChar :: IO Char
```

a. [3pt] Implement a function

```
ask :: String -> IO Bool
```

that prints a question (represented by a string), asks the user to enter a character, checks whether this character is 'y', and gives back 'True' if and only if the user entered 'y' and 'False' otherwise.

3 pt. **a.**

Recall the function

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f [] = return []
mapM f (a:as) = do b <- f a
                   bs <- mapM f as
                   return (b:bs)
```

that lets us map a monadic function over a list.

b. [3pt] Use 'mapM' to write a function

```
countYs :: [String] -> IO Int
```

that asks a lists of questions and counts the number of questions answered with 'y'.

3 pt. **b.**

c. [3pt] Desugar the do notation below to code using '>>=' instead.

```
weird lst = do let (anns, vals) = splitAt 5 lst
               ann <- anns
               vals
               return ('c' : ann, True)
```

3 pt. **c.**

d. [2pt] What is the type of the above expression 'weird'?

`weird ::` **d.** .... (2 pt.)

e. [2pt, bonus] What does

```
weird ["aap","noot","mies"]
```

compute?

e. .... (2 pt.)