[20211005] INFOFP - Functioneel programmeren - 1 - UITHOF

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur:	2 uur
Aantal vragen:	6

[20211005] INFOFP - Functioneel programmeren - 1 - UITHOF

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 6

1 In this question, we test a basic understanding of types and elementary programming with recursion and list comprehensions.

a. [2pt] We would like to define a safer version of the 'head' function that cannot crash; i.e. we would like to define a total function that returns the first element of a list if it exists, or a 'Nothing' otherwise. Please give the type signature of such a function -- note that we are not asking you to write down the definition of the function, only its type.

a.

b. [4pt] Define a function

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

such that 'lookupExtract k m' tries to look up whether some key k appears in a list 'm' of '(key, value)' pairs, and if so extracts it from the list. That is we return the found '(key,value)' pair and the list 'm' with this pair removed. If the key 'k' does not occur we return 'Nothing' and the original list 'm'.

For example,

```
lookupExtract 4 [(3,5),(4,123),(4,8),(5,2)]
```

returns

```
(Just (4,123),[(3,5),(4,8),(5,2)])
```

b.

c. [2pt] Sometimes, we are only interested in computing the first component of 'lookupExtract', that is the function 'lookupKV' such that

```
lookupKV k m = fst (lookupExtract k m))
```

Please show how this function 'lookupKV' can be defined more easily in terms of list comprehensions and 'safeHead' (and without using 'lookupExtract').

c.

2 In this question, we test your ability to write functions on lists using direct recursion and folds.

We take a look at buckets, which are useful, for example, to implement a linear time median function as well as in many other divide and conquer algorithms.

We wish to build a 'buckets' function that will take an Int 'k', and a list, and split the list into a non-empty list of 'buckets' (lists) of 'k' elements each. Only the first bucket may contain fewer than 'k' elements (and may even be empty). For example,

```
buckets 6 "Never.gonna.give.you.up"
```

would return
["Never",".gonna",".give.","you.up"]

We implement 'buckets' as:

```
buckets :: Int -> [a] -> [[a]]
buckets k = snd . bucketsH k
```

where

```
bucketsH :: Int -> [a] -> (Int,[[a]])
```

is a helper function in which 'bucketsH k xs' returns the length of the *first* bucket and the list of all buckets.
a. [5pt] Please implement 'bucketsH' using direct recusion.
a.
We can get back the original list by concatenating all buckets using the function 'concat'.
Sometimes, we wish to first map a function 'f :: [b] -> [b]' that transforms the buckets over the list of buckets before concatening the results.
That is, we are often interested in the function

```
concatMap :: ([b] -> [b]) -> [[b]] -> [b]
concatMap f bss = concat (map f bss)
```

b. [3pt] Please implement 'concatMap' directly using 'foldr' (rather than in terms of 'concat' and 'map').

b.

3 In this question, we test your ability to program using data structures and type classes.
Recall the 'Successor a' data structure that stores an ordered set s of elements of some type a such that we can efficiently answer successor queries, i.e.

```
 succOf :: Ord a => a -> Successor a -> Maybe a
```

such that 'succOf q s' returns the smallest element from s that is at least 'q' (if such an element exists).

Suppose we want a 'Map k v' data structure that stores (key,value) pairs of type '(k,v)' such that we can efficiently answer lookup queries. i.e.

```
 lookupInMap :: Ord k => k -> Map k v -> Maybe v
```

such that 'lookupInMap k m' returns a 'Just v' if (and only if) the (key,value) pair '(k,v)' is in the map 'm' (and a 'Nothing' otherwise).
We want to *implement* the 'Map' data structure using the 'Successor' data structure. To this end, we define:

```
data Key k v = Key k (Maybe v)
type Map k v = Successor (Key k v)
```

a. [4pt] Implement the function 'lookupInMap'. You may assume that 'Key k v' is an instance of 'Eq' and 'Ord'.

a.

b. [2pt] For the above implementation of 'lookupInMap' to work, 'Key k v' needs to be an instance of 'Eq' and 'Ord'. Please give the 'Eq' type class instance so that your implementation of 'lookupInMap' is correct. You do not also need to give the 'Ord' instance (to avoid repetition).

Hint: Note that the type signature of 'lookupInMap' does not say *anything* about the type of values 'v'.

b.

4 In this question, we test your understanding of some basic higher-order functions as well as their relation to list comprehensions.

a. [2pt] Please explain what the function

```
mystery = foldl (flip (:)) []
```

computes.

a.

b. [3pt] Please write the function

```
foo yzs = [y + z | (y, z) <- yzs, y > 2]
```

in terms of higher order functions, without using a list comprehension

b.

5 In this question, we test your understanding of type inference.

a. [5pt] Determine the type of the following expression or demonstrate that it is not well-typed. Please write down a full type derivation in your answer rather than merely giving the type of the expression.

```
foldl (flip (:))
```

Hint: first determine the type of

```
flip (:)
```

to help solve the full problem and to receive partial points.

a.

b. [2pt] What is the type of

```
concat . map concat
```

Note: 2pt for a correct answer, -1pt for a wrong answer, 0pt for "I don't know.".

b.

- **a.** (a -> b) -> [[a]] -> [b]
- **b.** [[[a]]] -> [a]
- **c.** (a -> [b]) -> [a] -> [b]
- **d.** [[a]] -> [a]
- **e.** I don't know.

6 In this question, we test your understanding of how to represent and program on trees in functional languages like Haskell.

Consider the 'Trie' data type that models a Tree whose **edges** are labeled with values of type 'Char' (such trees are typically called "tries"). Furthermore, every node may have an arbitrary number of children.

```
data Trie = Leaf Bool
          | Node Bool [(Char,Trie)]
              deriving (Show,Eq)
```

Tries can be used to compactly represent a collection of "words" (i.e. Strings/lists of Chars). In particular, every word is represented by (the labels on) a path from the root to a particular 'Node' (or 'Leaf').
The Boolean stored in each 'Node' (or 'Leaf') indicates if there is a word (in the collection) that ends there. For example, the 'Trie'

```
exampleTrie =
  Node False [('t', Node False [('o', Leaf True),
                                ('e', Node False [('a', Leaf True),
                                                  ('d', Leaf True),
                                                  ('n', Leaf True)])]),
              ('a', Leaf True),
              ('i', Node True [('n', Node True [('n', Leaf True)])])]
```

represents the set of words {"to", "tea", "ted", "ten", "a", "i", "in", "inn"}.
a. [4pt] Write a function

```
suffixes :: [Char] -> Trie -> Trie
```

that takes a word 'w', and reports the (sub)trie that represents all words in the collection that are suffixes of 'w', i.e. that start with 'w'.
Hint 1: for example, 'suffixes "te" exampleTrie' should return

```
Node False [('a', Leaf True),
            ('d', Leaf True),
            ('n', Leaf True)])])
```

and 'suffixes "inn" exampleTrie' should return

```
Leaf True
```

Hint 2: use the Prelude function

```
lookup :: Eq k => k -> [(k, v)] -> Maybe v
lookup k [] = Nothing
lookup k ((k', v) : kvs) | k == k' = Just v
```

```
                                        | otherwise = lookup k kvs
```

a.

b. [2pt] Write a function 'occurs' that checks whether a word occurs in a 'Trie'.
Hint: do not use direct recursion.

b.

c. [4pt] Complete the definition of the function

```
insert :: [Char] -> Trie -> Trie
```

that takes a word and inserts it into the Trie. This definition uses the function

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

for which 'lookupExtract k m' tries to look up whether some key k appears in a list 'm' of '(key, value)' pairs, and if so extracts it from the list. It returns the found '(key,value)' pair and the list 'm' with this pair removed. If the key 'k' does not occur it returns 'Nothing' and the original list 'm'. Indeed, you implemented this function in one of the other exercises.

```
insert [] (Leaf _) = c. .......... ()

insert [] (Node _ chs) = d. .......... ()

insert (c:cs) (Leaf b) = Node b [(c,insert cs $ Leaf False)]
insert (c:cs) (Node b chs) =
    Node b $ case lookupExtract c chs of
                 (Nothing, _) -> (c,insert cs $ Leaf False) : chs
                 (Just (_,t),ts) -> e. .......... ()
```