[20221006] INFOFP - Functioneel programmeren - 1 - USP

Course: BETA-INFOFP Functioneel programmeren (INFOFP)

Duration:	2 hours
Number of questions:	4

[20221006] INFOFP - Functioneel programmeren - 1 - USP

Course: Functioneel programmeren (INFOFP)

Number of questions: 4

1 Let us define a type synonym

```
type Point = Float
```

whose elements we think of as points on a line (the number line).

We define a datatype

```
data Interval = MkInterval Point -- starting point
                           Point -- end point
                              deriving (Show)
```

whose elements we think of as intervals in the number line demarked by a starting point and an endpoint. We use the convention that Intervals are closed in the sense that they contain both endpoints. You may assume that for `MkInterval start end`, we always have that `start <= end`.
a. [2pt] Write a function

```
contains :: Interval -> Point -> Bool
```

that checks if an interval contains a given point

2 pt. **a.**

b. [3pt] Write, using direct recursion, a function

```
stabs :: Point -> [Interval] -> [Interval]
```

that, given a Point and a list of Intervals returns all Intervals that are "stabbed" by the Point, that is, all Intervals that contain that Point.

3 pt. **b.**

c. [2pt] Now, rewrite "stabs" using higher order functions, without using direct recursion.

```
q `stabs` ints = c.  ....................................... (2 pt.)
```

d. [2pt] Write, using a list comprehension, a function

```
containments :: [Interval] -- list of intervals
                 -> [Point] -- list of points
                 -> [(Point,[Interval])] -- for every point, the
intervals that contain it.
```

That given a list of intervals, and a list of points, returns for each point the intervals it stabs.

```
containments ints points = d.  ............................. (2 pt.)
```

e. [3pt] Complete the definition below of a function

```
countContainments :: [(Point, [Interval])] -> Int
```

such that

```
totalIntersections :: [Interval] -> [Point] -> Int
totalIntersections ints points = countContainments $ containments
ints points
```

computes the total number of intersections between a list of intervals and a list of points.

```
countContainments = foldr f e where
 e = e.  ............................................. (1 pt.)
 f = f.  ............................................. (2 pt.)
```

Here, by an intersection between a list `ints` of intervals and a list `pts` of points, we mean a pair of a point `pt` of `pts` and an interval `int` of `ints` such that `int` contains `pt`. For example, we have that

```
totalIntersections  [MkInterval (-1) 4, MkInterval 6 9, MkInterval
0 2, MkInterval 2 3, MkInterval 1 4] [1, 9] == 4
```

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.
a. [5pt]

```
flip foldr
```

5 pt. **a.**
b. [6pt]

```
(.) . map
```

6 pt. **b.**

3 Consider the following data type modeling binary trees (that store elements in both the leaves and the internal nodes.)

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

a. [4pt] Complete the following definition

```
longestPath :: Tree a -> [a]
longestPath = snd . longestPath'

longestPath' :: Tree a -> (Int,[a])
```

by defining `longestPath'` such that `longestPath` computes the longest root to leaf path in the tree. We use the convention that the length of the longest root to leaf path in a `Leaf` has length 1.

4 pt. **a.**

b. [1pt] Please modify the `Tree a` data type to let us store/read out the length of the longest root-to leaf path of a subtree in the data type.

1 pt. **b.**

4 Consider the following type modelling non-empty circular lists.

```
data CircularList a = MkCircularList [a] -- consecutive subset of
the elements left of the focus
                                         -- in CCW (counter
clockwise) order
                                        a  -- current focus
                                      [a] -- consecutive subset of
the elements right of the focus
                                         -- in CW (clockwise) order
                         deriving (Eq, Show)
```

Observe that the same circular list can have multiple different representations, depending on where we "cut the circle".
For example,

```
clock5   = MkCircularList [4, 3, 2, 1] 5 [6, 7, 8, 9, 10, 11, 12]
clock5'  = MkCircularList [] 5 [6, 7, 8, 9, 10, 11, 12, 1, 2, 3, 4]
clock5'' = MkCircularList [4,3,2,1,12,11,10,9,8,7,6] 5 []
```

are three different representations of a clock that reads five o'clock (5 is the focus). And

```
triangle  = MkCircularList [1] 2 [3]
notAClock = MkCircularList [4, 3, 2, 1] 5 [7, 6, 8, 9, 10, 11, 12]
```

are circular lists that represent a triangle with vertices 1, 2, 3 and a clock with the numerals in the wrong order.
a. [2pt] Complete the definition below of a function

```
cwElements :: CircularList a -> [a]
```

that returns all elements in CW order, starting from the focus.
So, for example,

```
cwElements clock5   == [5, 6, 7, 8, 9, 10, 11, 12, 1, 2, 3, 4]
cwElements clock5'  == [5, 6, 7, 8, 9, 10, 11, 12, 1, 2, 3, 4]
cwElements clock5'' == [5, 6, 7, 8, 9, 10, 11, 12, 1, 2, 3, 4]
```

```
cwElements (MkCircularList ls x rs) = a.  .................. (2 pt.)
```

b. [3pt] Write a function

```
goCw :: CircularList a -> CircularList a
```

that moves the focus in clockwise direction.

So, for example,

```
goCw clock5   == MkCircularList [5, 4, 3, 2, 1] 6 [7, 8, 9, 10, 11,
12]
goCw clock5'  == MkCircularList [5] 6 [7, 8, 9, 10, 11, 12, 1, 2,
3, 4]
goCw clock5'' == MkCircularList [5] 6 [7, 8, 9, 10, 11, 12, 1, 2,
3, 4]
```

3 pt. **b.**

c. [3pt] Write a function

```
isShiftOf :: Eq a => CircularList a -> CircularList a -> Bool
```

to test if the first circular list is a shift of the second. For example,

```
clock5 `isShiftOf` clock5'   == True
clock5 `isShiftOf` clock5''   == True
clock5 `isShiftOf` triangle  == False
clock5 `isShiftOf` notAClock == False
```

You may assume that you have a function

```
allRotations :: CircularList a -> [CircularList a]
```

that computes all different shifts/rotations of a circular list (containing each rotation exactly once). Don't worry about efficiency.

3 pt. **c.**

d. [2pt] Please make `CircularList a` an instance of the `Eq` typeclass that tests if two `CircularList`s are the same up to rotations. (This is often a more useful `Eq` instance than the automatically derived one that merely tests structural equality.)

2 pt. **d.**

e. [2pt] Please complete the following definition of the function `allRotations` that you used in subquestion c. above.

```
allRotations c = take (size c) $ cwRotations c
size :: CircularList a -> Int
size (MkCircularList l x r) = length l + 1 + length r
```

by defining an appropriate function

```
cwRotations :: CircularList a -> [CircularList a]
```

2 pt. **e.**