[20231005] INFOFP - Functioneel programmeren - 1 - USP

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur: 2 uur en 30 minuten

Aantal vragen: 5

[20231005] INFOFP - Functioneel programmeren - 1 - USP

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 5

1 We start with a couple of basic programming questions about lists.

a. [6pt] Using direct recursion, implement the function

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

that creates a list from a predicate and another list. It inspects the original list and takes from it its elements until the moment when the predicate fails. Then it stops processing. For example,

```
>>> takeWhile (\x -> x <10) [1,5,1,8,20,1,30]
[1,5,1,8]
>>> takeWhile (\x -> x <10) [1,5,1]
[1,5,1]
```

a.

b. [6pt] Please complete the following implementation of the function

```
elem :: Eq a => a -> [a] -> Bool
```

that checks whether an element is present in a list:

```
elem y = foldr f e where
    e = b. .....()
    f = c. .....()
```

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. [8pt]

```
foldr filter
```

a.

b. [8pt]

```
foldl filter
```

b.

3 Let us define the following two types that allow us to model (axis-aligned) squares:

```
type Point = (Int, Int) -- points in the 2d plane represented through
their x and y coordinates

data Square = Square Point Int deriving (Show,Eq)

lowerLeft              :: Square -> Point
lowerLeft (Square l _) = l

width              :: Square -> Int
width (Square _ w) = w
```

a. [6pt] Please write a function

```
inSquare :: Point -> Square -> Bool
```

that tests if a point lies in (or on the boundary of) a square.

a.

b. [8pt] Write a function

```
intersects :: Square -> Square -> Bool
```

that tests if two (axis-aligned) squares intersect.
Hint: two (axis-aligned) squares intersect if and only if one square has a corner inside the other square.

b.

c. [4pt] Consider the following typeclass that represents partially ordered types

```
class Eq t => PartialOrd t where
  isLeq :: t -> t -> Bool
```

where `isLeq x y` returns `True` if and only if `x` appears before `y` (or is equal to `y`) in the partial order.

For example, we have the following instance for intervals (modeled by pairs of a lower bound and an upper bound):

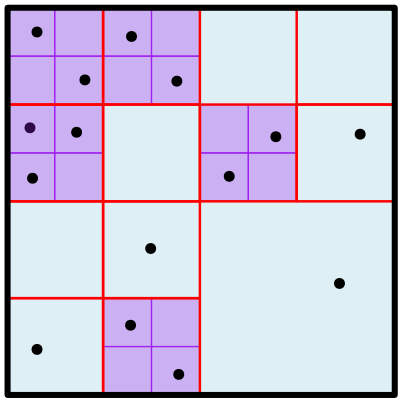
```
instance (Ord a) => PartialOrd (a,a) where
  isLeq (l,r) (l',r') = l' <= l && r <= r'
```

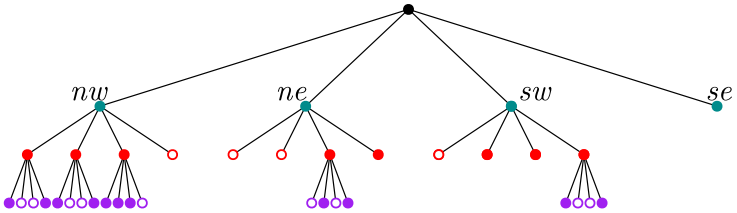
Please make `Square` an instance of `PartialOrd` so that `isLeq s1 s2` if (and only if) square s1 is contained in square s2. In other words, when the set of points in s1 form a subset of the points in s2.

c.

Given an square and a set of points. We can define a recursive decomposition of the square into four

equal-size subsquares until each square contains only one point. This defines a tree-structure known as a QuadTree. See the following figure that shows the recursive decomposition of the square on the left, and the resulting tree structure on the right.





We can then represent such a decomposition in Haskell using the following `QuadTree` type:

```
data QuadTree = EmptyLeaf
              | FullLeaf Point
              | Node Square
                     QuadTree QuadTree
                     QuadTree QuadTree
                 deriving (Show,Eq)
```

We can then represent the quadtree shown in the figure above as something like:

```
myQuadTree :: QuadTree
myQuadTree = Node blackSquare nw ne sw se
  where
     nw = Node blueNWSquare EmptyLeaf EmptyLeaf (Node ...) EmptyLeaf
     ne = ...
     sw = ...
     se = FullLeaf p
```

(we only give a partial definition to avoid clutter).

d. [6pt] Complete the following definition of a function

```
buildQuadTree :: Square -> [Point] -> QuadTree
```

which takes a Square and a list of points, and builds a QuadTree. In particular, we keep refining a square (node) until there is at most one point in the square left.

```
buildQuadTree s [] = d. ..........................................()
buildQuadTree s [p] = e. .........................................()
buildQuadTree s@(Square l@(x,y) w) pts = Node s (recurse nw) (recurse ne)
(recurse se) (recurse sw) where
```

```
  w' = f. ..................................................()
  square' p = Square p w'
  recurse s' = buildQuadTree s' (points s')
  points s' = g. ...........................................()
  nw = square' (x,y+w')
  ne = square' (x+w',y+w')
  se = square' (x+w',y)
  sw = square' l
```

e. [6pt] QuadTrees are somewhat useful to efficiently find points in some query region.

Write a function report, that takes an arbitrary query Square q, and a QuadTree, and reports all the points in the QuadTree that lie in (or on the boundary of) q. Make sure to visit the children of a node only when it intersects the query square.

```
report :: Square -> QuadTree -> [Point]
```

h.

4 In this question, we look at some higher order functions. In particular, we see how they can be useful for making lists more efficient.

We define the type synonym

```
type ListBuilder a = [a] -> [a]
```

These list-builders are sometimes also known as difference lists.
We can use this type to make computations with lists of type `[a]` more efficient. The idea is that the input list given to the `ListBuilder a` is the remaining suffix (that we currently may not know yet) that we combine with the part of the list that we did already receive.

We can turn any list into a list-builder:

```
asBuilder :: [a] -> ListBuilder a
asBuilder xs = \suffix -> xs ++ suffix
```

We tend to do this to convert the input (a list) we care about into a list-builder.
Next, we perform our particular computation using the list-builder representation.
This often allows us to easily write more efficient code than if we were working directly with lists.
Once we are done, we materialize the result to an actual list by running the builder on the empty list as input:

```
materialize :: ListBuilder a -> [a]
materialize builder = builder []
```

a. [2pt] Write a function

```
(+++) :: ListBuilder a -> ListBuilder a -> ListBuilder a
```

that implements the `(++)` function on list-builders. That is, such that

```
materialize (asBuilder xs +++ asBuilder ys)
```

equals

```
xs ++ ys
```

a.

b. [4pt] Write a function

```
concat' :: [ListBuilder a] -> ListBuilder a
```

that performs the equivalent of the `concat` function on list-builders.

b.

5 Consider the following function toList, that converts a tree into a list:

```
elems :: Tree a -> [a]
elems BLeaf = []
elems (BNode l x r) = elems l ++ [x] ++ elems r
```

where

```
data Tree a = BLeaf | BNode (Tree a) a (Tree a)
```

This function has the same issue as we saw with the initial implementation of reverse: it runs in $O(n^2)$ time, since we repeatedly concatenate lists.

a. [6pt] Write a function

```
fastElems :: Tree a -> [a]
```

that computes the same result as `elems` but avoids this issue and runs in linear time instead.

There are no futher questions (no b. etc.). This is the end of the exam. Have a nice evening!