[20251002] INFOFP - Functioneel programmeren - 1 - USP

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur:	2 uur
Aantal vragen:	4

[20251002] INFOFP - Functioneel programmeren - 1 - USP

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 4

1 In this question, we work with *lists of coefficients of a polynomial in a single variable x* where the i-th element is the coefficient of x^i. For example, we think of the list '[2, 0, 3]' as the polynomial '2 + 0 * x + 3 * x^2'.

a. [3pt] Without using explicit recursion, please implement a function

```
 degreeList :: (Eq a, Num a) => [a] -> Maybe Int
```

that returns the degree of the polynomial after removing trailing zeros.

Edge cases:
• If the list is empty OR all coefficients are zero, return Nothing.
• Otherwise return Just d, where d is the largest index with a nonzero coefficient.

Examples:

```
 degreeList [] == Nothing
 degreeList [0,0,0] == Nothing
 degreeList [5] == Just 0
 degreeList [5,0,0] == Just 0
 degreeList [5,0,3,20] == Just 3
 degreeList [5,0,3,20,0] == Just 3
```

You may use the following helper function stripZeros.

```
stripZeros :: (Eq a, Num a) => [a] -> [a]
stripZeros = reverse . dropWhile (== 0) . reverse
```

3 pt. **a.**

b. [5pt] Please implement a function

```
addListsPad :: Num a => [a] -> [a] -> [a]
```

that adds two coefficient lists element-wise, padding the shorter with zeros.
Clarification: This corresponds to polynomial addition. If one list ends, keep copying the remaining coefficients of the other list.

Examples:

```
 addListsPad [1,2,3] [4,5] == [5,7,3]
 addListsPad [1,2] [4,5,6] == [5,7,6]
 addListsPad [] [4,5,6] == [4,5,6]
 addListsPad [1,2,3] [] == [1,2,3]
 addListsPad [-1] [1] == [0]
```

5 pt. **b.**

c. [4pt] Without using recursion, please implement a function

```
nonZeroPositions :: (Eq a, Num a) => [a] -> [Int]
```

that returns the list of indices 'i' whose coefficient is nonzero (where indices are 0-based).
Examples:

```
 nonZeroPositions [0,3,0,5] == [1,3]
 nonZeroPositions [] == []
 nonZeroPositions [0,0,0] == []
```

4 pt. **c.**

d. [4pt] Please implement a function

```
eval :: Num a => [a] -> a -> a
```

that evaluates a polynomial given by its coefficients using a fold (no recursion). Your implementation should use what is known as 'Horner's method', and satisfy:
'eval [a0,a1,..,an] x = a0 + x*(a1 + x*(... + x*an))'

Examples:

```
 eval [5] 2 == 5
 eval [5,2,3] 2 == 5 + 2*2 + 3*2^2 == 21
```

4 pt. **d.**

Not all lists of 'a's should be interpreted as polynomials. Hence, we introduce the newtype 'Poly a' to make intent explicit.

```
newtype Poly a = P [a]
```

Consider the following 'Monoid' type class, representing an associative binary operation on values of the same type together with a unit element.

```
class Monoid a where
 -- | An associative operation.
   (<>) :: a -> a -> a
   mempty :: a
 -- | A unit element for <>
```

e. [4pt] Define a 'Monoid' instance for polynomials that performs *coefficient-wise addition* (padding with zeros as needed) as its binary operation. You may reuse your function from (b).

4 pt. **e.**

2 We now work with a binary-search-tree-like structure where *elements live in the leaves* and internal nodes carry only a routing key (a pivot). That key is used to direct the search; it is not itself stored as a leaf element

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

We maintain the following routing invariant (BST-style):
For any 'Node l k 'r:
• all elements in 'l' are strictly < 'k'
• all elements in 'r' are >= 'k' Elements are stored only in Leaves. The key 'k' serves as a routing pivot.

Example of a valid tree (storing integers):

```
Node (Leaf 1) 3 (Node (Leaf 3) 5 (Leaf 7))
```

It is valid because: 1 < 3 and all elements in the right subtree are >= 3. Note that the keys stored at internal nodes do not necessarily appear as elements of the tree (e.g. 5).
a. [3pt] Please implement the following function for checking whether an element is in a binary search tree:

```
member :: Ord a => a -> Tree a -> Bool
```

Follow the invariant above: compare against the routing key at 'Node' to decide to go left (< k) or right (>= k); base case checks equality at 'Leaf'.
Examples:

```
member 3 (Leaf 3) == True
member 2 (Node (Leaf 1) 3 (Leaf 4)) == False
member 3 (Node (Leaf 1) 3 (Node (Leaf 3) 5 (Leaf 7))) == True
```

3 pt. **a.**

b. [4pt] Please complete the implementation below for the function:

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

that produces the elements in ascending order (inorder traversal of leaves). Preferrably your implementation runs in linear time.

```
toAscList = go []
   where
   go acc (Leaf x)     = b.  ......................................(1 pt.)
   go acc (Node l _ r) = c.  ......................................(3 pt.)
```

Example:

```
 toAscList (Node (Leaf 1) 3 (Node (Leaf 3) 5 (Leaf 7))) == [1,3,7]
```

c. [3pt] Please implement a function

```
minOf :: Tree a -> a
```

that returns the leftmost leaf element (the minimum). Example:

```
minOf (Node (Leaf 2) 3 (Leaf 5)) == 2
```

3 pt. **d.**

d. [2pt] Please implement a function

```
join :: Ord a => Tree a -> Tree a -> Tree a
```

that given trees l and r such that every element of l is < every element of r, produce a valid BST with height +1 by choosing a routing key appropriately. Example:

```
join (Leaf 1) (Leaf 3) == Node (Leaf 1) 3 (Leaf 3)
```

2 pt. **e.**

Just e. [6pt] Complete 'buildBalanced' below so that, given a non-empty sorted list of n elements (in ascending order), it constructs a balanced binary search tree in a bottom-up manner by repeatedly pairing adjacent trees.

Complexity requirement: The intended approach runs in linear time O(n) (bottom-up pairing), rather than the O(n log n) of a naive top-down split-at-middle approach.

```
buildBalanced :: [a] -> Tree a
buildBalanced = combineByLevel . map f. ..........................(1 pt.)
  where
    combineByLevel :: [Tree a] -> Tree a
    combineByLevel ts = case g. ..............................(2 pt.) of
                          []     -> error "impossible"
                          [root] -> h. ...........................(1 pt.)
                          level  -> i. ...........................(1 pt.)

   pairUp (l:r:rest) = j. ..........................(1 pt.) : pairUp rest
   pairUp ts = ts
```

Examples:

```
Input list (sorted):
 [1, 2, 3, 4]

Output tree:
 Node (Node (Leaf 1) 2 (Leaf 2)) 3 (Node (Leaf 3) 4 (Leaf 4))

    (3)
    / \
  (2) (4)
  / \ / \
1 2  3  4


Input list (sorted):
 [1, 2, 3, 4, 5]

Output tree:
 Node (Node (Node (Leaf 1) 2 (Leaf 2)) 3 (Node (Leaf 3) 4 (Leaf 4))) 5
(Leaf 5)

     (5)
     / \
   (3)  5
   / \
 (2) (4)
 / \ / \
1  2 3  4
```

We would like to implement a function 'delete' that, if possible, removes a given element from a BST. Your function should be total, i.e. for any valid input BST (in which every element appears at most once) it should give a valid output; i.e. your function should not crash. Complete the following type signature for this function 'delete'. Make sure that your function returns something of an appropriate type.

f. [1pt] Please complete the type of delete below:

```
delete :: Ord a => a -> Tree a -> k.  .............................(1 pt.)
```

g. [5pt] Complete the following definition of 'delete'.
You may assume that the 'implementedElsewhere' function correctly handles a deletion when the element we are deleting appears in the right subtree (i.e. when x >= k).

```
delete x t@(Leaf y) = l.  .........................................(2 pt.)
delete x (Node l k r) | x < k = m.  ............(1 pt.)  (case delete x l of
                                     n.  .....................(1 pt.)  -> r
                                     o.  ...........(1 pt.)  -> Node l' k r
                                  )
                      | otherwise = implementedElsewhere x l k r
```

3 a. [4pt] The ADTs below are *given*.

```
data Pair a b = Pair a b
data Rose a = R a [Rose a]
data Foo a = A (a, Maybe a) | B (Either a (a -> a))
data Either a b = Left a | Right b
```

For each of the four expressions below, fill in ONLY:
• its most general Haskell type, or
• write “ill-typed” if it is not well-typed.
Do not include any derivation or explanation for this subpart.

```
e1 = Pair True 'x'
e1 :: a. ........................................................(1 pt.)

e2 = R 'a' [R 'b' []]
e2 :: b. .......................................................(1 pt.)

e3 = A (True, Just 'x')
e3 :: c. ........................................................(1 pt.)

e4 = B (Right id)
e4 :: d. .........................................................(1 pt.)
```

b. [5pt] For each function f1–f5, fill in ONLY its most general Haskell type or write “ill-typed” if it is not well-typed. Include typeclass constraints when needed. No justification needed.

```
f1 x y = (x, y)
f1 :: e. ........................................................(1 pt.)

f2 x y = if x == x then Just y else Nothing
f2 :: f. ........................................................(1 pt.)

f3 f x ys = f x : ys
f3 :: g. ........................................................(1 pt.)

f4 n y = const (y == y) (n + 0)
f4 :: h. ........................................................(1 pt.)

f5 x = x + 1 == x
f5 :: i. .........................................................(1 pt.)
```

c. [7pt] Give a full type inference derivation for the expression:

```
foldl bind
```

where

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

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

Please include key reasoning step as partial credits are awarded based on them.

7 pt. **j.**

4 [12pt] For each expression below, decide whether it is *exactly evaluates* to the target list:

```
target = [0,2,4,6,8]
```

Mark your choice per expression:
A = Correct (it evaluates to target),
B = Incorrect (it does not evaluate to target),
C = Don't know.

Scoring: +1 for a correct answer, −1 for a wrong answer, 0 for "Don't know".
Reminders:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ [] _ = []
zipWith _ _ [] = []

-- This is a simplified implementation
scanl :: (b -> a -> b) -> b -> [a] -> [b]
scanl f z = foldl (\acc x -> acc ++ [f (last acc) x]) [z]
```

1 pt. **a.** map (*2) [0..4]

- **a.** Correct (it evaluates to target)
- **b.** Incorrect (it does not evaluate to target)
- **c.** Don't know

1 pt. **b.** [2*x | x <- [0..10], x <= 4]

- **a.** Correct (it evaluates to target)
- **b.** Incorrect (it does not evaluate to target)
- **c.** Don't know

1 pt. **c.** [x*2 | x <- [0..10], even x]

- **a.** Correct (it evaluates to target)
- **b.** Incorrect (it does not evaluate to target)
- **c.** Don't know

1 pt. **d.** zipWith (+) [0,2,4,6,8] [0,0,0,0,0]

- **a.** Correct (it evaluates to target)
- **b.** Incorrect (it does not evaluate to target)
- **c.** Don't know

1 pt. **e.** `zipWith (*) [0..4] [0,2..8]`

- **a.** Correct (it evaluates to target)
- **b.** Incorrect (it does not evaluate to target)
- **c.** Don't know

1 pt. **f.** `take 5 (scanl (+) 0 [2,2,2,2,2])`

- **a.** Correct (it evaluates to target)
- **b.** Incorrect (it does not evaluate to target)
- **c.** Don't know

1 pt. **g.** `concat (map (\x -> [2*x]) [0..4])`

- **a.** Correct (it evaluates to target)
- **b.** Incorrect (it does not evaluate to target)
- **c.** Don't know

1 pt. **h.** `[0,2..8]`

- **a.** Correct (it evaluates to target)
- **b.** Incorrect (it does not evaluate to target)
- **c.** Don't know

1 pt. **i.** `map snd (zip [0..4] [0,2..10])`

- **a.** Correct (it evaluates to target)
- **b.** Incorrect (it does not evaluate to target)
- **c.** Don't know

1 pt. **j.** `[x+y | (x,y) <- zip [0..4] [0,2..8]]`

- **a.** Correct (it evaluates to target)
- **b.** Incorrect (it does not evaluate to target)
- **c.** Don't know

1 pt. **k.** `foldr (\x acc -> 2*x : acc) [] [0..4]`

- **a.** Correct (it evaluates to target)
- **b.** Incorrect (it does not evaluate to target)
- **c.** Don't know

1 pt. **l.** foldl (\acc x -> 2*x : acc) [] [0..4]

a. Correct (it evaluates to target)

b. Incorrect (it does not evaluate to target)

c. Don't know