WiSe 2020/2021

Funktionale Programmierung

3. Übungsblatt (für das Tutorium)

Prof. Dr. Margarita Esponda

Ziel: Auseinandersetzung mit polymorphe Funktionen, Listengeneratoren und Funktionen höherer Ordnung.

1. Aufgabe (4 Punkte)

Betrachten Sie folgende Funktionsdefinition, die die Menge aller möglichen Sublisten der Elemente einer Liste berechnet:

```
powerList [] = [[]]
powerList (x:xs) = powerList xs ++ map (x:) (powerList xs)
```

Schreiben Sie alle Reduktionsschritte für den folgenden Ausdruck:

```
powerList  [1, 2, 3, 4]
```

Losung:

```
powerList [1, 2, 3, 4]

    => powerList [2,3,4] ++ map (1:) (powerList [2,3,4])

    => (powerList [3,4] ++ map (2:) (powerList [3,4]))
         ++ map (1:) (powerList [3,4] ++ map (2:) (powerList [3,4]))

    => ((powerList [4] ++ map (3:) (powerList [4]))
         ++ map (2:) ((powerList [4] ++ map (3:) (powerList [4])))
         ++ map (1:) ((powerList [4] ++ map (3:) (powerList [4]))
         ++ map (2:) ((powerList [4] ++ map (3:) (powerList [4])))

    => ((powerList [] ++ map (4:) (powerList []))
                        ++ map (3:) (powerList [] ++ map (4:) (powerList []))
                        ++ map (2:) (((powerList [] ++ map (4:) (powerList []))
                        ++ map (3:) (powerList [] ++ map (4:) (powerList [])))
                        ++ map (1:) (((powerList [] ++ map (4:) (powerList []))
                        ++ map (3:) (powerList [] ++ map (4:) (powerList []))
                        ++ map (2:) (((powerList [] ++ map (4:) (powerList []))
                        ++ map (3:) (powerList [] ++ map (4:) (powerList [])))))

    => (( [[]] ++ map (4:) ( [[]] ))
                        ++ map (3:) ( [[]] ++ map (4:) ( [[]] ))
                        ++ map (2:) ((( [[]] ++ map (4:) ( [[]] )
                        ++ map (3:) ( [[]] ++ map (4:) ( [[]] ))))
                        ++ map (1:) ((( [[]] ++ map (4:) ( [[]] )
                        ++ map (3:) ( [[]] ++ map (4:) ( [[]] ))
                        ++ map (2:) ((( [[]] ++ map (4:) ( [[]] )
                        ++ map (3:) ( [[]] ++ map (4:) ( [[]] )))))
```

=> ([[], [4]] ++ map (3:) ([[], [4]]) ++ map (2:) (([[[], [4]] ++ map (3:) ([[], [4]])))
++ map (1:) (([[[], [4]] ++ map (3:) ([[], [4]]) ++ map (2:) (([[[], [4]]
++ map (3:) ([[], [4]]))))

=> [[], [4], [3], [3, 4]] ++ [[2], [2, 4], [2, 3], [2, 3,4]]
++ [[1], [1, 4], [1, 3], [1, 3, 4], [1, 2], [1, 2, 4], [1, 2, 3], [1, 2, 3, 4]]

=> **[[], [4], [3], [3, 4], [2], [2, 4], [2, 3], [2, 3, 4], [1], [1, 4], [1, 3], [1, 3, 4], [1, 2], [1, 2, 4], [1, 2, 3], [1, 2, 3, 4]]**

2. Aufgabe (2 Punkte)

Was ist die Normalform folgendes Ausdrucks? Berechnen Sie die Lösung ohne den Ausdruck in dem Haskell-Interpreter einzugeben. Begründen Sie Ihre Lösung.

```
[if x==y then 'o' else '.' | x <- [1..5], y <- [1..7], (x+y)<9]
```

	if x==y then 'o' else '.'	x	y	(x+y)<9
"o"	'o'	1	1	True
"o."	'.'	1	2	True
"o.."	'.'	1	3	True
"o..."	'.'	1	4	True
"o...."	'.'	1	5	True
"o....."	'.'	1	6	True
"o......"	'.'	1	7	True
"o......."	'.'	2	1	True
"o.......o"	'o'	2	2	True
"o.......o."	'.'	2	3	True
"o.......o.."	'.'	2	4	True
"o.......o..."	'.'	2	5	True
"o.......o...."	'.'	2	6	True
"o.......o...."		2	7	False
"o.......o....."	'.'	3	1	True
"o.......o......"	'.'	3	2	True
"o.......o......o"	'o'	3	3	True
"o.......o......o."	'.'	3	4	True

	if x==y then 'o' else '.'	x	y	(x+y)<9
"o.......o......o.."	'.'	3	5	True
"o.......o......o.."		3	6	False
"o.......o......o.."		3	7	False
"o.......o......o..."	'.'	4	1	True
"o.......o......o...."	'.'	4	2	True
"o.......o......o....."	'.'	4	3	True
"o.......o......o.....o"	'o'	4	4	True
		4	5	False
		4	6	False
		4	7	False
"o.......o......o.....o."	'.'	5	1	True
"o.......o......o.....o.."	'.'	5	2	True
"o.......o......o.....o..."	'.'	5	3	True
		5	4	False
"o.......o......o.....o...o"	'o'	5	5	True
		5	6	False
		5	7	False

3. Aufgabe (7 Punkte)

Definieren Sie eine polymorphe Funktion **poss**, die die Positionen eines Elements innerhalb einer Liste wiederum in einer Liste zurückgibt.

Anwendungsbeispiel:

poss 'e' "Freie Universität Berlin" => [2, 4, 10, 19]

a) Definieren Sie zuerst die Funktion nur unter Verwendung von expliziter Rekursion und Akkumulator-Technik.

Lösung:

```
poss :: Eq a => a -> [a] -> [Int]
poss a xs = poss' [] 0 a xs
        where
        poss' ps _ _ [] = ps
        poss' ps n a (x:xs) | a==x = poss' (n:ps) (n+1) a xs
                            | otherwise = poss' ps (n+1) a xs
```

b) Definieren Sie die Funktion unter sinnvoller Verwendung von Listengeneratoren und Funktionen höherer Ordnung.

Losung:

```
poss :: Eq a => a -> [a] -> [Int]
poss a xs = [ fst (n,x) | (n,x) <- zip [0..] xs, a==x]
```