WS 2020/2021

Funktionale Programmierung

3. Übungsblatt

Prof. Dr. Margarita Esponda

1. Aufgabe (3 Punkte)

Definieren Sie eine rekursive Funktion **(°)**, die bei Eingabe zweier natürlicher Zahlen **k** und **n**, die Zahl **k** **n**-mal potenziert. D.h. die Anzahl der **k**-Elemente im Potenzturm ist gleich **n**. Vergessen sie dabei nicht, dass die Potenz-Funktion rechtsassoziativ ist.

$k \circ n = k^{k^{\cdot^{\cdot^{\cdot^{k}}}}}$ (Der Turm auf der rechten Seite der Gleichung hat **n**-mal das Argument **k)**

Anwendungsbeispiel:

(°) 3 3 => 7625597484987

4 **°** 3 => 134078079299425970995740249982058461274793658205923933
77723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649006084096

Lösung:

```
(°) :: Integer -> Integer -> Integer
(°) k 0 = 1
(°) k n = k ^ ((°) k (n-1))
```

2. Aufgabe (3 Punkte)

Definieren Sie eine polymorphe rekursive Funktion **flatten**, die aus einer Liste von Listen eine einfache Liste erstellt.

Anwendungsbeispiel:

flatten [[8, 2], [3], [], [4, 5, 0, 1], [1] = [8, 2, 3, 4, 5, 0, 1, 1]

Lösung:

```
flatten :: [[a]] -> [a]
flatten [] = []
flatten (x:xs) = x ++ flatten xs
```

3. Aufgabe (4 Punkte)

Programmieren Sie eine polymorphe Funktion, die an einer bestimmten Position ein neues Element in eine Liste einfügt.

Verwenden Sie die **error**-Funktion für den Fall, dass die angegebene Positionszahl größer als die Länge der Liste ist.

Anwendungsbeispiel:

insert '3' 3 "Hello" => "Hel3lo"

Lösung:

```
insert :: a -> Int -> [a] -> [a]
insert elem n xs | n<0 || n>(length xs) = error "....."
                 | otherwise = (take n xs) ++ [elem] ++ (drop n xs)
```

4. Aufgabe (4 Punkte)
Wir haben in der Vorlesung die Funktion **bin2dec** definiert (siehe Vorlesungsfolien), die die Binärdarstellung einer positiven Zahl bekommt und daraus die Dezimalzahl berechnet.

Definieren Sie eine Funktion **toDecFrom**, die eine Zahl als Liste von einzelnen Ziffern, und die Basis **b** (mit **0<b<=10**) bekommt und die Dezimaldarstellung der Zahl berechnet. Bei falscher Eingabe der Ziffern oder der Basis müssen Sie mit Hilfe der **error**-Funktion entsprechende Fehlermeldung ausgeben.

Anwendungsbeispiel:

toDecFrom [1,3,2,1] 4 => 121

Lösung:

```
toDecFrom :: Int -> [Int] -> Int
toDecFrom   b   [] = 0
toDecFrom   b   xs  | b==1 && onlyOnes xs = length xs
                    | validDigits b xs = toDec b (reverse xs)
                    | otherwise = error "wrong digits or wrong base"
                                  where
                                    toDec b []     = 0
                                    toDec b (x:xs) = b*(toDec b xs) + x

validDigits :: Int -> [Int] -> Bool
validDigits b []     = True
validDigits b (x:xs) = (0<=x && x<b) && (validDigits b xs)

onlyOnes :: [Int] -> Bool
onlyOnes [] = True
onlyOnes (x:xs) = x==1 && onlyOnes xs
```

5. Aufgabe (6 Punkte)

Definieren Sie eine polymorphe Funktion **removeReps** , die wiederholte Elemente aus einer Liste entfernt, so dass am Ende jedes Element nur einmal vorkommt.

Anwendungsbeispiel:

removeReps [1, 1, 2, 0, 2, 1, 3, 1, 4] => [1, 2, 0, 3, 4]
removeReps "Freie Universität Berlin" => "Frei UnvstäBl"

Lösung:

```
removeReps :: Eq a => [a] -> [a]
removeReps [] = []
removeReps [x] = [x]
removeReps (x:xs) = x: removeReps(delReps x xs)
                    where
                     delReps x [] = []
                     delReps x (y:ys) | x==y = delReps x ys
                                      | otherwise = y: delReps x ys
```

6. Aufgabe (6 Punkte)
Schreiben Sie eine Funktion **tokenizer,** die aus einem einfachen Text die Liste aller Wörter des Textes berechnet. Der Text besteht nur aus Buchstaben und Trennzeichen. Mit Trennzeichen sind Kommata, Punkte, Fragezeichen und Leerzeichen gemeint.

Die Funktion soll folgende Signatur haben:

tokenizer :: [Char] -> [[Char]]

Lösung:

```
tokenizer :: [Char] -> [[Char]]
tokenizer [] = []
tokenizer ws = listOfWords [] [] ws
        where
        listOfWords acc [] []      = acc
        listOfWords acc word []    = acc++[word]
        listOfWords acc [] (x:rest)      | delimiter x = listOfWords acc [] rest
        listOfWords acc word (x:rest) | delimiter x  = listOfWords (acc++[word]) [] rest
                                      | otherwise    = listOfWords acc (word++[x]) rest

delimiter :: Char -> Bool
delimiter c = elem c [',', '.', '?', '!', ' ', '\n', ,\t']
```

7. Aufgabe (6 Punkte)

Definieren Sie eine polymorphe Funktion **groupEquals**, die die aufeinanderfolgenden gleichen Elemente einer Liste gruppiert und als Ergebnis eine Liste von Listen zurückgibt.

Anwendungsbeispiel:

groupEquals [1,1,2,2,1,2,2,1,1,1] => [[1,1],[2,2],[1],[2,2],[1,1,1]]

Lösung:

```
groupEquals :: (Eq a) => [a] -> [[a]]
groupEquals []  = []
groupEquals [x] = [[x]]
groupEquals (x1:x2:xs) | x1 == x2 = (x1 : y) : ys
                       | otherwise = [x1] : y : ys
                                   where
                                   (y:ys) = groupEquals (x2:xs)
```

8. Aufgabe (5 Bonuspunkte)
Definieren Sie eine Haskell-Funktion, die zwei positive Binärzahlen multipliziert.

Anwendungsbeispiel:

multBits [1,0,0,1] [1,0,1,0] => [1,0,1,1,0,1,0]

Wichtige Hinweise:

1) Verwenden Sie geeignete Namen für Ihre Variablen und Funktionsnamen, die den semantischen Inhalt der Variablen oder die Semantik der Funktionen wiedergeben.
2) Verwenden Sie vorgegebene Funktionsnamen, falls diese angegeben werden.
3) Kommentieren Sie Ihre Programme.
4) Verwenden Sie geeignete lokale Funktionen und Hilfsfunktionen in Ihren Funktionsdefinitionen.
5) Geben Sie für alle Funktionen die entsprechende Signatur an.
6) Schreiben Sie getrennte Test-Funktionen für alle Aufgaben.
7) Die Lösungen sollen elektronisch (nur Whiteboard-Upload) abgegeben werden. **Keine verspätete Abgabe per Email ist erlaubt.**