WS 2020/2021

Funktionale Programmierung

2. Übungsblatt

Prof. Dr. Margarita Esponda

1. Aufgabe (3 Punkte)

Gegeben sei folgende Funktionsdefinition:

```
bin2dec :: [Int] -> Int
bin2dec bits = bin2dec' 0 bits
                where
                    bin2dec' ac [b]    = 2*ac + b
                    bin2dec' ac (b:bs) = bin2dec' (2*ac + b) bs
```

Reduzieren Sie folgenden Ausdruck, schreiben Sie die einzelnen Schritte bis zur Normalform.

bin2dec [0,1,0,1,1,0] => …

Lösung:

```
bin2dec [0,1,0,1,1,0]  =>  bin2dec' 0 [0,1,0,1,1,0]
                       =>  bin2dec' (2*0 + 0) [1,0,1,1,0]
                       =>  bin2dec' 0 [1,0,1,1,0]
                       =>  bin2dec' (2*0 + 1) [0,1,1,0]
                       =>  bin2dec' 1 [0,1,1,0]
                       =>  bin2dec' (1*2 + 0) [1,1,0]
                       =>  bin2dec' 2 [1,1,0]
                       =>  bin2dec' (2*2 +1) [1,0]
                       =>  bin2dec' 5 [1,0]
                       =>  bin2dec' (5*2 + 1) [0]
                       =>  bin2dec' 11 [0]
                       =>  bin2dec' (11*2 + 0)
                       =>  22
```

2. Aufgabe (6 Punkte)

Die Zahl π kann mit der folgenden unendlichen Seriensumme berechnet werden:

$$\pi = \sum_{k=0}^{\infty} \frac{(2^{k+1})(k!)^2}{(2k+1)!}$$

Definieren Sie eine Haskell Funktion **roughlyPI**, die bei Eingabe einer natürlichen Zahl **k** die Seriensumme von **0** bis zum **k**-Wert berechnet.

Anwendungsbeispiel:

roughlyPI 1000 => 3.1415926535897922

Lösung:

```
roughlyPI :: Double -> Double
roughlyPI 0 = 2
roughlyPI n = e1/e2 + (roughlyPI (n-1))
                where
                  e1 = 2**(n+1)*(factorial n)**2
                  e2 = factorial (2*n +1)

                  factorial 0 = 1
                  factorial n = n * factorial (n-1)
```

3. Aufgabe (4 Punkte)

Schreiben Sie eine **rekursive** Funktion, die einen Text als Argument bekommt und alle Zeichen, die nicht Klammern sind, aus dem Text entfernt.

Anwendungsbeispiel:

onlyParenthesis "[(2+7.0)*a-(xyz), {word}]" => „[()(){}]"

Lösung 1):

```
onlyParenthesis :: [Char] -> [Char]
onlyParenthesis [] = []
onlyParenthesis (c:cs) | isParenthesis c = c:(onlyParenthesis cs)
                       | otherwise = onlyParenthesis cs

isParenthesis :: Char -> Bool
isParenthesis c = c=='(' || c==')' || c=='[' || c==']' || c=='{' || c=='}'
```

Lösung 2):

```
onlyParenthesis :: [Char] -> [Char]
onlyParenthesis [] = []
onlyParenthesis (c:cs) | isIn pList c = c:(onlyParenthesis cs)
                       | otherwise = onlyParenthesis cs
                                     where
                                     pList = "()[]{}"
                                     isIn [] c = False
                                     isIn (p:ps) c = c==p || isIn ps c
```

Lösung 3): (Richtig! Aber in diesem Übungsblatt noch nicht als Lösung erlaubt, weil eine explizite rekursive Lösung in der Aufgabenstellung gefordert wurde. D.h. noch keine Listengeneratoren waren erlaubt).

```
onlyParenthesis :: [Char] -> [Char]
onlyParenthesis cs = [c | c <- cs, elem c "()[]{}"]
```

4. Aufgabe (4 Punkte)

Eine Hexagonalzahl ist eine Zahl der Form $2n^2 - n$. Schreiben Sie eine **rekursive** Haskell-Funktion **hexagonalNums**, die bei Eingabe einer natürlichen Zahl ***n*** die ersten ***n*** Hexagonalzahlen in einer Liste zurückgibt.

Anwendungsbeispiel:

hexagonalNums 9 ⇒ [0, 1, 6, 15, 28, 45, 66, 91, 120, 153]

Lösung:

```
hexagonalNums :: Integer -> [Integer]
hexagonalNums n = hexaNums 0
        where
        hexaNums k | k>n =  []
                | otherwise =  (2*k*k - k):hexaNums (k+1)
```

Lösung 2): (Richtig! Aber entspricht nicht der Aufgabenstellung einer rekursiven Funktion zu definieren)

```
hexagonalNums :: Integer -> [Integer]
hexagonalNums n = [ 2*h*h-h | h <- [0..n]]
```

5. Aufgabe (5 Punkte)

Schreiben Sie eine **rekursive** Haskell-Funktion, die aus einer Liste von Zahlen den Durchschnitt aller Zahlen, die innerhalb des Intervalls [a, b] liegen, berechnet.

Anwendungsbeispiel:

averageInInterval 2 5 [2.0, 3.0, 5.0, 1.0, 0.0, 1.0] => 3.333

Lösung:

```
averageInInterval :: Double -> Double -> [Double] -> Double
averageInInterval a b xs = aveInt 0 0 xs
        where
        aveInt 0 0 [] = error "no average without numbers..."
        aveInt ave n [] = ave
        aveInt ave n (y:ys) | y>=a && y<= b = aveInt ((ave*n+y)/(n+1)) (n+1) ys
                            | otherwise = aveInt ave n ys
```

Lösung 2): (Richtig! Aber entspricht nicht der Aufgabenstellung einer rekursiven Funktion zu definieren)

```
averageInInterval :: Double -> Double -> [Double] -> Double
averageInInterval a b xs | lenOfList>0 = (sum numsInInt)/(fromIntegral(lenOfList))
                         | otherwise   = error "no posible average without numbers"
                                  where
                                    numsInInt = [x | x <- xs, x>=a, x<=b]
                                    lenOfList = length numsInInt
```

6. Aufgabe (4 Punkte)

Schreiben Sie eine Funktion, die bei Eingabe einer positiven ganze Zahl die Einsen der Binärstellung der Zahl addiert (Quersumme der Binärdarstellung der Zahl berechnet).

Lösung:

```
qsumeBin :: Int -> Int
qsumeBin 0 = 0
qsumeBin n = (mod n 2) + (qsumeBin (div n 2))
```

7. Aufgabe (4 Punkte)

Definieren Sie eine Haskell-Funktion, die bei Eingabe einer Zahl in Hexadezimal-Darstellung die Oktal-Darstellung der Zahl berechnet.

Die Zahl soll als Zeichenkette eingegeben werden.

Anwendungsbeispiel:

hex2okt "1F81F8" => „07700770"

Lösung:

```
hex2Okt :: [Char] -> [Char]
hex2Okt xs = bin2Oct (completeBits (hex2Bin xs))
             where
             completeBits bs | mod3>0    = (replicate (3 - mod3) '0') ++ bs
                             | otherwise = bs
                                           where
                                           mod3 = mod (length bs) 3
             hex2Bin [] = []
             hex2Bin (h:hs) = (look h hex2binTable) ++ (hex2Bin hs)
             bin2Oct [] = []
             bin2Oct (b1:b2:b3:bs) = (look (b1:b2:b3:[]) bin2octTable) : (bin2Oct bs)

look :: Eq a => a -> [(a, b)] -> b
look a [] = error "nothing found "
look a ((x,y):xs) | a==x = y
                  | otherwise = look a xs

hex2binTable :: [(Char, [Char])]
hex2binTable = [('0',"0000"), ('1',"0001"),('2',"0010"),('3',"0011"),
                ('4',"0100"), ('5',"0101"),('6',"0110"),('7',"0111"),
                ('8',"1000"), ('9',"1001"),('A',"1010"),('B',"1011"),
                ('C',"1100"), ('D',"1101"),('E',"1110"),('F',"1111")]

bin2octTable :: [([Char], Char)]
bin2octTable = [("000",'0'),("001",'1'),("010",'2'),("011",'3'),
                ("100",'4'),("101",'5'),("110",'6'),("111",'7')]
```

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.**