



Vorsemesterkurs Informatik

Sommersemester 2023

Aufgabenblatt Nr. 6b

Aufgabe 1 (Listen)

- a) Implementieren Sie in Haskell eine Funktion `viertes :: [a] -> a`, die das vierte Element einer Liste liefert. Hat die Eingabeliste weniger als vier Elemente, so soll eine Fehlermeldung generiert werden.

Lösung

```
viertes (x1:x2:x3:x4:xs) = x4
viertes _                 = error "nicht genug Elemente"

-- mit vordefinierten Funktionen
viertes' xs = if length xs < 4 then error "nicht genug Elemente"
              else head (tail (tail (tail xs)))
```

- b) Implementieren Sie in Haskell eine Funktion `ohneLetztes :: [a] -> [a]`, die für eine Liste der Länge n die ersten $n - 1$ Elemente der Liste (als Liste) liefert. Für den Fall $n = 0$ soll `ohneLetztes` eine Fehlermeldung ausgeben.

Lösung

```
-- rekursiv
ohneLetztes [] = error "ohneLetztes auf leere Liste angewendet"
ohneLetztes [x] = []
ohneLetztes (x:xs) = x:(ohneLetztes xs)

-- mit length und take
ohneLetztes [] = error "ohneLetztes auf leere Liste angewendet"
ohneLetztes xs = take ((length xs)-1) xs
```

- c) Implementieren Sie in Haskell eine Funktion `swap3 :: [a] -> [a]`, die als Eingabe eine Liste erhält und je drei nebeneinander liegende Elemente umdreht. Z.B. soll `vertausche [1,2,3,4,5,6,7,8]` als Ergebnis die Liste `[3,2,1,6,5,4,7,8]` liefern.

Lösung

```
swap3 (x:y:z:xs) = z:y:x:(swap3 xs)
swap3 xs         = xs
```

Aufgabe 2 (Listen und Tupel)

Die Zuckerraffinerie Rohr & Rübe hat verschiedene Lieferungen Zuckerrüben und Zuckerrohr bekommen. Die Daten einer Lieferung bestehen aus dem Paar

(Menge in Tonnen, Preis pro Tonne in EUR)

Definieren Sie in Haskell eine Funktion, die als Eingabe eine Liste von Lieferungen erhält und den Durchschnittspreis pro Tonne Zuckerrohstoff berechnet.

Hinweis: Definieren Sie zunächst eine Funktion, die das Paar (Gesamtmenge, Gesamtwert) berechnet.

Lösung

```
gesamt paar [] = paar
gesamt (gm,gw) ((m,w):xs) = gesamt (gm+m,gw+m*w) xs

avg_preis_pro_tonne xs = let (a,b) = gesamt (0,0) xs in b/a
oder kürzer:

gewMittel = (\(a,w) -> w/a) .
            foldr (\(anz1,wert1) (anz2,wert2) -> (anz1+anz2, anz1*wert1 + wert2)) (0,0)
```

Aufgabe 3 (Strings, Rekursion)

Ein *Palindrom* ist eine Zeichenkette die vorwärts und rückwärtsgelesen das gleiche Wort ergibt, z.B. AN-NA oder RENTNER. Implementieren Sie in Haskell eine *rekursive* Funktion `istPalindrom :: String -> Bool`, die eine Zeichenkette als String erwartet und `True` oder `False` liefert, je nachdem ob die Eingabe ein Palindrom ist oder nicht.

Lösung

```
palindrom [] = True
palindrom [x] = True
palindrom xs
  | head xs == last xs = palindrom (tail (ohneLetztes xs))
  | otherwise = False
```

Aufgabe 4 (Strings)

Implementieren Sie in Haskell eine Funktion, die einen Text als String erhält und alle Zeilen mit Zeilennummern versieht und den nummerierten Text als Ausgabe liefert.

Hinweis: Mit der vordefinierten Funktion `show` können Sie eine Zahl in einen String konvertieren. Die vordefinierten Funktionen `lines :: String -> [String]` und `unlines :: [String] -> String` könnten hilfreich sein.

Lösung

```
nummeriere xs = unlines (nummeriereZeilen (lines xs) 0)
nummeriereZeilen [] i = []
nummeriereZeilen (x:xs) i = (show i ++ x) : (nummeriereZeilen xs (i+1))
```

Aufgabe 5 (Listen und Typen)

Definieren Sie in Haskell je eine Funktion, die den folgenden Typ besitzt:

- [Bool]
- Bool -> [[Integer]]
- [Integer] -> [Integer]
- [[a]] -> a
- Bool -> [Bool -> Bool]

Lösung

Natürlich gibt es keine eindeutige Lösung, d.h. dies sind nur Beispiele:

```
f1 :: [Bool]
f1 = [True,False]
f2 :: Bool -> [[Integer]]
f2 x = if x then [[1],[2,3]] else [[2,3]]
f3 :: [Integer] -> [Integer]
f3 xs = [2*(head xs)]
f4 :: [[a]] -> a
f4 xs = head (concat (concat xs))
f5 :: Bool -> [Bool -> Bool]
f5 b = [(&&) b, (||) b]
```

Ganz schlau (oder faul) könnte man auch definieren

```
f1 :: [Bool]
f1 = undefined
f2 :: Bool -> [[Integer]]
f2 = undefined
f3 :: [Integer] -> [Integer]
f3 = undefined
f4 :: [[a]] -> a
f4 = undefined
f5 :: Bool -> [Bool -> Bool]
f5 = undefined
```

Da undefined den allgemeinsten Typ a hat und daher jeden Typ annehmen kann.

Aufgabe 6 (Türme von Hanoi)

In der Vorlesung wurde ein Haskell-Programm erläutert, das die Zugfolge für das „Türme von Hanoi“-Spiel berechnet (zu finden als Datei `hanoi.hs` im zip-Archiv `code3.zip`). Die Zugfolge ist eine Liste von

Zahlpaaren der Form (x, y) , wobei x und y Zahlen aus der Menge $\{1, 2, 3\}$ sind. Ein solches Paar meint: schiebe die oberste Scheibe vom Stapel x auf den Stapel y .

Ziel der Aufgabe ist es, in Haskell eine Funktion `simuliere` zu implementieren, die einen Zustand des Hanoi-Spiels und eine Zugfolge als Eingaben erhält und die einzelnen Züge simuliert.

Der Zustand des Spiel wird als 3-Tupel $(\text{start}, \text{ziel}, \text{hilf})$ dargestellt, wobei die einzelnen Komponenten die entsprechenden Stapel sind. Ein Stapel selbst wird als Liste von geordneten Zahlen dargestellt. Die Startsituation des „Türme von Hanoi“-Spiels mit einem Turm der Höhe n ist daher das Tupel $(([1, 2, \dots, n], [], [])$, da der Zielstapel und der Hilfsstapel am Anfang leer sind.

Die Ausgabe der Funktion `simuliere` ist die Liste der Zustände (beginnend mit dem Anfangszustand), die erreicht werden.

Der Typ von `simuliere` ist daher

```
([Int], [Int], [Int]) -> [(Int, Int)] -> [[Int], [Int], [Int]]
```

Ein Beispielaufruf für einen Turm der Höhe 3 ist:

```
*> simuliere ([1,2,3], [], []) (start_hanoi 3)
[[[1,2,3], [], []],
 [ [2,3], [1], [] ],
 [ [3], [1], [2] ],
 [ [3], [], [1,2] ],
 [ [], [3], [1,2] ],
 [ [1], [3], [2] ],
 [ [1], [2,3], [] ],
 [ [], [1,2,3], [] ]
]
```

Implementieren Sie `simuliere` in Haskell.

Lösung

Z.B. alle Fälle durchgehen:

```
simuliere (xs,ys,zs) [] = [(xs,ys,zs)]
simuliere (xs,ys,zs) ((1,2):ms) = (xs,ys,zs):(simuliere ((tail xs),((head xs):ys),zs) ms)
simuliere (xs,ys,zs) ((1,3):ms) = (xs,ys,zs):(simuliere ((tail xs),ys,((head xs):zs)) ms)
simuliere (xs,ys,zs) ((2,1):ms) = (xs,ys,zs):(simuliere ((head ys):xs),(tail ys),zs) ms)
simuliere (xs,ys,zs) ((2,3):ms) = (xs,ys,zs):(simuliere xs,(tail ys),((head ys):zs)) ms)
simuliere (xs,ys,zs) ((3,1):ms) = (xs,ys,zs):(simuliere ((head zs):xs),ys,(tail zs)) ms)
simuliere (xs,ys,zs) ((3,2):ms) = (xs,ys,zs):(simuliere xs,((head zs):ys),(tail zs)) ms)
```

Für Spezialisten mit `as-` und `lazy-`Pattern, dafür ohne `head` und `tail`:

```
simuliere' (xxs,yys,zzs) [] = [(xxs,yys,zzs)]
simuliere' (xxs@(x:xs),yys@(y:ys),zzs@(z:zs)) ((1,2):ms) = (xxs,yys,zzs):(simuliere' (xs,x:yys,zzs) ms)
simuliere' (xxs@(x:xs),yys@(y:ys),zzs@(z:zs)) ((1,3):ms) = (xxs,yys,zzs):(simuliere' (xs,yys,x:zzs) ms)
simuliere' (xxs@(x:xs),yys@(y:ys),zzs@(z:zs)) ((2,1):ms) = (xxs,yys,zzs):(simuliere' (y:xxs,ys,zzs) ms)
simuliere' (xxs@(x:xs),yys@(y:ys),zzs@(z:zs)) ((2,3):ms) = (xxs,yys,zzs):(simuliere' (xxs,ys,y:zzs) ms)
simuliere' (xxs@(x:xs),yys@(y:ys),zzs@(z:zs)) ((3,1):ms) = (xxs,yys,zzs):(simuliere' (z:xxs,yys,zs) ms)
simuliere' (xxs@(x:xs),yys@(y:ys),zzs@(z:zs)) ((3,2):ms) = (xxs,yys,zzs):(simuliere' (xxs,z:yys,zs) ms)
```