2. Übung (15.–18. April 2018)

Programmierung

Aufgabe 1 (AGS 12.1.55)

(a) Schreiben Sie eine Funktion `pack :: [Char] -> [[Char]]`, welche in einer Liste aufeinander folgende Wiederholungen des gleichen Werts in einer Teilliste zusammenfasst.
Z.B.: `pack ['a','a','b','b','b','a'] = [['a','a'], ['b','b','b'], ['a']]`.

(b) Schreiben Sie eine Funktion `encode :: [Char] -> [(Int, Char)]`, welche eine Liste lauflängenkodiert.
Z.B.: `encode ['a','a','b','b','b','a'] = [(2, 'a'), (3, 'b'), (1, 'a')]`.

(c) Schreiben Sie eine Funktion `decode :: [(Int, Char)] -> [Char]`, welche eine lauflängenkodierte Liste wieder dekodiert.
Z.B.: `decode [(2, 'a'), (3, 'b'), (1, 'a')] = ['a','a','b','b','b','a']`.

(d) Schreiben Sie eine Funktion `rotate :: [Int] -> Int -> [Int]`, so dass `rotate xs n` die Liste `xs` um `n` nach links rotiert.
Z.B: `rotate [1,2,3,4] 1 = [2,3,4,1]` oder `rotate [1,2,3] (-1) = [3,1,2]`.

Aufgabe 2 (AGS 12.1.53)

(a) Schreiben Sie eine Funktion `unwords :: [String] -> String`, welche eine Liste von Wörtern aneinanderfügt. Die einzelnen Wörter sollen dabei durch ein Leerzeichen voneinander getrennt werden.

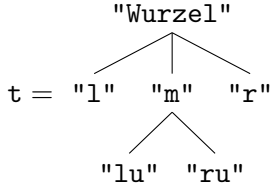
(b) Schreiben Sie eine Funktion `words :: String -> [String]`, welche den Eingabestring in seine Teilwörter zerlegt. Nehmen Sie dabei zur Vereinfachung an, dass Wörter nur durch Leerzeichen voneinander getrennt sind.

Hinweise:

- `String = [Char]`.
- `words` und `unwords` sind in der Prelude-Library bereits definiert. Schließen Sie also diese Funktionen beim Einbinden der Prelude mittels der Direktive `hiding` aus.

Aufgabe 3 (AGS 12.1.49 b–c)

(a) Gegeben sei der algebraische Datentyp `Tree` zur Darstellung von Bäumen, deren Knoten jeweils beliebig viele Kindbäume haben dürfen: `data Tree = Node String [Tree]`. Geben Sie für den rechtsstehenden Baum das entsprechende Element des Typs `Tree` an.



(b) Implementieren Sie die Funktion `level`, die für ein `n :: Int` und ein `t :: Tree` von links nach rechts alle Knoten aus `t` liefert, die `n` Kanten von der Wurzel entfernt sind, z.B.: `level 1 t = ["l", "m", "r"]`.

Zusatzaufgabe 1 (AGS 12.1.10)
Gegeben seien folgende Funktionen:

```
f1:: Int -> [Int]
f1 x = [x] ++ (f1 (x+1))

f2::  [Int] -> Int -> Int
f2 (x:xs) 1 = x
f2 (x:xs) i = f2 xs (i-1)
```

Stellen Sie die Operationsfolgen für den Aufruf `f2 (f1 1) 2` bei Benutzung der Evaluationsstrategien „eager evaluation“ und „lazy evaluation“ dar.

Zusatzaufgabe 2 (AGS 12.1.45 b–c)

(a) Welches Problem fällt Ihnen bei `[Int]` auf, wenn wir das Laufzeitverhalten beim Einfügen von Elementen am Anfang bzw. am Ende der Datenstruktur betrachten?

(b) Implementieren Sie nun mit den Listen von Haskell einen *Queue*-Datentyp. Eine Queue enthält eine Folge von Werten (bei uns: `Int`s) und bietet darüber hinaus Funktionen

- zum Prüfen ob die Queue leer ist:
  `isEmpty :: Queue -> Bool`,
- zum effizienten Einfügen eines Elements ans *Ende* der Queue:
  `enqueue :: Int -> Queue -> Queue`
- zum effizienten Zugriff auf das *Anfangs*element der Queue:
  `first :: Queue -> Int`
- zum Abspalten der Rest-Queue vom ersten Element:
  `rest :: Queue -> Queue`

Sie dürfen dabei die Funktion `reverse` zum Umkehren von Listen verwenden!