STARKER START ins Studium

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Fachbereich Informatik und Mathematik
Institut für Informatik

Vorsemesterkurs Informatik

Sommersemester 2020

Aufgabenblatt Nr. 6A

Aufgabe 1 (Listendarstellung)

- Die Listen-Darstellung $[a_1, \ldots, a_n]$ ist in Haskell nur syntaktischer Zucker. Die eigentliche (interne) Darstellung baut, die Listen rekursiv durch `[]` und `:` auf. D.h. $[a_1, \ldots, a_n]$ wird intern als a_1`:(`a_2`:(...:(`a_n`:[])))` dargestellt.

  Füllen Sie die folgende Tabelle, indem Sie die jeweiligen Listen einmal mit syntaktischem Zucker und einmal in der internen Darstellung aufschreiben. Geben Sie beide Darstellungen auch im GHCi zur Überprüfung ein.

Liste	Darstellung in der Form $[a_1, \ldots, a_n]$	interne Darstellung
Liste, welche die Zahlen von 1 bis 5 enthält	`[1,2,3,4,5]`	`1:(2:(3:(4:(5:[]))))`
Liste, welche die Zahlen von 3 bis 7 enthält		
Liste, die zweimal `True` und zweimal `False` enthält		
Liste, die die Buchstaben `'a'`, `'b'` und `'c'` enthält.		

Lösung

Liste	Darstellung in der Form $[a_1, \ldots, a_n]$	interne Darstellung
Liste, welche die Zahlen von 1 bis 5 enthält	`[1,2,3,4,5]`	`1:(2:(3:(4:(5:[]))))`
Liste, welche die Zahlen von 3 bis 7 enthält	`[3,4,5,6,7]`	`3:(4:(5:(6:(7:[]))))`
Liste, die zweimal `True` und zweimal `False` enthält	`[True,True,False,False]`	`True:(True:(False:(False:[]))`
Liste, die die Buchstaben `'a'`, `'b'` und `'c'` enthält.	`['a','b','c']`	`'a':('b':('c':[]))`

b) Listen können als Elemente auch selbst wieder Listen enthalten. Geben Sie für die folgenden Listen, jeweils die intere Darstellung an:

Darstellung in der Form `[`$a_1, \ldots, a_n$`]`	interne Darstellung
`[[1,2],[3],[4,5]]`	
`[[[1]],[[2],[3]]]`	

Lösung

`[[1,2],[3],[4,5]]`	`(1:2:[]):(3:[]):(4:5:[]):[]`
`[[[1]],[[2],[3]]]`	`((1:[]):[]) : ((2:[]):(3:[]):[]) : []`

c) Der *Typ* einer Liste hat immer die Form `[`t`]`, wobei t der Typ der Listenelemente ist. Z.B. hat die Liste `[1,2,3]` daher den Typ `[Integer]` und die Liste `[[True],[False]]` hat den Typ `[[Bool]]`.

Geben Sie jeweils Beispiellisten an, die den vorgegebenen Typ haben. Testen Sie Ihre Antwort, indem Sie im GHCi `:type` *ihre Liste*`::`*vorgegebener Typ* eingeben und prüfen, ob der GHCi die Eingabe ohne Fehler akzeptiert.

- `Bool`
- `[Bool]`
- `[[[Integer]]]`
- `[Integer -> Bool]`

Lösung

```
True::Bool :: Bool
Prelude> :type [True,False]::[Bool]
[True,False]::[Bool] :: [Bool]
Prelude> :type [[[1,2],[3,4]],[[5,6],[7]]]::[[[Integer]]]
[[[1,2],[3,4]],[[5,6],[7]]]::[[[Integer]]] :: [[[Integer]]]
Prelude> :type [even,odd]::[Integer -> Bool]
[even,odd]::[Integer -> Bool] :: [Integer -> Bool]
```

Aufgabe 2 (Listenfunktionen)

a) Implementieren Sie in Haskell eine Funktion `zweites :: [a] -> a`, die das zweite Element einer Liste liefert. Hat die Eingabeliste weniger als zwei Elemente, so soll eine Fehlermeldung generiert werden. Beachten Sie, dass eine Liste mit mindestens zwei Elementen aufgebaut ist als $(a_1 : (a_2 : \text{rest}))$, wobei a_1 das erste und a_2 das zweite Element ist. Mit einem solchen *Pattern* können Sie daher auf das zweite Element zugreifen.

Lösung

```
zweites (x:y:xs) = y
zweites _        = error "nicht genuegend Elemente"

-- mit vordefinierten Funktionen
zweites' xs = if length xs < 2 then error "nicht genuegend Elemente"
              else head (tail xs)
```

b) Implementieren Sie in Haskell eine Funktion `ersteVier :: [a] -> [a]`, die für eine Liste mit mindestens vier Elementen die ersten vier Elemente als Liste zurück liefert.

Tipp: Sie können die Funktion mithilfe von `take` implementieren oder durch Verwendung von Pattern.

Lösung

```
ersteVier = take 4
ersteVier' (x1:x2:x3:x4:xs) = [x1,x2,x3,x4]
ersteVier' _ = error "zu wenige Elemente
```

c) Implementieren Sie in Haskell eine Funktion `diffR`, die eine Liste von Zahlen erwartet und die Zahlen voneinander rechtsgeklammert abzieht, d.h. `diffR` $[a_1, \ldots, a_n]$ soll $(a_1 - (a_2 - (a_3 - (\ldots (a_{n-1} - a_n)))))$ berechnen. Z.B. ergibt `diffR [5,4,3,2,1]` als Ergebnis 3, denn $5-(4-(3-(2-1))) = 5-(4-(3-1)) = 5-(4-2) = 5-2 = 3$.

Tipp: Verwenden Sie Rekursion. Der Rekursionsanfang ist die leere Liste `[]`. Hier wissen wir bereits, dass `diffR []` gerade 0 ergibt. Für den Rekursionsschritt lassen Sie $(a_2 - (a_3 - (\ldots (a_{n-1} - a_n))))$ durch den rekursiven Aufruf berechnen und müssen dieses Ergebnis im Anschluss von a_1 abziehen.

Lösung

```
diffR [] = 0
diffR (x:xs) = x - (diffR xs)
```

d) Implementieren Sie in Haskell eine Funktion `diffL`, die eine Liste von Zahlen erwartet und die Zahlen voneinander linksgeklammert abzieht, d.h. `diffL` $[a_1, \ldots, a_n]$ soll $(((a_1 - a_2) - a_3) - \ldots) - a_n$ berechnen.

Tipp: Schreiben Sie zunächst eine Hilfsfunktion `diffLAccu`, die neben der Liste von Zahlen noch eine Zahl r als Argument erhält. Die Zahl r stellt gerade das bisher berechnete Zwischenergebnis dar. Z.B. nach der Verarbeitung der ersten drei Listenelemente ist r gerade das Ergebnis von $(a_1 - a_2) - a_3$. Für die Implementierung von `diffL` rufen Sie dann `diffLAccu` mit der Liste ohne erstes Listenelement und mit dem ersten Listenelement als Anfangswert für r auf.

Lösung

```
diffLAccu []     r = r
diffLAccu (x:xs) r = diffLAccu xs (r-x)
diffL (x:xs)       = diffLAccu xs x
diffL []           = 0
```

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

Tipp: Verwenden Sie Rekursion, wobei

- der Rekursionsanfang gerade Listen mit weniger als zwei Elementen sind. In diesem Fall geben Sie die Eingabeliste ohne Veränderung zurück.
- der Rekursionsschritt darin besteht, die *ersten beiden* Listenelemente zu vertauschen und die restlichen Vertauschungen ab dem dritten Element dem rekursiven Aufruf zu überlassen.

Lösung

```
vertausche (x:y:xs) = y:x:(vertausche xs)
vertausche xs       = xs
```

Aufgabe 3 (Strings)

Ein String ist eine Liste von Zeichen und kann daher genau wie andere Listen behandelt werden.

a) Implementieren Sie eine Funktion `anzahlA :: String -> Integer`, die einen String als Eingabe erhält und zählt wie oft der Buchstabe `'A'` im String enthalten ist.

Wie in den vorherigen Aufgaben sollten Sie Rekursion verwenden: Die leere Liste enthält gar keine Zeichen und daher wissen wir, dass `anzahlA` in diesem Fall immer 0 als Ergebnis liefert. Der Rekursionsschritt besteht darin, die Anzahl der 'A's für die Liste ohne das erste Element durch einen rekursiven Aufruf berechnen zu lassen und anschließend 1 oder 0 dazu zu addieren, jenachdem, ob das erste Zeichen ein `'A'` ist oder nicht.

Lösung

```
anzahlA [] = 0
anzahlA (x:xs) = (anzahlA xs) + (if x == 'A' then 1 else 0)
```

b) Implementieren Sie eine Funktion `alleUVWGross :: String -> String`, die einen String als Eingabe erhält und alle Vorkommen von `'u'`, `'v'` und `'w'` durch den entsprechenden Großbuchstaben `'U'`, `'V'` bzw. `'W'` ersetzt.

Tipp: Laufen Sie rekursiv durch die Liste von Zeichen.

Lösung

```
alleUVWGross [] = []
alleUVWGross ('u':xs) = 'U':(alleUVWGross xs)
alleUVWGross ('v':xs) = 'V':(alleUVWGross xs)
alleUVWGross ('w':xs) = 'W':(alleUVWGross xs)
alleUVWGross (x:xs)   = x:(alleUVWGross xs)
```