

tut02 - Freitag

Freitag, 17. April 2020 16:20



tut02

PROGRAMMIERUNG

ÜBUNG 2: LISTEN, ZEICHENKETTEN & BÄUME

Eric Kunze

eric.kunze@mailbox.tu-dresden.de

HINWEISE

- ▶ Übungsaufgaben können **freiwillig** abgegeben werden
- ▶ keine Bonuspunkte
- ▶ Abgaben in möglichst einer Datei und (solange es geht) nur Quelltexte

- ▶ Hinweise meinerseits nur noch auf meiner Website
oakoneric.github.io
- ▶ Fragen und Anmerkungen jederzeit
- ▶ einfache Fragen (Orga oder kleine Inhalte) gern schnell und unkompliziert über Telegram

Probleme aus der vergangenen Woche?

Listen & Zeichenketten in Haskell

LISTEN

Listen Wenn a ein Typ ist, dann bezeichnet [a] den Typ
"Liste mit Elementen vom Typ a", insbesondere haben
alle Elemente einer Liste den gleichen Typ

$[[\text{Int}]]$ z.B. $[[1,2,3,4],[2,4,6]]$

$[\text{Bool}]$ z.B. $[\text{True}, \text{False}, \text{True}]$

$[1, 3, \text{True}, "S"] \rightarrow \text{falsch}$

Int Bool String

2

LISTEN

Listen Wenn a ein Typ ist, dann bezeichnet [a] den Typ
"Liste mit Elementen vom Typ a", insbesondere haben
alle Elemente einer Liste den gleichen Typ

cons-Operator " : " Trennung von *head* und *tail* einer Liste

$[x_1, x_2, x_3, x_4, x_5] = x_1 : [x_2, x_3, x_4, x_5]$

(i) leere Liste: []

(ii) mind. 1 Element: $(x : xs)$

Int $x : xs$
 [Int]

2

LISTEN

Listen Wenn a ein Typ ist, dann bezeichnet [a] den Typ "Liste mit Elementen vom Typ a", insbesondere haben alle Elemente einer Liste den gleichen Typ

cons-Operator " :: " Trennung von *head* und *tail* einer Liste

[x₁ , x₂ , x₃ , x₄ , x₅] = x₁ : [x₂ , x₃ , x₄ , x₅]

Verkettungsoperator " ++ " Verkettung zweier Listen

gleichen Typs

[x₁ , x₂] ++ [x₃ , x₄ , x₅] = [x₁ , x₂ , x₃ , x₄ , x₅]

[Int] ++ [Int]

[3 , 5 , 8] ++ [9] = [3 , 5 , 8 , 9]

2

ZEICHEN & ZEICHENKETTEN

Zeichen

- ▶ Datentyp Char
- ▶ Eingabe in einfachen Anführungszeichen
- ▶ z.B. 'a', 'e', '3'

3

ZEICHEN & ZEICHENKETTEN

Zeichen

- ▶ Datentyp Char
- ▶ Eingabe in einfachen Anführungszeichen
- ▶ z.B. 'a', 'e', '3'

Zeichenketten

- ▶ Datentyp String = [Char]
- ▶ Eingabe in doppelten Anführungszeichen
- ▶ z.B. "hallo", "welt"
- ▶ Konkatenation von Zeichenketten:

```
"hallo" ++ "welt" = "hallo_welt"
"hallo" = 'h' : "allo"
          Char
```

3

ÜBUNG: DATENTYPEN

Bestimme den Datentyp:

► "prog" String = [Char]
► 5 Int

► 3 Char

► mod (gemeint ist die Modulo-Funktion)

► [2,4,6,8] [Int]

► [[a, "r", "i", "c"], [k, "u", "n", "z", "e"]]

~~[[Char], [Char]]~~

~~[[Int]]~~

[String]

= ~~[[[Char]]]~~

Argumente

Rückgabe

/

(Int → Int) → Int
(Int, Int) → Int

4

mod :: Int → (Int → Int)
mod m

(mod 5) 3

mod 5 :: Int → Int

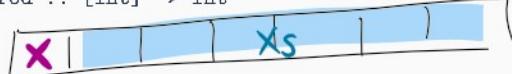
Übungsblatt 2

Aufgabe 1

AUFGABE 1 – TEIL (A)

Multiplikation einer Liste

prod :: [Int] → Int



$$(1) \text{ prod } [] = 1$$

$$(2) \text{ prod } (x:xs) = x * \text{prod } xs$$

Bsp.:

$$\text{prod } [2, 3, 4] \stackrel{(2)}{=} 2 * \text{prod } [3, 4] \quad (3) = 4 : []$$

$$\stackrel{2: [3, 4]}{=} 2 * 3 * 4 * \text{prod } []$$

$$\stackrel{(1)}{=} 2 * 3 * 4 * 1$$

↳ neutrale Element

5

AUFGABE 1 – TEIL (A)

Multiplikation einer Liste

prod :: [Int] -> Int

```
1 -- (a) Produkt der Listenelemente
2 prod :: [Int] -> Int
3 prod []      = 1
4 prod (x:xs) = x * prod xs
```

5

AUFGABE 1 – TEIL (B)

Umkehrung einer Liste

rev :: [Int] -> [Int]

$$\text{rev } [] = [] \quad (\text{Basisfall})$$
$$\text{rev } (x:xs) = \text{rev } xs ++ [x] \quad (\text{Rek-Fall})$$

Int [Int]

6

AUFGABE 1 – TEIL (B)

Umkehrung einer Liste

rev :: [Int] -> [Int]

```
1 rev :: [Int] -> [Int]
2 rev []      = []
3 rev (x:xs) = [rev xs] ++ [x]
```

$$\begin{aligned} \text{rev } [1,2,3] &\stackrel{3}{=} (\text{rev } [2,3]) ++ [1] \\ 1 : [2,3] &\stackrel{2}{=} (\text{rev } [3]) ++ [2] ++ [1] \\ &\stackrel{3}{=} ([\underbrace{\text{rev } [3]}_{[3]}] ++ [2]) ++ [1] \\ \text{func } (-1) &\stackrel{2}{=} (\underbrace{([])}_{[3]} ++ [3]) ++ [2] ++ [1] \\ &\quad \underbrace{[3]}_{[3,2]} \end{aligned}$$

6

AUFGABE 1 – TEIL (C)

Elemente einer Liste löschen

excl :: Int -> [Int] -> [Int]

↑ ↑ ↑
 | |
 | Ergebnis

`excl :: Int -> [Int] -> [Int]`

n | | | | | |

excl n (x:xs) 

$n == x = \text{excl } n \text{ } xs$

otherwise = x: excl n xs

otherwise $= x$: excl n xs

| otherwise = x: excl n xs

7

AUFGABE 1 – TEIL (C)

Elemente einer Liste löschen

```
excl :: Int -> [Int] -> [Int]
```

```

1 excl :: Int -> [Int] -> [Int]
2 excl _ [] = []
3 excl n (x:xs)
4   | x /= n      = x : excl n xs
5   | otherwise    = excl n xs

```

7

AUFGABE 1 – TEIL (D)

Sortierung einer Liste prüfen

isOrd :: [Int] -> Bool

x y ks

$\leq \rightarrow$ ja : teste ($y:xs$)

→ nein: falsch
→ ja: TRUE

isOrd [] = True
isOrd [x] = True

$\text{isOrd } (x:y:xs) = x <= y \quad \& \quad \text{isOrd } (y:xs)$

$[1, 3, 2, 4]$ richtig sortiert
 $x \leq y \rightarrow [3, 2, 4] \rightarrow$ richtig

log, und
^

8

AUFGABE 1 – TEIL (D)

Sortierung einer Liste prüfen

isOrd :: [Int] -> Bool

```
1 isOrd :: [Int] -> Bool
2 isOrd [] = True
3 isOrd [x] = True
4 isOrd (x:y:xs)
5   | x <= y = isOrd (y:xs)
6   | otherwise = False
```

8

AUFGABE 1 – TEIL (D)

Sortierung einer Liste prüfen

isOrd :: [Int] -> Bool

```
1 isOrd :: [Int] -> Bool
2 isOrd [] = True
3 isOrd [x] = True
4 isOrd (x:y:xs)
5   | x <= y = isOrd (y:xs)
6   | otherwise = False
```

```
1 isOrd' :: [Int] -> Bool
2 isOrd' [] = True
3 isOrd' [x] = True
4 isOrd' (x:y:xs) = x <= y && isOrd' (y:xs)
```

True/False

True && False && ... && False → False

8

AUFGABE 1 – TEIL (E)

sortiertes Zusammenfügen zweier (sortierten) Listen

merge :: [Int] -> [Int] -> [Int]



Vergleich $x \leq y$? → ja : nehme x und merge xs ($y:ys$)
 $(y < x)$ → nein nehme y und merge $(x:xs)$ ys

merge $xs@ (x:xs) ys@ (y:ys)$
| $x \leq y = x$: merge xs ($y:ys$)
| otherwise = y : merge $(x:xs) ys$

9

AUFGABE 1 – TEIL (E)

sortiertes Zusammenfügen zweier (sortierten) Listen

merge :: [Int] -> [Int] -> [Int]

```
1 merge :: [Int] -> [Int] -> [Int]
2 merge [] ys = ys
3 merge xs [] = xs
4 merge (x:xs) (y:ys)
5   | x < y      = x : merge xs (y:ys)
6   | otherwise    = y : merge (x:xs) ys
```

9

AUFGABE 1 – TEIL (E)

sortiertes Zusammenfügen zweier (sortierten) Listen

merge :: [Int] -> [Int] -> [Int]

```
1 merge :: [Int] -> [Int] -> [Int]
2 merge [] ys = ys
3 merge xs [] = xs
4 merge (x:xs) (y:ys)
5   | x < y      = x : merge xs (y:ys)
6   | otherwise    = y : merge (x:xs) ys
```

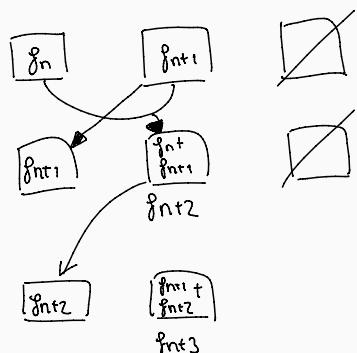
```
1 merge' :: [Int] -> [Int] -> [Int]
2 merge' [] ys = ys
3 merge' xs [] = xs
4 merge' xxss@(x:xs) yyss@(y:ys)
5   | x < y      = x : merge' xs yyss
6   | otherwise    = y : merge' xxss ys
```

9

AUFGABE 1 – TEIL (F)

(unendliche) Liste der Fibonacci-Zahlen

fibs :: [Int]



10

AUFGABE 1 – TEIL (F)

(unendliche) Liste der Fibonacci-Zahlen

fibs :: [Int]

```
1 fibs :: [Int] 1
2 fibs = fibs' 1
3   where fibs' n m = n : fibs' m (n+m)
```

take 7 fibs

10

AUFGABE 1 – TEIL (F)

(unendliche) Liste der Fibonacci-Zahlen

fibs :: [Int]

```
1 fibs :: [Int]
2 fibs = fibs' 0 1
3   where fibs' n m = n : fibs' m (n+m)
```

```
1 fib :: Int -> Int
2 fib 0 = 1
3 fib 1 = 1
4 fib n = fib (n-1) + fib (n-2)
5
6 fibs :: [Int]
7 fibs = fibAppend 0
8   where fibAppend x = fib x : fibAppend (x+1)
```

10

Übungsblatt 2

Aufgabe 2

AUFGABE 2 – TEIL (A)

Liste von Wörtern aneinanderfügen

join :: [String] -> String

["Hallo", "Welt"] ~~~ " HalloWelt"



x + + " " + + join xs

11

AUFGABE 2 – TEIL (A)

Liste von Wörtern aneinanderfügen

join :: [String] -> String

```
1 join :: [String] -> String
2 join [] = ""  
3 join [x] = x
4 join (x:xs) = x ++ ' ' : join xs
```

11

AUFGABE 2 – TEIL (B)

Trennung eines Strings in seine Wörter

unjoin :: String -> [String]

str = " Hallo Welt"
c —————↑cs —————
c ——————cs—g—

↓ speicher [] = [speicher]

↓ speicher (c:cs)
| c == ' ' = speicher : ↓ [] cs |

| otherwise = ↓ (speicher ++ [c]) cs |

12

AUFGABE 2 – TEIL (A)

Trennung eines Strings in seine Wörter

```
unjoin :: String -> [String]
```

```
1 unjoin :: String -> [String]
2 unjoin s = f [] s
3 where
4   f save [] = [save]
5   f save (c:cs)
6     | c == ' ' = save : f [] cs
7     | otherwise = f (save ++ [c]) cs
```

12

AUFGABE 2 – TEIL (A)

Trennung eines Strings in seine Wörter

```
unjoin :: String -> [String]
```

```
1 unjoin' :: String -> [String]
2 unjoin' [] = []
3 unjoin' [c] = if c == ' ' then [[]] else [[c]]
4 unjoin' (c:cs)
5   | c == ' ' = [] : unjoin' cs
6   | otherwise = let (s:ss) = unjoin' cs
                 in ((c:s):ss)
```

12

Übungsblatt 2

Aufgabe 3

ALGEBRAISCHE DATENTYPEN

- Ziel: problemspezifische Datenkonstruktoren
- z.B. in C: Aufzählungstypen
- funktionale Programmierung: algebraische Datentypen

Aufbau:

```
1 data Typename
2   = Con1 t11 ... t1k1
3   | Con2 t21 ... t2k2
4   | ...
5   | Conr tr1 ... trkr
```

- Typename ist ein Name (Großbuchstabe)
- Con1, ... Conr sind Datenkonstruktoren (Großbuchstabe)
- tij sind Typnamen (Großbuchstaben)

13

ALGEBRAISCHE DATENTYPEN – BEISPIELE

```
1 data Typename
2   = Con1 t11 ... t1k1
3   | Con2 t21 ... t2k2
4   | ...
5   | Conr tr1 ... trkr
```

```
1 data Season = Spring | Summer | Autumn | Winter
```

```
1 goSkiing :: Season -> Bool
2 goSkiing Winter = True
3 goSkiing _       = False
```

```
1 data TriBool = TriTrue | TriMaybe | TriFalse
```

14

AUFGABE 3

```
1 data BinTree = Branch Int BinTree BinTree | Nil
```

15

AUFGABE 3

```
1 data BinTree = Branch Int BinTree BinTree | Nil  
  
2 tree1 :: BinTree -- Suchbaum  
3 tree1 = Branch 5  
4   ( Branch 3  
5     (Branch 2 Nil Nil)  
6     (Branch 4 Nil Nil)  
7   )  
8   Branch 8  
9     ( Branch 7  
10    (Branch 6 Nil Nil)  
11    (Nil)  
12  )  
13  ( Branch 10  
14    (Nil)  
15    (Branch 13 Nil Nil)  
16  )  
17 )
```

15

AUFGABE 3 – TEIL (A)

Einfügen von Schlüsseln in einen Binärbaum

```
data BinTree = Branch Int BinTree BinTree | Nil  
insert :: BinTree -> [Int] -> BinTree
```

16

AUFGABE 3 – TEIL (A)

Einfügen von Schlüsseln in einen Binärbaum

```
data BinTree = Branch Int BinTree BinTree | Nil  
insert :: BinTree -> [Int] -> BinTree  
  
1 insert :: BinTree -> [Int] -> BinTree  
2 insert t      [] = t  
3 insert t (x:xs) = insert t' xs  
4   where  
5     t' = insertSingle t x  
6     insertSingle Nil           x = Branch x Nil Nil  
7     insertSingle (Branch y l r) x  
8       | x < y    = Branch y (insertSingle l x) r  
9       | otherwise = Branch y l           (insertSingle r x)
```

16

AUFGABE 3 – TEIL (B)

Test auf Baum-Gleichheit

```
data BinTree = Branch Int BinTree BinTree | Nil  
equal ::
```

17

AUFGABE 3 – TEIL (B)

Test auf Baum-Gleichheit

```
data BinTree = Branch Int BinTree BinTree | Nil  
equal :: BinTree -> BinTree -> Bool
```

```
1 equal :: BinTree -> BinTree -> Bool  
2 equal Nil           Nil          = True  
3 equal Nil           (Branch y l2 r2) = False  
4 equal (Branch x l1 r1) Nil      = False  
5 equal (Branch x l1 r1) (Branch y l2 r2)  
6   = (x == y) && (equal l1 l2) && (equal r1 r2)
```

17

ENDE

Fragen?

18