

# tut03 - Freitag

Freitag, 24. April 2020 15:25



tut03

## PROGRAMMIERUNG

### ÜBUNG 3: BÄUME & FUNKTIONEN HÖHERER ORDNUNG

---

Eric Kunze  
[eric.kunze@mailbox.tu-dresden.de](mailto:eric.kunze@mailbox.tu-dresden.de)

## 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)

1

## ALGEBRAISCHE DATENTYPEN - BEISPIELE

```
1 data Typename
2   = Con1 t11 ... t1k1
3   | Con2 t21 ... t2k2
4   | ...
5   | Conr tr1 ... trkr
      Con1   Con2   Con3   Con4
1 data Season = Spring | Summer | Autumn | Winter
1 goSkiing :: Season -> Bool
2 goSkiing Winter = True ←
3 goSkiing _       = False
1 data TriBool = TriTrue | TriMaybe | TriFalse
```

2

## AUFGABE 3

```
1 data BinTree = Branch Int BinTree BinTree | Nil
      ↑           Name          Knoten          "Blatt" / leerer Baum
      ↓           ↓             ↓             ↓
      Int          BinTree     BinTree        Nil
```

Diagramm eines Binären Baums:

- Ein Knoten ist entweder ein Int (Blatt) oder zwei BinTree.
- Ein BinTree ist entweder ein Blatt (Nil) oder ein Branch mit einem Int-Wert und zwei Kind-BinTrees.

Handgeschriebene Beispiele:

- tcastTree :: BinTree
- tcastTree = Branch 5 (Branch 3 Nil Nil)
- (Branch 8 Nil (Branch 12 Nil Nil))

Handgezeichnete Baumstruktur:

```
graph TD
    Root[5] --- Node1[3]
    Root --- Node2[8]
    Node1 --- Node3[12]
```

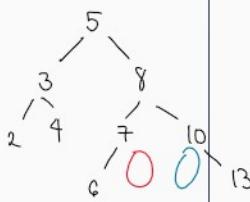
3

## AUFGABE 3

```

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

```



3

## AUFGABE 3 - TEIL (A)

### Einfügen von Schlüsseln in einen Binärbaum

```

data BinTree = Branch Int BinTree BinTree | Nil
insert :: BinTree -> [Int] -> BinTree
insert t [] = t  (nichts einfügen)
insert t (x:xs) = insert (insertSingle t x) xs

```

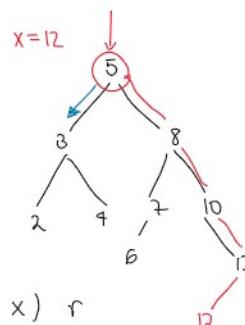
where

```

insertSingle :: BinTree -> Int -> BinTree
insertSingle Nil x = Branch x Nil Nil
insertSingle (Branch y l r) x
  → | x < y  = Branch y (insertSingle l x) r
  → | otherwise = Branch y l (insertSingle r x)

```

4



## 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)

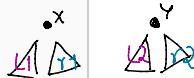
```

4

## AUFGABE 3 - TEIL (B)

### Test auf Baum-Gleichheit

```
data BinTree = Branch Int BinTree BinTree | Nil  
equal :: BinTree → BinTree → Bool  
equal Nil Nil = True  
equal Nil (Branch y l2 r2) = False  
equal (Branch x l1 r1) Nil = False  
equal (Branch x l1 r1) (Branch y l2 r2)  
= (x == y) && equal l1 l2  
&& equal r1 r2
```



5

## AUFGABE 3 - TEIL (B)

### Test auf Baum-Gleichheit

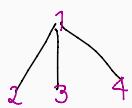
```
data BinTree = Branch Int BinTree BinTree | Nil  
equal :: BinTree → BinTree → Bool
```

```
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)
```

5

## Übungsblatt 3

### Aufgabe 1



Node 1 [Node 2[], Node 3[], ...]

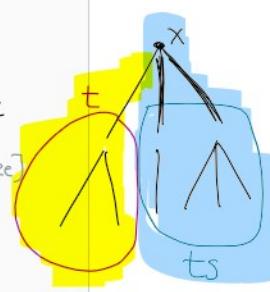
## AUFGABE 1 - TEIL (A)

### Anzahl der Blätter

```

data RoseTree = Node Int [RoseTree]
countLeaves :: RoseTree -> Int
countLeaves (Node [] []) = 1
countLeaves (Node [] t) = countLeaves t
countLeaves (Node x (t:ts)) = countLeaves t +
                                countLeaves (Node x ts)

```



6

## AUFGABE 1 - TEIL (A)

### Anzahl der Blätter

```

data RoseTree = Node Int [RoseTree]
countLeaves :: RoseTree -> Int

```

```

1 -- (a) Blaetter zaehlen
2 countLeaves :: RoseTree -> Int
3 countLeaves (Node [] []) = 1
4 countLeaves (Node [] t) = countLeaves t
5 countLeaves (Node x (t:ts)) = countLeaves t + countLeaves (Node x ts)
6

```

6

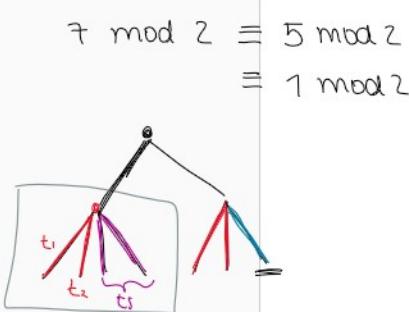
## AUFGABE 1 - TEIL (B)

### gerade Anzahl an Kindern

```

data RoseTree = Node Int [RoseTree]
evenNodes :: RoseTree -> Bool
evenNodes (Node [] []) = True
evenNodes (Node [] t) = False
evenNodes (Node x (t1:t2:ts))
= evenNodes t1 &&
  evenNodes t2 &&
  evenNodes (Node x ts)

```



7

## AUFGABE 1 - TEIL (B)

### gerade Anzahl an Kindern

```
data RoseTree = Node Int [RoseTree]
evenNodes :: RoseTree -> Bool
```

```
1 -- (b) gerade Anzahl an Kindern testen - Variante 1
2 evenNodes :: RoseTree -> Bool
3 evenNodes (Node _ []) = True
4 evenNodes (Node x [t]) = False
5 evenNodes (Node x (t1:t2:ts))
6   = evenNodes (Node x ts) && evenNodes t1 &&
7     evenNodes t2
```

7

## AUFGABE 1 - TEIL (B)

### gerade Anzahl an Kindern

```
data RoseTree = Node Int [RoseTree]
evenNodes :: RoseTree -> Bool
```

```
1 -- (b) gerade Anzahl an Kindern testen - Variante 2
2 evenNodes' :: RoseTree -> Bool
3 evenNodes' (Node _ []) = True ←
→ 4 evenNodes' (Node _ ts)
5   = mod (length ts) 2 == 0 && evenNodes'' ts
6   where
7     evenNodes'' :: [RoseTree] -> Bool
8     evenNodes'' [] = True
9     evenNodes'' (t:ts)
10    = evenNodes' t && evenNodes'' ts
```

7

## Funktionen advanced

## FUNKTIONEN

Wir kennen bereits einige Möglichkeiten Funktionen zu notieren. Hier seien einige weitere erwähnt.

**anonyme Funktionen.** Funktionen ohne konkreten Namen

z.B.  $(\lambda x \rightarrow x+1)$  ist die Addition mit 1

$$(\lambda x \rightarrow x+1) \quad 4 = 5$$

$x \mapsto x+1$   
 $4 \mapsto 4+1=5$

$$(\lambda x \rightarrow x*x) \quad 4 = 16$$

8

## FUNKTIONEN

Wir kennen bereits einige Möglichkeiten Funktionen zu notieren. Hier seien einige weitere erwähnt.

**anonyme Funktionen.** Funktionen ohne konkreten Namen

z.B.  $(\lambda x \rightarrow x+1)$  ist die Addition mit 1

$$(\lambda x \rightarrow x+1) \quad 4 = 5$$

**Operator  $\leftrightarrow$  Funktion** Aus Operatoren (wie z.B. +) kann man eine Funktion machen und vice versa.

► Operator → Funktion: Klammern drum herum

$$\begin{array}{l} 1 (+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\ 2 (+) x y = x + y \end{array}$$

► Funktion → Operator: Backticks `...` `mod`

$$5 `mod` 2 = 1 \quad \text{mod } 5 2 = 1$$

$$3 + 4 = ?$$

↑  
(+) 3 4 = ?

$$3 `(^)` 4 = ?$$

8

## FUNKTIONSKOMPOSITION

Analog zur mathematischen Notation  $f = g \circ h$  für

$f(x) = g(h(x))$  versteht auch Haskell das

Kompositionsprinzip mit dem Operator .

z.B.

$$\begin{array}{ll} 1 \text{ sqAdd} :: \text{Int} \rightarrow \text{Int} & +5 :: \text{Int} \rightarrow \text{Int} \\ 2 \text{ sqAdd} = (^2) . \underline{(+ 5)} & +5 x = x + 5 \end{array}$$

statt  $\text{sqAdd } x = (x + 5)^2$  für das Quadrat des fünften Nachfolgers

$$\begin{array}{ll} g(y) = y^2 & g = ^2 \\ h(x) = x + 5 & h = +5 \end{array}$$

9

## PARTIELLE APPLIKATION

Funktionen müssen nicht immer mit allen Argumenten versorgt werden. Lässt man (hintere) Argumente weg, so spricht man von Unterversorgung. Die Modulo Funktion hat eigentlich zwei Argumente. Lassen wir das zweite Argument weg, so liefert dies uns eine neue Funktion, die noch ein Argument entgegennimmt und sodann die Restberechnung ausführt.

```

1 mod :: Int -> (Int -> Int)
2 mod m(n) = ...
3
4 mod 10 :: Int -> Int
5 (mod 10) n = mod 10 n
6
7 (> 3) :: Int -> Bool
8 (> 3) x = x > 3
9
10 (7>3)5 =True

```

10

## FUNKTIONEN HÖHERER ORDNUNG — MAP

Funktionen können als Argumente von Funktionen auftreten. Wir lernen drei Basics kennen:

$(x : x_5)$				
$x_1$	$x_2$	$x_3$	$x_4$	$1$
$\rightarrow$	$f x_1$	$f x_2$	$f x_3$	$f x_4$

11

## FUNKTIONEN HÖHERER ORDNUNG — MAP

Funktionen können als Argumente von Funktionen auftreten. Wir lernen drei Basics kennen:

## Die Funktion $\text{map}$

- `map` ermöglicht es eine Funktion f auf alle Elemente einer Liste anzuwenden

```
1 map :: (Int -> Int) -> [Int] -> [Int]
2 map f [] = []
3 map f (x:xs) = f x : map f xs
```

► Beispiel square x = x \* x

```
map square [1,2,7,12,3,20] = [1,4,49,144,9,400]
(\x -> x*x)      sq 1 : map sq [2,7,12,3,20]
                  = sq 1 : sq 2 : map sq [7,12,3,20],
```

## FUNKTIONEN HÖHERER ORDNUNG — FILTER

**Die Funktion filter**  $p :: \text{Int} \rightarrow \text{Bool}$   
 $p = \text{odd}$

- $\text{filter } p \text{ xs}$  liefert eine Liste, die genau die Elemente von  $xs$  enthält, welche das Prädikat  $p$  erfüllen

```
1 filter :: (a -> Bool) -> [a] -> [a]
2 filter p xs = [x | x <- xs, p x]
```

$$\{x \mid x \in xs \wedge p(x) = \text{True}\}$$

- Beispiel.

```
1 filter odd [1, 2, 7, 12, 3, 20] = [1, 7, 3]
```

12

## FUNKTIONEN HÖHERER ORDNUNG — FOLDR

**Die Funktion foldr**



- $\text{foldr } f z \text{ xs}$  faltet eine Liste  $xs$  und verknüpft jeweils durch die Funktion  $f$ ; gestartet wird mit  $z$  und dem rechtesten Element

```
1 foldr :: (a -> b -> b) -> b -> [a] -> b
2 foldr f z [] = z
3 foldr f z (x:xs) = f x (foldr f z xs)
```

- Beispiel.

```
1 foldr (+) 3 [1, 2, 3, 4, 5] = 18 ✓
2 length xs = foldr (+) 0 (map (\x -> 1) xs)
```

$$\begin{array}{c} \{1, 2, 3, 4, 5\} \\ \diagdown \quad \diagup \\ 18 = 15 + 3 \end{array} \quad \begin{array}{c} xs [1|1|1|1|1] \\ \diagdown \quad \diagup \\ 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \end{array} \quad 0 \quad 13$$

## FUNKTIONEN HÖHERER ORDNUNG — ÜBERSICHT

- $\text{map}$  wendet Funktion auf alle Listenelemente an

```
1 map :: (a -> b) -> [a] -> [b]
2 map f [] = []
3 map f (x:xs) = f x : map f xs
```

- $\text{filter}$  wählt Listenelemente anhand einer Funktion aus

```
1 filter :: (a -> Bool) -> [a] -> [a]
2 filter p xs = [x | x <- xs, p x]
```

- $\text{foldr}$  faltet eine Liste mit Verknüpfungsfunktion (von rechts beginnend)

```
1 foldr :: (a -> b -> b) -> b -> [a] -> b
2 foldr f z [] = z
3 foldr f z (x:xs) = f x (foldr f z xs)
```

14

## Übungsblatt 3

### Aufgabe 2

#### AUFGABE 2

**Produkt der Quadrate aller geraden Zahlen einer Liste**

$f :: [Int] \rightarrow Int$

$\text{prod } x \cdot y = x * y$

$\text{sq } x = x * x$

$\text{even } x = (x \bmod 2) == 0$

$\text{filter}$  (Prelude)

$f xs = \text{foldr prod 1} (\text{map sq} (\text{filter even xs}))$

(\*)  $(\wedge 2)$

$\underbrace{\text{Liste gerade Zahlen}}$

$\underbrace{\text{Liste der Quadrate gerade Zahlen}}$

$\underbrace{\text{Produkt d. Quadrate gerade Zahlen}}$

$$\begin{aligned} \text{prod } x \cdot y &= x * y \\ \text{even } x &= x \bmod 2 == 0 \\ \text{sq } x &= x * x \end{aligned}$$

15

$$f xs = \text{foldr} \dots$$

#### AUFGABE 2

##### Produkt der Quadrate aller geraden Zahlen einer Liste

$f :: [Int] \rightarrow Int$

```
1 f :: [Int] -> Int
2 f xs
3 = foldr (+) 0 (map (^2) (filter ('mod' 2) == 0) xs))
```

15

## AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

$f :: [Int] \rightarrow Int$

```
1 f :: [Int] -> Int
2 f xs
3 = foldr (+) 0 (map (^2) (filter ('mod' 2) == 0) xs))
```

```
1 f' :: [Int] -> Int
2 f' xs = foldr (*) 1 (map (^2) (filter even xs))
```

15

## AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

$f :: [Int] \rightarrow Int$

$\text{mod } n \text{ m}$

```
1 f :: [Int] -> Int
2 f xs
3 = foldr (+) 0 (map (^2) (filter ('mod' 2) == 0) xs))
```

```
1 f' :: [Int] -> Int
2 f' xs = foldr (*) 1 (map (^2) (filter even xs))
```

```
1 f'' :: [Int] -> Int
2 f'' = foldr (*) 1 . map (^2) . filter even
```

```
1 f''' :: [Int] -> Int
2 f''' = foldr (*) 1 . map (^2) . filter ((== 0) . ('mod' 2))
```

15

## Übungsblatt 3

### Aufgabe 2

### AUFGABE 3

Faltung einer Liste von *links*

foldleft :: ( $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ )  $\rightarrow \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Int}$

neuer Startwert

Startwert | List | Ergebnis

$\text{foldleft } f \ x \ [ ] = x$

$\text{foldleft } f \ x \ (y:ys) = \text{foldleft } f \ (f \ x \ y) \ ys$

$\boxed{\text{foldleft} = \text{foldl}}$

16

### AUFGABE 3

#### Faltung einer Liste von *links*

foldleft :: ( $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ )  $\rightarrow \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Int}$

```
1 foldleft :: ( $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ )  $\rightarrow \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Int}$ 
2 foldleft f x []      = x
3 foldleft f x (y:ys) = foldleft f (f x y) ys
```

16

### ENDE

Fragen?

17