

PROGRAMMIERUNG

ÜBUNG 3: BÄUME & FUNKTIONEN HÖHERER ORDNUNG

Eric Kunze

`eric.kunze@mailbox.tu-dresden.de`

Übungsblatt 2 — Aufgabe 3

Algebraische Datentypen

ALGEBRAISCHE DATENTYPEN

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

Aufbau:

```
data Typename  
    = Con1 t11 ... t1k1  
    | Con2 t21 ... t2k2  
    | ...  
    | Conr tr1 ... trkr
```

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

ALGEBRAISCHE DATENTYPEN – BEISPIELE

```
data Typename
  = Con1 t11 ... t1k1
  | Con2 t21 ... t2k2
  | ...
  | Conr tr1 ... trkr
```

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

ALGEBRAISCHE DATENTYPEN – BEISPIELE

```
data Typename
  = Con1 t11 ... t1k1
  | Con2 t21 ... t2k2
  | ...
  | Conr tr1 ... trkr
```

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

```
goSkiing :: Season -> Bool
goSkiing Winter* = True
goSkiing _       = False
```

ALGEBRAISCHE DATENTYPEN – BEISPIELE

```
data Typename
  = Con1 t11 ... t1k1
  | Con2 t21 ... t2k2
  | ...
  | Conr tr1 ... trkr
```

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

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

```
data Weather = Sunny Int Int Bool | Cloudy Float
              | Rainy String Int
```

AUFGABE 3 – TEIL (A)

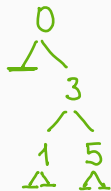
```
data BinTree = Branch (Int) BinTree BinTree | Nil deriving Show
```

↑
3-stellig

↑
0-stellig

```
mytree :: BinTree
```

```
mytree = Branch 0  
  (Nil)  
  (Branch 3  
    (Branch 1 Nil Nil)  
    (Branch 5 Nil Nil)  
  )
```



AUFGABE 3 – TEIL (A)

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

Ein Beispielbaum:

```
mytree :: BinTree
mytree = Branch 0
  ( Nil )
  ( Branch 3
    ( Branch 1 Nil Nil )
    ( Branch 5 Nil Nil )
  )
```

... erfüllt die Suchbaumeigenschaft.

Test auf Baum-Gleichheit

AUFGABE 3 – TEIL (B)

Test auf Baum-Gleichheit

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

$\text{equal Nil Nil} = \text{True}$

$\text{equal (Branch } _ _ _) \text{ Nil} = \text{False}$

$\text{equal Nil (Branch } _ _ _) = \text{False}$

$\text{equal (Branch } x \ l1 \ r1) \text{ (Branch } y \ l2 \ r2) =$
 $x == y \ \&\& \ \text{equal } l1 \ l2 \ \&\& \ \text{equal } r1 \ r2$



Test auf Baum-Gleichheit

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

```
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)
```

AUFGABE 3 – TEIL (C)

Einfügen von Schlüsseln in einen Binärbaum

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

$\text{insertSingle} :: \text{BinTree} \rightarrow \text{Int} \rightarrow \text{BinTree}$

$\text{insertSingle Nil } x = \text{Branch } x \text{ Nil Nil}$ \downarrow $3 = x$

$\text{insertSingle (Branch } y \text{ } l \text{ } r) x$

$| x < y = \text{Branch } y \text{ } (\text{insertSingle } l \text{ } x) \text{ } r$ \swarrow $5 = y$

$| \text{otherwise} = \text{Branch } y \text{ } l \text{ } (\text{insertSingle } r \text{ } x)$

$\text{insert } t \text{ } [] = t$

$\text{insert } t \text{ } (x:xs) = \text{insert } t' \text{ } xs$

 where $t' = \text{insertSingle } t \text{ } x$

Einfügen von Schlüsseln in einen Binärbaum

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

```
insert :: BinTree -> [Int] -> BinTree
insert t [] = t
insert t (x:xs) = insert t' xs
  where
    t' = insertSingle t x
    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)
```

Übungsblatt 3 — Aufgabe 1

Algebraische Datentypen

AUFGABE 1 – TEIL (A)

Anzahl der Blätter

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

[] für Blätter
↳ Node x []

$$\text{countLeaves (Node - [])} = 1$$

$$\text{countLeaves (Node - [t])} = \text{countLeaves t}$$

$$\text{countLeaves (Node x (t:ts))} =$$

$$\text{countLeaves t} + \text{countLeaves (Node x ts)}$$



AUFGABE 1 – TEIL (A)

Anzahl der Blätter

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

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

wichtig, sonst
zusätzliches
Blatt

AUFGABE 1 – TEIL (B)

gerade Anzahl an Kindern

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

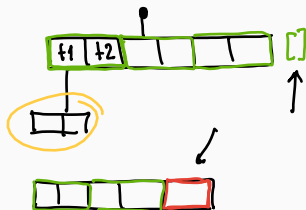
`evenNodes (Node _ []) = True`

`evenNodes (Node _ [t]) = False`

`evenNodes (Node x (t1:t2:ts))`

`= evenNodes t1 && evenNodes t2`

`&& evenNodes (Node x ts)`



gerade Anzahl an Kindern

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

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

AUFGABE 1 – TEIL (B)

gerade Anzahl an Kindern

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

```
evenNodes' :: RoseTree -> Bool
→ evenNodes' (Node _ []) = True
evenNodes' (Node _ ts)
  = mod (length ts) 2 == 0 && evenNodes'' ts
  where
    → evenNodes'' :: [RoseTree] -> Bool
      evenNodes'' [] = True
      evenNodes'' (t:ts)
        = evenNodes' t && evenNodes'' ts
```

Übungsblatt 3 — Aufgaben 2 & 3

Funktionen höherer Ordnung

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

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

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

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

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

- ▶ Operator \rightarrow Funktion: Klammern

$(+)$ $::$ $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$(+)$ $x\ y = x + y$

$x+y$ Operator

$f\ x\ y$ Funktion

- ▶ Funktion \rightarrow Operator: Backticks '...'

$5\ \text{'mod'}\ 2 = 1$

$t1\ \text{'equal'}\ t2$

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.

```
sqAdd :: Int -> Int
```

```
sqAdd = (^2) . (+ 5)
```

$$h(x) = x + 5$$

$$g(x) = x^2$$

$$f(x) = g(h(x))$$

$$f = g \circ h$$

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

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.

```
mod :: Int -> Int -> Int,  
(mod m) n = ...
```

```
mod 10 :: Int -> Int  
(mod 10) n = mod 10 n
```


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.

(Int, Int)
`mod :: Int -> (Int -> Int)`

`mod m n = ...`

`mod 10 :: Int -> Int`

`(mod 10) n = mod 10 n`

$(>) : Int \rightarrow Int \rightarrow \underline{Bool}$
 > 3

`(> 3) :: Int -> Bool`

`(> 3) x = x > 3`

FUNKTIONEN HÖHERER ORDNUNG — MAP

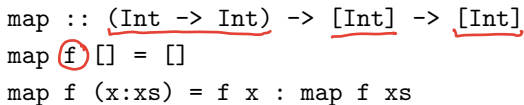
Funktionen können als Argumente von Funktionen auftreten. Wir lernen drei Basics kennen:

FUNKTIONEN HÖHERER ORDNUNG — MAP

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

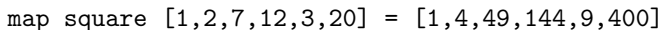
Die Funktion map

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



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

- ▶ *Beispiel.*



```
map square [1,2,7,12,3,20] = [1,4,49,144,9,400]
```

$(^2)$

Die Funktion filter

- ▶ `filter p xs` liefert eine Liste, die genau die Elemente von `xs` enthält, welche das Prädikat `p` erfüllen

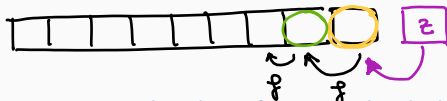
```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]
```

- ▶ *Beispiel.*

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

FUNKTIONEN HÖHERER ORDNUNG — FOLDR

Die Funktion foldr



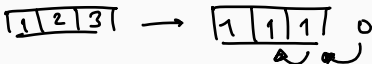
- ▶ `foldr f z xs` faltet eine Liste `xs` und verknüpft jeweils durch die Funktion `f`; gestartet wird mit `z` und dem rechten Element

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

- ▶ *Beispiel.*

```
foldr (+) 3 [1,2,3,4,5] = 18
```

```
length xs = foldr (+) 0 (map (\x -> 1) xs)
```



FUNKTIONEN HÖHERER ORDNUNG – ÜBERSICHT

- ▶ `map` wendet Funktion auf alle Listenelemente an

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

- ▶ `filter` wählt Listenelemente anhand einer Funktion aus

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

- ▶ `foldr` faltet eine Liste mit Verknüpfungsfunktion (von rechts beginnend)

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

AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

```
f :: [Int] -> Int
```

$\text{foldr } (*) \ 1 \left(\text{map } (^2) \left(\text{filter even } xs \right) \right)$

xs

$\text{even } xs = \text{length } xs \text{ `mod` } 2 == 0$

$\text{even} = (\text{`mod` } 2 == 0) \cdot \text{length}$

AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

```
f :: [Int] -> Int
```


```
f xs  
  = foldr (+) 0 (map (^2) (filter ('mod' 2) == 0) xs))
```


AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

```
f :: [Int] -> Int
```

```
f xs  
  = foldr (+) 0 (map (^2) (filter ('mod' 2) == 0) xs))
```



```
f' xs = foldr (*) 1 (map (^2) (filter even xs))
```

AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

```
f :: [Int] -> Int
```

```
f xs  
  = foldr (+) 0 (map (^2) (filter ('mod' 2) == 0) xs))
```

```
f' xs = foldr (*) 1 (map (^2) (filter even xs))
```

```
f'' = foldr (*) 1 . map (^2) . filter even
```

```
f'''  
  = foldr (*) 1 . map (^2) . filter (((= 0) . ('mod' 2)))
```

Faltung einer Liste von *links*

```
foldleft :: (Int -> Int -> Int) -> Int -> [Int] -> Int
```

Faltung einer Liste von *links*

```
foldleft :: (Int -> Int -> Int) -> Int -> [Int] -> Int
```

```
foldleft :: (Int -> Int -> Int) -> Int -> [Int] -> Int
```

```
foldleft f x [] = x
```

```
foldleft f x (y:ys) = foldleft f (f x y) ys
```

Fragen?