

PROGRAMMIERUNG

ÜBUNG 3: BÄUME & FUNKTIONEN HÖHERER ORDNUNG

Eric Kunze

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

Übungsblatt 2

Aufgabe 3

- ▶ Ziel: problemspezifische Datenkonstrukturen
- ▶ 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 Datenkonstrukturen (Großbuchstabe)
- ▶ tij sind Typnamen (Großbuchstaben)

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
```

AUFGABE 3

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

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     )( 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    )
```

AUFGABE 3 – TEIL (A)

Einfügen von Schlüsseln in einen Binärbaum

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

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)
```


Test auf Baum-Gleichheit

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

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)
```

Übungsblatt 3

Aufgabe 1

AUFGABE 1 – TEIL (A)

Anzahl der Blätter

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

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))
6  = countLeaves t + countLeaves (Node x ts)
```

gerade Anzahl an Kindern

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

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
```

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
```


Funktionen advanced

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

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

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

```
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 drum herum

```
1 (+) :: Int -> Int -> Int  
2 (+) x y = x + y
```

► Funktion \rightarrow Operator: Backticks ‘...’

```
1 5 `mod` 2 = 1
```

Analog zur mathematischen Notation $f = g \circ h$ für $f(x) = g(h(x))$ versteht auch Haskell das Kompositionsprinzip mit dem Operator `.`
z.B.

```
1 sqAdd :: Int -> Int  
2 sqAdd = (^2) . (+ 5)
```

statt `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.

```
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
```

FUNKTIONEN HÖHERER ORDNUNG — MAP

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

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

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

- *Beispiel.*

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

Die Funktion filter

- `filter p 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]
```

- *Beispiel.*

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


Die Funktion foldr

- `foldr f z 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)
```

FUNKTIONEN HÖHERER ORDNUNG – ÜBERSICHT

- `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
```

- `filter` wählt Listenelemente anhand einer Funktion aus

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

- `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)
```

Übungsblatt 3

Aufgabe 2

AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

`f :: [Int] -> Int`

AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

`f :: [Int] -> Int`

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

AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

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

AUFGABE 2

Produkt der Quadrate aller geraden Zahlen einer Liste

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

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

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

Übungsblatt 3

Aufgabe 2

AUFGABE 3

Faltung einer Liste von *links*

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

AUFGABE 3

Faltung einer Liste von *links*

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

```
1 foldleft :: (Int -> Int -> Int) -> Int -> [Int] -> Int
2 foldleft f x []          = x
3 foldleft f x (y:ys) = foldleft f (f x y) ys
```

Fragen?