# PROGRAMMIERUNG

## ÜBUNG 3: BÄUME & FUNKTIONEN HÖHERER ORDNUNG

Eric Kunze

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:**

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

- `Typename` ist ein Name (Großbuchstabe)
- `Con1, ... Conr` sind Datenkonstruktoren (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
```

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

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

# AUFGABE 3

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

```haskell
tree1 :: BinTree -- Suchbaum
tree1 = Branch 5
    (   Branch 3
        (Branch 2 Nil Nil)
        (Branch 4 Nil Nil)
    )( Branch 8
        ( Branch 7
            (Branch 6 Nil Nil)
            (Nil)
        )
        ( Branch 10
            (Nil)
            (Branch 13 Nil Nil)
        )
    )
```

# AUFGABE 3 – TEIL (A)

## 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)
```

## 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)
```

# Übungsblatt 3

## *Aufgabe 1*

## Anzahl der Blätter

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

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

## gerade Anzahl an Kindern

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

```
-- (b) gerade Anzahl an Kindern testen - Variante 1
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
```

```
-- (b) gerade Anzahl an Kindern testen - Variante 2
evenNodes' :: RoseTree -> Bool
evenNodes' (Node _ []) = True
evenNodes' (Node _ ts)
  = mod (length ts) 2 == 0 && evenNodes'' ts
```

# Funktionen advanced

# FUNKTIONEN

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

**anonyme Funktionen.** Funktionen ohne konkreten Namen

z.B. `(\x -> x+1)` ist die Addition mit 1

```
(\x -> 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

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

- Funktion $\rightarrow$ Operator: Backticks ‘...‘

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

# 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)
```

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.

```
mod :: Int -> Int -> Int
mod m n = ...

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

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

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]
```

## 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]
```

## 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

```
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)
```

# Übungsblatt 3

## *Aufgabe 2*

# AUFGABE 2

## Produkt der Quadrate aller geraden Zahlen einer Liste

`f :: [Int] -> Int`

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

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

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

```haskell
f''' :: [Int] -> Int
f''' =
  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`

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

# ENDE

**Fragen?**