

# PROGRAMMIERUNG

## ÜBUNG 2: LISTEN, ZEICHENKETTEN & BÄUME

---

Eric Kunze

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

# Übungsblatt 1

## *Zusatzaufgabe*

---

# ÜBUNGSBLATT 1 – ZUSATZAUFGABE

**Ziel:** Anzahl der vollständigen Binärbäume mit  $n$  Knoten

**Idee:** Wie erhalten wir volle Binärbäume? — Ein voller Binärbaum ist

- ▶ entweder ein Blatt
- ▶ oder er besteht aus einer Wurzel und *zwei* Kindern

**Umsetzung:**

- ▶ Rekursionsfall:  $n \geq 3$  Knoten
  - ▷ ein Wurzelknoten
  - ▷  $n - 1$  Knoten für linken und rechten Teilbaum (systematisch alle Möglichkeiten durchlaufen)
- ▶ Basisfall:
  - ▷  $n = 0$ : es gibt keinen Baum mit keinen Knoten
  - ▷  $n = 1$ : Baum mit einem Knoten = Blatt (davon gibt es genau einen)

# ÜBUNGSBLATT 1 – ZUSATZAUFGABE

```
countBinTrees :: Int -> Int
countBinTrees 0 = 0
countBinTrees 1 = 1
countBinTrees n = go (n-1)
  where
    go 0 = 0
    go m = go (m-1) + countBinTrees (n - 1 - m) *
              countBinTrees m
```

**Hinweis:** `go` durchläuft alle Möglichkeiten  $n - 1$  Knoten so auf zwei (Kind-)Bäume zu verteilen, dass der linke Teilbaum  $m$  Knoten und der rechte Teilbaum die übrigen  $n - 1 - m$  Knoten besitzt.

# Aufgabe 1

*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

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

## **Verkettungsoperator " ++ "**

Verkettung zweier Listen gleichen Typs

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

## Multiplikation einer Liste

```
prod :: [Int] -> Int
```

```
prod :: [Int] -> Int  
prod []      = 1  
prod (x:xs) = x * prod xs
```

## Umkehrung einer Liste

```
rev :: [Int] -> [Int]
```

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

### WICHTIG

- ▶ Element : [Liste]
- ▶ [Liste] ++ [Liste]



## AUFGABE 1 – TEIL (B)

### Sortierung einer Liste prüfen

```
isOrd :: [Int] -> Bool
```

```
isOrd :: [Int] -> Bool
isOrd [] = True
isOrd [x] = True
isOrd (x:y:xs)
  | x <= y    = isOrd (y:xs)
  | otherwise = False
```

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

## AUFGABE 1 – TEIL (C)

### sortiertes Zusammenfügen zweier (sortierten) Listen

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

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x < y      = x : merge xs (y:ys)
  | otherwise  = y : merge (x:xs) ys
```

Wir können Listen auch “benennen” — Rekursionsfall:

```
merge xxs@(x:xs) yys@(y:ys)
  | x < y      = x : merge xs yys
  | otherwise  = y : merge xxs ys
```

## AUFGABE 1 – TEIL (D)

### (unendliche) Liste der Fibonacci-Zahlen

```
fibs :: [Int]
```

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

fibs :: [Int]
fibs = fibAppend 0
      where fibAppend x = fib x : fibAppend (x+1)
```

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

# Aufgabe 2

## *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"
```

### Präfix - Test

```
isPrefix :: String -> String -> Bool
```

```
isPrefix :: String -> String -> Bool
isPrefix [] _ = True
isPrefix _ [] = False
isPrefix (p:ps) (c:cs) = p == c && isPrefix ps cs
```

### Vorkommen eines Patterns zählen

```
countPattern :: String -> String -> Int
```

```
countPattern :: String -> String -> Int
countPattern "" "" = 1
countPattern _  "" = 0
countPattern xs yys@(y:ys)
    | isPrefix xs yys = 1 + countPattern xs ys
    | otherwise       = countPattern xs ys
```

# 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)
- ▶ t<sub>ij</sub> 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 Weather = Sunny Int Int Bool | Cloudy Float
              | Rainy String Int
```

## 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

```
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)
```

### 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)
```

**Fragen?**