

# PROGRAMMIERUNG

## ÜBUNG 3: ZEICHENKETTEN & BÄUME

---

Eric Kunze

`eric.kunze@tu-dresden.de`

TU Dresden, 25.04.2022

letzte Änderung:  
24.04.2022, 15:25

# Aufgabe 1

## *Zeichen & Zeichenketten*

---

## Zeichen

- ▶ Datentyp Char
- ▶ Eingabe in einfachen Anführungszeichen
- ▶ z.B. 'a', 'e', '3'

## Zeichen

- ▶ Datentyp Char
- ▶ Eingabe in einfachen Anführungszeichen
- ▶ z.B. 'a', 'e', '3'

## Zeichenketten

- ▶ Datentyp String = [Char]
- ▶ Eingabe in doppelten Anführungszeichen
- ▶ z.B. "hallo", "welt"
- ▶ Konkatination von Zeichenketten:

```
1 || "hallo " ++ "welt" = "hallo welt"
```

### Präfix – Test

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

## Präfix – Test

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

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

### Vorkommen eines Patterns zählen

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

## Vorkommen eines Patterns zählen

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

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



# Aufgabe 2

## *Algebraische Datentypen*

---

# 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 Typkonstruktor
- ▶ Con1, ... Conr sind Datenkonstruktoren
- ▶ t<sub>ij</sub> sind Typnamen

Konstruktoren beginnen mit Großbuchstaben.

# ALGEBRAISCHE DATENTYPEN – BEISPIELE

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

# ALGEBRAISCHE DATENTYPEN – BEISPIELE

```
1 data Season = Spring | Summer | Autumn | Winter
2
3 goSkiing :: Season -> Bool
4 goSkiing Winter = True
5 goSkiing _      = False
```

# ALGEBRAISCHE DATENTYPEN – BEISPIELE

```
1 data Season = Spring | Summer | Autumn | Winter
2
3 goSkiing :: Season -> Bool
4 goSkiing Winter = True
5 goSkiing _      = False
6
7 data Weather = Sunny Int Int Bool | Cloudy Float
8                | Rainy String Int
```

# ALGEBRAISCHE DATENTYPEN – BEISPIELE

```
1 data Season = Spring | Summer | Autumn | Winter
2
3 goSkiing :: Season -> Bool
4 goSkiing Winter = True
5 goSkiing _      = False
6
7 data Weather = Sunny Int Int Bool | Cloudy Float
8              | Rainy String Int
9
10 data Bool = True | False
```

# ALGEBRAISCHE DATENTYPEN – BEISPIELE

```
1 data Season = Spring | Summer | Autumn | Winter
2
3 goSkiing :: Season -> Bool
4 goSkiing Winter = True
5 goSkiing _      = False
6
7 data Weather = Sunny Int Int Bool | Cloudy Float
8              | Rainy String Int
9
10 data Bool = True | False
11
12 data BinTree = Branch Int BinTree BinTree | Nil
```

- ▶ Konstruktor `Branch`: erzeugt Knoten mit Beschriftung und zwei Kindern  $\rightsquigarrow$  *Rekursionsfall*
- ▶ Konstruktor `Nil`: erzeugt leeren Baum  $\rightsquigarrow$  *Basisfall*

- ▶ Basisfall: nulläre Wertkonstrukturen (z.B. `Nil`)
- ▶ Rekursionsfall: Konstrukturen mit Stelligkeit  $> 0$  (z.B. `Branch`)



- ▶ Basisfall: nulläre Wertkonstruktoren (z.B. `Nil`)
- ▶ Rekursionsfall: Konstruktoren mit Stelligkeit  $> 0$  (z.B. `Branch`)

Man **erzeugt** konkrete Bäume, indem man `Int` und `BinTree` aus der Typdefinition durch konkrete Werte ersetzt, z.B. `Int` durch 3 und `BinTree` durch `Nil` oder `Branch` ....

- ▶ Basisfall: nulläre Wertkonstruktoren (z.B. Nil)
- ▶ Rekursionsfall: Konstruktoren mit Stelligkeit  $> 0$  (z.B. Branch)

Man **erzeugt** konkrete Bäume, indem man `Int` und `BinTree` aus der Typdefinition durch konkrete Werte ersetzt, z.B. `Int` durch 3 und `BinTree` durch `Nil` oder `Branch` ....

**Pattern Matching** funktioniert weiterhin; man nutzt dafür die Wertkonstruktoren (hier: `Branch` und `Nil`):

```
1 | foo :: BinTree -> ...  
2 | foo Nil = ...  
3 | foo (Branch x l r) = ...
```

# ALGEBRAISCHE DATENTYPEN – REKURSION

- ▶ Basisfall: nulläre Wertkonstruktoren (z.B. `Nil`)
- ▶ Rekursionsfall: Konstruktoren mit Stelligkeit  $> 0$  (z.B. `Branch`)

Man **erzeugt** konkrete Bäume, indem man `Int` und `BinTree` aus der Typdefinition durch konkrete Werte ersetzt, z.B. `Int` durch `3` und `BinTree` durch `Nil` oder `Branch` ....

**Pattern Matching** funktioniert weiterhin; man nutzt dafür die Wertkonstruktoren (hier: `Branch` und `Nil`):

```
1 | foo :: BinTree -> ...  
2 | foo Nil = ...  
3 | foo (Branch x l r) = ...
```

Um in GHCi eine **Ausgabe** der Bäume zu erhalten, muss `deriving Show` hinter die Typdefinition geschrieben werden.

## AUFGABE 2 – TEIL (A)

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

## AUFGABE 2 – TEIL (A)

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

Ein Beispielbaum:

```
1 mytree :: BinTree
2 mytree = Branch 0
3           ( Nil )
4           ( Branch 3
5               ( Branch 1 Nil Nil )
6               ( Branch 5 Nil Nil )
7           )
```

... erfüllt die Suchbaumeigenschaft.

### Test auf Baum-Gleichheit

### Test auf Baum-Gleichheit

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

## AUFGABE 2 – TEIL (B)

### Test auf Baum-Gleichheit

```
1 data BinTree = Branch Int BinTree BinTree | Nil
2 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)
```



### Einfügen von Schlüsseln in einen Binärbaum

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

## AUFGABE 2 – TEIL (C)

### Einfügen von Schlüsseln in einen Binärbaum

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

```
1 insert :: BinTree -> [Int] -> BinTree
2 insert t      [] = t
3 insert t (x:xs) = insert t' xs
4     where t' = insertSingle t x
5           insertSingle Nil          x = Branch x Nil
6           Nil
7           insertSingle (Branch y l r) x
8           | x < y      = Branch y (insertSingle l x)
9           |            = Branch y l (insertSingle r x)
10          | otherwise = Branch y l (insertSingle r x)
```

## AUFGABE 2 – TEIL (D)

### Levelorder-Traversierung

```
1 data BinTree = Branch Int BinTree BinTree | Nil
2 unwind :: BinTree -> [Int]
```

## AUFGABE 2 – TEIL (D)

### Levelorder-Traversierung

```
1 data BinTree = Branch Int BinTree BinTree | Nil
2 unwind :: BinTree -> [Int]
```

**Idee:** Nutze Liste von Bäumen als Zwischenspeicher (Hilfsfunktion) und hänge Kindbäume hinten an diese Liste an

## AUFGABE 2 – TEIL (D)

### Levelorder-Traversierung

```
1 data BinTree = Branch Int BinTree BinTree | Nil
2 unwind :: BinTree -> [Int]
```

**Idee:** Nutze Liste von Bäumen als Zwischenspeicher  
(Hilfsfunktion) und hänge Kindbäume hinten an diese Liste an

```
1 unwind :: BinTree -> [Int]
2 unwind t = go [t]
3   where
4     go [] = []
5     go ((Branch a Nil Nil) : ts) = a : go ts
6     go ((Branch a l r) : ts) = a : go (ts ++ [l,r])
```

**Fragen?**