

PROGRAMMIERUNG

ÜBUNG 2: LISTEN, ZEICHENKETTEN & BÄUME

Eric Kunze

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

- ▶ Übungsaufgaben können *freiwillig* abgegeben werden
- ▶ keine Bonuspunkte
- ▶ Abgaben in möglichst einer Datei und (solange es geht) nur Quelltexte
- ▶ Hinweise meinerseits nur noch auf meiner Website `oakoneric.github.io`
- ▶ Fragen und Anmerkungen jederzeit
- ▶ einfache Fragen (Orga oder kleine Inhalte) gern schnell und unkompliziert über Telegram

Probleme aus der vergangenen Woche?

Listen & Zeichenketten in Haskell

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

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

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

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

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

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:

```
"hallo " ++ "welt" = "hallo welt"
```


Bestimme den Datentyp:

- ▶ `"prog"`
- ▶ `5`
- ▶ `'3'`
- ▶ `mod` (gemeint ist die Modulo-Funktion)
- ▶ `[2,4,6,8]`
- ▶ `[["e", "r", "i", "c"], ["k", "u", "n", "z", "e"]]`

Übungsblatt 2

Aufgabe 1

Multiplikation einer Liste

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

Multiplikation einer Liste

`prod :: [Int] -> Int`

```
1  -- (a) Produkt der Listenelemente
2  prod :: [Int] -> Int
3  prod []      = 1
4  prod (x:xs) = x * prod xs
```

Umkehrung einer Liste

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

AUFGABE 1 – TEIL (B)

Umkehrung einer Liste

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

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

Elemente einer Liste löschen

```
excl :: Int -> [Int] -> [Int]
```

Elemente einer Liste löschen

`excl :: Int -> [Int] -> [Int]`

```
1 excl :: Int -> [Int] -> [Int]
2 excl _ [] = []
3 excl n (x:xs)
4   | x /= n    = x : excl n xs
5   | otherwise = excl n xs
```


Sortierung einer Liste prüfen

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

Sortierung einer Liste prüfen

`isOrd :: [Int] -> Bool`

```
1 isOrd :: [Int] -> Bool
2 isOrd [] = True
3 isOrd [x] = True
4 isOrd (x:y:xs)
5   | x <= y = isOrd (y:xs)
6   | otherwise = False
```

AUFGABE 1 – TEIL (D)

Sortierung einer Liste prüfen

isOrd :: [Int] -> Bool

```
1 isOrd :: [Int] -> Bool
2 isOrd [] = True
3 isOrd [x] = True
4 isOrd (x:y:xs)
5   | x <= y = isOrd (y:xs)
6   | otherwise = False
```

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

AUFGABE 1 – TEIL (E)

sortiertes Zusammenfügen zweier (sortierten) Listen

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

AUFGABE 1 – TEIL (E)

sortiertes Zusammenfügen zweier (sortierten) Listen

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

```
1 merge :: [Int] -> [Int] -> [Int]
2 merge [] ys = ys
3 merge xs [] = xs
4 merge (x:xs) (y:ys)
5   | x < y      = x : merge xs (y:ys)
6   | otherwise  = y : merge (x:xs) ys
```

AUFGABE 1 – TEIL (E)

sortiertes Zusammenfügen zweier (sortierten) Listen

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

```
1 merge :: [Int] -> [Int] -> [Int]
2 merge [] ys = ys
3 merge xs [] = xs
4 merge (x:xs) (y:ys)
5   | x < y      = x : merge xs (y:ys)
6   | otherwise  = y : merge (x:xs) ys
```

```
1 merge' :: [Int] -> [Int] -> [Int]
2 merge' [] ys = ys
3 merge' xs [] = xs
4 merge' xxs@(x:xs) yys@(y:ys)
5   | x < y      = x : merge' xs yys
6   | otherwise  = y : merge' xxs ys
```

AUFGABE 1 – TEIL (F)

(unendliche) Liste der Fibonacci-Zahlen

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

AUFGABE 1 – TEIL (F)

(unendliche) Liste der Fibonacci-Zahlen

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

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


AUFGABE 1 – TEIL (F)

(unendliche) Liste der Fibonacci-Zahlen

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

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

```
1 fib :: Int -> Int
2 fib 0 = 1
3 fib 1 = 1
4 fib n = fib (n-1) + fib (n-2)
5
6 fibs :: [Int]
7 fibs = fibAppend 0
8   where fibAppend x = fib x : fibAppend (x+1)
```

Übungsblatt 2

Aufgabe 2

AUFGABE 2 – TEIL (A)

Liste von Wörtern aneinanderfügen

```
join :: [String] -> String
```

Liste von Wörtern aneinanderfügen

```
join :: [String] -> String
```

```
1 join :: [String] -> String
2 join [] = ""
3 join [x] = x
4 join (x:xs) = x ++ ' ' : join xs
```

Trennung eines Strings in seine Wörter

```
unjoin :: String -> [String]
```

Trennung eines Strings in seine Wörter

`unjoin :: String -> [String]`

```
1 unjoin :: String -> [String]
2 unjoin s = f [] s
3   where
4     f save [] = [save]
5     f save (c:cs)
6       | c == ' ' = save : f [] cs
7       | otherwise = f (save ++ [c]) cs
```

AUFGABE 2 – TEIL (A)

Trennung eines Strings in seine Wörter

`unjoin :: String -> [String]`

```
1 unjoin'' :: String -> [String]
2 unjoin'' [] = []
3 unjoin'' [c] = if c == ' ' then [[]] else [[c]]
4 unjoin'' (c:cs)
5   | c == ' ' = [] : unjoin'' cs
6   | otherwise = let (s:ss) = unjoin'' cs
7                 in ((c:s):ss)
```

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     )(
8     Branch 8
9     (    Branch 7
10        (Branch 6 Nil Nil)
11        (Nil)
12    )
13    (    Branch 10
14        (Nil)
15        (Branch 13 Nil Nil)
16    )
17 )
```

Einfügen von Schlüsseln in einen Binärbaum

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

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)
```


Fragen?