

PROGRAMMIERUNG

ÜBUNG 2: LISTEN

Eric Kunze

`eric.kunze@tu-dresden.de`

TU Dresden, 20. April 2022

letzte Änderung:
20.04.2022, 18:17

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

```
1 countBinTrees :: Int -> Int
2 countBinTrees 0 = 0
3 countBinTrees 1 = 1
4 countBinTrees n = go (n-1)
5   where
6     go 0 = 0
7     go m = go (m-1) + countBinTrees (n - 1 - m) *
8               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

Binomialkoeffizient

Binomialkoeffizient

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$$

Binomialkoeffizient

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$$

```
bincoeff :: Int -> Int -> Int
```

Binomialkoeffizient

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$$

```
bincoeff :: Int -> Int -> Int
```

```
5 | fac :: Int -> Int
6 | fac n
7 |   | n < 1      = 1
8 |   | otherwise = n * fac (n-1)
```


AUFGABE 1

Binomialkoeffizient

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$$

`bincoeff :: Int -> Int -> Int`

```
5 | fac :: Int -> Int
```

```
6 | fac n
```

```
7 |   | n < 1      = 1
```

```
8 |   | otherwise = n * fac (n-1)
```

```
1 | bincoeff :: Int -> Int -> Int
```

```
2 | bincoeff n k = fac n 'div' (fac (n-k) * fac k)
```

Division

Der Operator `/` ist nur für Typen definiert, die gebrochene Zahlen darstellen können; `div` liefert dagegen die `Int`-Division.

Aufgabe 2

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

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

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

AUFGABE 2 – TEIL (A)

Produkt einer Liste

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

Produkt einer Liste

`prod :: [Int] -> Int`

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

Umkehrung einer Liste

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


Umkehrung einer Liste

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

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

WICHTIG

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

Elemente einer Liste löschen

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

AUFGABE 2 – TEIL (C)

Elemente einer Liste löschen

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

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

AUFGABE 2 – TEIL (D)

Sortierung einer Liste prüfen

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

AUFGABE 2 – 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
```

AUFGABE 2 – 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 2 – TEIL (E)

sortiertes Zusammenfügen zweier (sortierten) Listen

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

AUFGABE 2 – 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 2 – 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
```

Wir können Listen auch “benennen” — Rekursionsfall:

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

AUFGABE 2 – TEIL (F)

(unendliche) Liste der Fibonacci-Zahlen

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

AUFGABE 2 – TEIL (F)

(unendliche) Liste der Fibonacci-Zahlen

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

```
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)
```

AUFGABE 2 – TEIL (F)

(unendliche) Liste der Fibonacci-Zahlen

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

```
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)
```

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

Hinweis: `take 7 fibs` liefert die ersten 7 Fibonacci-Zahlen

Fragen?