Hochschule RheinMain

Funktionale Programmierung

04 Listen und Bäume

Prof. Dr. David Sabel

Sommersemester 2025

Stand der Folien: 20. Mai 2025

Hochschule RheinMain

REKURSIVE DATENSTRUKTUREN: LISTEN

Vorbemerkung

Hochschule RheinMain

- In aktuellen Haskell-Bibliotheken sind viele Listenfunktionen **verallgemeinert** für Instanzen von `Foldable`
- Listen sind Instanzen von `Foldable`
- Zur Verständlichkeit geben wir in diesem Kapitel die auf Listen spezialisierten Implementierungen und Typen an

Beispiel:

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1+(length xs)
```

Aber:

```
> stack exec -- ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
ghci> :t length
length :: Foldable t => t a -> Int
```

Listen von Zahlen: Syntaktischer Zucker in Haskell

Hochschule RheinMain

- [*start*..*end*]
  erzeugt die Liste der Zahlen von *start* bis *end*
  z.B. ergibt `[10..15]` die Liste `[10,11,12,13,14,15]`.
- [*start*..]
  erzeugt die unendliche Liste ab dem Wert *start*,
  z.B. erzeugt `[1..]` die Liste aller natürlichen Zahlen.
- [*start*, *next*..*end*]
  erzeugt die Liste $[start, start + \delta, start + 2 * \delta, \ldots, start + m \cdot \delta]$,
  wobei $\delta = next - start$ und solange Vielfache von δ dazuaddiert werden,
  bis der Endwert erreicht ist (überschritten, falls $\delta > 0$, unterschritten falls $\delta < 0$).
  Z.B. `[1,3..8]` erzeugt `[1,3,5,7]` und `[7,5..2]` erzeugt `[7,5,3]`.
- [*start*, *next*..]
  erzeugt unendlich lange Liste mit der Schrittweite $next - start$.

Implementierung für Integer

Hochschule RheinMain

```
-- [start..]
from :: Integer -> [Integer]
from start = fromThenDelta start 1

-- [start,next..]
fromThen :: Integer -> Integer -> [Integer]
fromThen start next = fromThenDelta start (next-start)

fromThenDelta :: Integer -> Integer  -> [Integer]
fromThenDelta start delta = start:(fromThenDelta (start+delta) delta)
```

Implementierung für Integer (2)

Hochschule RheinMain

```
-- [start..end]
fromTo :: Integer -> Integer -> [Integer]
fromTo start end = fromThenDeltaTo start 1 end

-- [start,next..end]
fromThenTo :: Integer -> Integer -> Integer  -> [Integer]
fromThenTo start next end  = fromThenDeltaTo start (next-start) end

fromThenDeltaTo :: Integer -> Integer -> Integer  -> [Integer]
fromThenDeltaTo start delta end = go start
 where go val
        | test val     = []
        | otherwise    = val:(go (val+delta))
       test val
        | delta >= 0 = val > end
        | otherwise  = val < end
```

Zeichen und Zeichenketten

Hochschule RheinMain

- Eingebauter Typ `Char` für Zeichen
- Darstellung: Einfaches Anführungszeichen, z.B. `'A'`
- Steuersymbole beginnen mit \, z.B. `\n`, `\t`
- Spezialsymbole `\\` und `\"`

Zeichen und Zeichenketten

Hochschule RheinMain

- Eingebauter Typ `Char` für Zeichen
- Darstellung: Einfaches Anführungszeichen, z.B. `'A'`
- Steuersymbole beginnen mit \, z.B. `\n`, `\t`
- Spezialsymbole `\\` und `\"`

Strings

- Vom Typ `String` = `[Char]` (Listen von Zeichen)
- Spezialsyntax `"Hallo"` ist gleich zu

  `['H','a','l','l','o']` bzw. `'H':('a':('l':('l':('o':[]))))`.

Zeichen und Zeichenketten (2)

Hochschule RheinMain

Nützliche Funktionen für `Char`: In der Bibliothek `Data.Char`
Z.B.:

```
ord :: Char -> Int
chr :: Int -> Char

isLower :: Char -> Bool
isUpper :: Char -> Bool
isAlpha :: Char -> Bool

toUpper :: Char -> Char
toLower :: Char -> Char
```

Beachte: Strings als Listen von Char sind ineffizient.

Für RealWorld-Anwendungen: Verwende `ByteStrings` (siehe `Data.ByteString`).

Hochschule RheinMain

STANDARD-LISTENFUNKTIONEN

Listen zusammenhängen

Hochschule RheinMain

Append: ++

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Beispiele:

```
*> [1..10] ++ [100..109]
[1,2,3,4,5,6,7,8,9,10,100,101,102,103,104,105,106,107,108,109]
*> [[1,2],[2,3]] ++ [[3,4,5]]
[[1,2],[2,3],[3,4,5]]
*> "Infor" ++ "matik"
"Informatik"
```

Laufzeitverhalten: linear in der Länge der ersten Liste

Zugriff auf Listenelement per Index

Hochschule RheinMain

```
(!!) :: [a] -> Int -> a
[]     !! _ = error "Index too large"
(x:xs) !! 0 = x
(x:xs) !! i = xs !! (i-1)
```

Beispiele:

```
*> [1,2,3,4,5]!!3
4
*> [0,1,2,3,4,5]!!3
3
*> [0,1,2,3,4,5]!!5
5
*> [1,2,3,4,5]!!5
*** Exception: Prelude.!!: index too large
```

Index eines Elements berechnen

Hochschule RheinMain

```
elemIndex :: (Eq a) ⇒ a -> [a] -> Maybe Int
elemIndex a xs = findInd 0 a xs
  where findInd i a [] = Nothing
        findInd i a (x:xs)
          | a == x    = Just i
          | otherwise = findInd (i+1) a xs
```

Beispiele:

```
*> elemIndex 1 [1,2,3]
Just 0
*> elemIndex 1 [0,1,2,3]
Just 1
*> elemIndex 1 [5,4,3,2]
Nothing
*> elemIndex 1 [1,4,1,2]
Just 0
```

Funktion auf alle Listenelemente anwenden

Hochschule RheinMain

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

Beispiele:

```
*> map (*3) [1..20]
[3,6,9,12,15,18,21,24,27,30,33,36,39,42,45,48,51,54,57,60]
*> map not [True,False,False,True]
[False,True,True,False]
*> map (^2) [1..10]
[1,4,9,16,25,36,49,64,81,100]
*> map toUpper "Informatik"
"INFORMATIK"
```

Elemente aus Liste filtern

Hochschule RheinMain

```
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs)
  | f x       = x:(filter f xs)
  | otherwise = filter f xs
```

Beispiele:

```
*> filter (> 15) [10..20]
[16,17,18,19,20]
*> filter isAlpha "FFP 2021"
"FFP"
*> filter (\x -> x > 5) [1..10]
[6,7,8,9,10]
```

Listenelemente entfernen

Hochschule RheinMain

```
remove p xs = filter (not . p) xs
```

Der Kompositionsoperator `(.)` ist definiert als:

```
(f . g) x = f (g x)
```

Beispiele:

```
*> remove p xs = filter (not . p) xs
*> remove (> 5) [1..10]
[1,2,3,4,5]
*> remove isAlpha "FFP 2021"
" 2021"
```

Länge einer Liste berechnen

Hochschule RheinMain

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1+(length xs)
```

Beispiele:

```
*> length "Informatik"
10
*> length [2..20002]
20001
*> length [1..]
^CInterrupted.
```

Bessere Variante von length

Hochschule RheinMain

Benötigt nur konstanten Platz:

```
length :: [a] -> Int
length xs = length_it xs 0

length_it []     acc = acc
length_it (_:xs) acc = let acc' = 1+acc
                       in seq acc' (length_it xs acc')
```

Umdrehen einer Liste

Hochschule RheinMain

Schlechte Variante: Laufzeit quadratisch!

```
reverse1 :: [a] -> [a]
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++
[x]
```

Umdrehen einer Liste

Hochschule RheinMain

Schlechte Variante: Laufzeit quadratisch!

```
reverse1 :: [a] -> [a]
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++
[x]
```

Besser mit Stack: Laufzeit linear

```
reverse :: [a] -> [a]
reverse xs = rev xs []
 where rev []     acc = acc
       rev (x:xs) acc = rev xs (x:acc)
```

Umdrehen einer Liste

Hochschule RheinMain

Schlechte Variante: Laufzeit quadratisch!

```
reverse1 :: [a] -> [a]
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++
[x]
```

Beispiele:

```
*> reverse [1..10]
[10,9,8,7,6,5,4,3,2,1]
*> reverse [1..]
^CInterrupted.
*> reverse "neffen neppen neffen"
"neffen neppen neffen"
```

Besser mit Stack: Laufzeit linear

```
reverse :: [a] -> [a]
reverse xs = rev xs []
 where rev []     acc = acc
       rev (x:xs) acc = rev xs (x:acc)
```

Listen aus demselben Element

Hochschule RheinMain

```
repeat :: a -> [a]
repeat x = x:(repeat x)

replicate :: Int -> a -> [a]
replicate 0 x = []
replicate i x = x:(replicate (i-1) x)
```

Beispiele

```
*> replicate 10 [1,2]
[[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2],[1,2]]
*> replicate 20 'A'
"AAAAAAAAAAAAAAAAAAAA"
```

n Elemente nehmen / verwerfen

Hochschule RheinMain

```
take :: Int -> [a] -> [a]
take i [] = []
take 0 xs = []
take i (x:xs) = x:(take (i-1) xs)
```

```
drop :: Int -> [a] -> [a]
drop i []     = []
drop 0 xs     = xs
drop i (x:xs) = drop (i-1) xs
```

Beispiele:

```
*> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
*> drop 5 "Informatik"
"matik"
*> take 5 (drop 4 [1..])
[5,6,7,8,9]
```

Nehmen / Verwerfen solange bis...

Hochschule RheinMain

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
 | p x       = x:(takeWhile p xs)
 | otherwise = []
```

```
dropWhile :: (a -> Bool) -> [a] ->
[a]
dropWhile p [] = []
dropWhile p  (x:xs)
 | p x        = dropWhile p xs
 | otherwise  = x:xs
```

Beispiele:

```
*> takeWhile (> 5) [5,6,7,3,6,7,8]
[]
*> takeWhile (> 5) [7,6,7,3,6,7,8]
[7,6,7]
*> dropWhile (< 10) [1..20]
[10,11,12,13,14,15,16,17,18,19,20]
```

Listen paaren und entpaaren

Hochschule RheinMain

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```

```
unzip :: [(a, b)] -> ([a], [b])
unzip [] = ([],[])
unzip ((x,y):xs) =
       let (xs',ys') = unzip xs
       in (x:xs',y:ys')
```

Beispiele:

```
*> zip [1..10] "Informatik"
[(1,'I'),(2,'n'),(3,'f'),(4,'o'),(5,'r'),(6,'m'),(7,'a'),(8,'t'),(9,'i'),(10,'k')]
*> unzip [(1,'I'),(2,'n'),(3,'f'),(4,'o'),(5,'r'),
          (6,'m'),(7,'a'),(8,'t'),(9,'i'),(10,'k')]
([1,2,3,4,5,6,7,8,9,10],"Informatik")|
```

Bemerkung zu `zip`

Hochschule RheinMain

- Man kann zwar `zip3`, `zip4` etc. definieren um 3, 4, ..., Listen in 3-Tupel, 4-Tupel, etc. einzupacken, aber:

  Man kann keine Funktion `zipN` für n Listen definieren

- Grund: diese Funktion wäre nicht getypt.
- Abhilfe liefert: Template Haskell (siehe späteres Kapitel)

Kombination von zip und map

Hochschule RheinMain

```
zipWith :: (a -> b -> c) -> [a]-> [b] -> [c]
zipWith f (x:xs) (y:ys) = (f x y) : (zipWith f xs ys)
zipWith _ _      _      = []
```

Damit kann man zip definieren:

```
zip = zipWith (\x y -> (x,y))
```

Anderes Beispiel:

```
vectorAdd :: (Num a) ⇒ [a] -> [a] -> [a]
vectorAdd = zipWith (+)
```

Faltungen

Hochschule RheinMain

- `foldl` $\otimes$ e $[a_1, \ldots, a_n]$ ergibt $(\ldots((e \otimes a_1) \otimes a_2)\ldots) \otimes a_n$
- `foldr` $\otimes$ e $[a_1, \ldots, a_n]$ ergibt $a_1 \otimes (a_2 \otimes (\ldots \otimes (a_n \otimes e)\ldots))$

Implementierung:

```
foldl ::  (a -> b -> a) -> a -> [b] -> a
foldl f e  []     = e
foldl f e (x:xs) = foldl f (e `f` x) xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (x:xs) = x `f` (foldr f e xs)
```

foldl und foldr sind identisch, wenn die Elemente und der Operator $\otimes$ ein Monoid mit neutralem Element e bilden.

Listen glätten

Hochschule RheinMain

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

Beachte: `foldl` bei append wäre ineffizienter!

Effizienter Links-Falten

Hochschule RheinMain

```
sum     = foldl (+) 0
product = foldl (*) 1
```

haben schlechten Platzbedarf, besser strikte Variante von `foldl`:

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f e []     = e
foldl' f e (x:xs) = let e' = e `f` x in e' `seq` foldl' f e' xs
```

Weitere Anwendungen von Fold

Hochschule RheinMain

Beachte die Allgemeinheit der Typen von foldl / foldr

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

z.B. sind alle Elemente ungerade?

```
  foldl (\xa xb -> xa && (odd xb)) True
```

`xa` und `xb` haben verschiedene Typen!

Analog mit foldr:

```
  foldr (\xa xb -> (odd xa) && xb) True
```

Varianten von foldl, foldr

Hochschule RheinMain

```
foldr1                  :: (a -> a -> a) -> [a] -> a
foldr1 _ []             =  error "foldr1 on an empty list"
foldr1 _ [x]            =  x
foldr1 f (x:xs)         =  f x (foldr1 f xs)

foldl1                  :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)         =  foldl f x xs
foldl1 _ []             =  error "foldl1 on an empty list"
```

Beispiele:

```
maximum :: (Ord a) ⇒ [a] -> a
maximum xs = foldl1 max xs

minimum :: (Ord a) ⇒ [a] -> a
minimum xs = foldl1 min xs
```

Liste aller Zwischenergebnisse von foldl und foldr

Hochschule RheinMain

- `scanl` $\otimes$ e $[a_1, a_2, ..., a_n] = [e, e \otimes a_1, (e \otimes a_1) \otimes a_2, \ldots]$
- `scanr` $\otimes$ e $[a_1, a_2, ..., a_n] = [\ldots, a_{n-1} \otimes (a_n \otimes e), a_n \otimes e, e]$

Es gilt:

- `last (scanl f e xs) = foldl f e xs`
- `head (scanr f e xs) = foldr f e xs`.

Implementierung von scanl und scanr

Hochschule RheinMain

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f e xs = e:(case xs of
                   [] ->  []
                   (y:ys) -> scanl f (e `f` y) ys)

scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr _ e []           =  [e]
scanr f e (x:xs)       =  f x q : qs
                           where qs@(q:_) = scanr f e xs
```

Anmerkung: „As“-Pattern `Var@Pat`

Beispiele

Hochschule RheinMain

```
*> scanr (++) [] [[1,2],[3,4],[5,6],[7,8]]
[[1,2,3,4,5,6,7,8],[3,4,5,6,7,8],[5,6,7,8],[7,8],[]]
*> scanl (++) [] [[1,2],[3,4],[5,6],[7,8]]
[[],[1,2],[1,2,3,4],[1,2,3,4,5,6],[1,2,3,4,5,6,7,8]]
*> scanl (+) 0 [1..10]
[0,1,3,6,10,15,21,28,36,45,55]
```

Beispiele zur Verwendung von `scan`

Hochschule RheinMain

Fakultätsfolge:

```
faks = scanl (*) 1 [1..]
```

Z.B.

```
*> take 5 faks
[1,1,2,6,24]
```

Beispiele zur Verwendung von `scan`

Hochschule RheinMain

Fakultätsfolge:

```
faks = scanl (*) 1 [1..]
```

Z.B.

```
*> take 5 faks
[1,1,2,6,24]
```

Funktion, die alle Restlisten einer Liste berechnet:

```
tails xs = scanr (:) [] xs
```

Z.B.

```
*> tails [1,2,3]
[[1,2,3],[2,3],[3],[]]
```

Partionieren und Quicksort

Hochschule RheinMain

```
partition p xs = (filter p xs, remove p xs)
```

Effizienter:

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
partition p [] = ([],[])
partition p (x:xs)
  | p x       = (x:r1,r2)
  | otherwise = (r1,x:r2)
  where (r1,r2) = partition p xs
```

Quicksort mit `partition`

```
quicksort :: (Ord a) ⇒ [a] -> [a]
quicksort []  = []
quicksort [x] = [x]
quicksort (x:xs) = let (kleiner,groesser) = partition (<x) xs
                   in quicksort kleiner ++ (x:(quicksort groesser))
```

Hochschule RheinMain

LISTEN ALS STRÖME

Listen als Ströme

Hochschule RheinMain

- Listen in Haskell können unendlich lang sein
- Daher kann man Listen auch als Ströme auffassen

Listen als Ströme

Hochschule RheinMain

- Listen in Haskell können unendlich lang sein
- Daher kann man Listen auch als Ströme auffassen
- Bei der Stromverarbeitung muss man aufpassen:
  Nie versuchen die gesamte Liste auszuwerten
- D.h. Funktionen sollen strom-produzierend sein.
- Grobe Regel:
  Funktion `f ::[Int]->[Int]` ist strom-produzierend, wenn `take n (f list)`
  für jede unendliche Liste `list` und jedes `n` terminiert

Listen als Ströme

Hochschule RheinMain

- Listen in Haskell können unendlich lang sein
- Daher kann man Listen auch als Ströme auffassen
- Bei der Stromverarbeitung muss man aufpassen:
  Nie versuchen die gesamte Liste auszuwerten
- D.h. Funktionen sollen strom-produzierend sein.
- Grobe Regel:
  Funktion `f ::[Int]->[Int]` ist strom-produzierend, wenn `take n (f list)`
  für jede unendliche Liste `list` und jedes `n` terminiert
- Ungeeignet daher: `reverse`, `length`, `fold`
- Geeignet: `map`, `filter`, `zipWith`, `take`, `drop`

Stromfunktionen für Strings

Hochschule RheinMain

- `words :: String -> [String]`
  Zerlegen einer Zeichenkette in eine Liste von Wörtern
- `lines :: String -> [String]`
  Zerlegen einer Zeichenkette in eine Liste der Zeilen
- `unlines :: [String] -> String`
  Einzelne Zeilen zu einem String zusammenfügen (mit Zeilenumbrüchen)

Beispiele:

```
*> words "Haskell ist eine funktionale Programmiersprache"
["Haskell","ist","eine","funktionale","Programmiersprache"]
*> lines "1234\n5678\n90"
["1234","5678","90"]
*> unlines ["1234","5678","90"]
"1234\n5678\n90\n"
```

Mischen zweier sortierter Ströme

Hochschule RheinMain

```
merge :: (Ord t) ⇒ [t] -> [t] -> [t]
merge []       ys  = ys
merge xs       []  = xs
merge a@(x:xs) b@(y:ys)
  | x ≤ y     = x:merge xs b
  | otherwise = y:merge a ys
```

Beispiel:

```
*> merge [1,3,5,6,7,9] [2,3,4,5,6]
[1,2,3,3,4,5,5,6,6,7,9]
```

Doppelte Elemente aus Strom entfernen

Hochschule RheinMain

```
nub xs  = nub' xs []
  where
    nub' [] _           = []
    nub' (x:xs) seen
        | x `elem` seen = nub' xs seen
        | otherwise     = x : nub' xs (x:seen)
```

Anmerkungen:

- `seen` merkt sich die bereits gesehenen Elemente
- Laufzeit von `nub` ist quadratisch

```
elem e []     = False
elem e (x:xs)
  | e == x    = True
  | otherwise = elem e xs
```

Doppelte Elemente aus sortierter Liste entfernen

Hochschule RheinMain

```
nubSorted (x:y:xs)
  | x == y    = nubSorted (y:xs)
  | otherwise = x:(nubSorted (y:xs))
nubSorted y = y
```

ist linear in der Länge der Liste.

Mischen der Vielfachen von 3,5 und 7

Hochschule RheinMain

```
*> nubSorted $ merge (map (3*) [1..])
*>              (merge (map (5*) [1..]) (map (7*) [1..]))
 [3,5,6,7,9,10,12,14,15,18,20,..
```

Listen als Wörterbuch

Hochschule RheinMain

```
lookup                  :: (Eq a) ⇒ a -> [(a,b)] -> Maybe b
lookup  key []          =  Nothing
lookup  key ((x,y):xys)
    | key == x          =  Just y
    | otherwise         =  lookup key xys
```

Beispiele:

```
*> lookup 5 [(1,'A'), (2,'B'), (4,'C'), (5,'F')]
Just 'F'
*> lookup 3 [(1,'A'), (2,'B'), (4,'C'), (5,'F')]
Nothing
```

Listen als Mengen: Any und All

Hochschule RheinMain

```
any _ []     = False
any p (x:xs)
 | (p x)     = True
 | otherwise = any xs
```

```
all _ []     = True
all p (x:xs)
 | (p x)     = all xs
 | otherwise = False
```

Beispiele:

```
*> all even [1,2,3,4]
False
*> all even [2,4]
True
*> any even [1,2,3,4]
True
```

Listen als Mengen: Löschen eines Elements

Hochschule RheinMain

```
delete :: (Eq a) ⇒ a -> [a] -> [a]
delete e (x:xs)
  | e == x    = xs
  | otherwise = x:(delete e xs)
```

Mengendifferenz: \\

```
(\\) :: (Eq a) ⇒ [a] -> [a] -> [a]
(\\) =  foldl (flip delete)
```

dabei dreht `flip` dreht die Argumente einer Funktion um:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f a b = f b a
```

Beispiele

Hochschule RheinMain

```
*> delete 3 [1,2,3,4,5,3,4,3]
[1,2,4,5,3,4,3]
*> [1,2,3,4,4] \\ [9,6,4,4,3,1]
[2]
*> [1,2,3,4] \\ [9,6,4,4,3,1]
[2]
```

Listen als Mengen: Vereinigung und Schnitt

Hochschule RheinMain

```
union :: (Eq a) ⇒ [a] -> [a] -> [a]
union xs ys =  xs ++ (ys \\ xs)

intersect :: (Eq a) ⇒ [a] -> [a] -> [a]
intersect xs ys = filter (\y -> any (== y) ys) xs

*> union [1,2,3,4,4] [9,6,4,3,1]
[1,2,3,4,4,9,6]
*> union [1,2,3,4,4] [9,6,4,4,3,1]
[1,2,3,4,4,9,6]
*> union [1,2,3,4,4] [9,9,6,4,4,3,1]
[1,2,3,4,4,9,6]
*> intersect [1,2,3,4,4] [4,4]
[4,4]
*> intersect [1,2,3,4] [4,4]
[4]
```

concat und map

Hochschule RheinMain

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f

*> concatMap (\x-> take x [1..]) [3..7]
[1,2,3,1,2,3,4,1,2,3,4,5,1,2,3,4,5,6,1,2,3,4,5,6,7]
```

Hochschule RheinMain

LIST COMPREHENSIONS

List Comprehensions

Hochschule RheinMain

- Spezielle Syntax zur Erzeugung und Verarbeitung von Listen
- ZF-Ausdrücke (nach der Zermelo-Fränkel Mengenlehre)

Syntax: `[Expr | qual1,...,qualn]`

- `Expr`: ein Ausdruck
- $FV($`Expr`$)$ sind durch `qual1,...,qualn` gebunden
- `quali` ist:
  - ein Generator der Form `pat <- Expr`, oder
  - ein Guard, d.h. ein Ausdruck booleschen Typs,
  - oder eine Deklaration lokaler Bindungen der Form `let x1=e1,...,xn=en` (ohne `in`-Ausdruck!) ist.

List Comprehensions: Beispiele

Hochschule RheinMain

- Liste der natürlichen Zahlen: `[x | x <- [1..]]`
- Kartesisches Produkt:
  `[ (wert,name) | wert <-[1..3], name <- ['a'..'b']]` erzeugt
  `[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]`.
- Reihenfolge wichtig:
  `[ (wert,name) | name <- ['a'..'b'], wert <-[1..3]]` erzeugt
  `[(1,'a'),(2,'a'),(3,'a'),(1,'b'),(2,'b'),(3,'b')]`.
- `[(x,y) | x <- [1..], y <- [1..]]` erzeugt $(\mathbb{N} \times \mathbb{N})$

Beispiele (2)

Hochschule RheinMain

```
*> take 10 [(x,y) | x <- [1..], y <- [1..]]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10)]|
*> [x | x <- [1..], odd x]
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31, ^CInterrupted
*> take 20 [y | x <- [1..], let y = x*x, y > 100, y < 1000]
[121,144,169,196,225,256,289,324,361,400,441,484,529,576,625,676,729,784,841,900]
*> [a | (a,_,_,_) <- [(x,x,y,y) | x <- [1..3], y <- [1..3]]]
[1,1,1,2,2,2,3,3,3]
```

Map, Filter, Concat mit List Comprehensions

Hochschule RheinMain

```
map f xs = [f x | x <- xs]

filter p xs = [x | x <- xs, p x]

concat xss = [y | xs <- xss, y <- xs]
```

List Comprehensions: Quicksort

Hochschule RheinMain

```
qsort (x:xs) =    qsort [y | y <- xs, y ≤ x]
              ++ [x]
              ++ qsort [y | y <- xs, y > x]
qsort x = x
```

Übersetzung in ZF-freies Haskell

Hochschule RheinMain

```
[ e | True ]          =   [e]
[ e | q ]             =   [ e | q, True ]
[ e | b, Q ]          =   if b then [ e | Q ] else []
[ e | p <- l, Q ]     =   let ok p = [ e | Q ]
                              ok _ = []
                          in concatMap ok l
[ e | let decls, Q ]  =   let decls in [ e | Q ]
```

wobei

- `ok` eine neue Variable,
- `b` ein Guard,
- `q` ein Generator, eine lokale Bindung oder ein Guard (nicht `True`)
- `Q` eine Folge von Generatoren, Deklarationen und Guards.

Übersetzung in ZF-freies Haskell: Beispiel

Hochschule RheinMain

```
[x*y | x <- xs, y <- ys, x > 2, y < 3]
= let ok x = [x*y | y <- ys, x > 2, y < 3]
      ok _ = []
  in concatMap ok xs
= let ok x = let ok' y = [x*y | x > 2, y < 3]
                 ok' _ = []
             in concatMap ok' ys
      ok _ = []
  in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

Hochschule RheinMain

```
[x*y | x <- xs, y <- ys, x > 2, y < 3]
= let ok x = [x*y | y <- ys, x > 2, y < 3]
      ok _ = []
  in concatMap ok xs
= let ok x = let ok' y = [x*y | x > 2, y < 3]
                 ok' _ = []
             in concatMap ok' ys
      ok _ = []
  in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

Hochschule RheinMain

```
[x*y | x <- xs, y <- ys, x > 2, y < 3]
=  let ok x = [x*y | y <- ys, x > 2, y < 3]
       ok _ = []
   in concatMap ok xs
=  let ok x = let ok' y = [x*y | x > 2, y < 3]
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

Hochschule RheinMain

```
= let ok x = let ok' y = if x > 2 then [x*y | y < 3] else []
                 ok' _ = []
             in concatMap ok' ys
      ok _ = []
  in concatMap ok xs
= let ok x = let ok' y = if x > 2 then [x*y | y < 3, True] else []
                 ok' _ = []
             in concatMap ok' ys
      ok _ = []
  in concatMap ok xs
= let ok x = let ok' y = if x > 2 then
                           (if y < 3 then [x*y | True] else [])
                         else []
                 ok' _ = []
             in concatMap ok' ys
      ok _ = []
  in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

Hochschule RheinMain

```
=  let ok x = let ok' y = if x > 2 then [x*y | y < 3] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
=  let ok x = let ok' y = if x > 2 then [x*y | y < 3, True] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
=  let ok x = let ok' y = if x > 2 then
                            (if y < 3 then [x*y | True] else [])
                          else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

Hochschule RheinMain

```
=  let ok x = let ok' y = if x > 2 then [x*y | y < 3] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
=  let ok x = let ok' y = if x > 2 then [x*y | y < 3, True] else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
=  let ok x = let ok' y = if x > 2 then
                            (if y < 3 then [x*y | True] else [])
                          else []
                  ok' _ = []
              in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Übersetzung in ZF-freies Haskell: Beispiel

Hochschule RheinMain

```
=  let ok x = let ok' y = if x > 2 then
                            (if y < 3 then [x*y] else [])
                          else []
               ok' _ = []
           in concatMap ok' ys
       ok _ = []
   in concatMap ok xs
```

Die Übersetzung ist nicht optimal, da Listen generiert und wieder abgebaut werden und bei `x <- xs` unnötige Pattern-Fallunterscheidung

Hochschule RheinMain

REKURSIVE DATENSTRUKTUREN: BÄUME

Rekursive Datenstrukturen: Bäume

Hochschule RheinMain

Binäre Bäume mit (polymorphen) Blattmarkierungen:

```
data BBaum a = Blatt a | Knoten (BBaum a) (BBaum a)
 deriving(Eq,Show)
```

`BBaum` ist Typkonstruktor, `Blatt` und `Knoten` sind Datenkonstruktoren

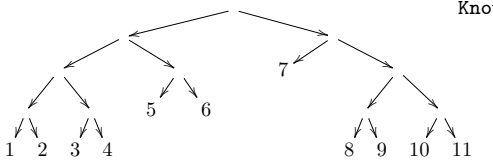
Rekursive Datenstrukturen: Bäume

Hochschule RheinMain

Binäre Bäume mit (polymorphen) Blattmarkierungen:

```
data BBaum a = Blatt a | Knoten (BBaum a) (BBaum a)
 deriving(Eq,Show)
```

`BBaum` ist Typkonstruktor, `Blatt` und `Knoten` sind Datenkonstruktoren




```
beispielBaum =
  Knoten (Knoten (Knoten
                   (Knoten (Blatt 1) (Blatt 2))
                   (Knoten (Blatt 3) (Blatt 4)))
                 (Knoten (Blatt 5) (Blatt 6)))
         (Knoten (Blatt 7)
                 (Knoten
                   (Knoten (Blatt 8) (Blatt 9))
                   (Knoten (Blatt 10) (Blatt 11))))
```

Summe aller Blattmarkierungen

Hochschule RheinMain

```
bSum (Blatt a) = a
bSum (Knoten links rechts) = (bSum links) + (bSum rechts)
```

Ein Beispielaufruf:

```
*> bSum beispielBaum
66
```

Liste der Blätter

Hochschule RheinMain

```
bRand (Blatt a) = [a]
bRand (Knoten links rechts) = (bRand links) ++ (bRand rechts)
```

Test:

```
*> bRand beispielBaum
[1,2,3,4,5,6,7,8,9,10,11]
```

Map auf Bäumen

Hochschule RheinMain

```
bMap f (Blatt a) = Blatt (f a)
bMap f (Knoten l r) = Knoten (bMap f l) (bMap f r)
```

Beispiel:

```
*> bMap (^2) beispielBaum
Knoten (Knoten (Knoten (Knoten (Blatt 1) (Blatt 4))
 (Knoten (Blatt 9) (Blatt 16))) (Knoten (Blatt 25) (Blatt 36)))
 (Knoten (Blatt 49) (Knoten (Knoten (Blatt 64) (Blatt 81))
 (Knoten (Blatt 100) (Blatt 121))))
```

Die Anzahl der Blätter eines Baumes:

```
anzahlBlaetter = bSum . bMap (\x -> 1)
```

Element-Test

Hochschule RheinMain

```
bElem e (Blatt a)
  | e == a            = True
  | otherwise         = False
bElem e (Knoten l r) = (bElem e l) || (bElem e r)
```

Einige Beispielaufrufe:

```
*> 11 `bElem` beispielBaum
 True
*> 1 `bElem` beispielBaum
 True
*> 20 `bElem` beispielBaumm
 False
*> 0 `bElem` beispielBaum m
 False
```

Fold auf Bäumen

Hochschule RheinMain

```
bFold op (Blatt a) = a
bFold op (Knoten a b) = op (bFold op a) (bFold op b)
```

Damit kann man z.B. die Summe und das Produkt berechnen:

```
*> bFold (+) beispielBaum
66
*> bFold (*) beispielBaum
39916800
```

Allgemeineres Fold auf Bäumen

Hochschule RheinMain

```
foldbt :: (a -> b -> b) -> b -> BBaum a  -> b
foldbt op a (Blatt x)    = op x a
foldbt op a (Knoten x y) = (foldbt op (foldbt op a y) x)
```

Der Typ des Ergebnisses kann anders sein als der Typ der Blattmarkierung
Zum Beispiel: Rand eines Baumes:

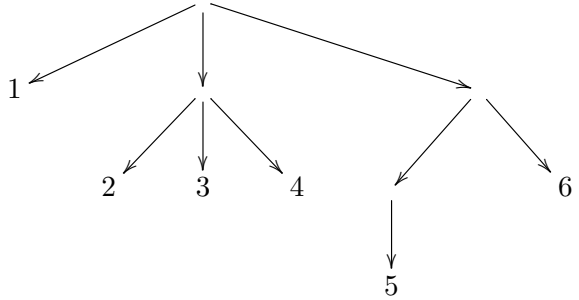
```
*> foldbt (:) [] beispielBaum
[1,2,3,4,5,6,7,8,9,10,11]
```

N-äre Bäume

Hochschule RheinMain

```
data NBaum a = NBlatt a | NKnoten [NBaum a]
 deriving(Eq,Show)

beispiel = NKnoten [NBlatt 1,
                    NKnoten [NBlatt 2, NBlatt 3, NBlatt 4],
                    NKnoten [NKnoten [NBlatt 5], NBlatt 6]]
```



Bäume mit Knotenmarkierungen

Hochschule RheinMain

Beachte: `BBaum` und `NBaum` haben nur Markierungen der Blätter!

Bäume mit Markierung aller Knoten

```
data BinBaum a = BinBlatt a | BinKnoten a (BinBaum a) (BinBaum a)
 deriving(Eq,Show)
```

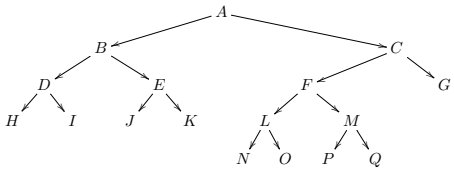
Bäume mit Knotenmarkierungen

Hochschule RheinMain

Beachte: `BBaum` und `NBaum` haben nur Markierungen der Blätter!

Bäume mit Markierung aller Knoten

```
data BinBaum a = BinBlatt a | BinKnoten a (BinBaum a) (BinBaum a)
 deriving(Eq,Show)
```




```
beispielBinBaum =
 BinKnoten 'A'
  (BinKnoten 'B'
     (BinKnoten 'D' (BinBlatt 'H') (BinBlatt 'I'))
     (BinKnoten 'E' (BinBlatt 'J') (BinBlatt 'K'))
  )
  (BinKnoten 'C'
     (BinKnoten 'F'
        (BinKnoten 'L' (BinBlatt 'N') (BinBlatt 'O'))
        (BinKnoten 'M' (BinBlatt 'P') (BinBlatt 'Q'))
     )
     (BinBlatt 'G')
  )
```

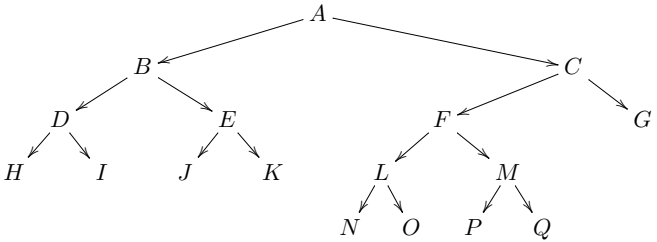
Funktionen auf BinBaum (1)

Hochschule RheinMain

Knoten in Preorder-Reihenfolge (Wurzel, links, rechts):

```
preorder :: BinBaum t -> [t]
preorder (BinBlatt a)      = [a]
preorder (BinKnoten a l r) = a:(preorder l) ++ (preorder r)

*> preorder beispielBinBaum
"ABDHIEJKCFLNOMPQG"
```



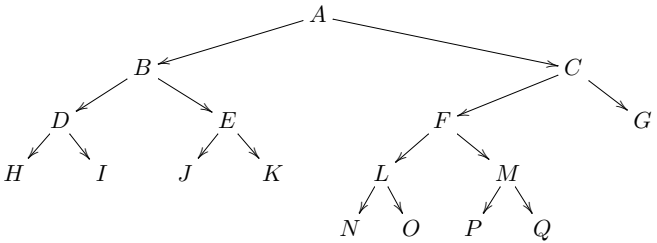
Funktionen auf `BinBaum` (2)

Hochschule RheinMain

Knoten in Inorder-Reihenfolge (links, Wurzel, rechts):

```
inorder :: BinBaum t -> [t]
inorder (BinBlatt a) = [a]
inorder (BinKnoten a l r) = (inorder l) ++ a:(inorder r)

*> inorder beispielBinBaum
"HDIBJEKANLOFPMQCG"
```



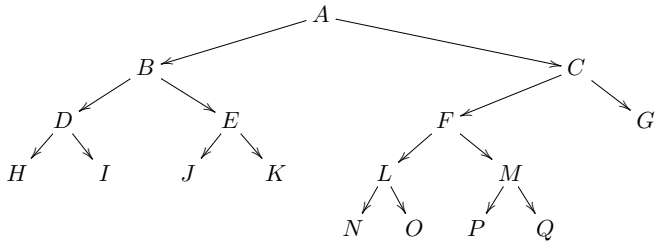
Funktionen auf BinBaum (3)

Hochschule RheinMain

Knoten in Post-Order Reihenfolge (links, rechts, Wurzel)

```
postorder (BinBlatt a) = [a]
postorder (BinKnoten a l r) =
   (postorder l) ++ (postorder r) ++ [a]
```

```
*> postorder  beispielBinBaum
"HIDJKEBNOLPQMFGCA"
```



Funktionen auf `BinBaum` (2)

Hochschule RheinMain

Level-Order (Stufenweise, wie Breitensuche)

Schlecht (alle Stufen werden von oben her neu berechnet)

```
levelorderSchlecht b =
   concat [nodesAtDepthI i b | i <- [0..depth b]]
 where
  nodesAtDepthI 0 (BinBlatt a) = [a]
  nodesAtDepthI i (BinBlatt a) = []
  nodesAtDepthI 0 (BinKnoten a l r) = [a]
  nodesAtDepthI i (BinKnoten a l r) =
         (nodesAtDepthI (i-1) l) ++ (nodesAtDepthI (i-1) r)
  depth (BinBlatt _) = 0
  depth (BinKnoten _ l r) = 1+(max (depth l) (depth r))

*> levelorderSchlecht  beispielBinBaum
"ABCDEFGHIJKLMNOPQ"
```

Funktionen auf `BinBaum` (3)

Hochschule RheinMain

Level-Order (Stufenweise, wie Breitensuche)

Besser:

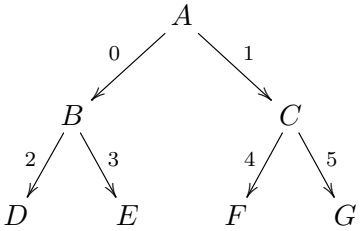
```
levelorder b = loForest [b]
 where
  loForest xs = map root xs ++ loForest (concatMap subtrees xs)
  root (BinBlatt a) = a
  root (BinKnoten a _ _) = a
  subtrees (BinBlatt _) = []
  subtrees (BinKnoten _ l r) = [l,r]
```

```
*> levelorder  beispielBinBaum
"ABCDEFGHIJKLMNOPQ"
```

Bäume mit Knoten und Kantenmarkierungen

Hochschule RheinMain

```
data BinBaumMitKM a b =
   BiBlatt a
 | BiKnoten a (b, BinBaumMitKM a b) (b,BinBaumMitKM a b)
 deriving(Eq,Show)
```




```
beispielBiBaum =
  BiKnoten 'A'
     (0,BiKnoten 'B'
           (2,BiBlatt 'D')
           (3,BiBlatt 'E'))
     (1,BiKnoten 'C'
           (4,BiBlatt 'F')
           (5,BiBlatt 'G'))
```

Funktion auf BinBaumMitKM

Hochschule RheinMain

Map mit 2 Funktionen

```
biMap f g (BiBlatt a) = BiBlatt (f a)
biMap f g (BiKnoten a (kl,links) (kr,rechts) =
 BiKnoten (f a) (g kl, biMap f g links) (g kr, biMap f g rechts)
```

Beispiel

```
*> biMap toLower even beispielBiBaum
BiKnoten 'a'
  (True,BiKnoten 'b' (True,BiBlatt 'd') (False,BiBlatt 'e'))
  (False,BiKnoten 'c' (True,BiBlatt 'f') (False,BiBlatt 'g'))
```

Syntaxbäume

Hochschule RheinMain

Auch Syntaxbäume sind Bäume

Beispiel: Einfache arithmetische Ausdrücke:

$$\begin{array}{lcl} E & ::= & (E + E) \mid (E * E) \mid Z \\ Z & ::= & 0Z' \mid \ldots \mid 9Z' \\ Z' & ::= & \varepsilon \mid Z \end{array}$$

Als Haskell-Datentyp (infix-Konstruktoren müssen mit `:` beginnen)

```
data ArEx = ArEx :+: ArEx
          | ArEx :*: ArEx
          | Zahl Int
```

alternativ

```
data ArEx = Plus ArEx ArEx
          | Mult ArEx ArEx
          | Zahl Int
```

Z.B. $(3 + 4) * (5 + (6 + 7))$ als Objekt vom Typ `ArEx`:

```
((Zahl 3) :+: (Zahl 4)) :*: ((Zahl 5) :+: ((Zahl 6) :+: (Zahl 7)))
```

Interpreter für Arithmetische Ausdrücke

Hochschule RheinMain

```
interpArEx :: ArEx -> Int
interpArEx (Zahl i) = i
interpArEx (e1 :+: e2) = (interpArEx e1) + (interpArEx e2)
interpArEx (e1 :*: e2) = (interpArEx e1) * (interpArEx e2)
```

Z.B.:

```
*Main> interpArEx (((Zahl 3) :+: (Zahl 4)) :*:
                   ((Zahl 5) :+: ((Zahl 6) :+: (Zahl 7))))
126
```