GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Grundlagen der Programmierung 2 B

Haskell: Listen-Komprehensionen

Prof. Dr. Manfred Schmidt-Schauß

Sommersemester 2017

Listenausdrücke, Listen-Komprehensionen

Analog zu Mengenausdrücken, aber Reihenfolge der Listenelemente im Resultat wird festgelegt.
`[x | x <- [0,1,..]]` ist analog zu $\{ x \mid x \in \mathbb{N}\}$

Beispiele:

`[x`	`| x <- xs]`	entspricht	`xs`
`[f x`	`| x <- xs]`	entspricht	`map f xs`
`[x`	`| x <- xs, p x]`	entspricht	`filter p xs`
`[(x,y)`	`| x <- xs, y <-ys]`	entspricht	`xs` „kreuzprodukt“ `ys` für endliche Listen
`[y`	`| x <- xs, y <-x]`	entspricht	`concat xs`

Syntax:
[⟨Ausdruck⟩ ”|” {⟨Generator⟩ | ⟨Filter⟩}{, {⟨Generator⟩ | ⟨Filter⟩}}*]

Terminalzeichen sind '[' ',' ']' und das Zeichen ” |”

Listen-Komprehensionen

[Resultatausdruck | Generatoren,Testfunktionen]

Generator: `v <- liste` liefert die Elemente von `liste`
Test: auf True/False. Element wird akzeptiert/nicht akz.

Wirkungsweise: die Generatoren liefern nach und nach die Elemente aller Listen.
Testfunktionen müssen `True` liefern, damit der Resultatausdruck in die Resultatliste aufgenommen wird.

Neue lokale Variablen sind möglich;
deren Geltungsbereich ist rechts von der Einführung

Listen-Komprehensionen: Beispiel

Koordinaten:

```
[(waagrecht,senkrecht) | waagrecht <- "abcdefgh",
                         senkrecht <- [1..8]]
```

Resultat: `[('a',1),('a',2),('a',3),('a',4),('a',5),`
`('a',6),('a',7),('a',8),('b',1),...]`

Listen-Komprehensionen: Beispiel

```
[(x,y) | x <- [1..10], even x, y <- [2..6], x < y]
```

Resultat: `[(2,3),(2,4),(2,5),(2,6),(4,5),(4,6)]`

Begründung: Erzeugungsreihenfolge:

x	1	2	2	2	2	2	3	4	4	4	4	4	5	6	...
y		2	3	4	5	6		2	3	4	5	6		2	...
?	N	N	Y	Y	Y	Y	N	N	N	N	Y	Y	N	N	...

Listen-Komprehensionen: Beispiel

```
*Main> [(x,y) | x <- [1..9], y <- [1..x]] ↩

[(1,1),
(2,1),(2,2),
(3,1),(3,2),(3,3),
(4,1),(4,2),(4,3),(4,4),
(5,1),(5,2),(5,3),(5,4),(5,5),
(6,1),(6,2),(6,3),(6,4),(6,5),(6,6),
(7,1),(7,2),(7,3),(7,4),(7,5),(7,6),(7,7),
(8,1),(8,2),(8,3),(8,4),(8,5),(8,6),(8,7),(8,8),
(9,1),(9,2),(9,3),(9,4),(9,5),(9,6),(9,7),(9,8),(9,9)]
```


Listen-Komprehensionen: Beispiel

Liste der nicht durch 2,3,5 teilbaren Zahlen. Die erste Nicht-Primzahl darin ist 49:

```
*Main> [x | x <- [2..], x `rem` 2 /= 0, x `rem` 3 /= 0,
x `rem` 5 /= 0] ↩
[7,11,13,17,19,23,29,31,37,41,43,47,49,53,59,61,67,71,73,77,...
```


Online Interface zu Listen-Komprehensionen

- Gut bedienbares spezielles Interface zum Ausprobieren von List-Komprehensionen
- Mit Beispiel-Aufgaben, verschiedene Varianten
- Entstanden im Rahmen der BSc-Arbeit von Robin Kreuzig

`http://www.ki.informatik.uni-frankfurt.de/listcomprehension`

Listen-Komprehensionen:

Beispiel Primzahlen

```
primes = 2: [x | x <- [3,5..],
     and (map (\t-> x `mod` t /= 0)
              (takeWhile (\y -> y^2 <= x) primes))]
```


Ressourcenbedarf von Listenfunktionen

(Bei verzögerter Auswertung)
Drei Möglichkeiten der Auswertung:

Komplett-Auswertung z.B. beim Drucken der Ergebnisliste im Interpreter

Rückgrat-Auswertung nur soviel, wie `length` von der Ergebnisliste benötigt.

Kopf-Auswertung (einfache. Ausw.) nur soviel, dass die Frage „Ist die Ergebnisliste leer oder nicht leer“ beantwortet werden kann.

Ressourcenbedarf von `map`

`xs` sei ausgewertete Liste der Länge n

```
map f xs:
```

Bei Rückgratauswertung: $O(n)$ Reduktionsschritte
Bei Komplett-Auswertung: # Reduktionsschritte abhängig von f

Bei Rückgratauswertung: $O(n)$ Speicherbedarf
Bei Komplett-Auswertung: Speicherbedarf abhängig von f

Ressourcenbedarf von `append`

`xs,ys` seien ausgewertete Listen der Länge m und n.
Bei Rückgrat-Auswertung:
`xs ++ ys` benötigt $O(m+n)$ Reduktionsschritte

$O(m)$ Platz: nur das Rückgrat der Liste `xs` muss kopiert werden

Listen-Funktion: reverse

Umdrehen einer Liste:

```
reverse_naiv []        = []
reverse_naiv (x:xs)    = (reverse_naiv xs) ++ [x]
```

Reduktionsschritte: $O(n^2)$, wenn n Länge der Liste, bei Rückgratauswertung

Begründung: $[a_1, \ldots, a_n]$
wird nach einigen Reduktionsschritten zu
$([a_n]$++$[a_{n-1}])$ ++) $\ldots$ ++ $[a_1]$
$[a_n, a_{n-1}, \ldots, a_{k-1}]$++$[a_k]$ braucht $k-1$ Reduktionsschritte.
Da $1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}$, ergibt sich $O(n^2)$.

Effizientes Umdrehen einer Liste

```
reverse xs = foldl (\x y -> y:x) [] xs
```

Oder

```
reverse  xs = reverse_lin xs []
reverse_lin [] ys        = ys
reverse_lin (x:xs) ys    = (reverse_lin xs (x:ys)
```

Iterativer Prozess für Listen:

```
reverse [1,3,5,7]
reverse_lin (1:3:5:7:[]) []
reverse_lin (3:5:7:[]) (1:[])
reverse_lin (5:7:[]) (3:1:[])
reverse_lin (7:[]) (5:3:1:[])
reverse_lin [] (7:5:3:1:[])
(7:5:3:1:[])
```


Weitere Listen-Funktionen

Restliste nach n-tem Element:

```
drop 0 xs            = xs
drop _ []            = []
drop n (_:xs) | n>0  = drop (n-1) xs
drop _ _             = error "Prelude.drop: negative argument"
```

Bildet Liste der Paare der Elemente:

```
zip              :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs)   = (a,b) : zip as bs
zip _ _             = []
```

aus Liste von Paaren ein Paar von Listen:

```
unzip  :: [(a,b)] -> ([a],[b])
unzip   = foldr (\(a,b) (as,bs) -> (a:as, b:bs)) ([], [])
```


Beispiele

```
*Main> drop 10 [1..100] ↩
[11,12,...]

*Main> zip "abcde" [1..] ↩
[('a',1),('b',2),('c',3),('d',4),('e',5)]

*Main> unzip (zip "abcdefg" [1..]) ↩
("abcdefg",[1,2,3,4,5,6,7])
```