Listenprogrammierung in Haskell

PD Dr. David Sabel

Goethe-Universität Frankfurt am Main

29. September 2015

Listen

- Liste = Folge von Elementen
- z.B. `[True,False,False,True,True]` und `[1,2,3,4,5,6]`
- In Haskell sind nur homogene Listen erlaubt:
  Alle Elemente haben den gleichen Typ
- z.B. verboten: `[True,'a',False,2]`

```
*Main> [True,'a',False,2]

<interactive>:1:6:
  Couldn't match expected type 'Bool' against inferred type 'Char'
  In the expression: 'a'
  In the expression: [True, 'a', False, 2]
  In the definition of 'it': it = [True, 'a', False, ....]
```

Listen: Typ

- Allgemeiner Typ: [a]
- Das a bezeichnet den Typ der Elemente
- z.B. Liste von Zeichen: Typ [Char] usw.

```
*Main> :type [True,False]
[True,False] :: [Bool]

*Main> :type ['A','B']
['A','B'] :: [Char]

*Main> :type [1,2,3]
[1,2,3] :: (Num t) => [t]
```

Listen: Typ

- Allgemeiner Typ: [a]
- Das a bezeichnet den Typ der Elemente
- z.B. Liste von Zeichen: Typ [Char] usw.

```
*Main> :type [True,False]
[True,False] :: [Bool]

*Main> :type ['A','B']
['A','B'] :: [Char]

*Main> :type [1,2,3]
[1,2,3] :: (Num t) => [t]
```

Auch möglich:

```
*Main> :type [double, doubleEven, \x -> x]
[double, doubleEven, \x -> x] :: [Integer -> Integer]

*Main> :type [[True,False], [False,True,True], [True,True]]
[[True,False], [False,True,True], [True,True]] :: [[Bool]]
```

Listen erstellen

- Eckige Klammern und Kommas z.B. `[1,2,3]`
- Das ist jedoch nur Syntaktischer Zucker

Listen sind **rekursiv** definiert:

- die **leere Liste** `[]` („Nil“)
- `x:xs` („Cons“)
  - `x` ist erstes Listenelement
  - `xs` ist Restliste

  Dann ist `x:xs` Liste mit *n* Elementen beginnend mit `x` und anschließend folgen die Elemente aus `xs`

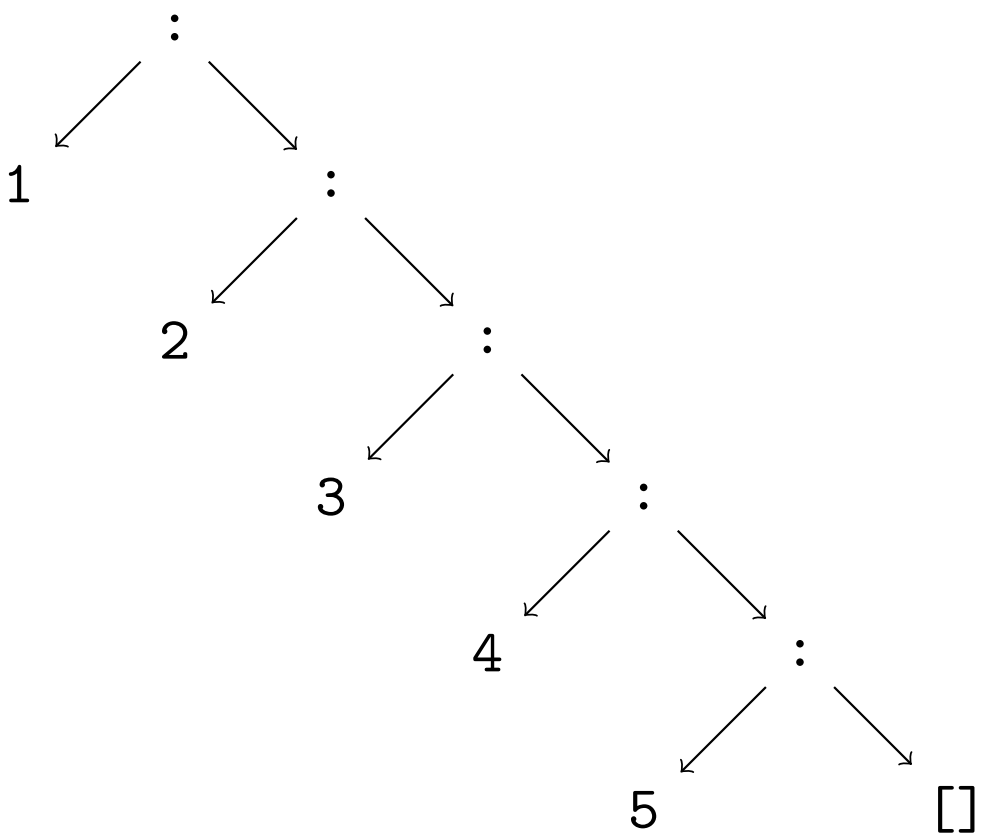
`[1,2,3]` ist Abkürzung für `1:(2:(3:[]))`

Typen:

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

Interne Darstellung der Listen

```
[1,2,3,4,5] = 1:(2:(3:(4:(5:[]))))
```



Beispiel

Konstruiere die Listen der Zahlen $n, n-1 \ldots, 1$:

```
nbis1 :: Integer -> [Integer]
nbis1 0 = []
nbis1 n = n:(nbis1 (n-1))
```

```
*Main> nbis1 0
[]
*Main> nbis1 1
[1]
*Main> nbis1 10
[10,9,8,7,6,5,4,3,2,1]
*Main> nbis1 100
[100,99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,81,80,79,
 78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,62,61,60,59,58,57,
 56,55,54,53,52,51,50,49,48,47,46,45,44,43,42,41,40,39,38,37,36,35,
 34,33,32, 31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,
 12,11,10,9,8,7,6,5,4,3,2,1]
```

Listen zerlegen

Vordefiniert:

- `head :: [a] -> a` liefert das erste Element einer nicht-leeren Liste
- `tail :: [a] -> [a]` liefert den Schwanz einer nicht-leeren Liste.
- `null :: [a] -> Bool` testet, ob eine Liste leer ist

```
*> head [1,2]
1
*> tail [1,2,3]
[2,3]
*> head []
*** Exception: Prelude.head: empty list
*> null []
True
*> null [4]
False
```

Beispiel

Funktion, die das letzte Element einer Liste liefert.

```
last :: [a] -> a
last xs =
  | null xs          = error "Liste ist leer"
  | null (tail xs)   = head xs
  | otherwise        = last (tail xs)
```

```
Main> last [True,False,True]
True
*Main> last [1,2,3]
3
*Main> last (nbis1 1000)
1
*Main> last [[1,2,3], [4,5,6], [7,8,9]]
[7,8,9]
```

Listen zerlegen mit Pattern

In

$$f\ par_1\ \ldots\ par_n = rumpf$$

dürfen par_i auch sog. Pattern sein.

Z.B.

```
myHead []     = error "empty list"
myHead (x:xs) = x
```

Auswertung von `myHead (1:(2:[]))`

- Das erste Pattern das zu `(1:(2:[]))` passt („matcht“) wird genommen
- Dies ist `(x:xs)`. Nun wird anhand des Patterns zerlegt:

  `x` = 1

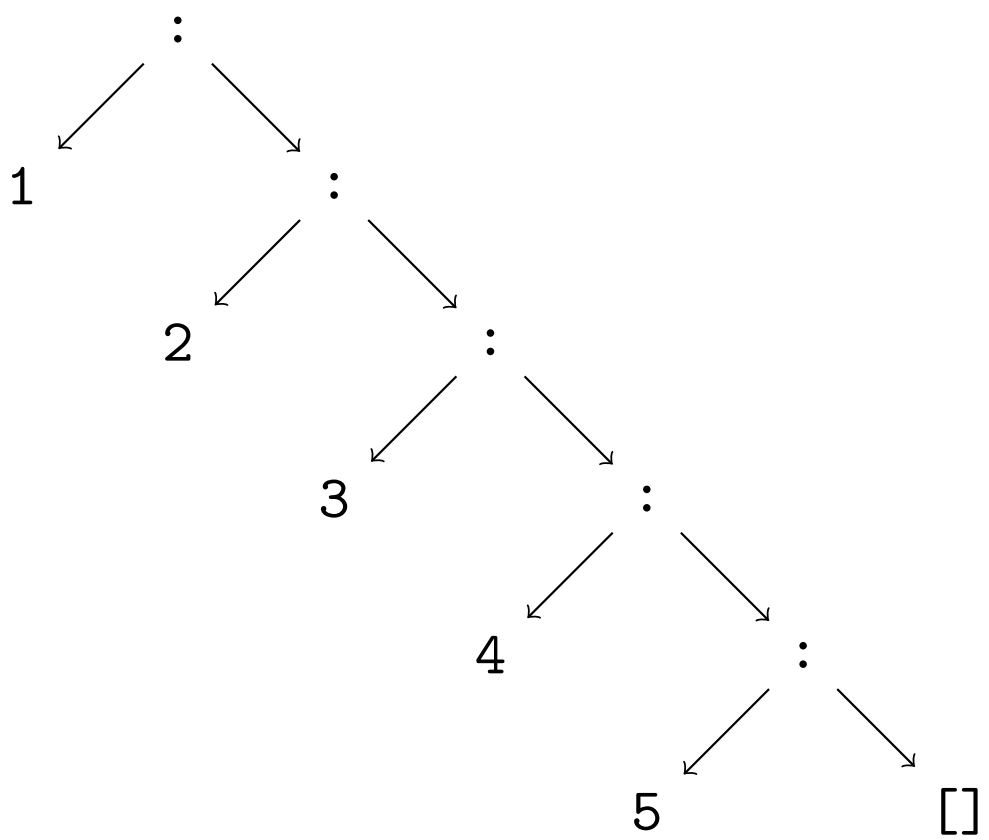
  `xs` = `(2:[])`

Pattern-Matching

Pattern []

[]

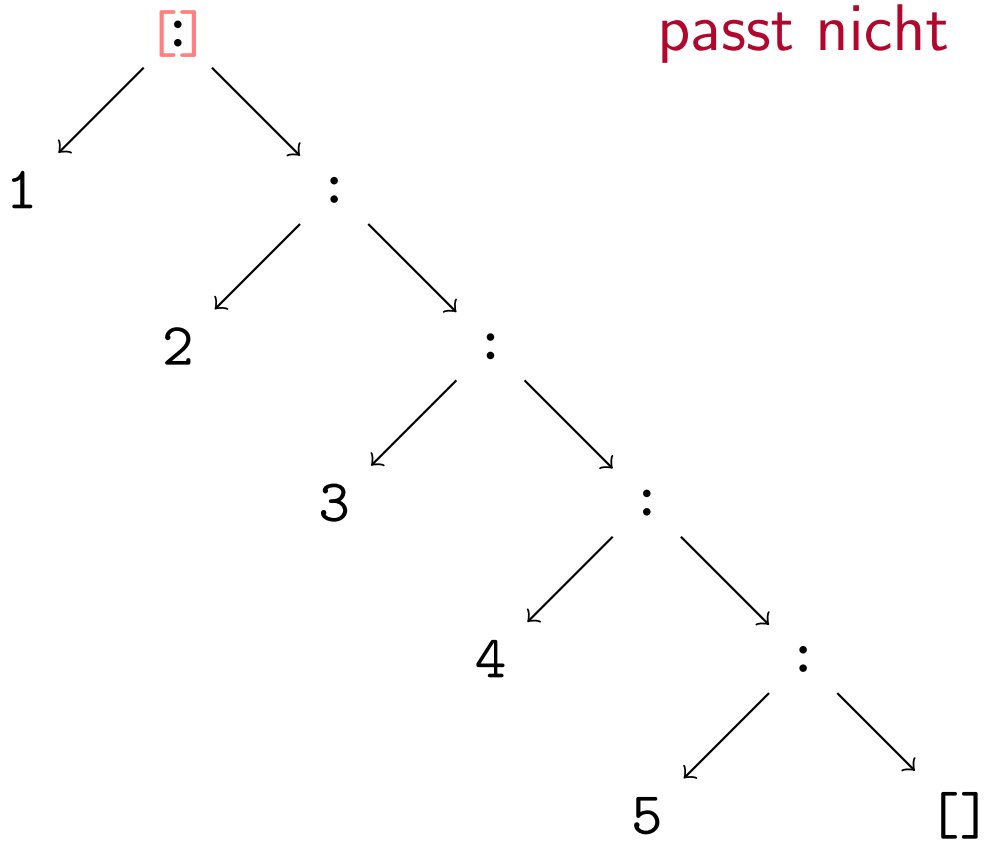
Liste



Pattern-Matching

Pattern `[]`

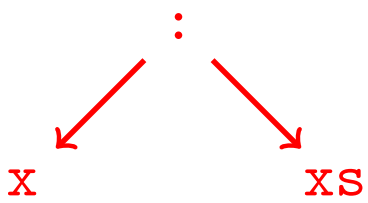
Liste



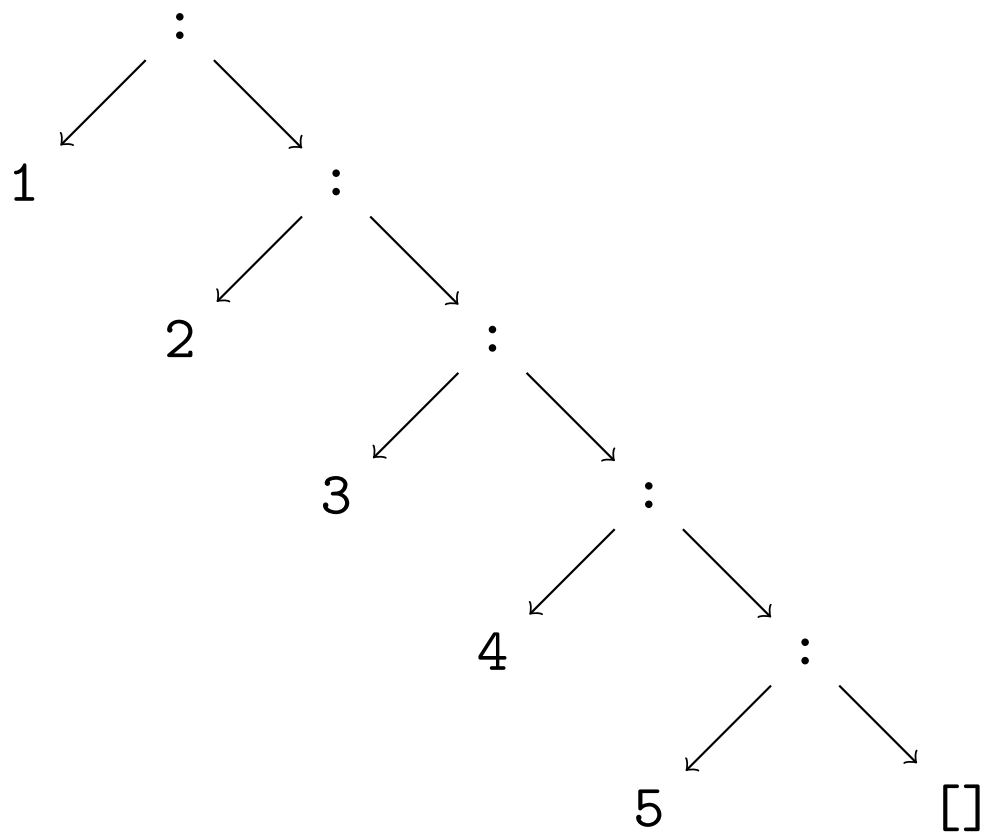
passt nicht

Pattern-Matching

Pattern `x:xs`



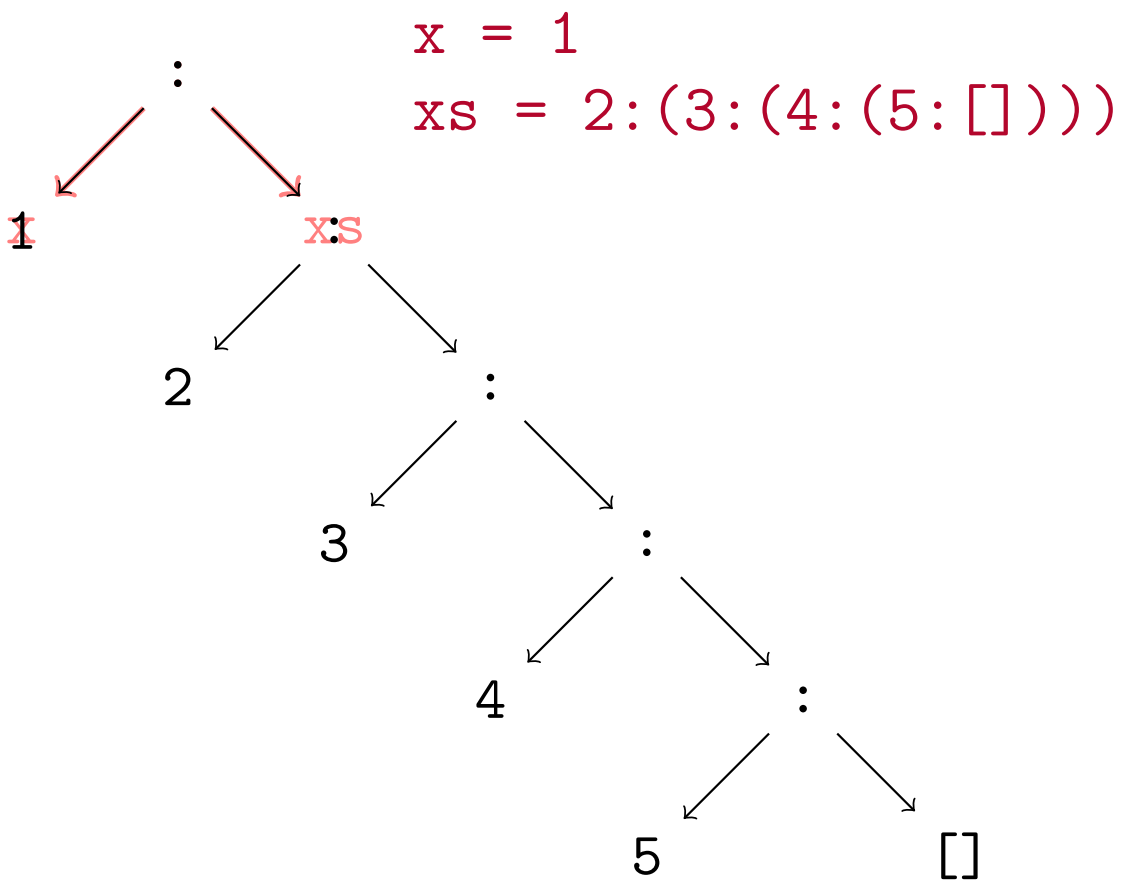
Liste



Pattern-Matching

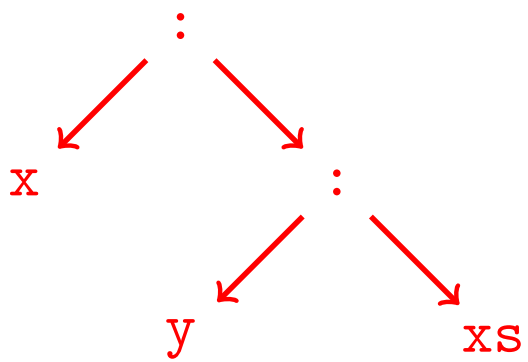
Pattern `x:xs`

Liste

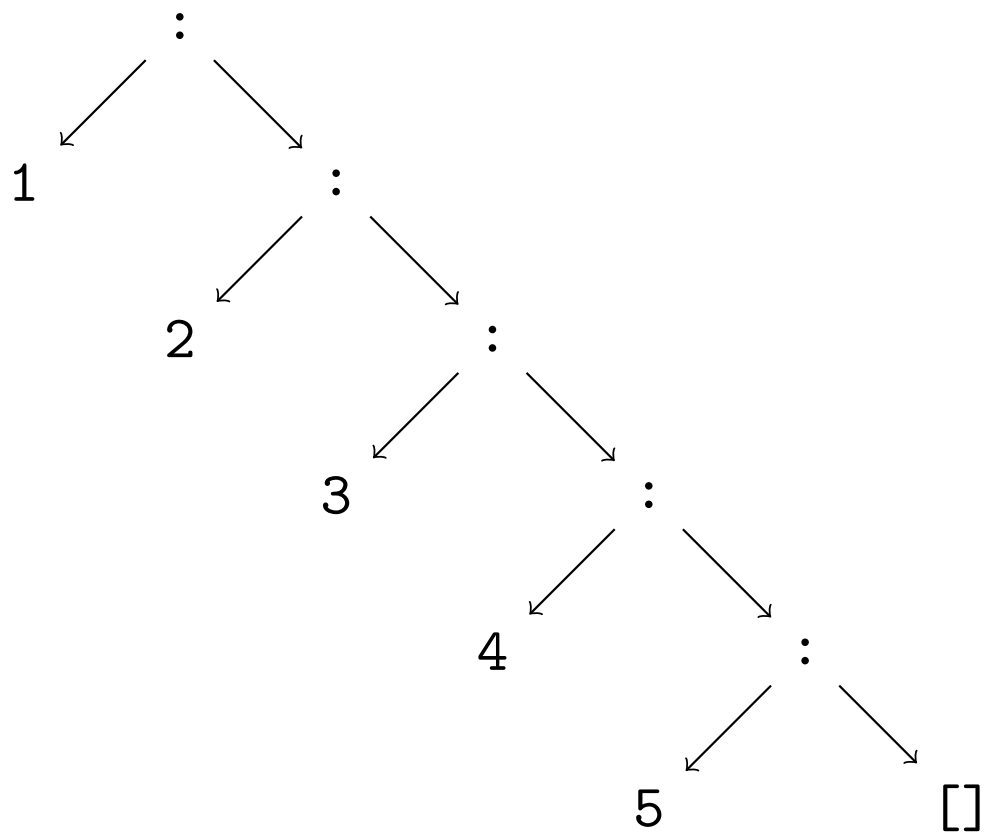


Pattern-Matching

Pattern `x:(y:xs)`



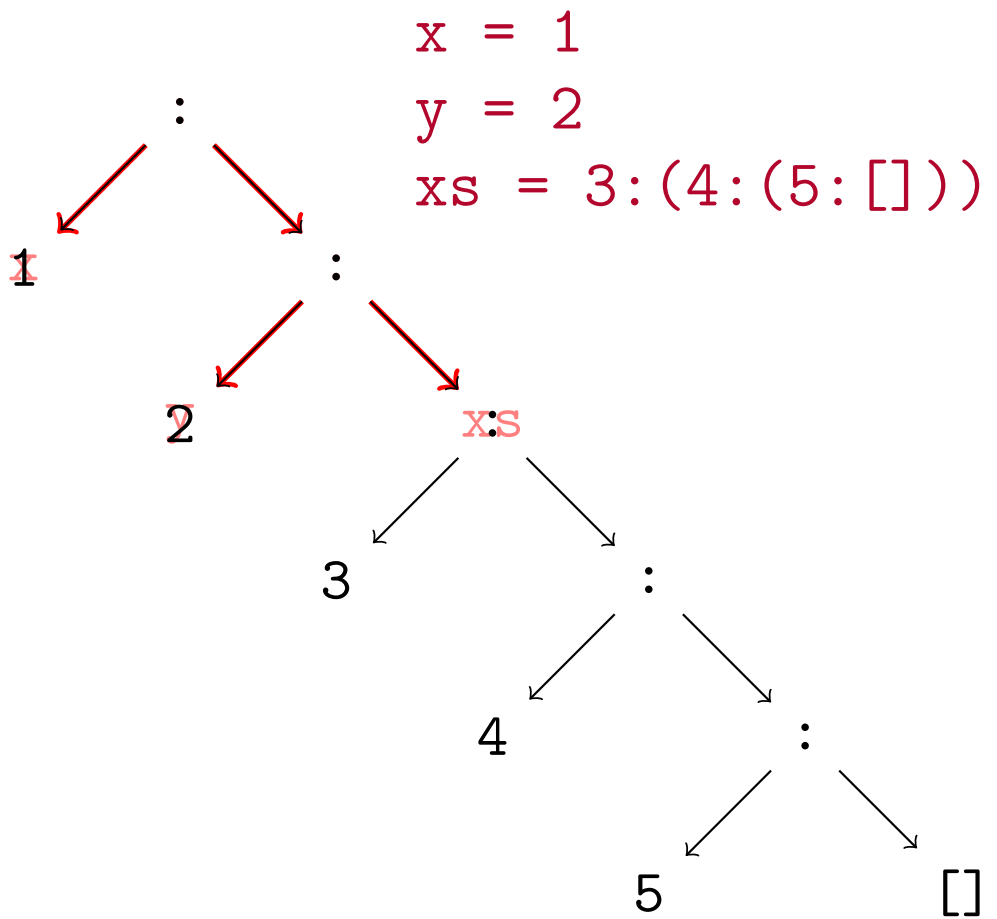
Liste



Pattern-Matching

Pattern `x:(y:xs)`

Liste



Beispiele: Pattern-Matching

```
isEmpty []     = True
isEmpty (x:xs) = False

-- alternativ:
isEmpty2 []    = True
isEmpty2 xs    = False -- matcht immer!

-- falsch!:
wrongIsEmpty xs    = False
wrongIsEmpty []    = True
```

```
 Warning: Pattern match(es) are overlapped
          In the definition of ‘wrongIsEmpty’: wrongIsEmpty [] = ...
*Main> wrongIsEmpty [1]
False
*Main> wrongIsEmpty []
False
*Main> isEmpty []
True
```

Letztes Element mit Pattern:

```
last2 []     = error "leere Liste"
last2 (x:[]) = x
last2 (x:xs) = last2 xs
```

`(x:[])` passt nur für einelementige Listen, alternativ `[x]`

Verschachtelte Pattern

Die Köpfe aller inneren Listen:

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

`((x:xs):ys)` ist ein verschachteltes Pattern!
Beispiel

```
*Main> heads [[1,2,3],[4,5,6],[],[7,8]]
[1,4,7]
```

Die map-Funktion

- map wendet eine Funktion auf alle Listenelemente an
- Definition von map:

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

- Beispiele:

```
*Main> map (\x -> x*x) [1..10]
[1,4,9,16,25,36,49,64,81,100]
*Main> map toUpper "Der Hund beisst die Katze."
"DER HUND BEISST DIE KATZE."
```

Die `filter`-Funktion

- `filter` liefert die Listenelemente, die das Prädikat erfüllen
- Definition von `filter`:

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

- Beispiele:

```
*Main> filter even [1..10]
[2,4,6,8,10]
*Main> filter isUpper "Der Hund beisst die Katze."
"DHK"
```

Einige vordefinierte Listenfunktionen (Auswahl)

- `length :: [a] -> Int` berechnet die Länge einer Liste
- `take :: Int -> [a] -> [a]` erwartet eine Zahl k und eine Liste xs und liefert die Liste der erste k Elemente von xs
- `drop :: Int -> [a] -> [a]` erwartet eine Zahl k und eine Liste xs und liefert xs ohne die der ersten k Elemente.
- `(++) :: [a] -> [a] -> [a]` „append“: hängt zwei Listen aneinander, kann infix in der Form `xs ++ ys` verwendet werden.
- `concat :: [[a]] -> [a]` glättet eine Liste von Listen. Z.B. `concat [xs,ys]` ist gleich zu `xs ++ ys`.
- `reverse :: [a] -> [a]` dreht die Reihenfolge der Elemente einer Liste um.

Strings

"Hallo Welt" ist nur syntaktischer Zucker für

```
['H','a','l','l','o',' ','W','e','l','t']
```

bzw.

```
'H':('a':('l':('l':('o':(' '(:'(W':('e':(l':('t':[])))))))))
```

```
*Main> head "Hallo Welt"
'H'
*Main> tail "Hallo Welt"
"allo Welt"
*Main> null "Hallo Welt"
False
*Main> null ""
True
*Main> last "Hallo Welt"
't'
```

Funktionen auf Strings

- `words :: String -> [String]`: Zerlegt eine Zeichenkette in eine *Liste von Worten*
- `unwords :: [String] -> String`: Macht aus einer Liste von Worten einen einzelnen String.
- `lines :: String -> [String]`: Zerlegt eine Zeichenkette in eine *Liste von Zeilen*
- `unlines :: [String] -> String`: Macht aus einer Liste von Zeilen einen einzelnen String.

Z.B.

```
numWords :: String -> Int
numWords text = length (words text)
```

Aufgabe

Aufgabe

Implementiere in Haskell eine Funktion `spal` die entscheidet, ob die Eingabe **ein Satzpalindrom** ist, d.h. ein Satz ist der Vorwärts wie rückwärts gelesen gleich ist, wenn man Leer- und Satzzeichen und Groß-/Kleinschreibung ignoriert.
Z.B. ist "Trug Tim eine so helle Hose nie mit Gurt?" ein Satzpalindrom.

Strukturiertes Vorgehen:

- Was sind die Eingaben und Ausgaben von `spal`?
- Wie lautet der Typ von `spal`?
- Welche Teilprobleme sind zum Palindromtest zu lösen?

Aufgabe

Aufgabe

Implementiere in Haskell eine Funktion `spal` die entscheidet, ob die Eingabe **ein Satzpalindrom** ist, d.h. ein Satz ist der Vorwärts wie rückwärts gelesen gleich ist, wenn man Leer- und Satzzeichen und Groß-/Kleinschreibung ignoriert.
Z.B. ist "Trug Tim eine so helle Hose nie mit Gurt?" ein Satzpalindrom.

Strukturiertes Vorgehen:

- Was sind die Eingaben und Ausgaben von `spal`?
  - Eingabe: Eine Zeichenkette vom Typ `String`
  - Ausgabe: Eine Wahrheitwert vom Typ `Bool`
- Wie lautet der Typ von `spal`?
  - `spal :: String -> Bool`
- Welche Teilprobleme sind zum Palindromtest zu lösen?
  - Normaler Palindromtest
  - Entfernen von Leerzeichen und Satzzeichen
  - Groß-/Kleinschreibung ignorieren

Satzpalindrom-Test

```
import Data.Char

-- Normaler Palindromtest
pal xs = reverse xs == xs

-- Entfernen von Leer- und Satzzeichen
remove = filter isAlphaNum

-- alles Kleinschreiben
tolower = map toLower

spal = pal . tolower . remove
```

Paare und Tupel

- Paare in Haskell: `(e1,e2)` z.B. `(1,'A')`
- Die Typen der Komponenten dürfen verschieden sein.

```
Main> :type ("Hallo",True)
("Hallo",True) :: ([Char], Bool)
Main> :type ([1,2,3],'A')
([1,2,3],False) :: (Num t) => ([t], Char)
*Main> :type (last, "Hallo" ++ "Welt")
(last, "Hallo" ++ "Welt") :: ([a] -> a, [Char])
```

Paare und Tupel (2)

Zugriffsfunktionen:

- `fst :: (a,b) -> a` liefert das linke Element eines Paares.
- `snd :: (a,b) -> b` liefert das rechte Element eines Paares.

```
*Main> fst (1,'A')
1
*Main> snd (1,'A')
'A'
*Main>
```

Pattern-Matching auf Paaren

```
eigenesFst (x,y) = x
eigenesSnd (x,y) = y
paarSumme  (x,y) = x+y
```

Tupel

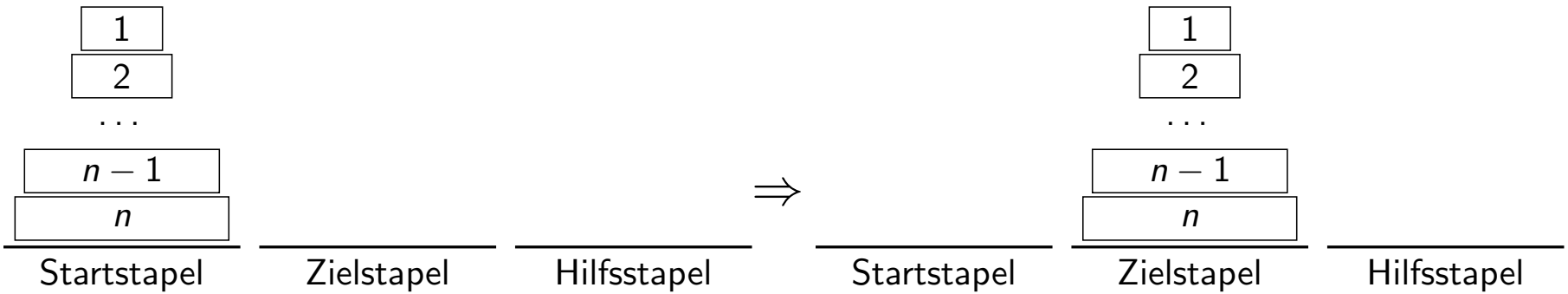
Wie Paare, aber mit mehr Komponenten.

```
*Main> :set +t
*Main> ('A',True,'B')
('A',True,'B')
it :: (Char, Bool, Char)
*Main> ([1,2,3],(True,'A',False,'B'),'B')
([1,2,3],(True,'A',False,'B'),'B')
it :: ([Integer], (Bool, Char, Bool, Char), Char)
```

Auch hier kann man Pattern verwenden:

```
erstes_aus_vier_tupel (w,x,y,z) = w
-- usw.
viertes_aus_vier_tupel (w,x,y,z) = z
```

Beispiel: Türme von Hanoi



Startzustand **Endzustand**

Lösen durch Rekursion: Rekursionsschritt

Lösen durch Rekursion: Rekursionsschritt

1. Verschiebe den Turm der Höhe $n-1$ **rekursiv** auf den Hilfsstapel

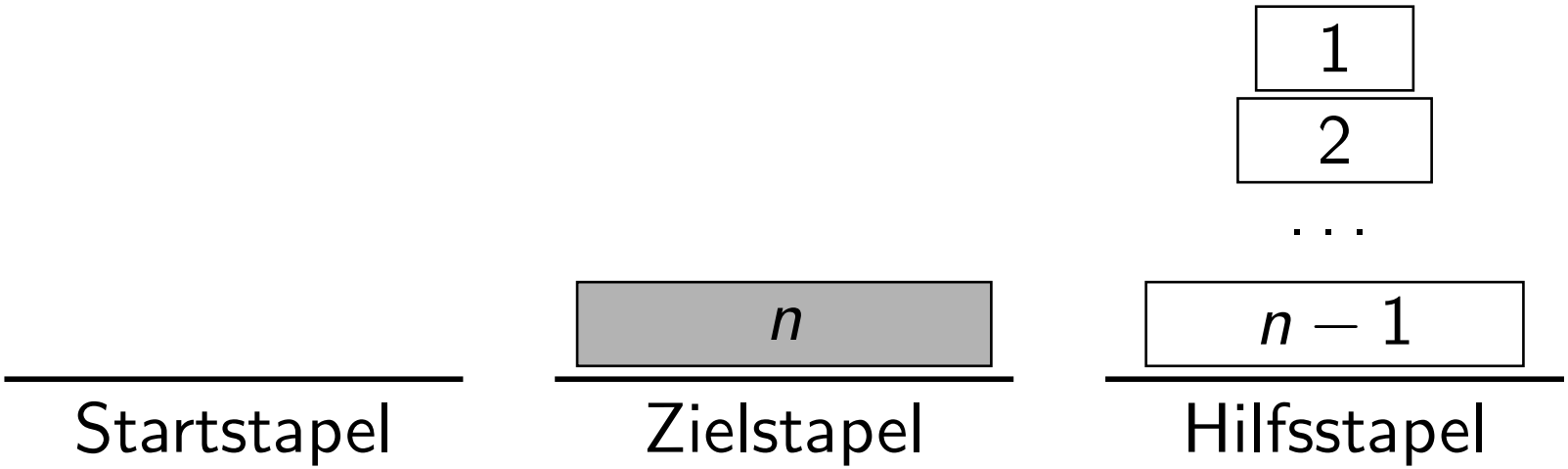
Lösen durch Rekursion: Rekursionsschritt

1. Verschiebe den Turm der Höhe $n-1$ **rekursiv** auf den Hilfsstapel

Lösen durch Rekursion: Rekursionsschritt

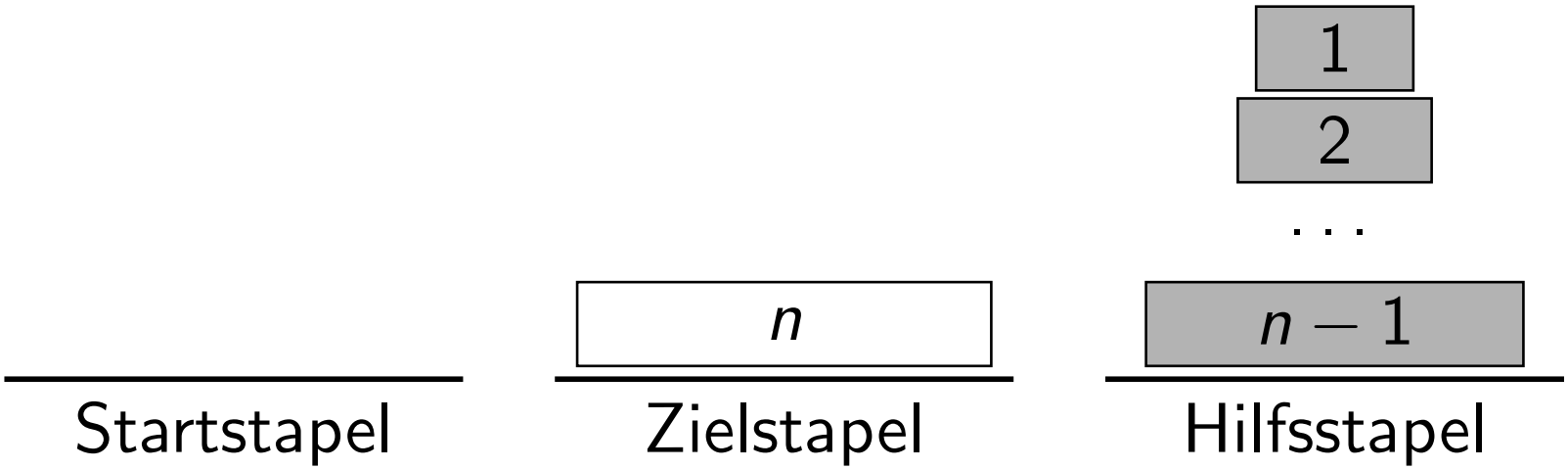
2. Verschiebe Scheibe n auf den Zielstapel

Lösen durch Rekursion: Rekursionsschritt



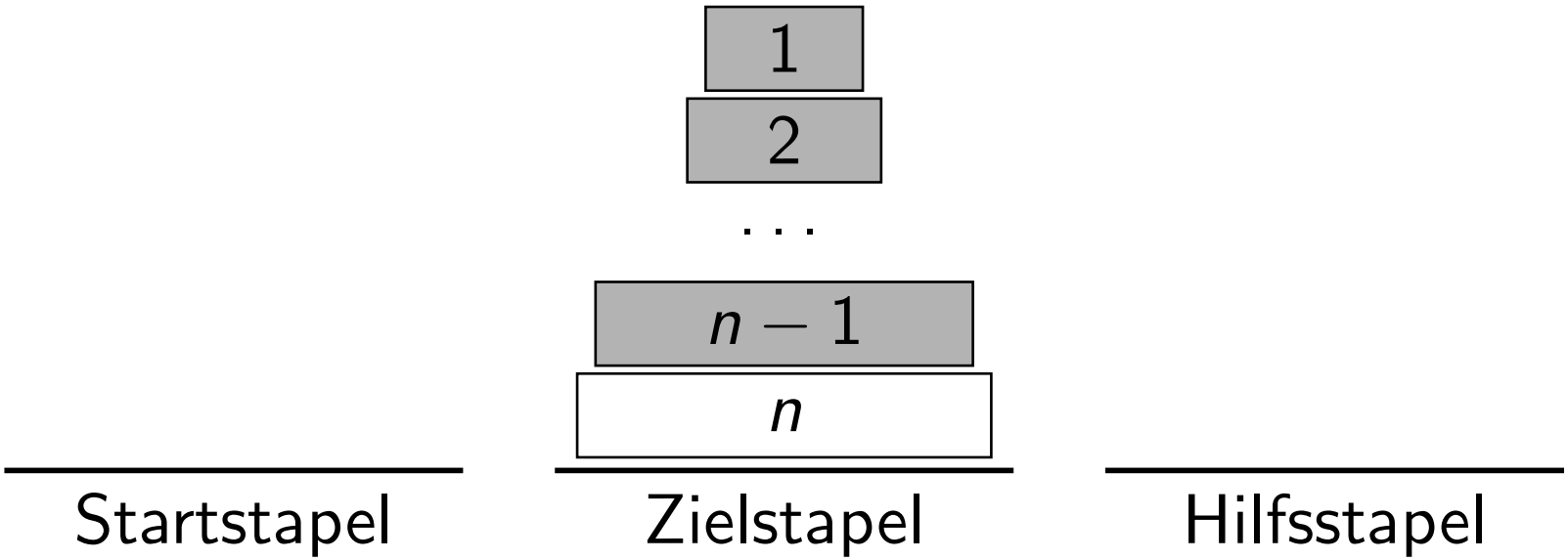
2. Verschiebe Scheibe n auf den Zielstapel

Lösen durch Rekursion: Rekursionsschritt



3. Verschiebe den Turm der Höhe $n-1$ **rekursiv** auf den Zielstapel

Lösen durch Rekursion: Rekursionsschritt



3. Verschiebe den Turm der Höhe $n-1$ **rekursiv** auf den Zielstapel

Türme von Hanoi in Haskell

Pseudo-Algorithmus:

verschiebe(n,start,ziel,hilf)

1. Wenn $n > 1$, dann verschiebe(n-1,start,hilf,ziel)
2. Schiebe Scheibe n von start auf ziel
3. Wenn $n > 1$, dann verschiebe(n-1,hilf,ziel,start)

Modellierung in Haskell

- Stapel sind benannt
  (am Anfang start = `"S"`, ziel = `"Z"`, hilf = `"H"`)
- Funktion `hanoi` erhält
  - Zahl n = Höhe des Stapels der verschoben werden soll
  - die drei Stapel
- Ausgabe: Liste von Zügen.
  Ein Zug ist ein Paar (x, y)
  = Schiebe oberste Scheibe vom Stapel x auf Stapel y

Die Funktion hanoi

```
-- Basisfall: 1 Scheibe verschieben
hanoi 1 start ziel hilf = [(start,ziel)]

-- Allgemeiner Fall:
hanoi n start ziel hilf =
   -- Schiebe den Turm der Hoehe n-1 von start zu hilf:
   (hanoi (n-1) start hilf ziel)     ++
   -- Schiebe n. Scheibe von start auf ziel:
   [(start,ziel)]                    ++
   -- Schiebe Turm der Hoehe n-1 von hilf auf ziel:
   (hanoi (n-1) hilf ziel start)
```

Starten mit

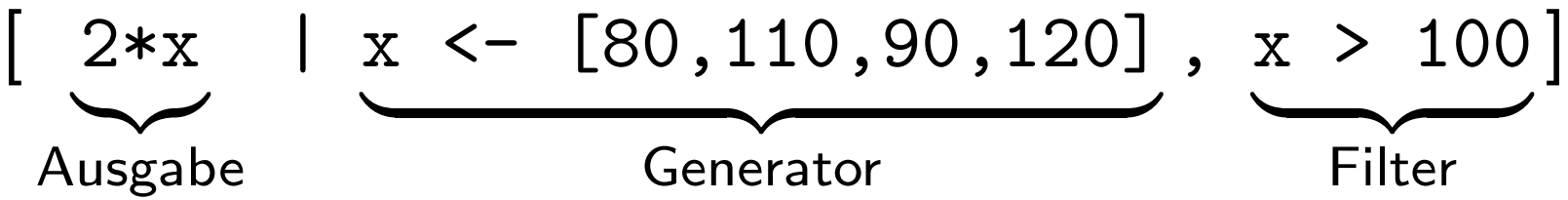
```
start_hanoi n = hanoi n "S" "Z" "H"
```

Beispiel mit 4 Scheiben

```
*Main> start_hanoi 4
[("S","H"),("S","Z"),("H","Z"),("S","H"),("Z","S"),
 ("Z","H"),("S","H"),("S","Z"),("H","Z"),("H","S"),
 ("Z","S"),("H","Z"),("S","H"),("S","Z"),("H","Z")]
```

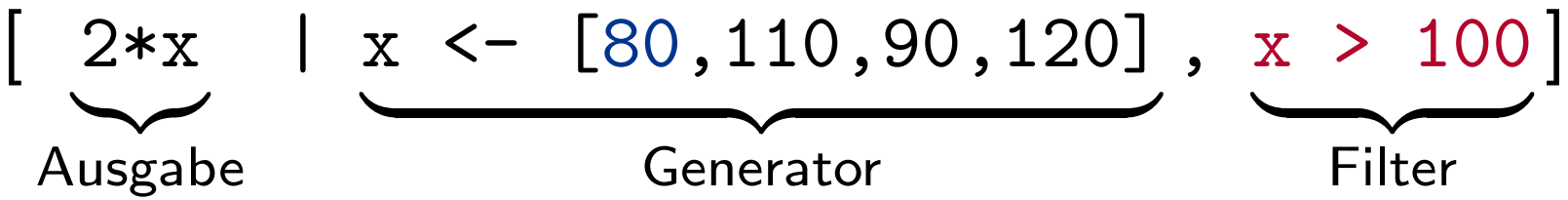
List Comprehensions

Listenerzeugung angelehnt an Zermelo-Fraenkel-Mengenausdrücke



List Comprehensions

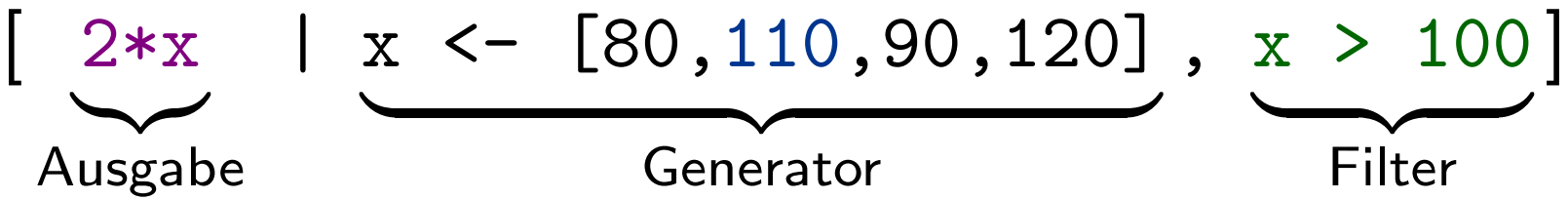
Listenerzeugung angelehnt an Zermelo-Fraenkel-Mengenausdrücke



ergibt [

List Comprehensions

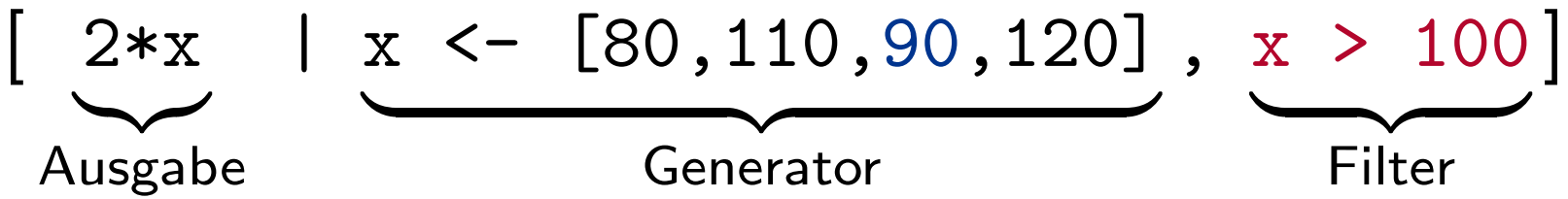
Listenerzeugung angelehnt an Zermelo-Fraenkel-Mengenausdrücke



ergibt [220,

List Comprehensions

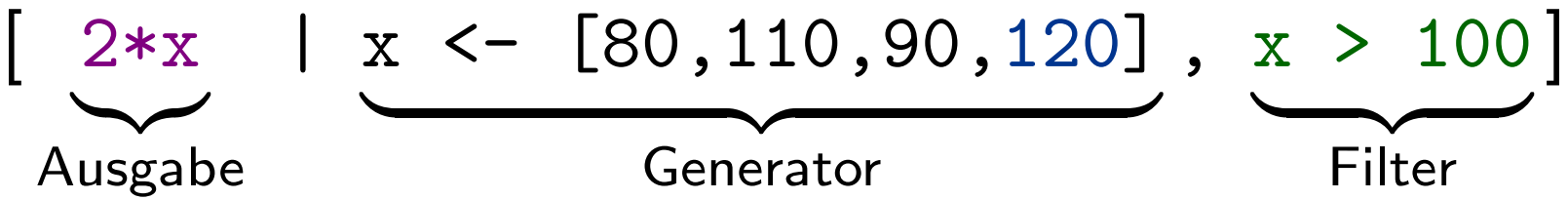
Listenerzeugung angelehnt an Zermelo-Fraenkel-Mengenausdrücke



ergibt [220,

List Comprehensions

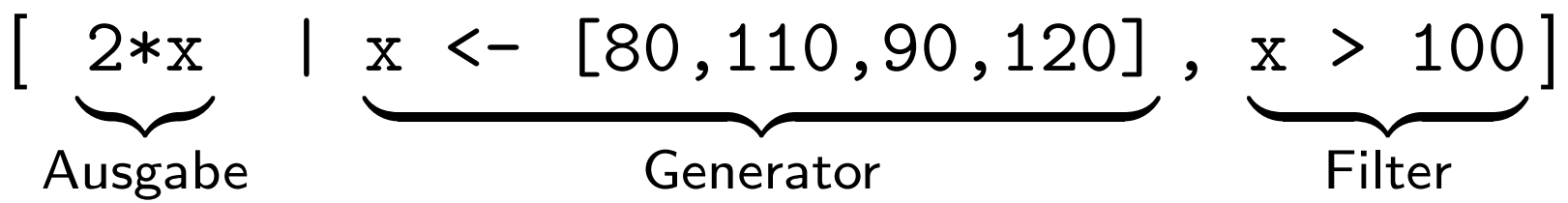
Listenerzeugung angelehnt an Zermelo-Fraenkel-Mengenausdrücke



ergibt [220,240]

List Comprehensions

Listenerzeugung angelehnt an Zermelo-Fraenkel-Mengenausdrücke



ergibt `[220,240]`

Beispiel:

```
[(x*x,y) | x <- [1..5]
         , y <- "aBcDeF"
         , even x
         , isUpper y]
```

ergibt

```
[(4,'B'),(4,'D'),(4,'F'),(16,'B'),(16,'D'),(16,'F')]
```