

Programmierung mit Haskell

Vorsemerkurs
Sommersemester 2022
Ronja Düffel

04. April 2022

Rekursion

```

erste_rekursive_Funktion x =
  if x <= 0 then 0           -- Rekursionsanfang
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt

```

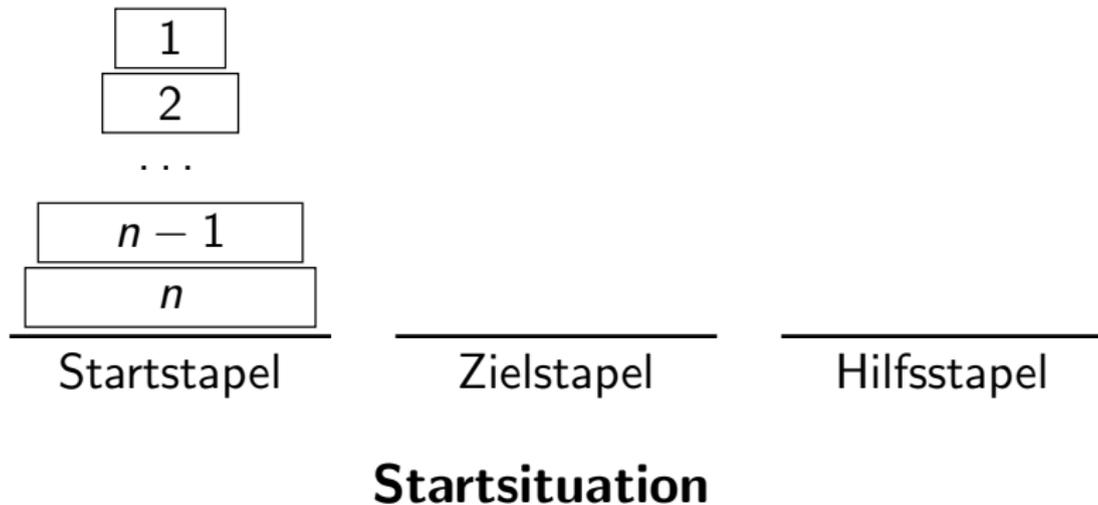
Ein Beispiel nachvollziehen:

```

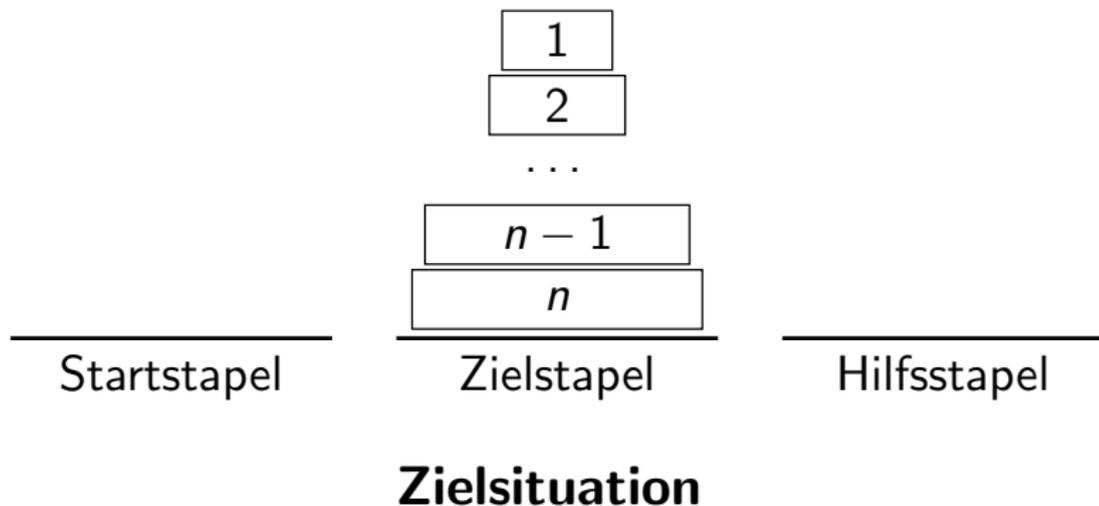
erste_rekursive_Funktion 5
= 5 + erste_rekursive_Funktion 4
= 5 + (4 + erste_rekursive_Funktion 3)
= 5 + (4 + (3 + erste_rekursive_Funktion 2))
= 5 + (4 + (3 + (2 + erste_rekursive_Funktion 1)))
= 5 + (4 + (3 + (2 + (1 + erste_rekursive_Funktion 0))))
= 5 + (4 + (3 + (2 + (1 + 0))))
= 15

```

Rekursion: Türme von Hanoi



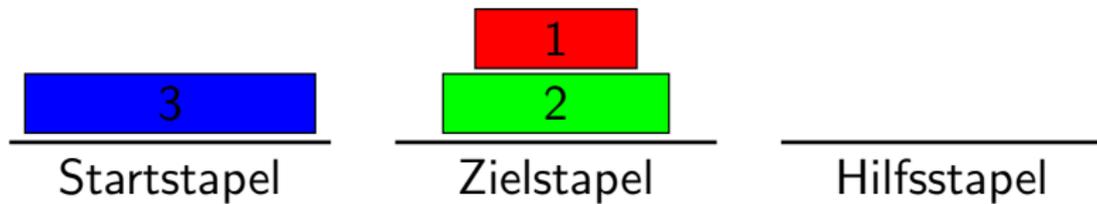
Rekursion: Türme von Hanoi

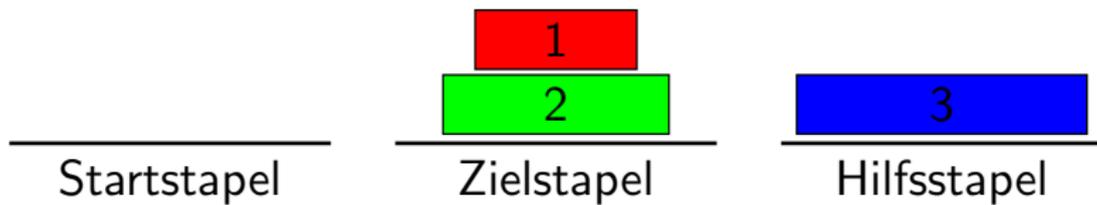


Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

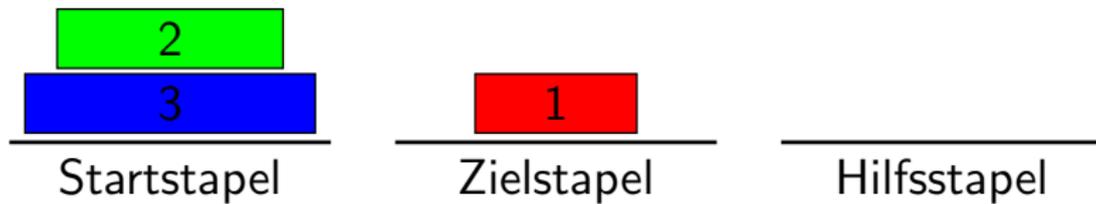
Beispiel $n = 3$ 

Beispiel $n = 3$ 

keine korrekte Lösung

Beispiel $n = 3$ 

zurück zum Anfang

Beispiel $n = 3$ 

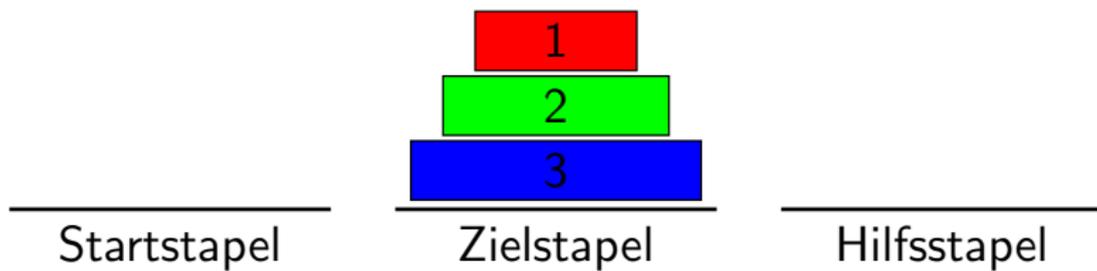
Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Korrekte Lösung

Lösen durch Rekursion: Rekursionanfang



$n = 1$: Verschiebe Scheibe von Startstapel auf Zielstapel

Lösen durch Rekursion: Rekursionanfang



$n = 1$: Verschiebe Scheibe von Startstapel auf Zielstapel

Lösen durch Rekursion: Rekursionanfang



$n = 1$: Verschiebe Scheibe von Startstapel auf Zielstapel

Lösen durch Rekursion: Rekursionsschritt



Lösen durch Rekursion: Rekursionsschritt



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

Lösen durch Rekursion: Rekursionsschritt



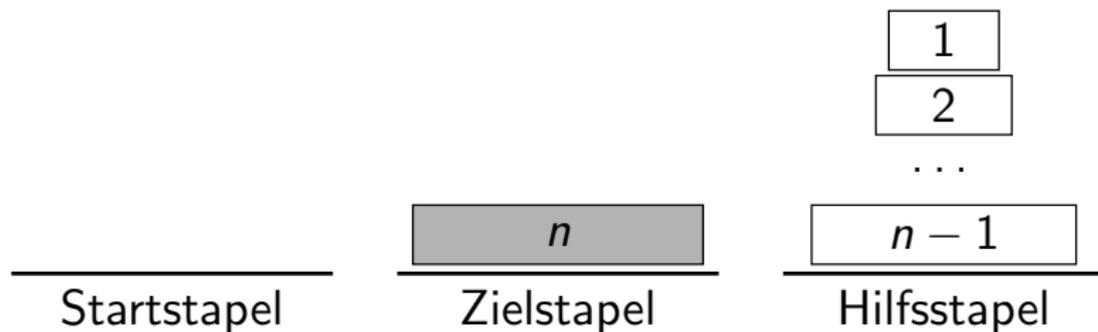
1. Verschiebe den Turm der Höhe $n - 1$ **rekursiv** auf den Hilfsstapel

Lösen durch Rekursion: Rekursionsschritt



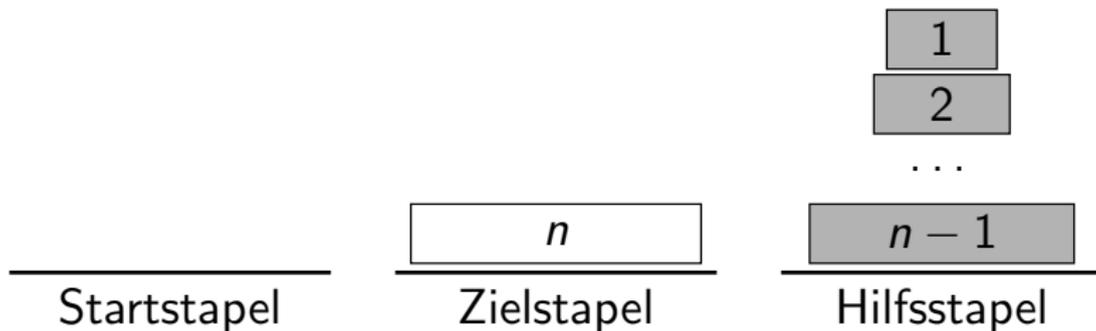
2. Verschiebe Scheibe n auf den Zielstapel

Lösen durch Rekursion: Rekursionsschritt



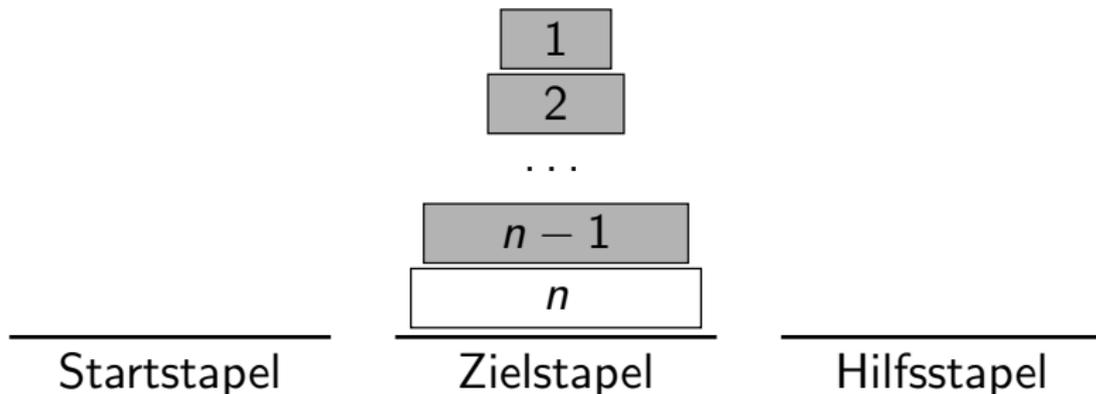
2. Verschiebe Scheibe n auf den Zielstapel

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

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)

- Rekursionanfang ist bei $n = 1$: keine rekursiven Aufrufe
- Beachte: Zwei rekursive Aufrufe pro Rekursionsschritt
- Haskell-Implementierung: Später

Rekursion, weitere Beispiele

Programmieren mit Haskell

Programmieren mit Listen

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]

```
Prelude> [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 erstellen

- Eckige Klammern und Kommata z.B. [1,2,3]

Listen erstellen

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

Listen erstellen

- Eckige Klammern und Kommata z.B. $[1, 2, 3]$
- Das ist jedoch nur **Syntaktischer Zucker**

Listen sind **rekursiv** definiert:

- „Rekursionsanfang“ ist die **leere Liste** $[]$ („Nil“)
- „Rekursionsschritt“ mit $:$ („Cons“)
 - x ein **Listenelement**
 - xs eine **Liste** (mit $n - 1$ Elementen)

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

$[1, 2, 3]$ ist in Wahrheit $1:(2:(3:[]))$

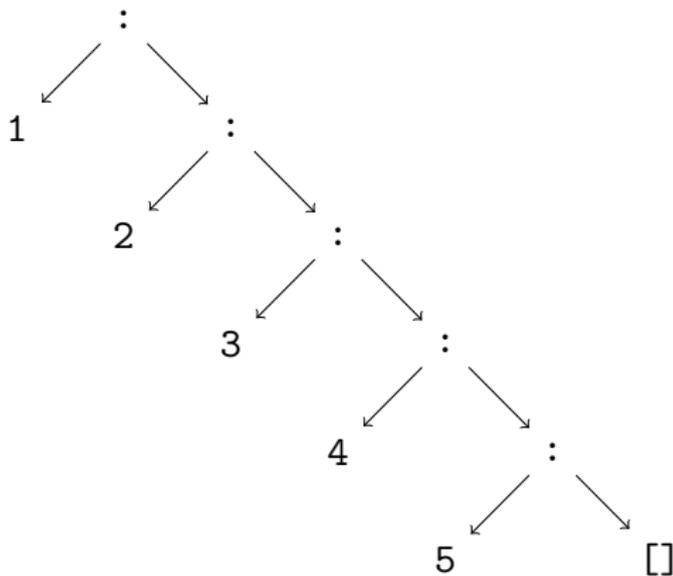
Typen:

$[] :: [a]$

$(:) :: a \rightarrow [a] \rightarrow [a]$

Interne Darstellung der Listen

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



Beispiel

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

```
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 die nicht-leere Eingabeliste ohne das erste Element.
- `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.

Beispiel

Funktion, die das letzte Element einer Liste liefert.

Listen zerlegen mit Pattern

In

$$f \text{ } par_1 \dots par_n = rumpf$$

dürfen par_i auch sog. **Pattern** sein.

Z.B.

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

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

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

$$x = 1$$

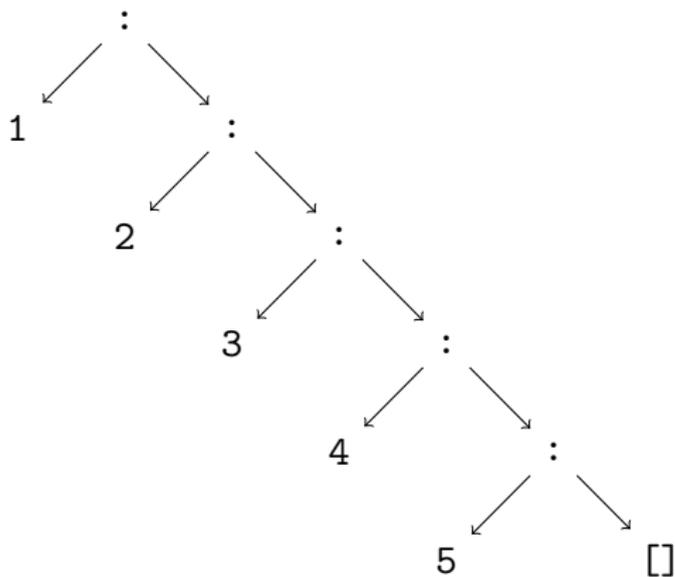
$$xs = (2:[])$$

Pattern-Matching

Pattern []

[]

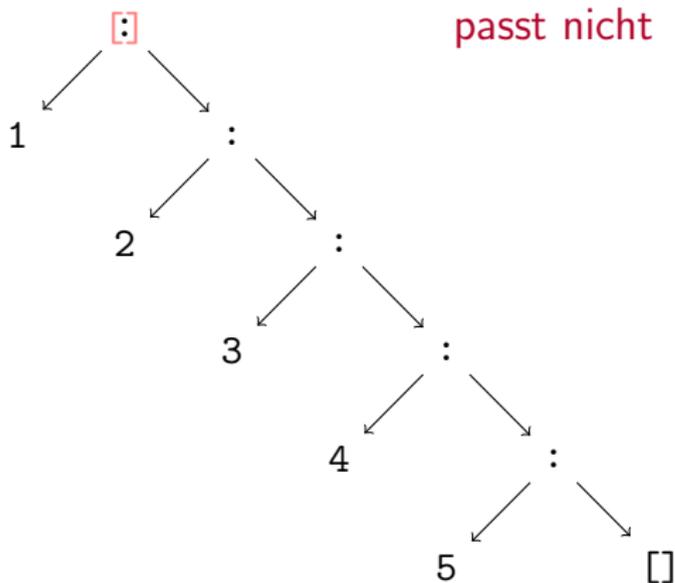
Liste



Pattern-Matching

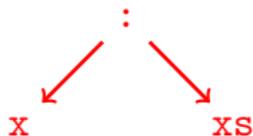
Pattern []

Liste

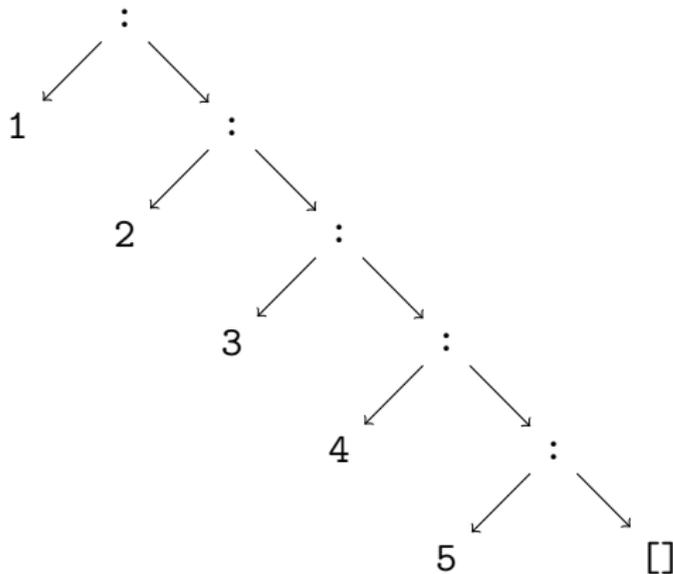


Pattern-Matching

Pattern $x:xs$



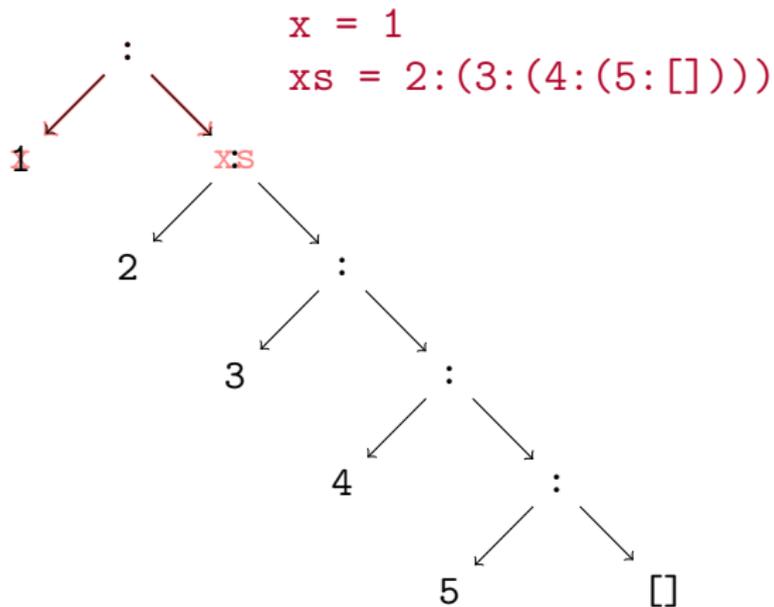
Liste



Pattern-Matching

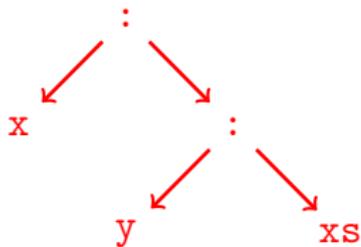
Pattern $x:xs$

Liste

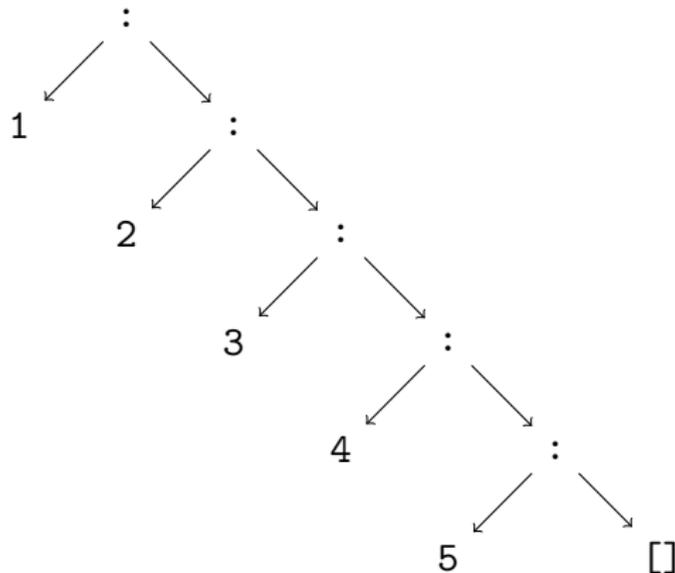


Pattern-Matching

Pattern $x:(y:xs)$



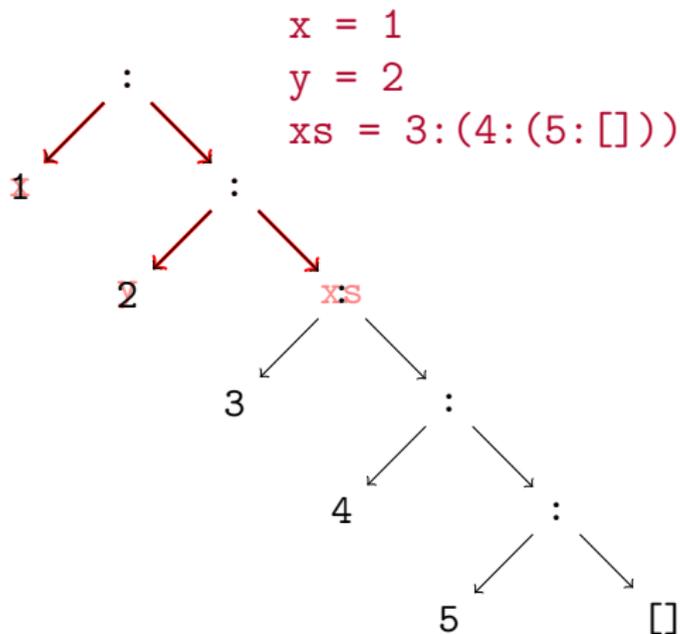
Liste



Pattern-Matching

Pattern $x:(y:xs)$

Liste



Letztes Element mit Pattern:

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 ersten 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.

Nochmal 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> letztesElement "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*

Z.B.

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

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 (letztesElement, "Hallo" ++ "Welt")
(letztesElement, "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

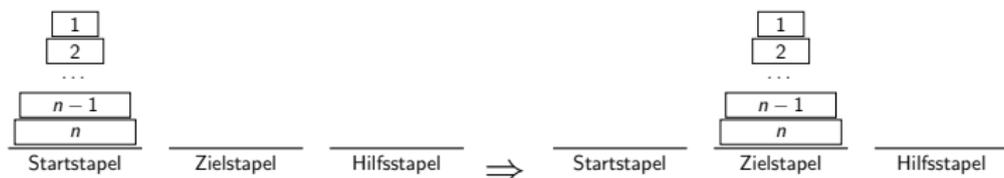
Wie Paare, aber mit mehr Komponenten.

```
*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
```

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 durchnummeriert
(am Anfang $\text{start} = 1$, $\text{ziel} = 2$, $\text{hilf} = 3$)
- 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 1 2 3
```

Ein Beispiel

```
*Main> start_hanoi 4  
[(1,3),(1,2),(3,2),(1,3),(2,1),(2,3),(1,3),(1,2),  
(3,2),(3,1),(2,1),(3,2),(1,3),(1,2),(3,2)]
```

Fragen?

?