Starker Start ins Studium

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Vorsemesterkurs Informatik

Grundlagen der Programmierung in Haskell

SoSe 2012

Stand der Folien: 28. März 2012

Übersicht

1. Ausdrücke und Typen
2. Funktionen
3. Rekursion
4. Listen
5. Paare und Tupel

Programmieren in Haskell

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Haskell-Programmieren:

- Im Wesentlichen formt man **Ausdrücke**
- z.B. arithmetische Ausdrücke `17*2+5*3`
- Ausführung: Berechnet den Wert eines Ausdrucks

```
*> 17*2+5*3 ↵
49
```

- Ausdrücke **zusammensetzen** durch:

  Anwendung von Funktionen auf Argumente,
  dabei sind Werte die kleinsten „Bauteile“

Programmieren in Haskell (2)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- In Haskell hat jeder Ausdruck (und Unterausdruck) einen **Typ**
- Typ = Art des Ausdrucks
  z.B. Buchstabe, Zahl, Liste von Zahlen, Funktion, ...
- Die Typen müssen zueinander passen:
  Z.B. verboten

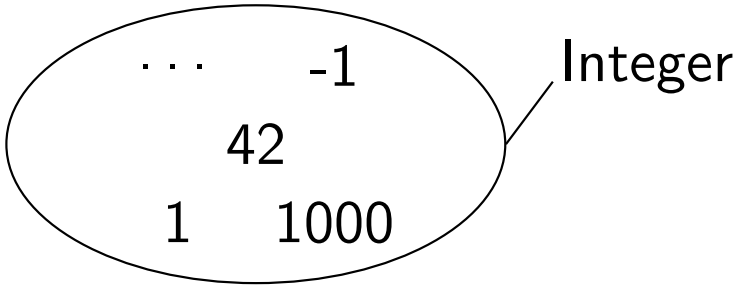
```
1 + "Hallo"
```

  Die Typen passen nicht zusammen (Zahl und Zeichenkette)

Typen

Andere Sichtweise:

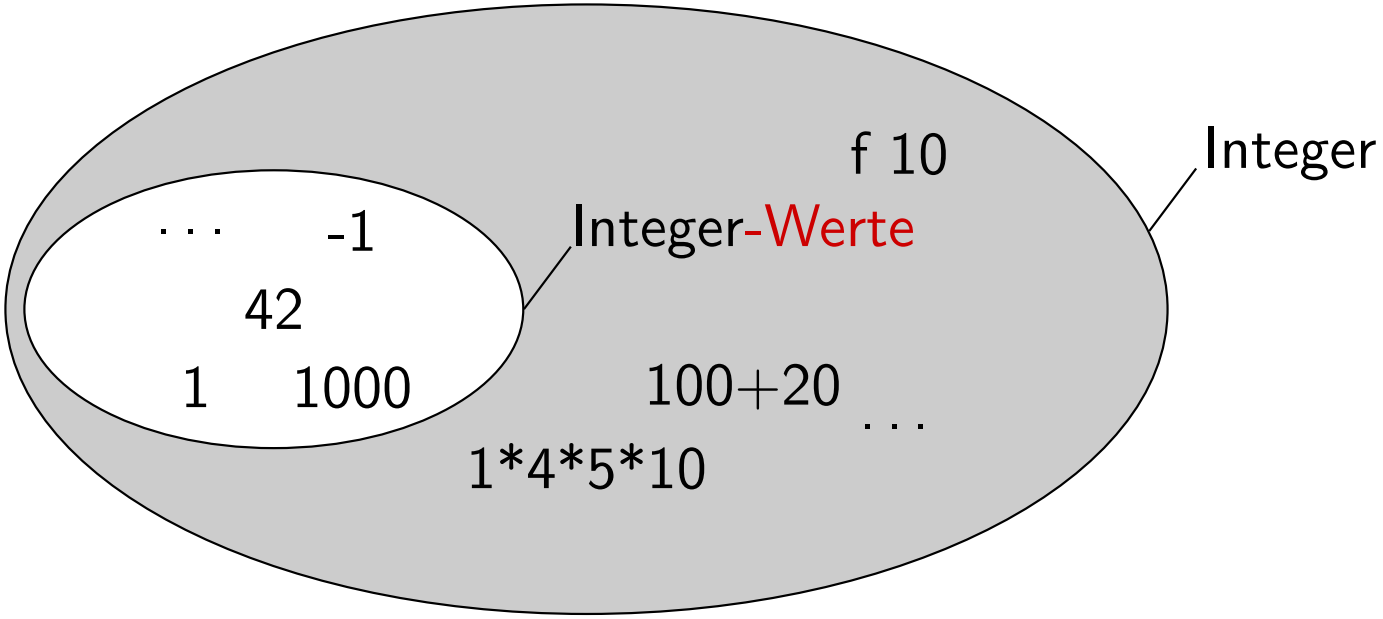
Typ = Menge von Werten



Typen

Andere Sichtweise:

Typ = Menge von ~~Werten~~ **Ausdrücken**



Typen (2)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Im GHCi Typen anzeigen lassen:

```
Prelude> :type 'C' ↵
'C' :: Char
```

Sprechweisen:

- „`'C'` hat den Typ `Char`“
- „`'C'` ist vom Typ `Char`“
- „`'C'` gehört zum Typ `Char`“

- `Char` ist der Typ in Haskell für Zeichen (engl. Character)
- Typnamen beginnen immer mit einem Großbuchstaben

Typen (3)

- Im GHCi: `:set +t`
  führt dazu, dass mit jedem Ergebnis auch dessen Typ gedruckt wird.

```
*Main> :set +t ↵
*Main> zwei_mal_Zwei ↵
4
it :: Integer
*Main> oft_fuenf_addieren ↵
55
it :: Integer
```

Typen (4)

- Der Typ `Integer` stellt beliebig große ganze Zahlen dar
- Man kann Typen auch selbst angeben:

Schreibweise *Ausdruck* : : Typ

```
*Main> 'C'::Char ↵
'C'
it :: Char
*Main> 'C'::Integer ↵

<interactive>:1:0:
  Couldn't match expected type `Integer'
  against inferred type `Char'
  In the expression: 'C' :: Integer
  In the definition of `it': it = 'C' :: Integer
```

Wahrheitswerte: Der Datentyp Bool

- Werte (Datenkonstruktoren) vom Typ `Bool`:
  - `True` steht für „wahr“
  - `False` steht für „falsch“
- Operationen (Funktionen):
  - Logische Negation: `not`: liefert True für False und False für True
  - Logisches Und: `a && b`: nur `True`, wenn `a` und `b` zu `True` auswerten
  - Logisches Oder: `a || b`: `True`, sobald `a` oder `b` zu `True` auswertet.

```
*Main> not True ↵
False
it :: Bool
*Main> not False ↵
True
it :: Bool
*Main> True && True ↵
True
it :: Bool
```

```
*Main> False && True ↵
False
it :: Bool
*Main> False || False ↵
False
it :: Bool
*Main> True || False ↵
True
it :: Bool
```

Ganze Zahlen: `Int` und `Integer`

- Der Typ `Int` umfasst ganze Zahlen beschränkter Größe (je nach Rechner, z.B. -2^{31} bis $2^{31} - 1$)
- Der Typ `Integer` umfasst ganze Zahlen beliebiger Größe

Ganze Zahlen: `Int` und `Integer`

- Der Typ `Int` umfasst ganze Zahlen beschränkter Größe (je nach Rechner, z.B. -2^{31} bis $2^{31}-1$)
- Der Typ `Integer` umfasst ganze Zahlen beliebiger Größe
- Darstellung der Zahlen ist identisch z.B. 1000
- Defaulting: `Integer`, wenn es nötig ist, sonst offenlassen:

Ganze Zahlen: Int und Integer

- Der Typ Int umfasst ganze Zahlen beschränkter Größe (je nach Rechner, z.B. -2^{31} bis $2^{31}-1$)
- Der Typ Integer umfasst ganze Zahlen beliebiger Größe
- Darstellung der Zahlen ist identisch z.B. 1000
- Defaulting: Integer, wenn es nötig ist, sonst offenlassen:

```
Prelude> :type 1000
1000 :: (Num t) => t
Prelude> :set +t
Prelude> 1000
1000
it :: Integer
```

- In etwa 1000 ist vom Typ t, wenn t ein numerischer Typ ist.
- Genauer: (Num t) => ist eine sog. Typklassenbeschränkung
- Int und Integer sind numerische Typen (haben Instanzen für Num)

Gleitkommazahlen

- Typen `Float`und `Double`(mit doppelter Genauigkeit)
- Kommastelle wird mit e. Punkt dargestellt
- Typklasse dazu: `Fractional`

```
Prelude> :type 10.5 ↵
10.5 :: (Fractional t) => t
Prelude> 10.5 ↵
10.5
it :: Double
```

- Beachte: Das Rechnen mit solchen Kommazahlen ist ungenau!

Zeichen und Zeichenketten

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- Der Typ `Char` repräsentiert Zeichen
  Darstellung: Zeichen in einfache Anführungszeichen, z.B. `'A'`
- Spezielle Zeichen (Auswahl):

Darstellung in Haskell	Zeichen, das sich dahinter verbirgt
`'\\'`	Backslash \
`'\''`	einfaches Anführungszeichen '
`'\"'`	doppeltes Anführungszeichen "
`'\n'`	Zeilenumbruch
`'\t'`	Tabulator

- Zeichenketten: Typ `String`
  Dastellung in doppelten Anfühungszeichen, z.B. `"Hallo"`.
- Genauer ist der Typ `String` gleich zu `[Char]`
  d.h. eine Liste von Zeichen (Listen behandeln wir später)

Beispiel Zeichenketten


```
> :set +t
> "Ein \n\'mehrzeiliger\'\nText mit \"Anfuehrungszeichen\""
"Ein \n'mehrzeiliger'\nText mit \"Anfuehrungszeichen\""
it :: [Char]
> putStrLn "Ein \n\'mehrzeiliger\'\nText mit \"Anfuehrungszeichen\""
Ein
'mehrzeiliger'
Text mit "Anfuehrungszeichen"
it :: ()
```

Operatoren auf Zahlen

- Operatoren auf Zahlen in Haskell:
  Addition `+`, Substraktion `-`, Multiplikation `*` und Division `/`
- Beispiele: `3 * 6`, `10.0 / 2.5`, `4 + 5 * 4`
- Beim `-` muss man aufpassen, da es auch für negative Zahlen benutzt wird

```
Prelude> 2 * -2 ↵

<interactive>:1:0:
    Precedence parsing error
        cannot mix ‘*’ [infixl 7] and prefix ‘-’ [infixl 6]
        in the same infix expression
Prelude> 2 * (-2) ↵
-4
```

Vergleichsoperationen

Gleichheitstest `==` und
Ungleichheitstest `/=`

```
Prelude> 1 == 3 ↵
False
Prelude> 3*10 == 6*5 ↵
True
Prelude> True == False ↵
False
Prelude> False == False ↵
True
Prelude> 2*8 /= 64 ↵
True
Prelude> 2+8 /= 10 ↵
False
Prelude> True /= False ↵
True
```

größer: `>`, größer oder gleich: `>=`
kleiner: `<`, kleiner oder gleich: `<=`

```
Prelude> 5 >= 5 ↵
True
Prelude> 5 > 5 ↵
False
Prelude> 6 > 5 ↵
True
Prelude> 4 < 5 ↵
True
Prelude> 4 < 4 ↵
False
Prelude> 4 <= 4 ↵
True
```

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Funktionen

Funktionen verwenden, Funktionstypen, Funktionen definieren

Funktionen

- Sei f eine Funktion, die den Rest einer Division mit Rest berechnet.
- In der Mathematik würde man z.B. schreiben: $f(10, 3) = 1$

Funktionen

- Sei f eine Funktion, die den Rest einer Division mit Rest berechnet.
- In der Mathematik würde man z.B. schreiben: $f(10, 3) = 1$
- Schreibweise in Haskell: `f 10 3` oder auch `(f 10 3)`
- Grund: Man darf in Haskell partiell anwenden, z.B. `(f 10)`

Funktionen

- Sei f eine Funktion, die den Rest einer Division mit Rest berechnet.
- In der Mathematik würde man z.B. schreiben: $f(10, 3) = 1$
- Schreibweise in Haskell: `f 10 3` oder auch `(f 10 3)`
- Grund: Man darf in Haskell partiell anwenden, z.B. `(f 10)`
- Obige Funktion heißt in Haskell: `mod`
- Passend dazu: `div`: Ganzzahligen Anteil der Division mit Rest

```
Prelude> mod 10 3 ↵
1
Prelude> div 10 3 ↵
3
Prelude> mod 15 5 ↵
0
Prelude> div 15 5 ↵
3
Prelude> (div 15 5) + (mod 8 6) ↵
5
```

Präfix und Infix

- `+`, `*`, ... werden infix verwendet:
  zwischen den Argumenten, z.B. `5+6`
- `mod`, `div` werden präfix verwendet:
  vor den Argumenten, z.B. `mod 10 3`

Präfix und Infix

- `+`, `*`, ... werden infix verwendet:
  zwischen den Argumenten, z.B. `5+6`
- `mod`, `div` werden präfix verwendet:
  vor den Argumenten, z.B. `mod 10 3`
- Präfix-Operatoren infix verwenden:
  In Hochkommata setzen (⇧ + `)
  z.B. `10 ‘mod‘ 3`
- Infix-Operatoren präfix verwenden:
  In runde Klammern setzen
  z.B. `(+) 5 6`

Funktionstypen (1)

- Auch Funktionen haben einen Typ
- Funktion hat mehrere Eingaben und eine Ausgabe
- Jeder der Eingaben und die Ausgabe haben einen Typ
- Verkettung durch ->

Funktionstypen (1)

- Auch Funktionen haben einen Typ
- Funktion hat mehrere Eingaben und eine Ausgabe
- Jeder der Eingaben und die Ausgabe haben einen Typ
- Verkettung durch ->

```
Prelude> :type not ↵
not :: Bool -> Bool
Prelude> :type (&&) ↵
(&&) :: Bool -> Bool -> Bool
Prelude> :type (||) ↵
(||) :: Bool -> Bool -> Bool
```

- `not`: Eine Eingabe vom Typ `Bool` und Ausgabe vom Typ `Bool`
- `(&&)`: Zwei Eingaben vom Typ `Bool` und Ausgabe vom Typ `Bool`
- `(||)`: Zwei Eingaben vom Typ `Bool` und Ausgabe vom Typ `Bool`

Funktionstypen (2)

Allgemein: f erwartet n Eingaben, dann ist der Typ von f:

$$f :: \underbrace{\text{Typ}_1}_{\substack{\text{Typ des}\\ \text{1. Arguments}}} \texttt{ -> } \underbrace{\text{Typ}_2}_{\substack{\text{Typ des}\\ \text{2. Arguments}}} \texttt{ -> } \ldots \texttt{ -> } \underbrace{\text{Typ}_n}_{\substack{\text{Typ des}\\ n\text{. Arguments}}} \texttt{ -> } \underbrace{\text{Typ}_{n+1}}_{\substack{\text{Typ des}\\ \text{Ergebnisses}}}$$

- `->` in Funktionstypen ist rechts-geklammert
- `(&&) :: Bool -> Bool -> Bool`
  entspricht `(&&) :: Bool -> (Bool -> Bool)`
  und nicht `(&&) :: (Bool -> Bool) -> Bool`
- Das passt zur partiellen Anwendung:

```
(&&)    ::       Bool -> (Bool -> Bool)
       True :: Bool
-------------------------------------
((&&) True) ::          (Bool -> Bool)
```

Funktionstypen (3)

Typen von `mod` und `div`:

```
mod :: Integer -> Integer -> Integer
div :: Integer -> Integer -> Integer
```

In Wirklichkeit:

```
Prelude> :type mod ↵
mod :: (Integral a) => a -> a -> a
Prelude> :type div ↵
div :: (Integral a) => a -> a -> a
```

In etwa: Für alle Typen `a` die `Integral`-Typen sind,
hat `mod` den Typ `a -> a -> a`

Funktionstypen (4)

```
Prelude> :type (==) ↵
(==) :: (Eq a) => a -> a -> Bool
Prelude> :type (<) ↵
(<) :: (Ord a) => a -> a -> Bool
Prelude> :type (<=) ↵
(<=) :: (Ord a) => a -> a -> Bool
Prelude> :type (+) ↵
(+) :: (Num a) => a -> a -> a
Prelude> :type (-) ↵
(-) :: (Num a) => a -> a -> a
```

Funktionstypen (5)

Die Typen müssen stets passen, sonst gibt's einen Typfehler:

```
Prelude> :type not
not :: Bool -> Bool
Prelude> not 'C'
<interactive>:1:4:
  Couldn't match expected type 'Bool' against inferred type 'Char'
  In the first argument of 'not', namely 'C'
  In the expression: not 'C'
  In the definition of 'it': it = not 'C'
```

Manchmal "merkwürdige" Fehlermeldung:

```
Prelude> not 5
<interactive>:1:4:
    No instance for (Num Bool)
      arising from the literal '5' at <interactive>:1:4
    Possible fix: add an instance declaration for (Num Bool)
    In the first argument of 'not', namely '5'
    In the expression: not 5
    In the definition of 'it': it = not 5
```

Funktionen selbst definieren

```
verdopple x = x + x
```

In der Mathematik würde man schreiben $\texttt{verdopple}(x) = x + x$

Funktionen selbst definieren

```
verdopple x = x + x
```

In der Mathematik würde man schreiben $\texttt{verdopple}(x) = x + x$

Allgemein:

funktion_Name par_1 ... par_n = *Haskell_Ausdruck*

wobei

- par_i: Formale Parameter, z.B. Variablen `x`, `y`, ...
- Die par_i dürfen rechts im *Haskell_Ausdruck* verwendet werden
- *funktion_Name* muss mit Kleinbuchstaben oder einem Unterstrich beginnen

Funktionen selbst definieren


```
verdopple :: Integer -> Integer
verdopple x = x + x
```

In der Mathematik würde man schreiben `verdopple`$(x) = x + x$

Allgemein:

funktion_Name par_1 ... par_n = *Haskell_Ausdruck*

wobei

- par_i: Formale Parameter, z.B. Variablen `x`, `y`, ...
- Die par_i dürfen rechts im *Haskell_Ausdruck* verwendet werden
- *funktion_Name* muss mit Kleinbuchstaben oder einem Unterstrich beginnen

Man darf auch den Typ angeben!

Funktion testen

```
Prelude> :load programme/einfacheFunktionen.hs ↵
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs)
Ok, modules loaded: Main.
*Main> verdopple 5 ↵
10
*Main> verdopple 100 ↵
200
*Main> verdopple (verdopple (2*3) + verdopple (6+9)) ↵
84
```

Fallunterscheidung: if-then-else

Syntax: `if` b `then` e_1 `else` e_2

```
verdoppleGerade :: Integer -> Integer
verdoppleGerade x = if even x then verdopple x else x
```

`even` testet, ob eine Zahl gerade ist:

```
even x = x `mod` 2 == 0
```

```
*Main> :reload ↵
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )
Ok, modules loaded: Main.
*Main> verdoppleGerade 50 ↵
100
*Main> verdoppleGerade 17 ↵
17
```

Fallunterscheidung: if-then-else (2)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Verschachteln von if-then-else:

```
jenachdem :: Integer -> Integer
jenachdem x = if x < 100 then 2*x else
              if x <= 1000 then 3*x else x
```

Falsche Einrückung:

```
jenachdem x =
if x < 100 then 2*x else
  if x <= 1000 then 3*x else x
```

```
Prelude> :reload ↵
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )

programme/einfacheFunktionen.hs:9:0:
    parse error (possibly incorrect indentation)
Failed, modules loaded: none.
Prelude>
```

Noch ein Beispiel


```
verdoppeln_oder_verdreifachen :: Bool -> Integer -> Integer
verdoppeln_oder_verdreifachen b x =
  if b then 2*x else 3*x
```

```
*Main> verdoppeln_oder_verdreifachen True 10
20
*Main> verdoppeln_oder_verdreifachen False 10
30
```

`verdoppeln` mithilfe von `verdoppeln_oder_verdreifachen`:

```
verdoppeln2 :: Integer -> Integer
verdoppeln2 x = verdoppeln_oder_verdreifachen True x
-- oder auch:
verdoppeln3 :: Integer -> Integer
verdoppeln3 = verdoppeln_oder_verdreifachen True
```

`verdoppeln3`: keine Eingabe, **Ausgabe ist eine Funktion**

Higher-Order Funktionen

- D.h.: **Rückgabewerte** dürfen in Haskell auch Funktionen sein
- Auch Argumente (Eingaben) dürfen Funktionen sein:

```
wende_an_und_addiere f x y = (f x) + (f y)
```

```
*Main> wende_an_und_addiere verdopple 10 20 ↵
60
*Main> wende_an_und_addiere jenachdem 150 3000 ↵
3450
```

Daher spricht man auch von Funktionen höherer Ordnung!

Nochmal Typen

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Typ von `wende_an_und_addiere`

```
wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer -> Integer
```

```
wende_an_und_addiere f x y = (f x) + (f y)
```

Nochmal Typen

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Typ von `wende_an_und_addiere`

$$\texttt{wende_an_und_addiere} :: \underbrace{\texttt{(Integer -> Integer)}}_{\text{Typ von } \texttt{f}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\text{Typ von } \texttt{x}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\text{Typ von } \texttt{y}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\text{Typ des Ergebnisses}}$$

$$\texttt{wende_an_und_addiere f x y} = \texttt{(f x)} + \texttt{(f y)}$$

Nochmal Typen

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Typ von `wende_an_und_addiere`

```
wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer -> Integer
```

```
wende_an_und_addiere f x y = (f x) + (f y)
```

Achtung: Im Typ

```
(Integer -> Integer) -> Integer -> Integer -> Integer
```

darf man die Klammern nicht weglassen:

```
Integer -> Integer -> Integer -> Integer -> Integer
```

denn das entspricht

```
Integer -> (Integer -> (Integer -> (Integer -> Integer))).
```

Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da Typvariablen in Haskell erlaubt sind, spricht man von polymorphen Typen

Für Typvariablen kann man Typen einsetzen!

Polymorphe Typen

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da Typvariablen in Haskell erlaubt sind, spricht man von polymorphen Typen

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (a -> a) -> a -> a
```

Polymorphe Typen

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da Typvariablen in Haskell erlaubt sind, spricht man von polymorphen Typen

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

Polymorphe Typen

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da Typvariablen in Haskell erlaubt sind, spricht man von polymorphen Typen

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B. `a = Bool`

```
zweimal_anwenden :: (a -> a) -> a -> a
```

Polymorphe Typen

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da Typvariablen in Haskell erlaubt sind, spricht man von polymorphen Typen

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B. `a = Bool`

```
zweimal_anwenden :: (Bool -> Bool) -> Bool -> Bool
```

Polymorphe Typen


```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da Typvariablen in Haskell erlaubt sind, spricht man von polymorphen Typen

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B. `a = Bool`

```
zweimal_anwenden :: (Bool -> Bool) -> Bool -> Bool
```

z.B. `a = Char -> Char`

```
zweimal_anwenden :: (a -> a) -> a -> a
```

Polymorphe Typen

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da Typvariablen in Haskell erlaubt sind, spricht man von polymorphen Typen

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B. `a = Bool`

```
zweimal_anwenden :: (Bool -> Bool) -> Bool -> Bool
```

z.B. `a = Char -> Char`

```
zweimal_anwenden::((Char->Char)->(Char->Char))->(Char->Char)->(Char->Char)
```

Polymorphe Typen (2)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Der GHCi setzt beim Anwenden automatisch die richtigen Typen ein

```
*> :type verdopple
verdopple :: Integer -> Integer
*> zweimal_anwenden verdopple 10
40
*> :type not
not :: Bool -> Bool
*> zweimal_anwenden not True
True
*> :type vergleiche True
vergleiche True :: Bool -> String
*> zweimal_anwenden (vergleiche True) True
<interactive>:1:18:
  Couldn't match expected type 'Bool' against inferred type 'String'
  In the first argument of 'zweimal_anwenden', namely '(vergleiche True)'
  In the expression: zweimal_anwenden (vergleiche True)
```

Formale Parameter

Gesucht:

Funktion erhält zwei Eingaben und liefert
`"Die Eingaben sind gleich"`,
wenn die beiden Eingaben gleich sind.

Formale Parameter

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Gesucht:

Funktion erhält zwei Eingaben und liefert
`"Die Eingaben sind gleich"`,
wenn die beiden Eingaben gleich sind.

Falscher Versuch:

```
sonicht x x = "Die Eingaben sind gleich!"
```

```
    Conflicting definitions for ‘x’
    In the definition of ‘sonicht’
Failed, modules loaded: none.
```

Die formalen Parameter müssen unterschiedliche Namen haben.

```
vergleiche x y =
  if x == y then "Die Eingaben sind gleich!" else ""
```

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Rekursion

„*Wer Rekursion verstehen will, muss Rekursion verstehen.*“

Rekursion

Eine Funktion ist **rekursiv**, wenn sie sich selbst aufrufen kann.

```
f x y z = ... (f a b c) ...
```

oder z.B. auch

```
f x y z = ... (g a b) ...

g x y = ... (f c d e) ...
```

Rekursion (2)

Bei Rekursion muss man aufpassen:

```
endlos_eins_addieren x = endlos_eins_addieren (x+1)
```

```
    endlos_eins_addieren 0
--> endlos_eins_addieren 1
--> endlos_eins_addieren 2
--> endlos_eins_addieren 3
--> endlos_eins_addieren 4
--> ...
```

`endlos_eins_addieren 0` terminiert nicht!

Rekursion (3)

So macht man es richtig:

- **Rekursionsanfang**: Der Fall, für den sich die Funktion nicht mehr selbst aufruft, sondern abbricht
- **Rekursionsschritt**: Der rekursive Aufruf

Darauf achten, dass der Rekursionsanfang stets erreicht wird.

Beispiel:

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

Rekursion (4)

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

Was berechnet `erste_rekursive_Funktion`?

- `erste_rekursive_Funktion` n ergibt 0 für $n \leq 0$
- $n > 0$?

Testen:

```
*Main> erste_rekursive_Funktion 5
15
*Main> erste_rekursive_Funktion 10
55
*Main> erste_rekursive_Funktion 11
66
*Main> erste_rekursive_Funktion 12
78
*Main> erste_rekursive_Funktion 1
1
```

Rekursion (5)

```
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 (6)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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

Allgemein:

```
  erste_rekursive_Funktion x
= x + erste_rekursive_Funktion (x-1)
= x + (x-1)  + erste_rekursive_Funktion (x-2))
= x + (x-1) + (x-2) + erste_rekursive_Funktion (x-3)))
= ...
```

Rekursion (6)

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

Allgemein:

```
  erste_rekursive_Funktion x
= x + erste_rekursive_Funktion (x-1)
= x + (x-1)  + erste_rekursive_Funktion (x-2))
= x + (x-1) + (x-2) + erste_rekursive_Funktion (x-3)))
= ...
```

Das ergibt $x + (x-1) + (x-2) + \ldots + 0 = \sum_{i=0}^{x} i$

Rekursion (7)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Warum ist Rekursion nützlich?

- Man kann damit schwierige Probleme einfach lösen

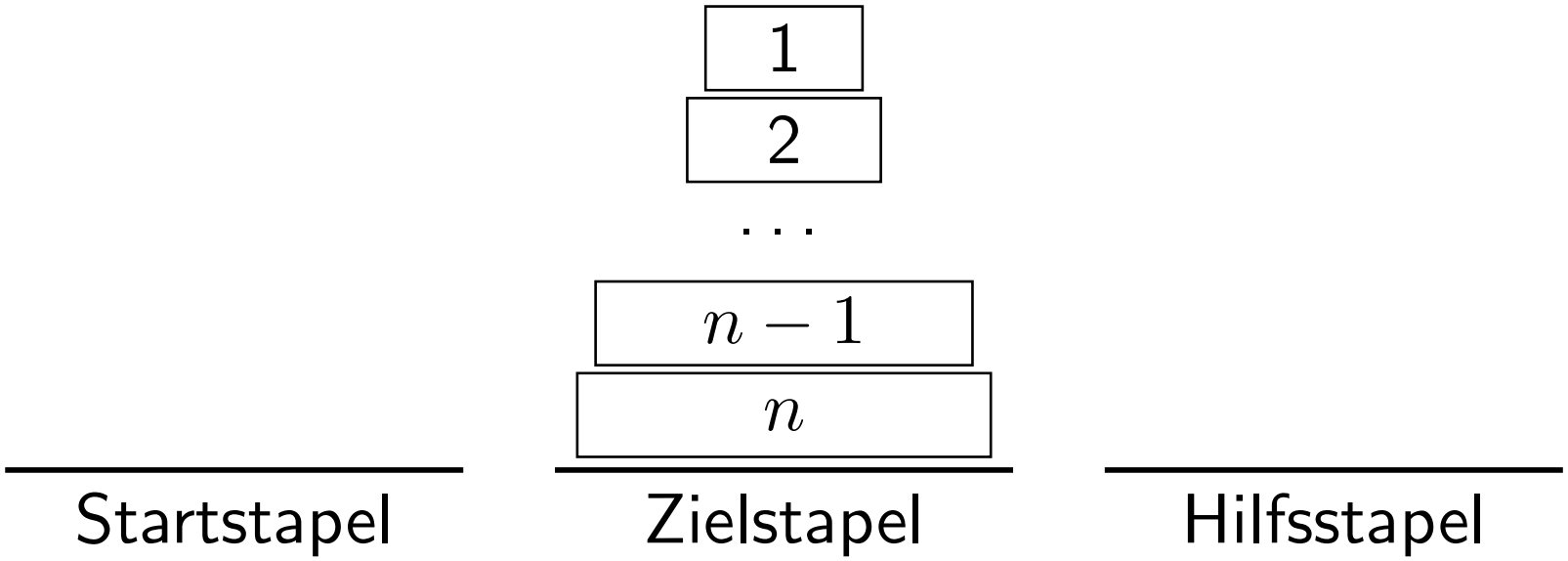
Wie geht man vor? (z.B. implementiere $f(n) = \sum_{i=1}^{n} i$)

- Rekursionsanfang:
  Der einfache Fall, für den man die Lösung direkt kennt
  $f(0) = \sum_{i=0}^{0} i = 0$
- Rekursionsschritt:
  Man löst ganz wenig selbst, bis das Problem etwas kleiner ist.
  Das (immer noch große) Restproblem erledigt die Rekursion,
  $f(n) = \sum_{i=0}^{n} i = x + \sum_{i=0}^{n-1} i = n + f(n-1)$ (für $x > 0$)

Rekursion: Türme von Hanoi

Startsituation

Rekursion: Türme von Hanoi



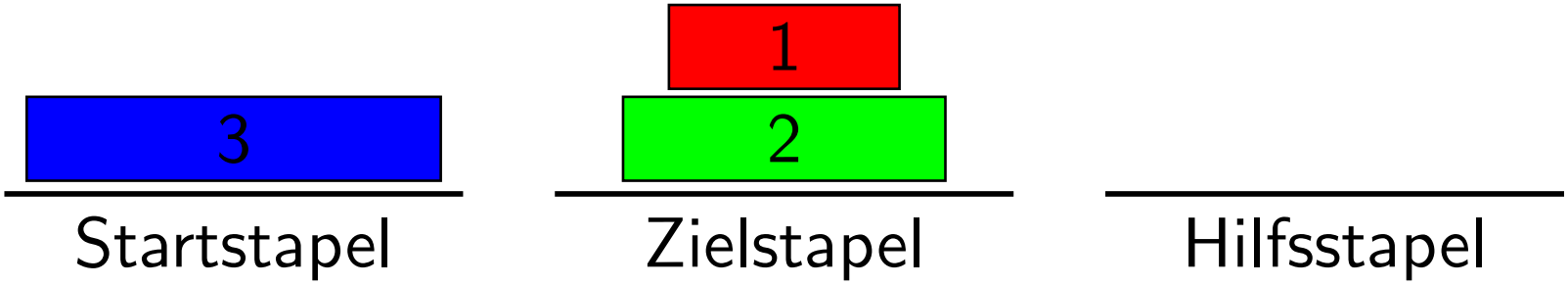
Zielsituation

Beispiel $n = 3$

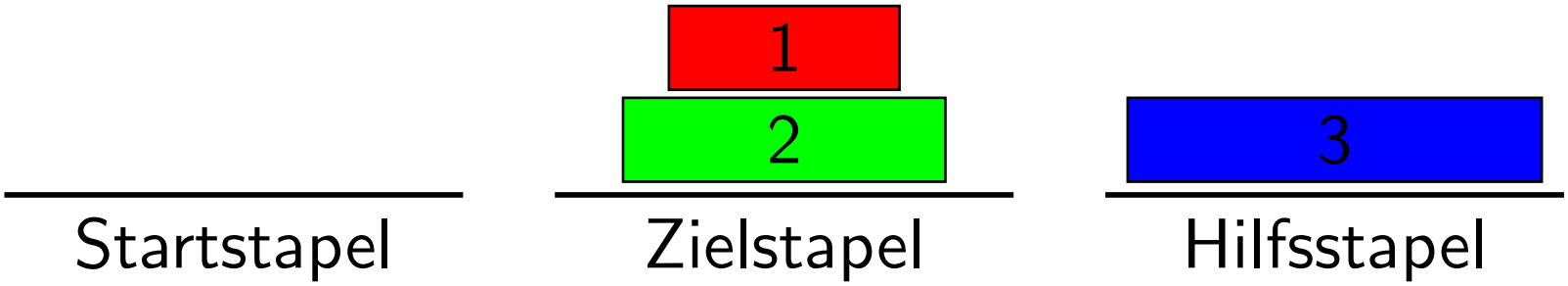
Beispiel $n = 3$

Beispiel $n = 3$

Beispiel $n = 3$



Beispiel $n = 3$



Beispiel $n = 3$

Beispiel $n = 3$

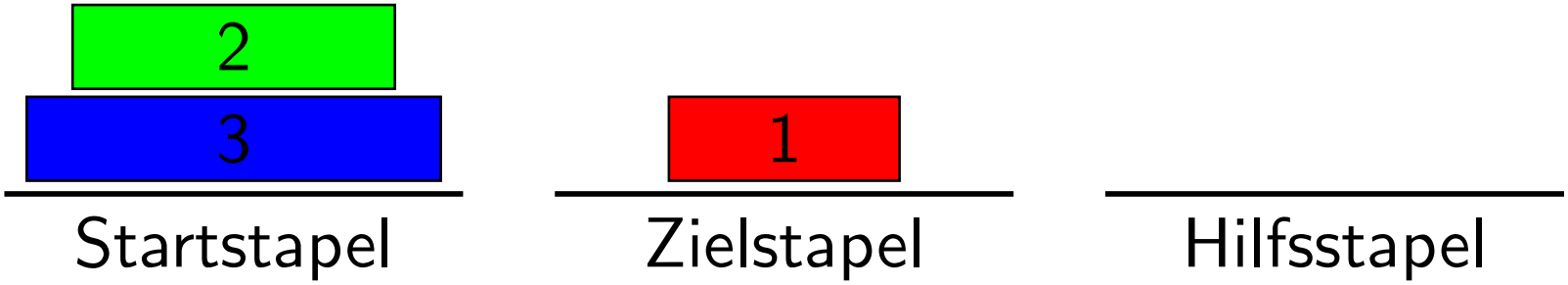
Beispiel $n = 3$

keine korrekte Lösung

Beispiel $n = 3$

zurück zum Anfang

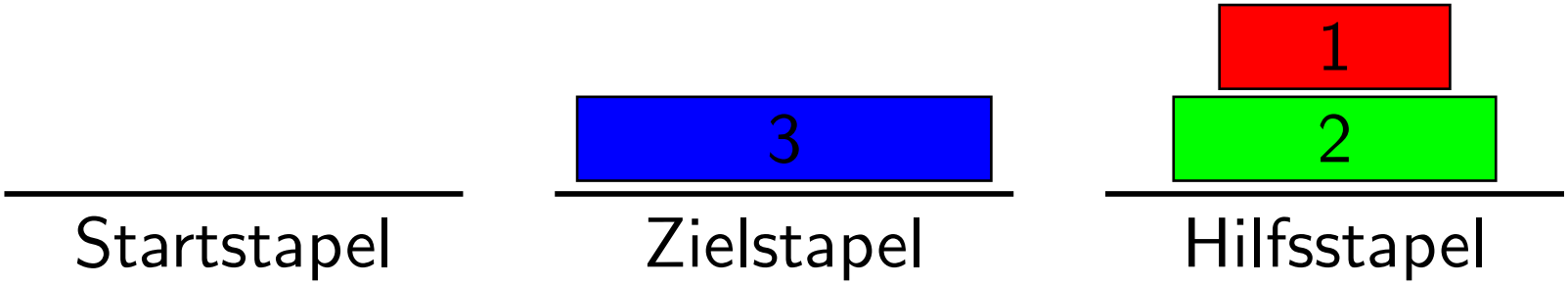
Beispiel $n = 3$



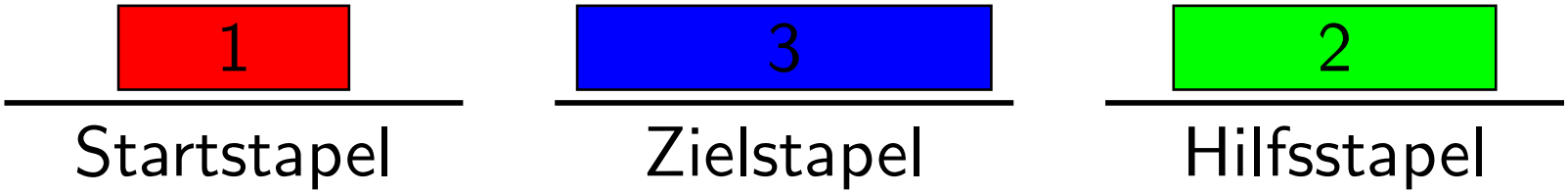
Beispiel $n = 3$

Beispiel $n = 3$

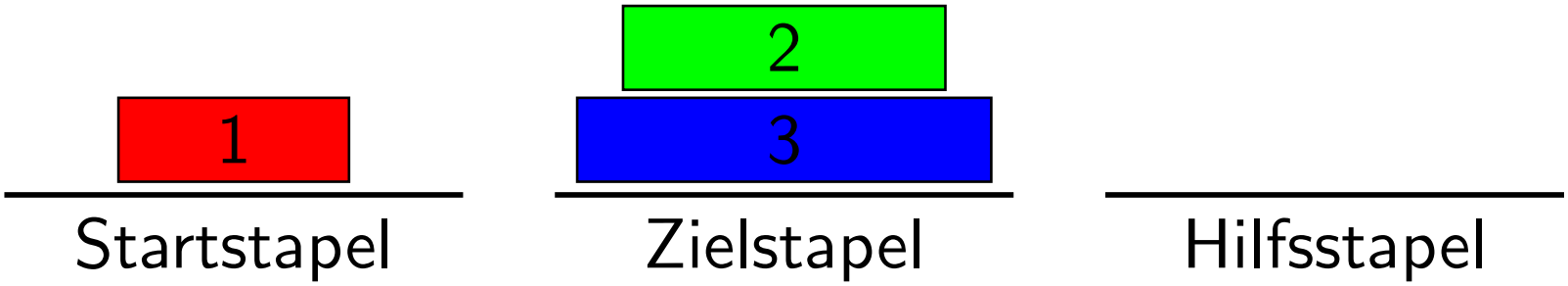
Beispiel $n = 3$



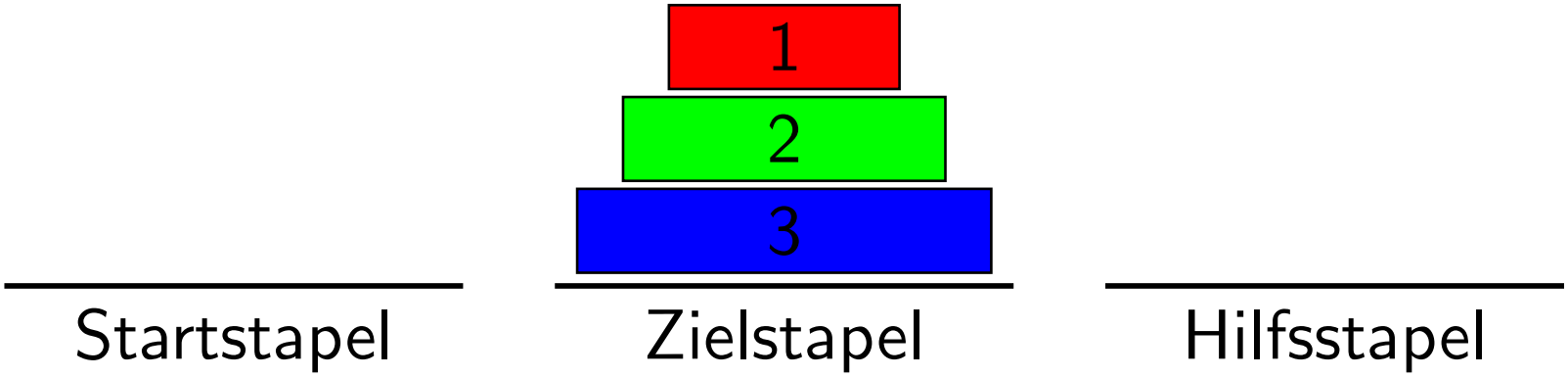
Beispiel $n = 3$



Beispiel $n = 3$



Beispiel $n = 3$



Korrekte Lösung

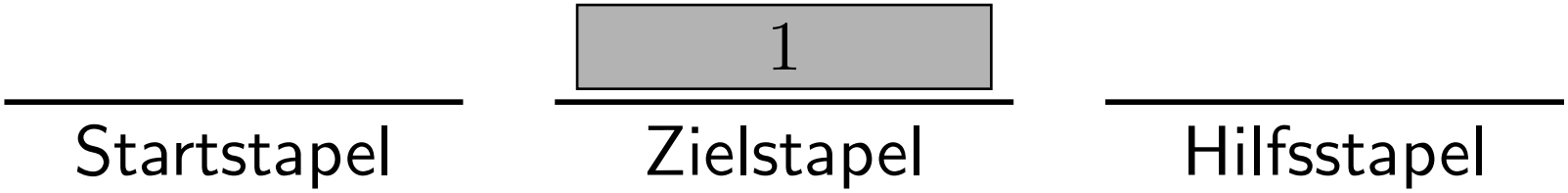
Lösen durch Rekursion: Rekursionanfang

$n = 1$: 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

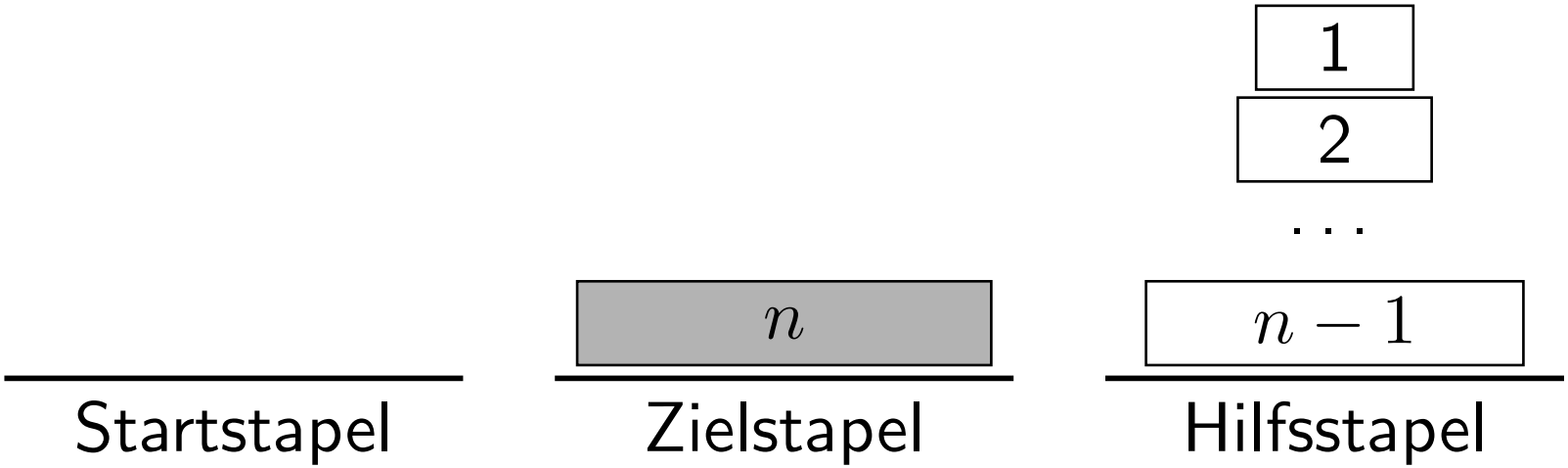
Lösen durch Rekursion: Rekursionsschritt

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

Lösen durch Rekursion: Rekursionsschritt

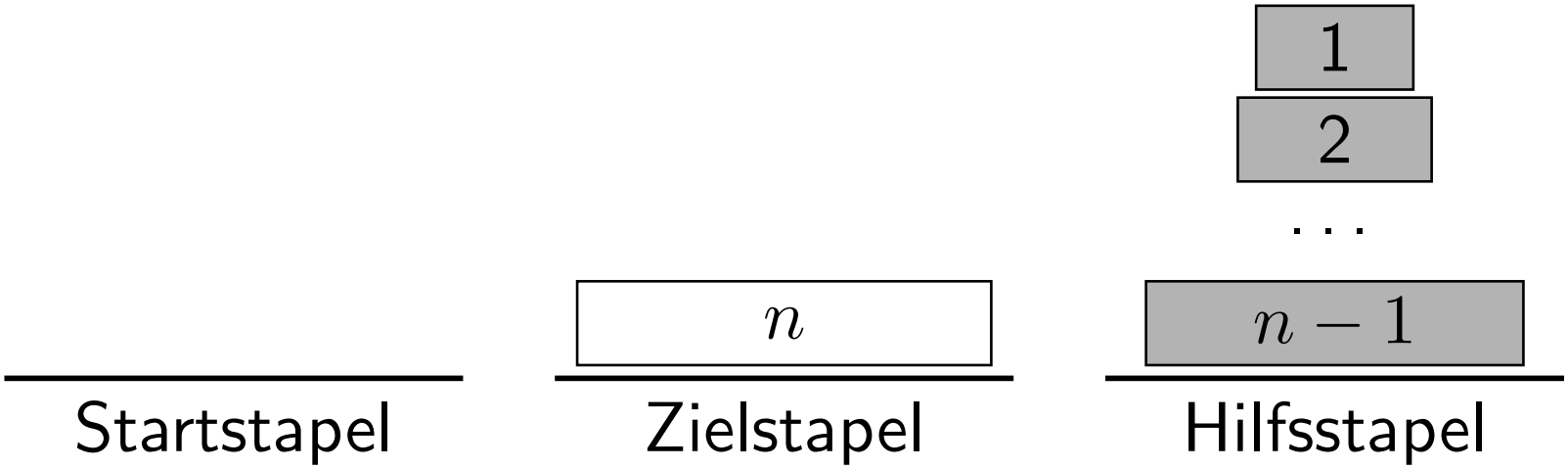
2. Verschiebe Scheibe n auf den Zielstapel

Lösen durch Rekursion: Rekursionsschritt



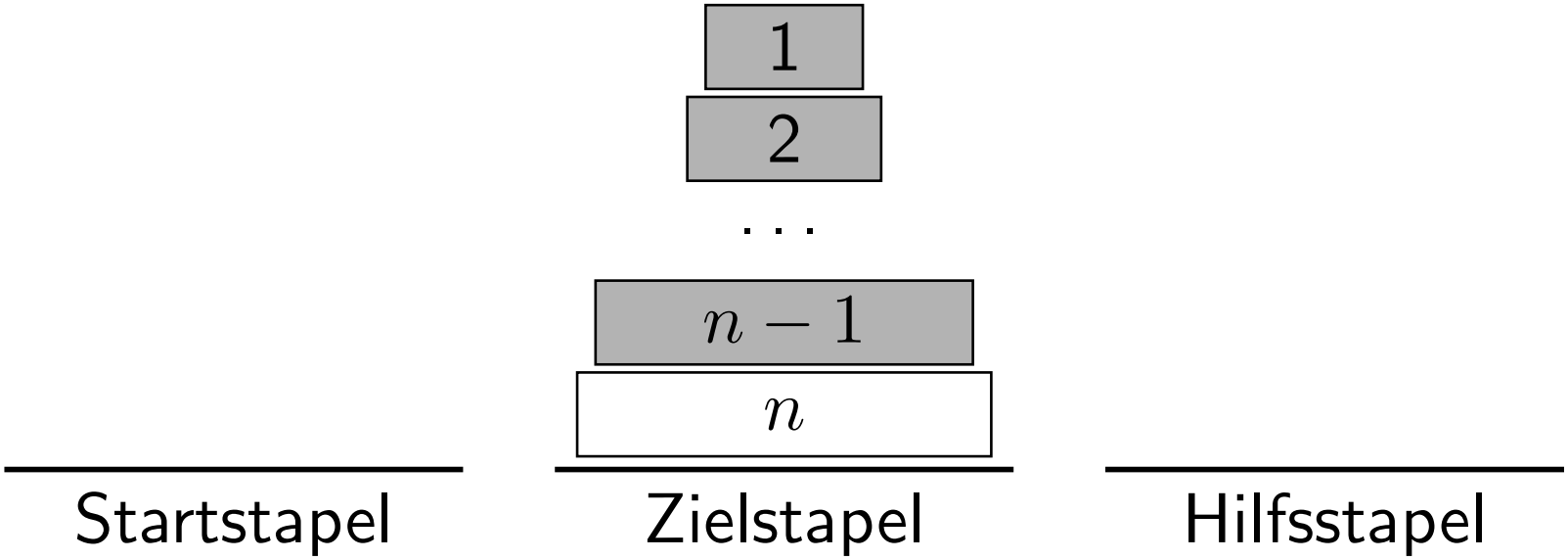
2. Verschiebe Scheibe n auf den Zielstapel

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

Beispiel: n-mal Verdoppeln

- Rekursionsanfang:

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n =
```

Rekursion, weitere Beispiele

Beispiel: n-mal Verdoppeln

- Rekursionsanfang: $n = 0$: Gar nicht verdoppeln

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n =
  if n == 0 then x
```

Rekursion, weitere Beispiele

Beispiel: n-mal Verdoppeln

- Rekursionsanfang: $n = 0$: Gar nicht verdoppeln
- Rekursionsschritt: $n > 0$: Einmal selbst verdoppeln, die restlichen $n - 1$ Verdopplungen der Rekursion überlassen

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n =
  if n == 0 then x
  else n_mal_verdoppeln (verdopple x) (n-1)
```

Pattern matching (auf Zahlen)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n =
  if n == 0 then x
  else n_mal_verdoppeln (verdopple x) (n-1)
```

Man darf statt Variablen auch Pattern in der Funktionsdefinition verwenden, und mehrere Definitionsgleichungen angeben. Die Pattern werden von unten nach oben abgearbeitet.

```
n_mal_verdoppeln2 :: Integer -> Integer -> Integer
n_mal_verdoppeln2 x 0 = x
n_mal_verdoppeln2 x n = n_mal_verdoppeln2 (verdopple x) (n-1)
```

Falsch:

```
n_mal_verdoppeln2 :: Integer -> Integer -> Integer
n_mal_verdoppeln2 x n = n_mal_verdoppeln2 (verdopple x) (n-1)
n_mal_verdoppeln2 x 0 = x
```

Guards

Guards (Wächter): Boolesche Ausdrücke,
die die Definition der Funktion festlegen

```
f x1 ... xm
  | 1. Guard  = e1
  ...
  | n. Guard  = en
```

- Abarbeitung von oben nach unten
- Erster Guard, der zu `True` auswertet, bestimmt die Definition.

```
n_mal_verdoppeln3 :: Integer -> Integer -> Integer
n_mal_verdoppeln3 x n
  | n == 0     = x
  | otherwise = n_mal_verdoppeln3 (verdopple x) (n-1)
```

Vordefiniert: `otherwise = True`

Die Error-Funktion

```
n_mal_verdoppeln3 :: Integer -> Integer -> Integer
n_mal_verdoppeln3 x n
  | n == 0    = x
  | otherwise = n_mal_verdoppeln3 (verdopple x) (n-1)
```

Was passiert bei negativem `n`?

Die Error-Funktion


```
n_mal_verdoppeln3 :: Integer -> Integer -> Integer
n_mal_verdoppeln3 x n
  | n == 0    = x
  | otherwise = n_mal_verdoppeln3 (verdopple x) (n-1)
```

Was passiert bei negativem `n`?

- `error :: String -> a`

```
n_mal_verdoppeln4 :: Integer -> Integer -> Integer
n_mal_verdoppeln4 x n
  | n < 0     = error "Negatives Verdoppeln verboten!"
  | n == 0    = x
  | otherwise = n_mal_verdoppeln4 (verdopple x) (n-1)
```

```
*Main> n_mal_verdoppeln4 10 (-10)
*** Exception:
 in n_mal_verdoppeln4: negatives Verdoppeln ist verboten
```

Beispiel

In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab. Wieviel Rehe gibt es im Wald am 1.1. des Jahres n?

Beispiel

In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab. Wieviel Rehe gibt es im Wald am 1.1. des Jahres n?

Rekursionsanfang: Jahr 1, 10 Rehe

```
anzahlRehe 1 = 10
```

Beispiel

In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab. Wieviel Rehe gibt es im Wald am 1.1. des Jahres n?

Rekursionsanfang: Jahr 1, 10 Rehe
Rekursionsschritt: Sei k = Anzahl Rehe am 1.1. des Jahres $n-1$

```
anzahlRehe 1 = 10
anzahlRehe n =
                        anzahlRehe (n-1)
```

Beispiel

In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab. Wieviel Rehe gibt es im Wald am 1.1. des Jahres n?

Rekursionsanfang: Jahr 1, 10 Rehe

Rekursionsschritt: Sei k = Anzahl Rehe am 1.1. des Jahres $n-1$

$$\text{Jahr } n\text{: } \begin{cases} 3*k-17, & \text{falls } n-1 \text{ kein zweites Jahr} \\ (3*k-17)/2, & \text{falls } n-1 \text{ ein zweites Jahr} \end{cases}$$

```
anzahlRehe 1 = 10
anzahlRehe n = if even (n-1)
               then (3*(anzahlRehe (n-1))-17) `div` 2
               else 3*(anzahlRehe (n-1))-17
```

Beispiel (2)

```
*Main> anzahlRehe 1
10
*Main> anzahlRehe 2
13
*Main> anzahlRehe 3
11
*Main> anzahlRehe 4
16
*Main> anzahlRehe 5
15
*Main> anzahlRehe 6
28
*Main> anzahlRehe 7
33
*Main> anzahlRehe 8
82
*Main> anzahlRehe 9
114
*Main> anzahlRehe 10
325
*Main> anzahlRehe 50
3626347914090925
```

Let-Ausdrücke: Lokale Definitionen

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

```
anzahlRehe 1 = 10
anzahlRehe n = if even (n-1)
               then (3*(anzahlRehe (n-1))-17) `div` 2
               else 3*(anzahlRehe (n-1))-17
```

Mit `let`:

```
anzahlRehe2 1 = 10
anzahlRehe2 n = let k = (3*anzahlRehe2 (n-1))-17
                in  if even (n-1) then k `div` 2
                    else k
```

Let-Ausdrücke: Lokale Definitionen (2)

Allgemeiner:

$$
\begin{array}{llcl}
\texttt{let} & \mathit{Variable1} & = & \mathit{Ausdruck1} \\
 & \mathit{Variable2} & = & \mathit{Ausdruck2} \\
 & \ldots & & \\
 & \mathit{VariableN} & = & \mathit{AusdruckN} \\
\texttt{in} & \text{Ausdruck} & &
\end{array}
$$

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Listen und Programmieren mit Listen

Listen

- Liste = Folge von Elementen
- z.B. `[True, False, False, True, True]` und `[1,2,3,4,5,6]`
- In Haskell 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: Typ

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

```
Prelude> :type [True,False]
[True,False] :: [Bool]
Prelude> :type ['A','B']
['A','B'] :: [Char]
Prelude> :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.

```
Prelude> :type [True,False]
[True,False] :: [Bool]
Prelude> :type ['A','B']
['A','B'] :: [Char]
Prelude> :type [1,2,3]
[1,2,3] :: (Num t) => [t]
```

Auch möglich:

```
*Main> :type [verdopple, verdoppleGerade, jenachdem]
[verdopple, verdoppleGerade, jenachdem] :: [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:

- „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`

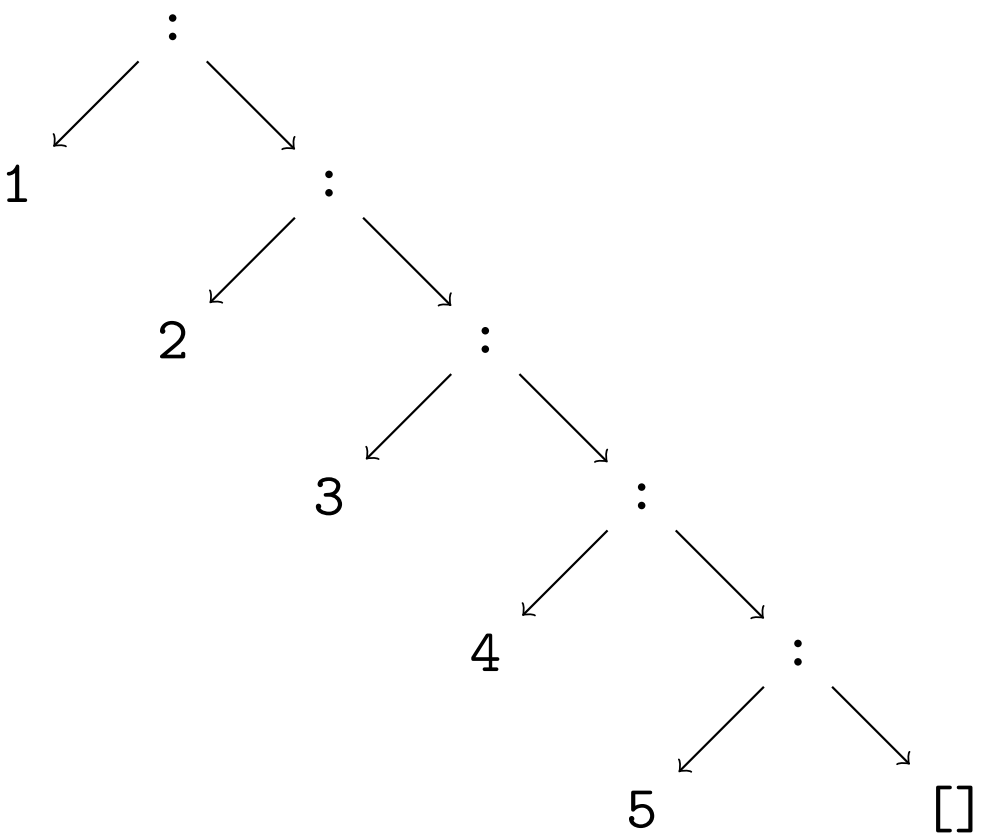
`[1,2,3]` ist in Wahrheit `1:(2:(3:[]))`

Typen:

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

```
letztesElement :: [a] -> a
letztesElement xs = if null xs then
                       error "Liste ist leer"
                    else
                      if null (tail xs) then head xs
                      else letztesElement (tail xs)
```

```
Main> letztesElement [True,False,True]
True
*Main> letztesElement [1,2,3]
3
*Main> letztesElement (nbis1 1000)
1
*Main> letztesElement [[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.

```
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`

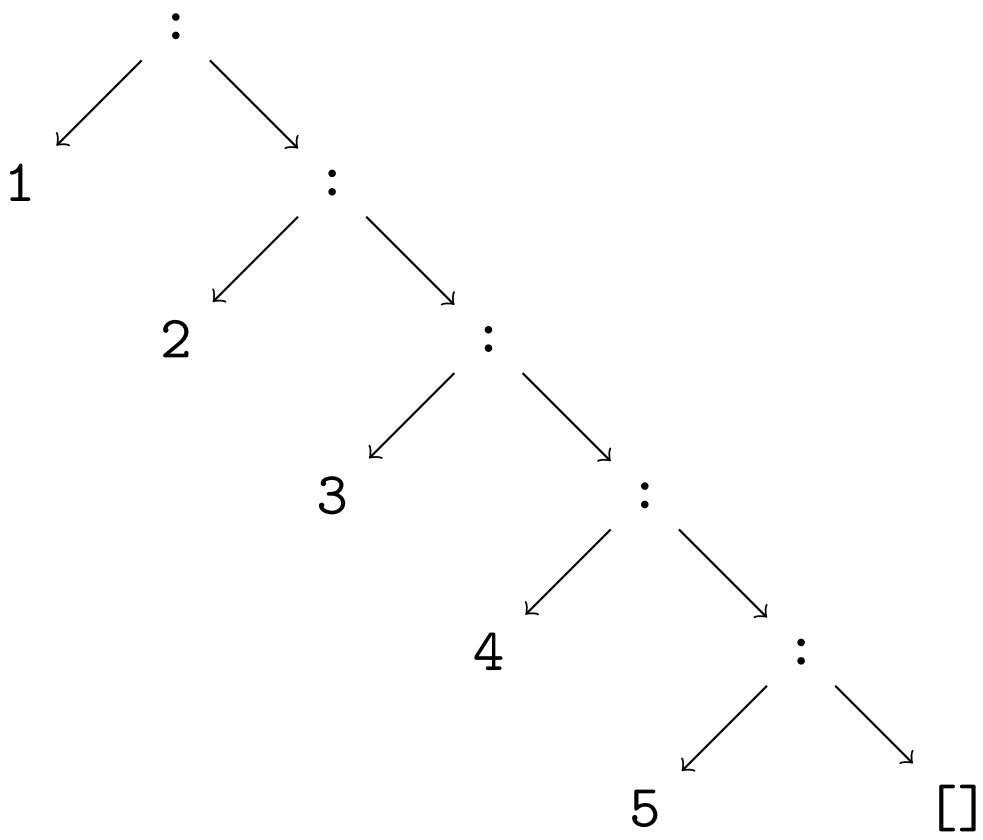
  `xs` = `(2:[])`

Pattern-Match

Pattern []

[]

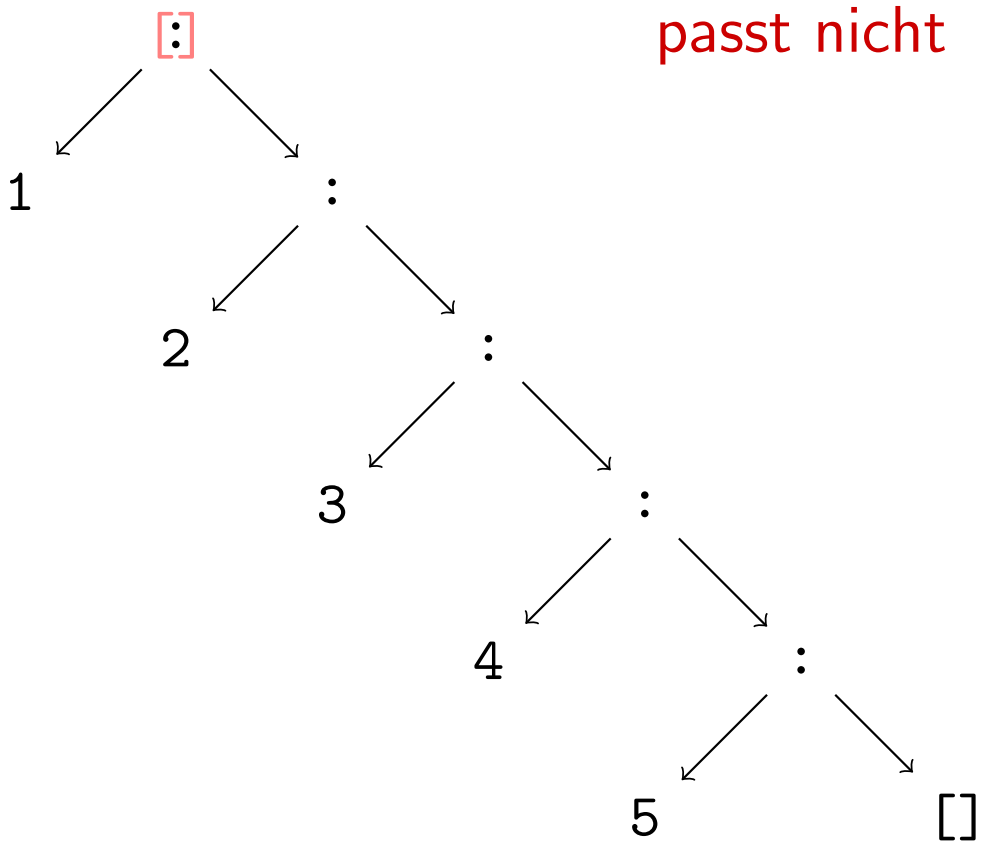
Liste



Pattern-Match

Pattern []

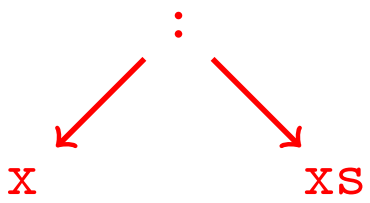
Liste



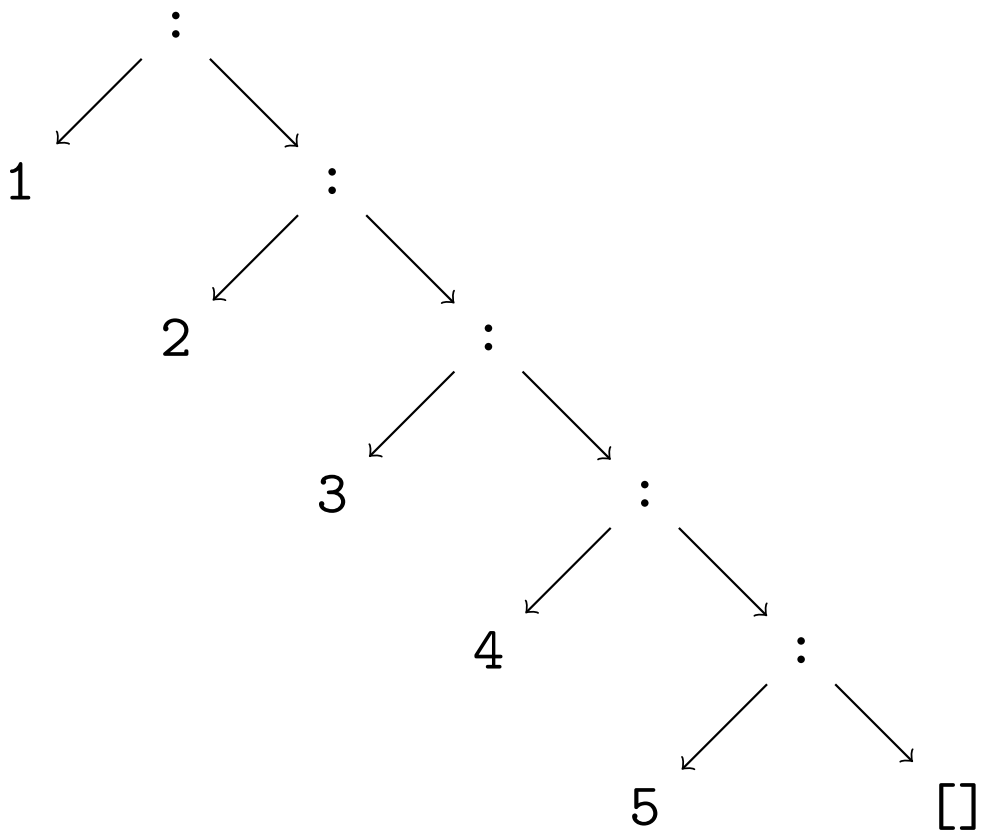
passt nicht

Pattern-Match

Pattern `x:xs`



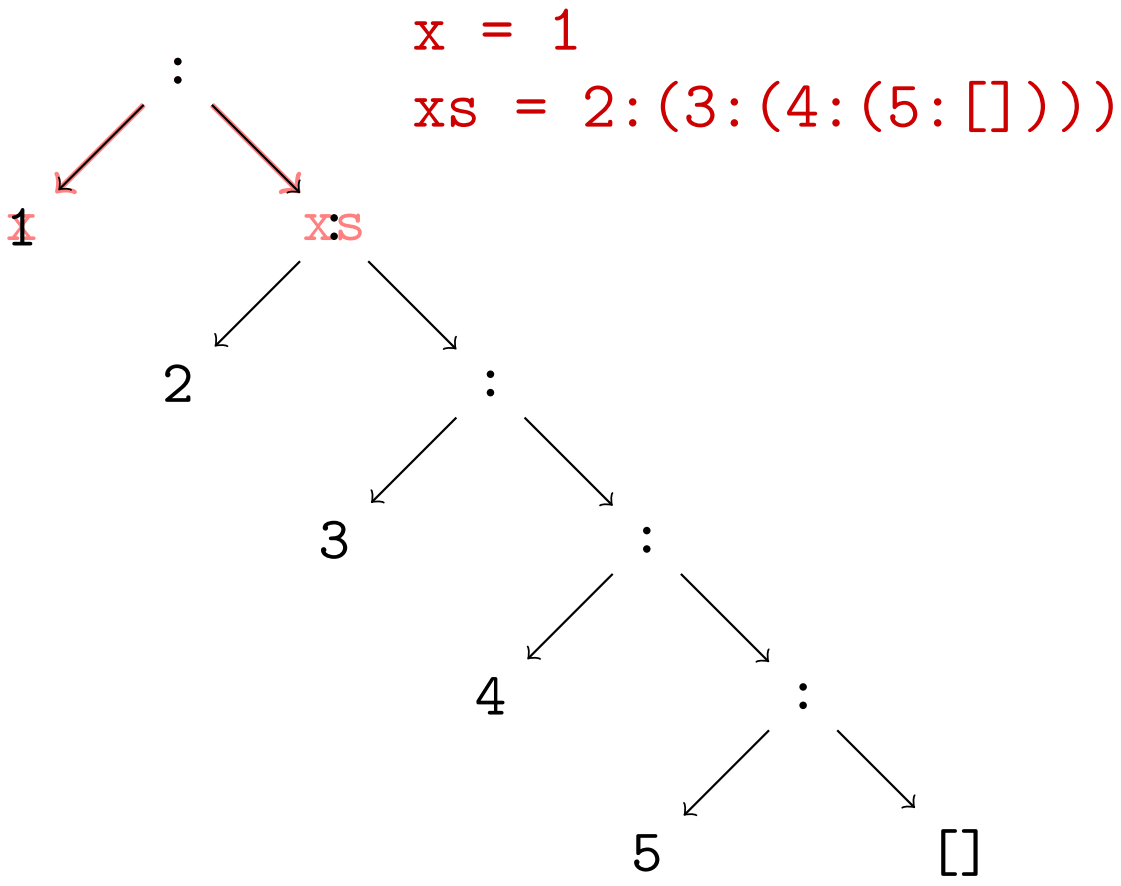
Liste



Pattern-Match

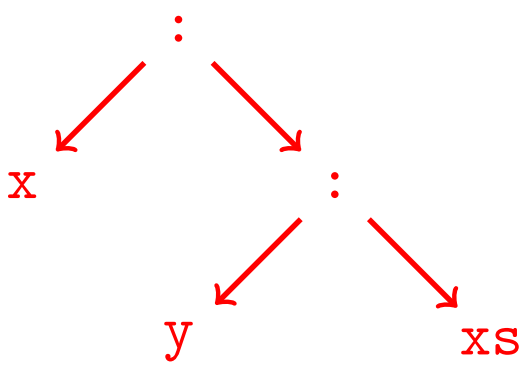
Pattern `x:xs`

Liste

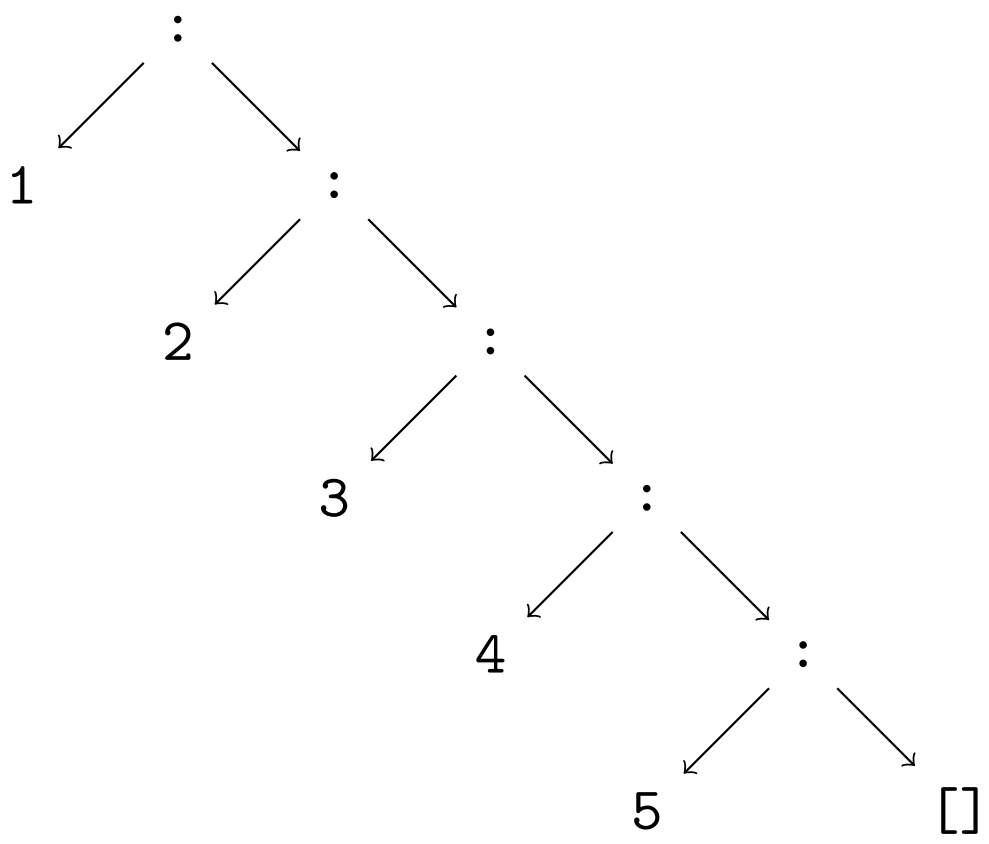


Pattern-Match

Pattern `x:(y:xs)`



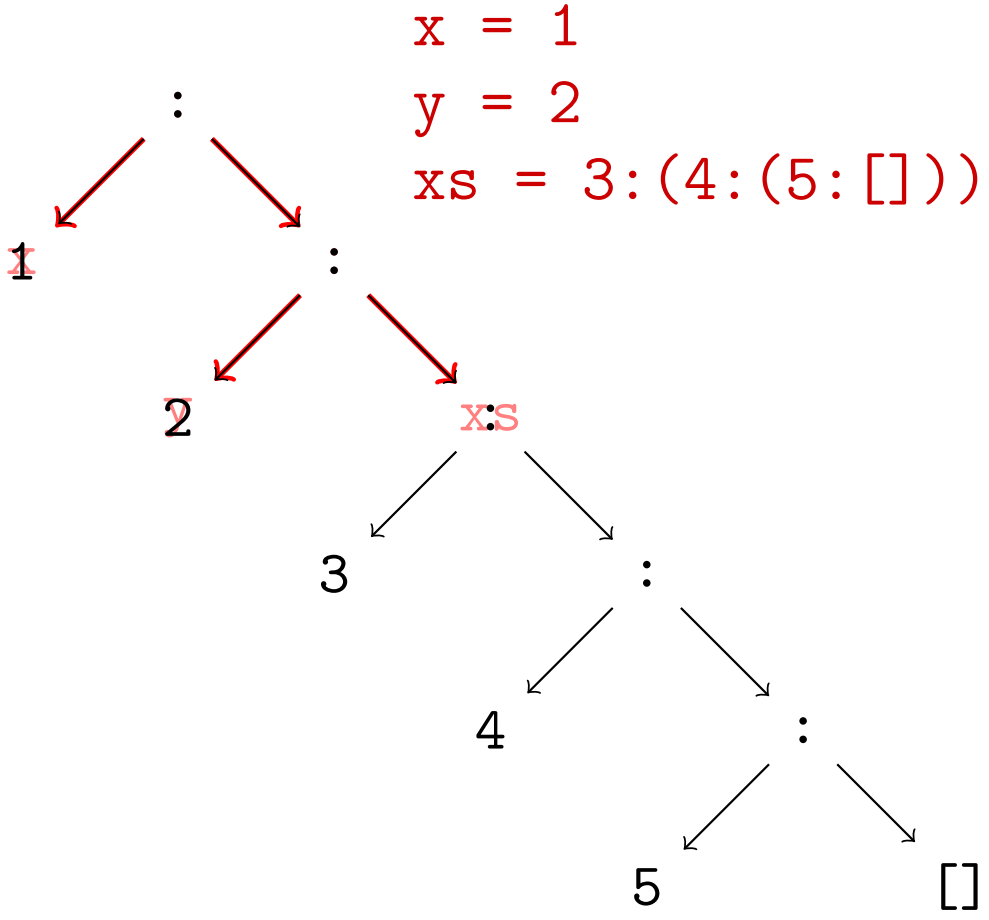
Liste



Pattern-Match

Pattern `x:(y:xs)`

Liste



Beispiele: Pattern-Match


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

-- alternativ:
eigenesNull2 []     = True
eigenesNull2 xs     = False -- matcht immer!

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

```
 Warning: Pattern match(es) are overlapped
          In the definition of ‘falschesNull’: falschesNull [] = ...
Ok, modules loaded: Main.
*Main> falschesNull [1]
False
*Main> falschesNull []
False
*Main> eigenesNull []
True
```

Letztes Element mit Pattern:

```
letztesElement2 []     = error "leere Liste"
letztesElement2 (x:[]) = x
letztesElement2 (x:xs) = letztesElement2 xs
```

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

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.

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` nimmt ein Paar und liefert das linke Element.
- `snd :: (a,b) -> b` liefert das rechte Element eines Paares.

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

Pattern-Match auf Paaren

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

Tupel

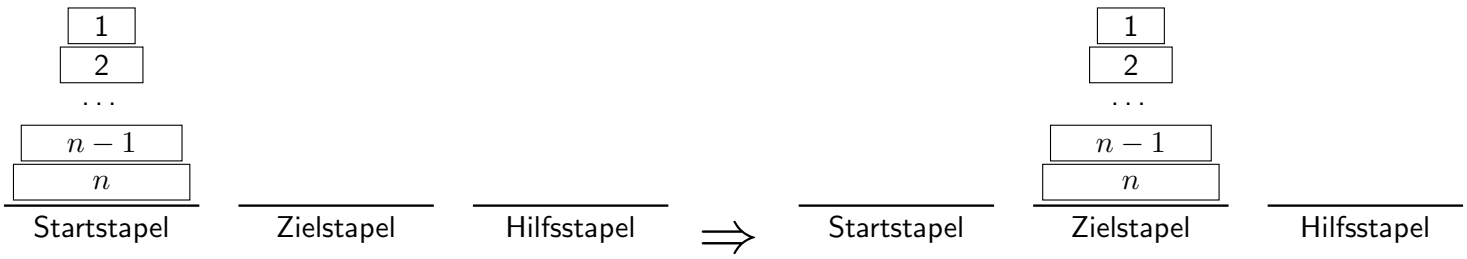
Wie Paare, aber mit mehr Komponenten.

```
*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 start $= 1$, ziel $= 2$, 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)]
```