# Programmierung und Modellierung mit Haskell

## Teil 3: Rekursion, Terminierung, Induktion

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

18. April 2018

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

# Teil 3: Rekursion, Terminierung, Induktion

# Rekursion

Man kann eine Funktion $f : A \to B$ durch einen Ausdruck definieren, der selbst Funktionsanwendungen von $f$ enthält.

*Beispiele*

- Fakultät:

$$\text{fakultät}(n) = \begin{cases} 1, \text{ falls } n = 0 \\ n \cdot \text{fakultät}(n-1), \text{ sonst} \end{cases}$$

- Summe der ersten $n$ natürlichen Zahlen: Dreieckszahlen

$$\text{trinum}(n) = \begin{cases} 0, \text{ falls n=0} \\ n + \text{trinum}(n-1), \text{ sonst} \end{cases}$$

Dies bezeichnet man als **rekursive Definition**. Auswerten durch Einsetzen der definierenden Gleichungen von links nach rechts:

$$\begin{aligned} \text{fakultät}(3) &\rightsquigarrow 3 \cdot \text{fakultät}(3-1) \rightsquigarrow 3 \cdot \text{fakultät}(2) \\ &\rightsquigarrow 3 \cdot 2 \cdot \text{fakultät}(2-1) \rightsquigarrow 3 \cdot 2 \cdot \text{fakultät}(1) \\ &\rightsquigarrow 3 \cdot 2 \cdot 1 \cdot \text{fakultät}(0) \rightsquigarrow 3 \cdot 2 \cdot 1 \cdot 1 \rightsquigarrow 6 \end{aligned}$$

# Rekursion

Man kann eine Funktion $f : A \to B$ durch einen Ausdruck definieren, der selbst Funktionsanwendungen von $f$ enthält.

*Beispiele*

- Fakultät:

```
fakultät 0 = 1
fakultät n = n * fakultät (n-1)
```

- Summe der ersten $n$ natürlichen Zahlen: Dreieckszahlen

```
trinum 0 = 0
trinum n = n + trinum (n-1)
```

Dies bezeichnet man als **rekursive Definition**. Auswerten durch Einsetzen der definierenden Gleichungen von links nach rechts:

$$\begin{aligned}\text{fakultät}(3) &\rightsquigarrow 3 \cdot \text{fakultät}(3-1) \rightsquigarrow 3 \cdot \text{fakultät}(2)\\ &\rightsquigarrow 3 \cdot 2 \cdot \text{fakultät}(2-1) \rightsquigarrow 3 \cdot 2 \cdot \text{fakultät}(1)\\ &\rightsquigarrow 3 \cdot 2 \cdot 1 \cdot \text{fakultät}(0) \rightsquigarrow 3 \cdot 2 \cdot 1 \cdot 1 \rightsquigarrow 6\end{aligned}$$

# Substitution

Wir schreiben $e[t/x]$ um auszudrücken, dass wir ...

- im Ausdruck $e$ ...
- alle *freien* Vorkommen von Variable $x$ ...
- durch Teilausdruck $t$ ersetzen.

## Beispiel I

`(\u -> u*2) 3` $\rightsquigarrow$ `(u*2)`$[3/u]$ $=$ `(2*3)` $\rightsquigarrow$ `6`

## Allgemeine Auswerteregeln für Funktionsanwendung

Für beliebige Variablen $v_i$, Ausdrücke $e_i$ und Funktionssymbole $f_i$:

- `(\`$v_1$`->`$e_1$`)` $e_2 \rightsquigarrow e_1[e_2/v_1]$
- $f_1\ e_2 \rightsquigarrow e_1[e_2/v_1]$ falls $f_1$ definiert durch: $f_1\ v_1 = e_1$

# Substitution

Wir schreiben $e[t/x]$ um auszudrücken, dass wir ...

- im Ausdruck $e$ ...
- alle *freien* Vorkommen von Variable $x$ ...
- durch Teilausdruck $t$ ersetzen.

## Beispiel II

`(\u->(\w->u+w) 3) 4` $\rightsquigarrow$ `((\w->u+w)[3/u]) 4`

$\rightsquigarrow$ `(\w->3+w) 4` $\rightsquigarrow$ `3+w[4/w]` $\rightsquigarrow$ `3+4` $\rightsquigarrow$ `7`

## Allgemeine Auswerteregeln für Funktionsanwendung

Für beliebige Variablen $v_i$, Ausdrücke $e_i$ und Funktionssymbole $f_i$:

- `(\`$v_1$`->`$e_1$`)` $e_2 \rightsquigarrow e_1[e_2/v_1]$
- $f_1\ e_2 \rightsquigarrow e_1[e_2/v_1]$ falls $f_1$ definiert durch: $f_1\ v_1 = e_1$

# Substitution

Wir schreiben $e[t/x]$ um auszudrücken, dass wir ...

- im Ausdruck $e$ ...
- alle *freien* Vorkommen von Variable $x$ ...
- durch Teilausdruck $t$ ersetzen.

## Beispiel III

`(\u->(\w->u+u) 3) 4` $\rightsquigarrow$ `((\w->u+u)[3/u]) 4`

$\rightsquigarrow$ `(\w->3+3) 4` $\rightsquigarrow$ `3+3[4/w]` $\rightsquigarrow$ `3+3` $\rightsquigarrow$ `6`

Die zu ersetzende Variable muss nicht immer vorkommen.

## Allgemeine Auswerteregeln für Funktionsanwendung

Für beliebige Variablen $v_i$, Ausdrücke $e_i$ und Funktionssymbole $f_i$:

- `(\`$v_1$`->`$e_1$`)` $e_2 \rightsquigarrow e_1[e_2/v_1]$
- $f_1\ e_2 \rightsquigarrow e_1[e_2/v_1]$ falls $f_1$ definiert durch: $f_1\ v_1 = e_1$

# Substitution

Wir schreiben $e[t/x]$ um auszudrücken, dass wir ...

- im Ausdruck $e$ ...
- alle *freien* Vorkommen von Variable $x$ ...
- durch Teilausdruck $t$ ersetzen.

## Beispiel IV

`(\w->(\w->w+w) 3) 4` $\rightsquigarrow$ `((\w->w+w)[3/w]) 4`

$\rightsquigarrow$ `(\w->w+w) 4` $\rightsquigarrow$ `w+w[4/w]` $\rightsquigarrow$ `4+4` $\rightsquigarrow$ `8`

Gebundene Vorkommen werden nicht ersetzt!

## Allgemeine Auswerteregeln für Funktionsanwendung

Für beliebige Variablen $v_i$, Ausdrücke $e_i$ und Funktionssymbole $f_i$:

- `(\`$v_1$`->`$e_1$`)` $e_2$ $\rightsquigarrow e_1[e_2/v_1]$
- $f_1\ e_2$ $\rightsquigarrow e_1[e_2/v_1]$ falls $f_1$ definiert durch: $f_1\ v_1 = e_1$

# Substitutionsmodell

Das **Substitutionsmodell** erklärt das Verhalten von rein funktionalen Sprachen, solange keine Fehler auftreten:

- Man wählt einen Teilausdruck und wertet diesen gemäß den geltenden Rechenregeln aus.
- Eine Funktionsanwendung wird durch den definierenden Rumpf ersetzt. Dabei werden die formalen Parameter im Rumpf durch die Argumentausdrücke ersetzt ("substitutiert").
- Dies wiederholt man, bis keine auswertbaren Teilausdrücke mehr vorhanden sind.

Es gibt verschiedene unterschiedliche Strategien, den als nächstes auszuwertenden Teilausdruck auszuwählen.

Dies ist ein rekursiver Algorithmus!

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`, wobei wir zur Verdeutlichung den zu bearbeitenden Teilausdruck unterstreichen:

```
bar 1 2 (succ 3)
```

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`, wobei wir zur Verdeutlichung den zu bearbeitenden Teilausdruck unterstreichen:

`bar 1 2 (succ 3)` $\rightsquigarrow$ `bar 1 2 (3 + 1)`

Wir ersetzen „`succ 3`“ durch „`x + 1`$[3/x]$“ gleich „`3 + 1`“
D.h. wir ersetzen die Anwendung der Funktion `succ` mit Argument `3` durch den definierenden Funktionsrumpf von `succ`, wobei wir alle freien vorkommen der Variable `x` durch das Argument `3` ersetzen.

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`, wobei wir zur Verdeutlichung den zu bearbeitenden Teilausdruck unterstreichen:

$$\texttt{bar 1 2 }\underline{\texttt{(succ 3)}} \rightsquigarrow \texttt{bar 1 2 }\underline{\texttt{(3 + 1)}} \rightsquigarrow \underline{\texttt{bar}}\texttt{ 1 2 4}$$

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`, wobei wir zur Verdeutlichung den zu bearbeitenden Teilausdruck unterstreichen:

`bar 1 2 (succ 3)` $\rightsquigarrow$ `bar 1 2 (3 + 1)` $\rightsquigarrow$ `bar 1 2 4`
$\rightsquigarrow$ `if 0 < succ 1 then 4 else 1+2`

Im definierenden Rumpf von `bar` substituieren wir: $[1/x, 2/y, 4/z]$

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`, wobei wir zur Verdeutlichung den zu bearbeitenden Teilausdruck unterstreichen:

$$\texttt{bar 1 2 }\underline{\texttt{(succ 3)}}\texttt{ } \rightsquigarrow \texttt{ bar 1 2 }\underline{\texttt{(3 + 1)}}\texttt{ } \rightsquigarrow \texttt{ }\underline{\texttt{bar}}\texttt{ 1 2 4}$$
$$\rightsquigarrow \texttt{ if 0 < }\underline{\texttt{succ 1}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{ if 0 < }\underline{\texttt{(1 + 1)}}\texttt{ then 4 else 1+2}$$

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`, wobei wir zur Verdeutlichung den zu bearbeitenden Teilausdruck unterstreichen:

$$\texttt{bar 1 2 }\underline{\texttt{(succ 3)}}\rightsquigarrow \texttt{bar 1 2 }\underline{\texttt{(3 + 1)}}\rightsquigarrow \underline{\texttt{bar}}\texttt{ 1 2 4}$$
$$\rightsquigarrow \texttt{if 0 < }\underline{\texttt{succ 1}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{if 0 < }\underline{\texttt{(1 + 1)}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{if }\underline{\texttt{0 < 2}}\texttt{ then 4 else 1+2}$$

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`, wobei wir zur Verdeutlichung den zu bearbeitenden Teilausdruck unterstreichen:

$$\texttt{bar 1 2 }\underline{\texttt{(succ 3)}}\rightsquigarrow \texttt{bar 1 2 }\underline{\texttt{(3 + 1)}}\rightsquigarrow \underline{\texttt{bar}}\texttt{ 1 2 4}$$
$$\rightsquigarrow \texttt{if 0 < }\underline{\texttt{succ 1}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{if 0 < }\underline{\texttt{(1 + 1)}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{if }\underline{\texttt{0 < 2}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{if True then 4 else }\underline{\texttt{1+2}}$$

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`, wobei wir zur Verdeutlichung den zu bearbeitenden Teilausdruck unterstreichen:

$$\texttt{bar 1 2 }\underline{\texttt{(succ 3)}}\rightsquigarrow \texttt{bar 1 2 }\underline{\texttt{(3 + 1)}}\rightsquigarrow \underline{\texttt{bar}}\texttt{ 1 2 4}$$
$$\rightsquigarrow \texttt{if 0 < }\underline{\texttt{succ 1}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{if 0 < }\underline{\texttt{(1 + 1)}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{if }\underline{\texttt{0 < 2}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{if True then 4 else }\underline{\texttt{1+2}}$$
$$\rightsquigarrow \underline{\texttt{if}}\texttt{ True }\underline{\texttt{then}}\texttt{ 4 }\underline{\texttt{else}}\texttt{ 3}$$

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`, wobei wir zur Verdeutlichung den zu bearbeitenden Teilausdruck unterstreichen:

$$\texttt{bar 1 2 }\underline{\texttt{(succ 3)}}\rightsquigarrow \texttt{bar 1 2 }\underline{\texttt{(3 + 1)}}\rightsquigarrow \underline{\texttt{bar}}\texttt{ 1 2 4}$$
$$\rightsquigarrow \texttt{if 0 < }\underline{\texttt{succ 1}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{if 0 < }\underline{\texttt{(1 + 1)}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{if }\underline{\texttt{0 < 2}}\texttt{ then 4 else 1+2}$$
$$\rightsquigarrow \texttt{if True then 4 else }\underline{\texttt{1+2}}$$
$$\rightsquigarrow \underline{\texttt{if}}\texttt{ True }\underline{\texttt{then}}\texttt{ 4 }\underline{\texttt{else}}\texttt{ 3}$$
$$\rightsquigarrow \texttt{4}$$

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Bei der Auswertung des Ausdrucks `bar 1 2 (succ 3)` hätten wir aber auch andere Teilausdrücke unterstreichen und auswerten können:

`bar 1 2 (succ 3)`

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Bei der Auswertung des Ausdrucks `bar 1 2 (succ 3)` hätten wir aber auch andere Teilausdrücke unterstreichen und auswerten können:

```
bar 1 2 (succ 3)
⇝ if 0 < succ 1 then (succ 3) else 1+2
```

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Bei der Auswertung des Ausdrucks `bar 1 2 (succ 3)` hätten wir aber auch andere Teilausdrücke unterstreichen und auswerten können:

```
bar 1 2 (succ 3)
⇝ if 0 < succ 1 then (succ 3) else 1+2
⇝ if 0 < 1+1 then (succ 3) else 1+2
```

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Bei der Auswertung des Ausdrucks `bar 1 2 (succ 3)` hätten wir aber auch andere Teilausdrücke unterstreichen und auswerten können:

```
bar 1 2 (succ 3)
↝ if 0 < succ 1 then (succ 3) else 1+2
↝ if 0 < 1+1 then (succ 3) else 1+2
↝ if 0 < 2 then (succ 3) else 1+2
```

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Bei der Auswertung des Ausdrucks `bar 1 2 (succ 3)` hätten wir aber auch andere Teilausdrücke unterstreichen und auswerten können:

```
bar 1 2 (succ 3)
⇝ if 0 < succ 1 then (succ 3) else 1+2
⇝ if 0 < 1+1 then (succ 3) else 1+2
⇝ if 0 < 2 then (succ 3) else 1+2
⇝ if True then (succ 3) else 1+2
```

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Bei der Auswertung des Ausdrucks `bar 1 2 (succ 3)` hätten wir aber auch andere Teilausdrücke unterstreichen und auswerten können:

```
bar 1 2 (succ 3)
↝ if 0 < succ 1 then (succ 3) else 1+2
↝ if 0 < 1+1 then (succ 3) else 1+2
↝ if 0 < 2 then (succ 3) else 1+2
↝ if True then (succ 3) else 1+2
↝ succ 3
```

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Bei der Auswertung des Ausdrucks `bar 1 2 (succ 3)` hätten wir aber auch andere Teilausdrücke unterstreichen und auswerten können:

```
bar 1 2 (succ 3)
⇝ if 0 < succ 1 then (succ 3) else 1+2
⇝ if 0 < 1+1 then (succ 3) else 1+2
⇝ if 0 < 2 then (succ 3) else 1+2
⇝ if True then (succ 3) else 1+2
⇝ succ 3
⇝ 3 + 1
```

# Beispiel

```
succ x = x + 1
bar x y z = if 0 < succ x then z else x+y
```

Bei der Auswertung des Ausdrucks `bar 1 2 (succ 3)` hätten wir aber auch andere Teilausdrücke unterstreichen und auswerten können:

```
bar 1 2 (succ 3)
⇝ if 0 < succ 1 then (succ 3) else 1+2
⇝ if 0 < 1+1 then (succ 3) else 1+2
⇝ if 0 < 2 then (succ 3) else 1+2
⇝ if True then (succ 3) else 1+2
⇝ succ 3
⇝ 3 + 1 ⇝ 4
```

Ergebnis bleibt gleich; Anzahl der Schritte unterscheidet sich.

⇒ Kapitel 11

# Fibonacci-Zahlen

```
fib 0 = 0
fib n | n <= 1    = 1
      | otherwise = fib (n-1) + fib (n-2)

> fib 8
21
> fib 9
34
```

- `fib` $n$ liefert die $n$-te Fibonacci-Zahl $F_n$:
  $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$
- `fib` $n$ liefert z.B. die Hasenpopulation nach $n$ Monaten unter der Annahme, dass Hasen jeden Monat einen Nachkommen haben, dies aber erst ab dem zweiten Lebensmonat.
- `fib` $n$ liefert auch die Zahl der Möglichkeiten, ein $2 \times n$ Zimmer mit $1 \times 2$ Kacheln zu fliesen.

# Collatz Vermutung

Für alle $n \in \mathbb{N}$ gibt es ein $i \in \mathbb{N}$ mit $f^i(n) = 1$.

Lothar Collatz, 1910-90, dt. Mathematiker

$$f(n) = \begin{cases} \frac{n}{2} & \text{falls } n \text{ gerade} \\ 3n+1 & \text{sonst} \end{cases}$$

Es ist unbekannt, ob man für jedes $n \in \mathbb{N}$ ein $i \in \mathbb{N}$ findet.

# Collatz Vermutung

Für alle $n \in \mathbb{N}$ gibt es ein $i \in \mathbb{N}$ mit $f^i(n) = 1$.

Lothar Collatz, 1910-90, dt. Mathematiker

$$f(n) = \begin{cases} \frac{n}{2} & \text{falls } n \text{ gerade} \\ 3n+1 & \text{sonst} \end{cases}$$

Es ist unbekannt, ob man für jedes $n \in \mathbb{N}$ ein $i \in \mathbb{N}$ findet.
Falls es ein $i$ gibt, dann auch ein kleinstes. Wir versuchen dieses rekursiv zu berechnen:

```
f n | even n          = n `div` 2
    | otherwise       = 3*n+1

min_i 1 = 0
min_i n = 1 + min_i (f n)

> [min_i n | n <- [2..10]++[26,27,28,29]]
[1,7,2,5,8,16,3,19,6,10,111,18,18]
```

# Collatz Vermutung

Für alle $n \in \mathbb{N}$ gibt es ein $i \in \mathbb{N}$ mit $f^i(n) = 1$.

Lothar Collatz, 1910-90, dt. Mathematiker

$$f(n) = \begin{cases} \frac{n}{2} & \text{falls } n \text{ gerade} \\ 3n+1 & \text{sonst} \end{cases}$$

Es ist unbekannt, ob man für jedes $n \in \mathbb{N}$ ein $i \in \mathbb{N}$ findet.
Falls es ein $i$ gibt, dann auch ein kleinstes. Wir versuchen dieses rekursiv zu berechnen:

```
f n | n `mod` 2 == 0 = n `div` 2
    | otherwise      = 3*n+1

min_i 1 = 0
min_i n = 1 + min_i (f n)

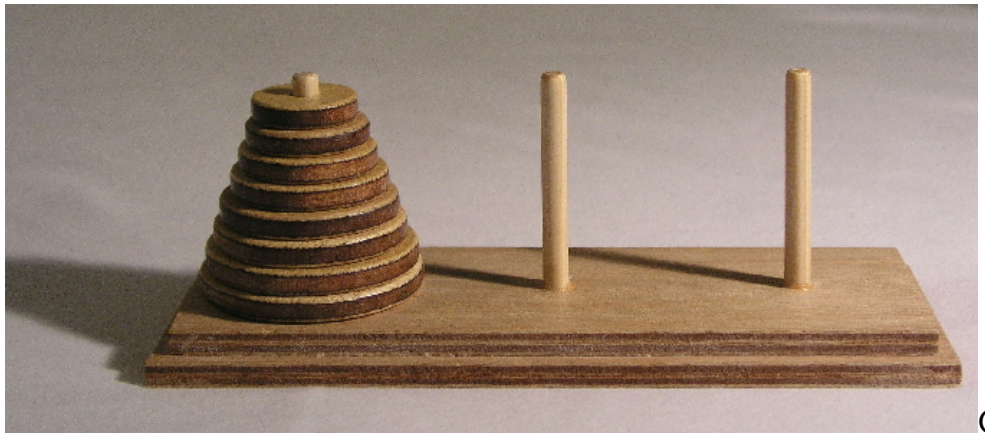
> [min_i n | n <- [2..10]++[26,27,28,29]]
[1,7,2,5,8,16,3,19,6,10,111,18,18]
```

# Türme von Hanoi

Es gibt drei senkrechte Stäbe. Auf dem ersten liegen $n$ gelochte Scheiben von nach oben hin abnehmender Größe.

Man soll den ganzen Stapel auf den dritten Stab transferieren, darf aber immer nur jeweils eine Scheibe entweder nach ganz unten oder auf eine größere legen.

Angeblich sind in Hanoi ein paar Mönche seit Urzeiten mit dem Fall $n = 64$ befasst.



Quelle: Wikipedia

# Lösung

Für $n = 1$ kein Problem.

Falls man schon weiß, wie es für $n - 1$ geht, dann schafft man mit diesem Rezept die obersten $n - 1$ Scheiben auf den zweiten Stab (die unterste Scheibe fasst man dabei als "Boden" auf.).

Dann legt man die größte nunmehr freie Scheibe auf den dritten Stapel und verschafft unter abermaliger Verwendung der Vorschrift für $n - 1$ die restlichen Scheiben vom mittleren auf den dritten Stapel.

# Lösung in Haskell

- "Türme" werden durch Zahl 1,2,3 repräsentiert
- "Befehle" durch Paare $(i, j)$: "Bewege Scheibe von $i$ nach $j$"
- "Befehlsfolgen" werden durch Listen repräsentiert.

```
hanoi 1 i j = [(i,j)]
hanoi n i j = hanoi n' i otherT ++ [(i,j)] ++ hanoi n' otherT j
  where n'      = n-1
        otherT = 1+2+3-i-j                          -- other tower
```

```
> hanoi 3 1 2
[(1,2),(1,3),(2,3),(1,2),(3,1),(3,2),(1,2)]
```

```
> hanoi 4 1 2
[(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)]
```

# Allgemeines Muster einer Rekursion

Eine rekursive Definition einer Funktion $f : A \to B$ hat die Form

$$f(a) = E[f, a]$$

wobei im Ausdruck $E[f, a]$, also dem Funktionsrumpf, sowohl das Argument $a$, als auch (endlich viele) Aufrufe an die definierte Funktion $f$ selbst vorkommen dürfen.

## Gegenbeispiele

Keine rekursiven Definitionen sind also:

- $f(n) = \begin{cases} 0, \text{ falls Programm für } f \text{ kürzer als } n \text{ KB} \\ 1, \text{ sonst} \end{cases}$

  Zugriff nicht in Form von Aufrufen

- $f(n) = \begin{cases} 1, \text{ falls } f(i) = 0 \text{ für alle } i \in \mathbb{N} \\ 0, \text{ sonst} \end{cases}$

  unendliche viele Aufrufe

# Allgemeines Muster einer Rekursion

Eine rekursive Definition einer Funktion $f : A \to B$ hat die Form

$$f(a) = E[f, a]$$

wobei im Ausdruck $E[f, a]$, also dem Funktionsrumpf, sowohl das Argument $a$, als auch (endlich viele) Aufrufe an die definierte Funktion $f$ selbst vorkommen dürfen.

## Gegenbeispiele

Keine rekursiven Definitionen sind also:

- $f(n) = \begin{cases} 0, \text{ falls Programm für } f \text{ kürzer als } n \text{ KB} \\ 1, \text{ sonst} \end{cases}$

  Zugriff nicht in Form von Aufrufen

- $f(n) = \begin{cases} 1, \text{ falls } f(i) = 0 \text{ für alle } i \in \mathbb{N} \\ 0, \text{ sonst} \end{cases}$

  unendliche viele Aufrufe

# Partialität

Rekursive Definitionen liefern im allgemeinen nur partielle Funktionen:

- Die Funktionen `trinum` und `fakultät` von Folie 3.3 liefern bei negativen Eingaben kein Ergebnis:

```
> trinum (-3)
Heap exhausted;
```

Grund: $\texttt{trinum}(-2) \leadsto -2 + \texttt{trinum}(-3) \leadsto$
$-2 - 3 + \texttt{trinum}(-4) \leadsto -2 - 3 - 4 + \texttt{trinum}(-5) \leadsto \ldots$.

- Gleiches gilt für die Funktion

```
waldläufer x = waldläufer x
```

- Bei der "$3n+1$"-Funktion (`min_i`, Folie 3.9): weiß man nicht, ob sie wirklich für alle $n$ definiert ist.

# Partialität

Rekursive Definitionen liefern im allgemeinen nur partielle Funktionen:

- Die Funktionen `trinum` und `fakultät` von Folie 3.3 liefern bei negativen Eingaben kein Ergebnis:

```
> trinum (-3)
Heap exhausted;
```

*Tipp:* Manche Computer werden unbenutzbar, wenn ein Programm den kompletten Speicher belegt.
GHC und GHCI erlaubt es, den maximal verwendeten Speicher zu beschränken, z.B. hier auf 2 GB: `> ghci file.hs +RTS -M2g`

```
waldläufer x = waldläufer x
```

- Bei der "$3n+1$"-Funktion (`min_i`, Folie 3.9); weiß man nicht, ob sie wirklich für alle $n$ definiert ist.

# Partialität

Rekursive Definitionen liefern im allgemeinen nur partielle Funktionen:

- Die Funktionen `trinum` und `fakultät` von Folie 3.3 liefern bei negativen Eingaben kein Ergebnis:

```
> trinum (-3)
Heap exhausted;
```

Grund: $\texttt{trinum}(-2) \leadsto -2 + \texttt{trinum}(-3) \leadsto$
$-2 - 3 + \texttt{trinum}(-4) \leadsto -2 - 3 - 4 + \texttt{trinum}(-5) \leadsto \ldots.$

- Gleiches gilt für die Funktion

```
waldläufer x = waldläufer x
```

- Bei der "$3n + 1$"-Funktion (`min_i`, Folie 3.9); weiß man nicht, ob sie wirklich für alle $n$ definiert ist.

# Partialität

Rekursive Definitionen liefern im allgemeinen nur partielle Funktionen:

- Die Funktionen `trinum` und `fakultät` von Folie 3.3 liefern bei negativen Eingaben kein Ergebnis:

```
> trinum (-3)
Heap exhausted;
```

Grund: $\texttt{trinum}(-2) \leadsto -2 + \texttt{trinum}(-3) \leadsto$
$-2 - 3 + \texttt{trinum}(-4) \leadsto -2 - 3 - 4 + \texttt{trinum}(-5) \leadsto \ldots.$

- Gleiches gilt für die Funktion

```
waldläufer x = waldläufer x
```

- Bei der "$3n + 1$"-Funktion (`min_i`, Folie 3.9); weiß man nicht, ob sie wirklich für alle $n$ definiert ist.

# Partialität

Rekursive Definitionen liefern im allgemeinen nur partielle Funktionen:

- Die Funktionen `trinum` und `fakultät` von Folie 3.3 liefern bei negativen Eingaben kein Ergebnis:

```
> trinum (-3)
Heap exhausted;
```

  Grund: $\texttt{trinum}(-2) \leadsto -2 + \texttt{trinum}(-3) \leadsto$
  $-2 - 3 + \texttt{trinum}(-4) \leadsto -2 - 3 - 4 + \texttt{trinum}(-5) \leadsto \ldots$.

- Gleiches gilt für die Funktion

```
waldläufer x = waldläufer x
```

- Bei der "$3n+1$"-Funktion (`min_i`, Folie 3.9); weiß man nicht, ob sie wirklich für alle $n$ definiert ist.

# Abstiegsfunktion



Gegeben sei eine rekursive Funktion

$$f : A \to B$$
$$f(x) = E[f, x]$$

Um festzustellen ob $f$ für alle Argumente einer gewählten Teilmenge $A' \subseteq A$ definiert ist, kann man eine **Abstiegsfunktion** verwenden. Zuerst prüfen wir, ob $A'$ sinnvoll gewählt wurde.

AUF) Unter der Voraussetzung $x \in A'$ wird im Ausdruck $E[f, x]$ die Funktion $f$ nur mit Argumenten $y \in A'$ aufgerufen.

DEF) Für $x \in A'$ ist der Ausdruck $E[f, x]$ definiert unter der Annahme, dass die getätigten Aufrufe von $f$ alle definiert sind.

Damit weiß man erst einmal, dass alle Funktionsanwendung $f\ x$ für $x \in A'$ vernünftig bleiben (keine unvollständigen Patterns, etc.), aber $A' \subseteq \mathrm{dom}(f)$ muss deshalb noch nicht gelten!

Gegenbeispiel: $E[f, x] = f\ x$

# Abstiegsfunktion



Gegeben sei eine rekursive Funktion

$$f : A \to B$$
$$f(x) = E[f, x]$$

Um festzustellen ob $f$ für alle Argumente einer gewählten Teilmenge $A' \subseteq A$ definiert ist, kann man eine **Abstiegsfunktion** verwenden. Zuerst prüfen wir, ob $A'$ sinnvoll gewählt wurde.

AUF) Unter der Voraussetzung $x \in A'$ wird im Ausdruck $E[f, x]$ die Funktion $f$ nur mit Argumenten $y \in A'$ aufgerufen.

DEF) Für $x \in A'$ ist der Ausdruck $E[f, x]$ definiert unter der Annahme, dass die getätigten Aufrufe von $f$ alle definiert sind.

Damit weiß man erst einmal, dass alle Funktionsanwendung $f\,x$ für $x \in A'$ vernünftig bleiben (keine unvollständigen Patterns, etc.), aber $A' \subseteq \mathrm{dom}(f)$ muss deshalb noch nicht gelten!

Gegenbeispiel: $E[f, x] = f(x)$

# Abstiegsfunktion



Sei nun zusätzlich $m : A \to \mathbb{N}$ eine Funktion mit $A' \subseteq \mathrm{dom}(m)$ und der folgenden Eigenschaft:

ABST) Im Rumpf $E[f, x]$ wird $f$ nur für solche $y \in A'$ aufgerufen, für welche $m(y) < m(x)$ gilt.

Man bezeichnet so ein $m$ als **Abstiegsfunktion**.

AUF+DEF+ABST: Dann gilt $A' \subseteq \mathrm{dom}(f)$.

Die Abstiegsfunktion bietet also ein Maß für die Argumente einer Funktion, welches für alle Argumente von rekursiven Ausdrücken immer kleiner wird.

*Erinnerung an Folie 1.40:* $\mathrm{dom}(f)$ bezeichnet den Definitionsbereich (engl. Domain) einer Funktion $f$

# Beispiel Fakultät

$$f : \mathbb{Z} \to \mathbb{Z}$$
$$f(x) = \begin{cases} 1, \text{ falls } x = 0 \\ x \cdot f(x-1), \text{ sonst} \end{cases}$$

Wir wählen $A' = \mathbb{N}$ und $m(x) = \max(x, 0)$ mit $m : \mathbb{Z} \to \mathbb{N}$.
In $E[f, x]$ wird $f$ einmal mit Argument $x - 1$ aufgerufen, falls $x \neq 0$ und gar nicht aufgerufen, falls $x = 0$.

AUF Wenn $x \in A'$ und $x \neq 0$ (nur dann kommt es zum Aufruf), dann ist $x - 1 \in A'$;

DEF Der Rumpf $E[f, x]$ enthält nur überall definierte Ausdrücke;

ABST Wenn $x \in A'$ und $x \neq 0$, so ist $m(x - 1) < m(x)$.

Also gilt $\mathbb{N} \subseteq \text{dom}(f)$: die durch obiges Schema definierte rekursive Funktion terminiert für alle $x \in \mathbb{N}$.
Über Argumente $x \in \mathbb{Z} \setminus \mathbb{N}$ wird hier nichts ausgesagt.

# Beispiel für kompliziertere Abstiegsfunktion

Gegeben ist die rekursive Funktion $f$ als:

$$foosum(i, n, a) = \begin{cases} a, \text{ falls } i = n \\ foosum(i + 1, n, a + i), \text{ sonst} \end{cases}$$

$$foosum(3, 6, 10) \rightsquigarrow foosum(4, 6, 13) \rightsquigarrow foosum(5, 6, 17)$$
$$\rightsquigarrow foosum(6, 6, 22) \rightsquigarrow 22$$

Wenn man wählt

- $A' = \{(i, n, a) \mid n \in \mathbb{N}, 0 \leq i \leq n, a \in \mathbb{Z}\}$
- $m(i, n, a) = \max(n - i, 0)$

kann man die Terminierung von $f$ auf $A'$ beweisen.

# Beispiel für kompliziertere Abstiegsfunktion

Gegeben ist die rekursive Funktion $f$ als:

$$foosum(i, n, a) = \begin{cases} a, \text{ falls } i = n \\ foosum(i + 1, n, a + i), \text{ sonst} \end{cases}$$

$$foosum(3, 6, 10) \rightsquigarrow foosum(4, 6, 13) \rightsquigarrow foosum(5, 6, 17)$$
$$\rightsquigarrow foosum(6, 6, 22) \rightsquigarrow 22$$

Wenn man wählt

- $A' = \{(i, n, a) \mid n \in \mathbb{N}, 0 \leq i \leq n, a \in \mathbb{Z}\}$
- $m(i, n, a) = \max(n - i, 0)$

kann man die Terminierung von $f$ auf $A'$ beweisen.

# Gegenbeispiele

Warum muss die Abstiegsfunktion nach $\mathbb{N}$ abbilden?

## Gegenbeispiel

Diese Funktion terminiert nie, auch nicht für positive Eingaben:

```
f :: Integer -> Integer
f x = f (x+1)
```

Für $x \geq 0$ werden folgende Funktionen bei jedem rekursiven Aufruf echt kleiner:

- $m'(x) = -x$ ist keine Abstiegsfunktion wegen $m' : A \to \mathbb{Z}$
- $m''(x) = \frac{1}{x}$ ist keine Abstiegsfunktion wegen $m'' : A \to \mathbb{Q}^+ \setminus \{0\}$

Das Problem besteht darin, dass weder $\mathbb{Z}$ noch $\mathbb{Q}^+ \setminus \{0\}$ ein kleinstes Element besitzen. $\mathbb{N}$ hat ein kleinstes Element, welches den Abschluß der Rekursion garantiert.
Eine Abstiegsfunktion $m''' : A \to \mathbb{N}$ kann es hier nicht geben.

# Verschiedene Formen der Rekursion

Eine rekursive Definition $f(x) = E[f, x]$ heißt:

- **linear**, wenn in $E[f, x]$ die Funktion $f$ höchstens einmal aufgerufen wird.
  Z.B. wenn in jedem Zweig höchstens ein Aufruf steht.
- **endständig**, wenn $E[f, x]$ eine Fallunterscheidung ist und jeder Zweig, in dem $f$ aufgerufen wird, von der Form $f(G)$ ist, wobei $G$ keine weiteren Aufrufe von $f$ enthält
- **mehrfach rekursiv**, wenn $E[f, x]$ möglicherweise mehrere Aufrufe von $f$ (im selben Zweig) enthält
- **verschachtelt rekursiv**, wenn die Argumente $y$ mit denen $f$ in $E[f, x]$ aufgerufen wird, selbst weitere Aufrufe von $f$ enthalten
- **wechselseitig rekursiv**, wenn im Rumpf eine andere (rekursive) Funktion aufgerufen wird, welche ihrerseits $f$ in ihrem Rumpf verwendet

# Lineare Rekursion

Bei **linearer Rekursion** findet im Rumpf höchstens ein rekursiver Aufruf statt.

## Beispiele:

- Summe, Fakultät
- ```
  collatz x
    | x <=1      = 0
    | even x     = 1 + collatz (x `div` 2)
    | otherwise = 1 + collatz (3 * x + 1)
  ```
  Auch linear, da es nur einen Aufruf *pro Zweig* gibt.

## Gegenbeispiele:

- Fibonacci hat zwei Aufrufe im Rumpf
- McCarthy's 91-Funktion

```
mc91 n
  | n > 100    = n-10
  | otherwise = mc91(mc91(n+11))
```

John McCarty, 1927-2011, amer. Informatiker


```

# Endständige Rekursion

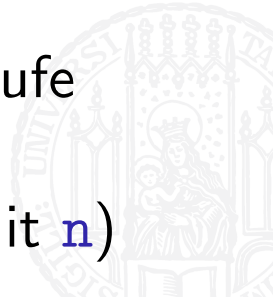
= Lineare Rekursion + Zweige aller Fallunterscheidungen sind rekursive Aufrufe, deren Ergebnisse nicht weiterverarbeitet werden.

## Beispiele

- ```
  foosum i n a
    | i == n    = a
    | otherwise = foosum (i+1) n (a+i)
  ```
- ```
  collatz' acc x
    | x <=1     = acc
    | even x    = collatz' (acc+1) (x `div` 2)
    | otherwise = collatz' (acc+1) (3 * x + 1)
  ```

## Gegenbeispiele

- Alle nicht-linearen Rekursionen
- `fakultät`, `trinum`, `collatz`, da das Ergebnis der Aufrufe dort noch weiterverarbeitet wird
  (z.B. bei `fakultät` durch Multiplikation mit `n`)



# Endrekursiv dank Akkumulator

Endrekursion kann oft effizient abgearbeitet werden.
*Grund:* Keine großen Zwischenergebnisse im Substitutionsmodell

Beispiel:

$$\texttt{collatz'}\ 0\ 11 \rightsquigarrow \texttt{collatz'}\ 1\ 34 \rightsquigarrow \texttt{collatz'}\ 2\ 17 \rightsquigarrow$$
$$\texttt{collatz'}\ 3\ 52 \rightsquigarrow \texttt{collatz'}\ 4\ 26 \rightsquigarrow \texttt{collatz'}\ 5\ 13 \rightsquigarrow \ldots$$

Viele Rekursionen lassen sich in endständige Form bringen, wenn man das vorläufige Ergebnis in einem zusätzlichen Argument aufsammelt; so ein Argument nennt man auch **Akkumulator**.

Beispiel: `foosum` ist eine endständige Version von `trinum`

```
foosum i n a | i == n     = a
             | otherwise = foosum (i+1) n (a+i)
```

$$\texttt{foosum}(0,5,5) \rightsquigarrow \texttt{foosum}(1,5,5) \rightsquigarrow \texttt{foosum}(2,5,6) \rightsquigarrow$$
$$\texttt{foosum}(3,5,8) \rightsquigarrow \texttt{foosum}(4,5,11) \rightsquigarrow \texttt{foosum}(5,5,15) \rightsquigarrow 15$$

# Endrekursiv dank Akkumulator

Endrekursion kann oft effizient abgearbeitet werden.
*Grund:* Keine großen Zwischenergebnisse im Substitutionsmodell

Beispiel:

$$\texttt{collatz'}\ 0\ 11 \rightsquigarrow \texttt{collatz'}\ 1\ 34 \rightsquigarrow \texttt{collatz'}\ 2\ 17 \rightsquigarrow$$
$$\texttt{collatz'}\ 3\ 52 \rightsquigarrow \texttt{collatz'}\ 4\ 26 \rightsquigarrow \texttt{collatz'}\ 5\ 13 \rightsquigarrow \ldots$$

Viele Rekursionen lassen sich in endständige Form bringen, wenn man das vorläufige Ergebnis in einem zusätzlichen Argument aufsammelt; so ein Argument nennt man auch **Akkumulator**.

Beispiel: `foosum` ist eine endständige Version von `trinum`

```
foosum i n a | i == n     = a
             | otherwise = foosum (i+1) n (a+i)
```

$$\texttt{foosum}(0,5,5) \rightsquigarrow \texttt{foosum}(1,5,5) \rightsquigarrow \texttt{foosum}(2,5,6) \rightsquigarrow$$
$$\texttt{foosum}(3,5,8) \rightsquigarrow \texttt{foosum}(4,5,11) \rightsquigarrow \texttt{foosum}(5,5,15) \rightsquigarrow 15$$

# Endrekursiv dank Akkumulator

Endrekursion kann oft effizient abgearbeitet werden.
*Grund:* Keine großen Zwischenergebnisse im Substitutionsmodell

Beispiel:

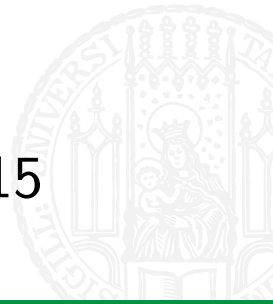
$$\texttt{collatz'}\ 0\ 11 \rightsquigarrow \texttt{collatz'}\ 1\ 34 \rightsquigarrow \texttt{collatz'}\ 2\ 17 \rightsquigarrow$$
$$\texttt{collatz'}\ 3\ 52 \rightsquigarrow \texttt{collatz'}\ 4\ 26 \rightsquigarrow \texttt{collatz'}\ 5\ 13 \rightsquigarrow \ldots$$

Viele Rekursionen lassen sich in endständige Form bringen, wenn man das vorläufige Ergebnis in einem zusätzlichen Argument aufsammelt; so ein Argument nennt man auch **Akkumulator**.

Beispiel: `foosum` ist eine endständige Version von `trinum`

```
foosum i n a | i == n     = a
             | otherwise = foosum (i+1) n (a+i)
```

$$\texttt{foosum}(0,5,5) \rightsquigarrow \texttt{foosum}(1,5,5) \rightsquigarrow \texttt{foosum}(2,5,6) \rightsquigarrow$$
$$\texttt{foosum}(3,5,8) \rightsquigarrow \texttt{foosum}(4,5,11) \rightsquigarrow \texttt{foosum}(5,5,15) \rightsquigarrow 15$$



# Endrekursiv dank Akkumulator

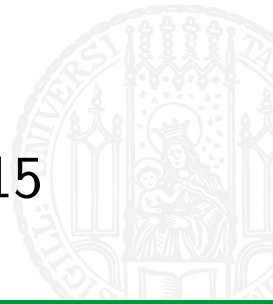
## Im Vergleich die nicht-endrekursive Version:

$$\begin{aligned}
\texttt{trinum}\ 5 &\rightsquigarrow 5 + (\texttt{trinum}\ 4) \\
&\rightsquigarrow 5 + (4 + (\texttt{trinum}\ 3)) \\
&\rightsquigarrow 5 + (4 + (3 + (\texttt{trinum}\ 2))) \\
&\rightsquigarrow 5 + (4 + (3 + (2 + (\texttt{trinum}\ 1)))) \\
&\rightsquigarrow 5 + (4 + (3 + (2 + (1 + (\texttt{trinum}\ 0))))) \\
&\rightsquigarrow 5 + (4 + (3 + (2 + (1 + 0)))) \rightsquigarrow 15
\end{aligned}$$

**Beispiel:** `foosum` ist eine endständige Version von `trinum`

```
foosum i n a | i == n     = a
             | otherwise = foosum (i+1) n (a+i)
```

$$\texttt{foosum}(0,5,5) \rightsquigarrow \texttt{foosum}(1,5,5) \rightsquigarrow \texttt{foosum}(2,5,6) \rightsquigarrow$$
$$\texttt{foosum}(3,5,8) \rightsquigarrow \texttt{foosum}(4,5,11) \rightsquigarrow \texttt{foosum}(5,5,15) \rightsquigarrow 15$$



# Endständige Rekursion und Iteration

Endrekursion entspricht im wesentlichen einer While-Schleife:

```
foosum i n a | i == n    = a
             | otherwise = foosum (i+1) n (a+i)
```

kann man ohne Rekursion in einer imperativen Sprache schreiben als

```
int foosum(int i, int n, int a) {
    while (!(i=n)) {
      a = a+i;
      i = i+1;
    }
  return a;
}
```

# Endständige Rekursion und Iteration

Endrekursion entspricht im wesentlichen einer While-Schleife:

```
foosum i n a | i == n    = a
             | otherwise = foosum (i+1) n (a+i)
```

Zur Demonstration der Abstiegsfunktion wurde `foosum` zuvor absichtlich *unnötig kompliziert* gewählt!

Eine einfachere endrekursive Variante von `trisum` ist:

```
trisum' :: Integer -> Integer
trisum' n = ts' 0 n
  where
    ts' :: Integer -> Integer -> Integer
    ts' acc 0 = acc
    ts' acc m = ts' (m-1) (acc+m)
```

# Beispiel: Potenzierung

Rekursives Programm zur Potenzierung:

```
potenz :: Double -> Int -> Double
potenz _x 0 = 1
potenz  x n = x * potenz x (n-1)
```

Endrekursive verallgemeinerte Version:

```
potenz2 :: Double -> Int -> Double
potenz2 x n = potenz2' 1 n
  where
    potenz2' :: Double -> Int -> Double
    potenz2' acc 0 = acc
    potenz2' acc m = potenz2' (x*acc) (m-1)
```

Die Funktion `potenz2' acc n` berechnet $x^n \cdot acc$.

Da `potenz2'` eine lokale Definition ist, ist `x` ohne Übergabe verfügbar; dadurch fehlt ein zusätzlicher `Double`-Parameter.

# Beispiel: Potenzierung

Rekursives Programm zur Potenzierung:

```
potenz :: Double -> Int -> Double
potenz _x 0 = 1
potenz  x n = x * potenz x (n-1)
```

Endrekursive verallgemeinerte Version:

```
potenz2 :: Double -> Int -> Double
potenz2 x n = potenz2' 1 n
  where
    potenz2' :: Double -> Int -> Double
    potenz2' acc 0 = acc
    potenz2' acc m = potenz2' (x*acc) (m-1)
```

Die Funktion `potenz2' acc n` berechnet $x^n \cdot acc$.

Da `potenz2'` eine lokale Definition ist, ist `x` ohne Übergabe verfügbar; dadurch fehlt ein zusätzlicher `Double`-Parameter.

# Beispiel: Potenzierung

Rekursives Programm zur Potenzierung:

```
potenz :: Double -> Int -> Double
potenz _x 0 = 1
potenz  x n = x * potenz x (n-1)
```

Endrekursive verallgemeinerte Version:

```
potenz2 :: Double -> Int -> Double
potenz2 x n = potenz2' 1 n
  where
    potenz2' :: Double -> Int -> Double
    potenz2' acc 0 = acc
    potenz2' acc m = potenz2' (x*acc) (m-1)
```

Die Funktion `potenz2' acc n` berechnet $x^n \cdot acc$.

Da `potenz2'` eine lokale Definition ist, ist `x` ohne Übergabe verfügbar; dadurch fehlt ein zusätzlicher `Double`-Parameter.

# Schnelle Potenzierung

Tatsächlich geht es aber noch besser:

```
potenz3 :: Double -> Int -> Double
potenz3 _ 0 = 1
potenz3 x n
  | even n    =     x_to_halfn_sq
  | otherwise = x * x_to_halfn_sq
  where
    x_to_halfn_sq = square (potenz3 x (n `div` 2))
    square y      = y * y
```

Begründung: $x^{2k} = (x^k)^2$ und $x^{2k+1} = (x^k)^2 \cdot x$.

Aufgabe: Schafft es jemand, eine endrekursive Version davon anzugeben, ohne dabei in die Standardbibliothek zu spicken?

# Rekursion mit Listen

Rekursion ist nicht auf Zahlen beschränkt:

```
length :: [a] -> Int
length []     = 0
length (_h:t) = 1 + length t
```

```
length [1,2,3]
↝ 1 + (length [2,3])
↝ 1 + 1 + (length [3])
↝ 1 + 1 + 1 + (length [])
↝ 1 + 1 + 1 + 0
↝ 3
```

# Rekursion mit Listen

Auch das Ergebnis muss keine Zahl sein:

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

```
reverse "fly"
↝ reverse "ly" ++ "f"
↝ reverse "y" ++ "l" ++ "f"
↝ reverse "" ++ "y" ++ "l" ++ "f"
↝ "" ++ "y" ++ "l" ++ "f"
↝ "ylf"
```

Rekursion eignet sich hervorragend dazu, Listen zu bearbeiten!

# Beispiel: Sortieren durch Einfügen

```
insert :: Int -> [Int] -> [Int]
insert x    [] = [x]
insert x (y:ys)
  | x <= y    = x : y : ys
  | otherwise = y : insert x ys
```

```
insert 2 [1,3,4,5]
⇝ 1 : insert 2 [3,4,5]
⇝ 1 : 2 : 3 : [4,5]
= [1,2,3,4,5]
```

```
sort :: [Int] -> [Int]
sort    []  = []
sort (x:xs) = insert x (sort xs)
```

# Beispiel: Sortieren durch Einfügen

```
insert :: Int -> [Int] -> [Int]
insert x     [] = [x]
insert x (y:ys)
  | x <= y     = x : y : ys
  | otherwise = y : insert x ys
```

```
insert 2 [1,3,4,5]
⇝ 1 : insert 2 [3,4,5]
⇝ 1 : 2 : 3 : [4,5]
= [1,2,3,4,5]
```

```
sort :: [Int] -> [Int]
sort     []  = []
sort (x:xs) = insert x (sort xs)
```

# Rekursion mit mehreren Listen

Wir können auch mehrere Listen auf einmal berarbeiten:

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

```
zip "abc" [1,2,3,4]
⇝ ('a',1): zip "bc" [2,3,4]
⇝ ('a',1):('b',2): zip "c" [3,4]
⇝ ('a',1):('b',2):('c',3): zip "" [4]
⇝ ('a',1):('b',2):('c',3):[]
=     [('a',1),('b',2),('c',3)]
```

# Richtlinien zur Rekursion

Viele Studenten haben anfangs Probleme damit, bewusst rekursive Funktionen zu schreiben.
*G.Hutton schreibt dazu:* Es ist wie Fahrrad fahren, wenn man es nicht kann sieht es unmöglich aus; und wenn man es kann, ist wirklich einfach.

1. Explizit **Typ der Funktion** hinschreiben hilft!
2. **Fallunterscheidungen** auflisten. Bei Listen ist fast immer klar, dass zwischen leerer und nicht-leerer Liste unterschieden werden muss; bei natürlichen Zahlen erfordert oft die Null oder Eins eine Sonderbehandlung
3. **Einfach Fälle** zuerst programmieren, die rekursiven Fälle kommen dann oft von alleine.
4. In **rekursiven Fällen** überlegen, was zur Verfügung steht!
5. Am Ende dann **verallgemeinern und vereinfachen**.

# Beispiel: drop

`drop` ist eine Funktion der Standardbibliothek, welche eine Zahl *n* und eine Liste *xs* als Argumente nimmt, und als Ergebnis eine Liste zurückliefert, bei der die ersten *n* Elemente fehlen.

Online-Demonstration ergab den Code:

```
myDrop :: Int -> [a] -> [a]
myDrop 0 []     = []
myDrop 0 (h:t)  = h:t
myDrop n []     = []
myDrop n (h:t)  = myDrop (n-1) t
```

. . . welchen wir anschliessend leicht vereinfachen konnten zu:

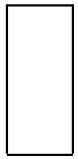
```
myDrop 0 xs     = xs
myDrop _ []     = []
myDrop n (_:t)  = myDrop (n-1) t
```

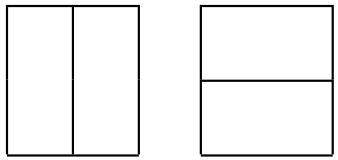
# Zusammenfassung Rekursion

- Rekursion als Mittel zur Definition von Funktionen
- Rekursion im Substitutionsmodell
- Rekursive Funktionen sind oft nicht überall definiert. Mit einer Abstiegsfunktion kann man die Definiertheit an bestimmten Stellen nachweisen.
- Verschiedene Arten der rekursiven Definition, insbesondere endständige Rekursion.
- Rekursion als Mittel zur Problemlösung (Bsp. Hanoi)
- Rekursion für den Umgang mit Listen (Bsp. Sortieren)

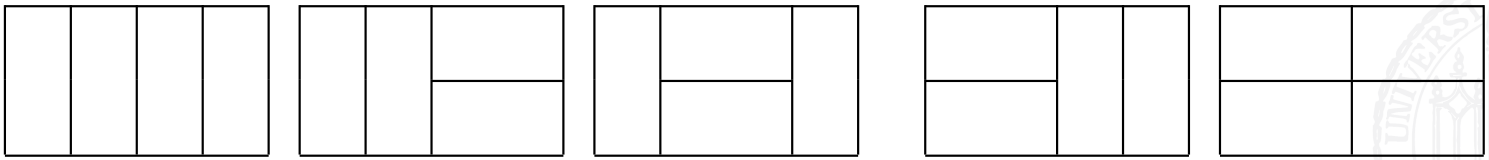
# Kachelungen Zählen

Wie viele Möglichkeiten gibt es einen $n \times 2$ Korridor mit $2 \times 1$ Kacheln zu belegen?

$n = 1$ 

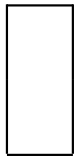
$n = 2$ 

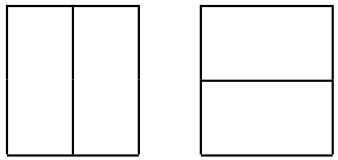
$n = 3$ 

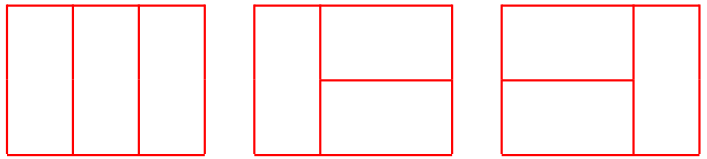
$n = 4$ 

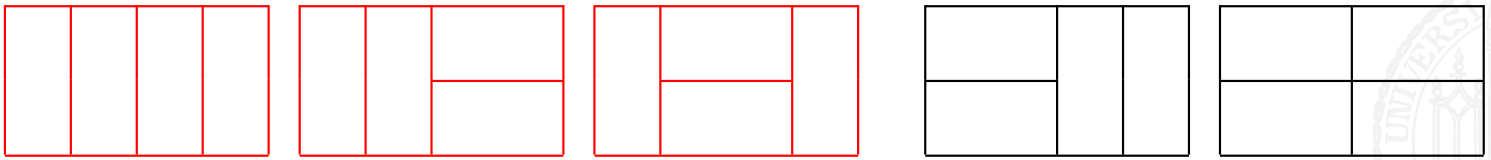
# Kachelungen Zählen

Wie viele Möglichkeiten gibt es einen $n \times 2$ Korridor mit $2 \times 1$ Kacheln zu belegen?

$n = 1$ 

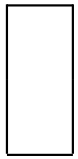
$n = 2$ 

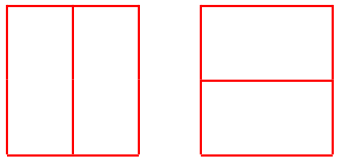
$n = 3$ 

$n = 4$ 

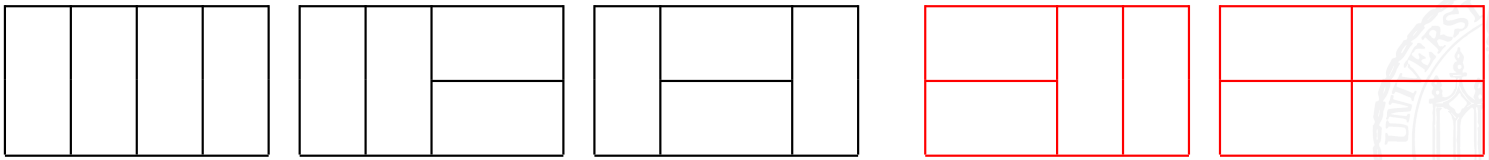
# Kachelungen Zählen

Wie viele Möglichkeiten gibt es einen $n \times 2$ Korridor mit $2 \times 1$ Kacheln zu belegen?

$n = 1$ 

$n = 2$ 

$n = 3$ 

$n = 4$ 

# Induktionsprinzip: Beispiel 1

Wir möchten zeigen, dass die Zahl der möglichen Kachelungen eines $n \times 2$ Korridors mit $2 \times 1$ Kacheln tatsächlich gleich $F_n$ ist; wobei $F_0 = F_1 = 1$ und $F_n = F_{n-1} + F_{n-2}$, falls $n > 1$.

siehe Folie 3.8 "Fibonacci-Zahlen"

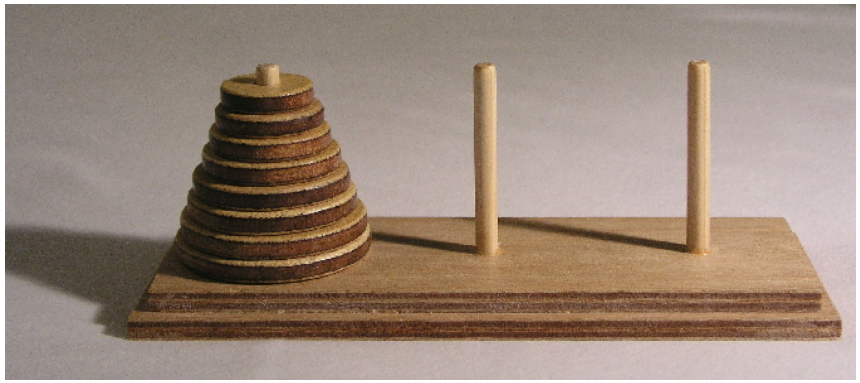
- Für $n = 0$ und $n = 1$ stimmt die Behauptung trivialerweise.
- Wenn $n > 1$, so gibt es zwei Möglichkeiten, die Kachelung zu beginnen: Man beginnt mit einer quergestellten Kacheln und hat dann noch eine $(n-1) \times 2$ Fläche zu kacheln.
- Oder man beginnt mit zwei längsgestellten Kacheln und hat dann noch eine $(n-2) \times 2$ Fläche zu kacheln.
- "per Induktion" oder "rekursiv" können wir annehmen, dass die Zahl der Möglichkeiten hierfür $F_{n-1}$ bzw. $F_{n-2}$ ist,
- Also ergeben sich $F_{n-1} + F_{n-2} = F_n$ Möglichkeiten, w.z.b.w.

Zwei Möglichkeiten zu Beginnen:

# Induktionsprinzip: Beispiel 2

Wir möchten zeigen, dass der auf Folie 3.11 angegebene Algorithmus tatsächlich das Hanoi-Problem löst.

- Für $n = 1$ stimmt die Behauptung.
- Wenn $n > 1$, so schafft der erste Aufruf "per Induktion" die obersten $n - 1$ Scheiben auf den Hilfsstapel;
  der nächste Befehl schafft die unterste Scheibe ans Ziel;
  der zweite Aufruf schafft wiederum "per Induktion" die $n - 1$ Scheiben vom Hilfsstapel ans Ziel.



Quelle: Wikipedia

# Induktionsprinzip: Beispiel 3

Wir möchten zeigen, dass jede Befehlsfolge für das Hanoi-Puzzle, welche $n$ Scheiben von einem Stapel auf einen anderen schafft, mindestens $2^n - 1$ Befehle enthält.

- Für $n = 1$ stimmt das: $2^1 - 1 = 1$; eine Scheibe ist zu bewegen.
- Wenn $n > 1$, so muss irgendwann die unterste Scheibe bewegt werden und zwar auf einen freien Stapel; also eine Bewegung.
  - Vorher müssen aber die $n - 1$ darüberliegenden Scheiben auf den jeweils anderen verschafft wurden, damit dort Platz ist. "Per Induktion" erfordert das mindestens $2^{n-1} - 1$ Befehle.
  - Anschließend müssen diese $n - 1$ Scheiben auch ans Ziel, macht noch einmal mindestens $2^{n-1} - 1$ Befehle.

  Insgesamt also mindestens $2(2^{n-1} - 1) + 1 = 2^n - 1$ Befehle.

# Das Induktionsprinzip

Ein Induktionsbeweis ist eigentlich ein rekursiver Beweis. Man darf die zu beweisende Aussage selbst benutzen. Allerdings muss die Kette solcher Rückgriffe irgendwann zum Ende kommen. Dafür bietet sich eine Abstiegsfunktion an.

## Induktionsprinzip

Es sei $A$ eine Menge, $P$ eine Eigenschaft auf $A$ (formal $P \subseteq A$) und $m : A \to \mathbb{N}$ eine Funktion, sodass für alle $a \in A$ gilt:

$$\text{wenn alle } y \in A \text{ mit } m(y) < m(a) \text{ in } P \text{ sind,}$$
$$\text{dann auch } a \in P$$

Dann gilt $P = A$, also alle Elemente von $A$ haben Eigenschaft $P$.

Insbesondere muss natürlich für alle $a$ mit $m(a) = 0$ direkt $a \in P$ gezeigt werden, da es in diesem Fall keine $y$ mit $m(y) < m(a)$ gibt.

# Beweis des Induktionsprinzips

## Induktionsprinzip

Es sei $A$ eine Menge, $P$ eine Eigenschaft auf $A$ (formal $P \subseteq A$) und $m : A \to \mathbb{N}$ eine Funktion, sodass für alle $a \in A$ gilt:

$$\text{wenn alle } y \in A \text{ mit } m(y) < m(a) \text{ in } P \text{ sind,}$$
$$\text{dann auch } a \in P$$

Dann gilt $P = A$, also alle Elemente von $A$ haben Eigenschaft $P$.

## Beweis

Wenn $P \neq A$ gelten würde, so sei $a \in A \setminus P$ ein Gegenbeispiel mit minimalem $m$-Wert. Für alle $y$ mit $m(y) < m(a)$ gelte also $y \in P$. Nach Annahme gilt dann aber auch $a \in P$. Ein Widerspruch.

# Anmerkungen

- Oft ist $A = \mathbb{N}$ und $m(a) = a$. Dann darf man also verwenden, dass die Behauptung $y \in P$ für alle $y < a$ schon gezeigt ist und hat dann $a \in P$ daraus herzuleiten.
- Bisweilen erlaubt man hier nur den Rückgriff auf $a - 1$ und nicht auf beliebiges $y < a$.
- Man kann auch Induktionsprinzipien für andere Maß-Werte als $\mathbb{N}$ betrachten. Es kommt nur darauf an, dass jede echt absteigende Kette von Maß-Werten irgendwann zum Ende kommt.
  *Beispiel:* Paare von natürlichen Zahlen mit der lexikographischen Ordnung.
- In den meisten Fällen kommt man aber mit $\mathbb{N}$-wertigen Abstiegsfunktionen aus.

# Beispiel: Endständige Rekursion

```
foosum (i,n,a) | i>=n      = a
               | otherwise = foosum (i+1,n,a+i)
```

Man zeige durch Induktion, dass $\texttt{foosum}(i, n, a) = a + \sum_{j=i}^{n-1} j$.

- Als Abstiegsfunktion nimmt man $m(i, n, a) = \max(n - i, 0)$.
- Falls $m(i, n, a) = 0$ so gilt $i \geq n$. Damit ist die Summe leer und wir haben $\texttt{foosum}(i, n, a) = a = a + \sum_{j=n}^{n-1} j$
- Falls $m(i, n, a) > 0$ so gilt $i < n$ und damit nach der Definition

$$\texttt{foosum}(i, n, a) \;=\; \texttt{foosum}(i + 1, n, a + i)$$

Wegen $m(i, n, a) > m(i + 1, n, a + i)$ folgern wir mit Induktion

$$\texttt{foosum}(i + 1, n, a + i) \;=\; a + i + \sum_{j=i+1}^{n-1} j \;=\; a + \sum_{j=i}^{n-1} j$$

# Partielle Korrektheit

Oft hat man eine rekursive Funktion $f(a) = E[f, a]$ gegeben und möchte eine Aussage der Form $\forall a \in D.Q(a, f(a))$ zeigen, wobei $D$ der Definitionsbereich von $f$ ist.

Es genügt dann, für jedes $a$ zu zeigen, dass $Q(a, f(a))$ erfüllt ist unter der Annahme, dass für alle rekursiven Aufrufe $f(a_1), \ldots, f(a_n)$, die in $E[f, a]$ vorkommen, bereits $Q(a_i, f(a_i))$ erfüllt ist.

Man kann nämlich dann als Abstiegsfunktion die Zahl der Auswertungsschritte nehmen. Auf dem Definitionsbereich ist diese erklärt und für alle rekursiven Aufrufe strikt kleiner.

Damit haben wir bewiesen: *Falls* die Berechnung terminiert, dann erfüllt das Ergebnis die gewünschte Eigenschaft.

Merke: Will man eine rekursive Funktion verifizieren, so darf man die rekursiven Aufrufe bereits als korrekt annehmen.

# Zusammenfassung Induktion

- Induktion als "rekursiver Beweis" mit überall definierter Abstiegsfunktion
- *Beispiele:* Kachelungen, Türme von Hanoi, Endrekursion
- Induktion ist ein Beweisverfahren um Aussagen über unendliche Mengen zu treffen.
- Partielle Korrektheit als Spezialfall der Induktion