Freie Universität Berlin

ALP I

ıktionen höherer Ordnung

Teil 2

SS 2013

Prof. Dr. Margarita Esponda

nktionen höherer Ordnung

an, wir möchten alle Zahlen innerhalb einer
nder addieren

```
l:: (Num a) => [a] -> a
l [] = 0
l (x:xs) = x + addAll xs
```

l-Operation über alle Elemente einer Liste

```
ll:: [Bool] -> Bool
ll [] = True
ll (x:xs) = x && (trueAll xs)
```

nktionen höherer Ordnung

eiten von beiden Funktionen sind:

Operator

nter Wert, wenn die Liste leer ist.

es Rekursions-Muster

ine verallgemeinerte Funktion definieren, die

ne löst

```
trueAll  = betweenAll  (&&)  True
addAll   = betweenAll  (+)   0
multAll  = betweenAll  (*)   1
```

Verallgemeinerungen sind immer gut!

ktionen höherer Ordnung

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

Operation

Wert der Funktion,
wenn die Liste leer ist

```
All  f   k  []  = k
All  f   k  (x:xs) = f  x  (betweenAll  f  k  xs)
```

Funktionen höherer Ordnung

```
a -> a) -> a -> [a] -> a
= k
xs) = f  x  (betweenAll  f  k  xs)
```

: $[x_1, x_2, \ldots, x_{n-1}, x_n]$

x_1 (betweenAll f k $[x_2, \ldots, x_{n-1}, x_n]$)

x_1 (f x_2 (betweenAll f k $[x_3, \ldots, x_{n-1}, x_n]$))

x_1 (f x_2 (f x_3 (betweenAll f k $[x_4, \ldots, x_{n-1}, x_n]$)))

...

x_1 (f x_2 (f x_3 (..... (f x_{n-1} (f x_n (betweenAll f k **[]**)))...)

x_1 (f x_2 (f x_3 (... (f x_{n-1} (**f x_n k**)))...)

...

x_1 **w_2**

nktionen höherer Ordnung

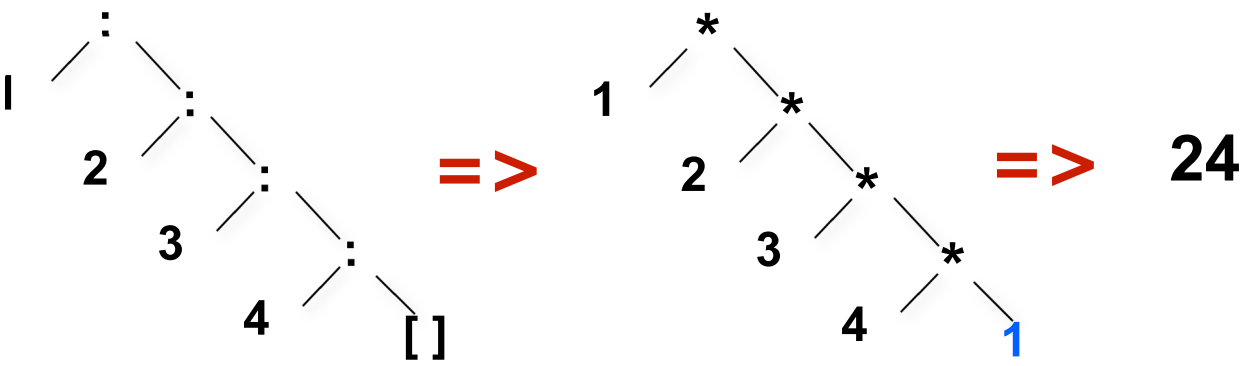
foldr-Funktion

st bereits eine allgemeine Funktion

:, die Faltungs-Operator genannt wird

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

1 [1,2,3,4]



Faltungs-Operatoren

] ⇒ (*) 1 (foldr (*) 1 [2,3,4])
⇒ (*) 1 ((*) 2 (foldr (*) 1 [3,4]))
⇒ (*) 1 ((*) 2 ((*) 3 (foldr (*) 1 [4])))
⇒ (*) 1 ((*) 2 ((*) 3 ((*) 4 (foldr (*) 1 []))))
⇒ (*) 1 ((*) 2 ((*) 3 ((*) 4 1)))
⇒ (*) 1 ((*) 2 ((*) 3 4))
⇒ (*) 1 ((*) 2 12)
⇒ (*) 1 24
⇒ 24

r (*) 1 [1..n]

nktionen höherer Ordnung

tandard-Funktionen von Haskell können mit ltungs-Operators definiert werden:

```
sum :: (Num a) => [a] -> a
sum         =  foldr  (+)  0

product  :: (Num a) => [a]   -> a
product     =  foldr  (*)  1

or :: [Bool] -> Bool
or          =  foldr  (||)  False

or :: [Bool] -> Bool
and         =  foldr  (&&)  True
```

Natur rekursiver Funktionen

nktionen haben oft folgende allgemeine Form:

0 = c

(n+1) = h (f n)

· Definitionen wird oft als Strukturelle
er die natürlichen Zahlen bezeichnet.

e Natur rekursiver Funktionen

definition dieser Form über die natürlichen
us wie folgt:

1+1+ … +1+0 = die natürliche Zahl n.

) mit **c** und **(1+)** mit **h** ersetzen, bekommen wir
druck

h(h(h(…h(h(c))…))),

nal auf **c** = f(0) angewendet wird.

```
: 0
: (1+) (f n)
```

```
f   0     = c
f (n+1)   = h (f n)
```

e Natur rekursiver Funktionen

Faltungsfunktion stellt eine Verallgemeinerung
onen mit dieser einfachen Grundform dar:

```
natFold :: (a->a) -> a -> Integer -> a
natFold  h  c  0 = c
natFold  h  c  (n+1) = h (natFold  h  c  n)
```

nz-Funktion:

$$potenz(n,m) = n^m \quad für \quad n,m \in \mathbb{N}$$

```
potenz :: Integer → Integer → Integer
potenz  n  m  =  natFold (*n) 1 m
```

Rekursionsarten

rsion

ktionen, die in jedem Zweig ihrer Definition maximal
en Aufruf beinhalten, werden als linear rekursiv

(*tail recursion*)

e Funktionen werden als endrekursive Funktionen
enn der rekursive Aufruf in jedem Zweig der
etzte Aktion zur Berechnung der Funktion ist.

nktionen höherer Ordnung

foldl-Funktion

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

$,x_2, \ldots,x_n] \Rightarrow$ **foldl** f (f **z** x_1) $[x_2,\ldots,x_n]$

$\Rightarrow$ **foldl** f (f (f **z** x_1) x_2) $[x_3,\ldots,x_n]$

...

$\Rightarrow$ **foldl** f (f...(f (f (f **z** x_1) x_2) x_3)...) []

$\Rightarrow$ (f...(f (f (f **z** x_1) x_2) x_3)...)

nktionen höherer Ordnung

```
z
= foldl f (f z x) xs
```

foldl (*) **1** [8,6,4] ⇒ **foldl** (*) ((*) **1** 8) [6,4]

⇒ **foldl** (*) **8** [6,4]

⇒ **foldl** (*) ((*) 8 6) [4]

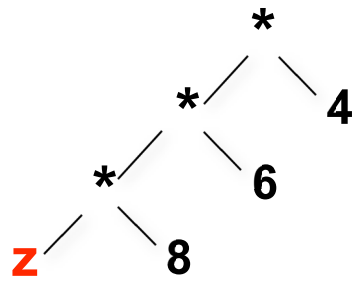
⇒ **foldl** (*) **48** [4]

⇒ **foldl** f ((*) 48 4) []

⇒ **foldl** f **192** []

⇒ **192**

⇒



Die foldl-Funktion

ispiel von Endrekursion

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

Hier werden Zwischenergebnisse akkumuliert und weitergeleitet.

Faltungs-Operatoren können sehr leicht
e Funktionen definiert werden.

```
Ord a) => [a] -> a
xs)  =  foldl  max  x  xs
```

```
[a] -> Int
= foldl addOne 0 xs
        where
         addOne a b = a + 1
```

```
teger -> Integer -> Integer
  = foldl  (*)  1  (take  n  [b,b..b])
```

spiele endrekursiver Funktionen

iel einer nicht endrekursiven Definition ist:
nition der reverse-Funktion

[a]

rev xs ++ [x]

aufwand von rev:

.., x_n]	=> rev [x_2, …, x_n] ++ [x_1]	1
	=> rev [x_3, …, x_n] ++ [x_2] ++ [x_1]	1
	. . .	
	=> [x_n] ++ [x_{n-1}] ++ [x_2] ++ [x_1]	1
	=> [] ++ [x_n] ++ … ++ [x_2] ++ [x_1]	1

bis hier (n+1) Reduktionen!

aufwand von rev

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

Reduktionen!

Reduktionen

$_1$] ++ [x_{n-1}] ++ … ++ [x_2] ++ [x_1]	
[x_{n-1}] ++ … ++ [x_2] ++ [x_1]	1
++ … ++ [x_2] ++ [x_1]	2
x_{n-2}] ++ … ++ [x_2] ++ [x_1]	3
	.
… ,x_1]	n

Quadratischer Ausführungsaufwand!

ahl der Reduktionen ist:

$$= \sum_{i=1}^{n+1} i = \frac{(n+2)(n+1)}{2} = \frac{1}{2}n^2 + 3n + 1 \in O(n^2)$$

ine effizientere Version von rev

```
ev_helper xs []

_helper [] ys = ys
_helper (x:xs) ys = rev_helper xs (x:ys)
```

and:

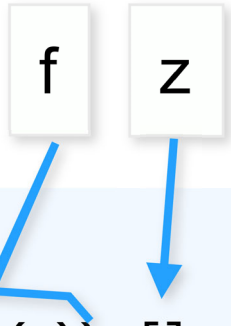
Reduktionen

…, x_n] => rev_helper [x_1,…,x_n] []	1	n
=> rev_helper [x_2,…,x_n] (x_1:[])	1	
=> rev_helper [x_3,…,x_n] (x_2:x_1:[])	1	
. . .	…	
=> (x_n:, … ,x_2:x_1:[])	1	n
=> (x_n:, … ,x_2:[x_1])	1	
. . .	…	
=> (x_n:, … , x_3:[x_2,x_1])	1	

lineare Komplexität — 2n = **O(n)**

```
    = z
xs) = foldl f (f z x) xs
```

-Funktion mit
erator




```
eloaded :: [a]  ->  [a]
eloaded xs = foldl  (flip (:))  []  xs
```

tion
e
ür die

```
flip  :: (a -> b -> c) -> b -> a -> c
flip  f  x  y    =   f  y  x
```

lauf:

ded $[x_1, x_2, \ldots, x_n]$

=> foldl (flip (:)) **[]** $[x_1, x_2, \ldots, x_n]$

2 => foldl (flip (:)) ((flip (:)) [] x_1) $[x_2, x_3, \ldots, x_n]$

=> foldl (flip (:)) ((:) x_1 []) $[x_2, x_3, \ldots, x_n]$

=> foldl (flip (:)) (x_1:[]) $[x_2, x_3, \ldots, x_n]$

=> foldl (flip (:)) [x_1] $[x_2, x_3, \ldots, x_n]$

2 => foldl (flip (:)) ((flip (:)) [x_1] x_2) $[x_3, \ldots, x_n]$

=> foldl (flip (:)) ((:) x_2 [x_1]) $[x_3, \ldots, x_n]$

=> foldl (flip (:)) (x_2:[x_1]) $[x_3, \ldots, x_n]$

=> foldl (flip (:)) [x_2, x_1] $[x_3, \ldots, x_n]$

2 => . . .

nktionen höherer Ordnung

gsbeispiel der zipWith-Funktion:

dukt von zwei Vektoren $\mathbf{v_1} \cdot \mathbf{v_2}$

$(x_1, x_2, .. , x_n)$

$(y_1, y_2, .. , y_n)$

$\mathbf{v_2 = x_1 \cdot y_1 + x_2 \cdot y_2 + .. + x_n \cdot y_n}$

```
d ::[Int] -> [Int] -> Int
d xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

ınktionen höherer Ordnung

unktion berechnet die Fibonacci-Zahlen in

it O(n)

```
 [Integer]
 0 : 1 : zipWith (+) fibs (tail fibs)
```

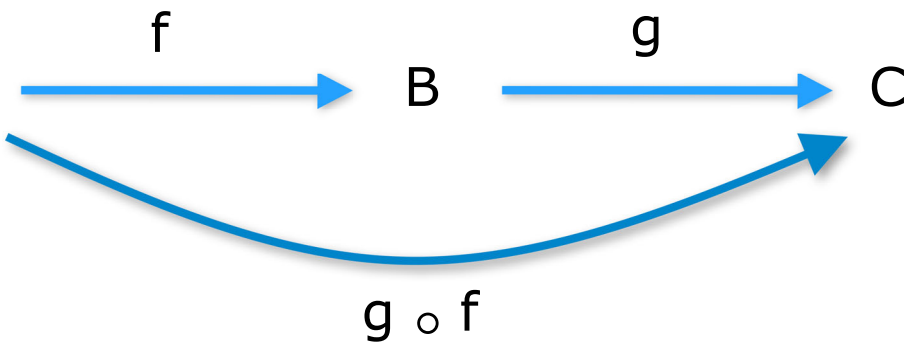
```
          0 : 1 : 1 : 2 : 3 : 5 : 8 : . . .
)s        1 : 1 : 2 : 3 : 5 : . . .
:h (+)    1 : 2 : 3 : 5 : 8 : . . .
```

ıdungsbeispiel:

```
take  40  fibs
```

nktionen höherer Ordnung

Funktionskomposition



.) :: (b → c) → (a → b) → (a → c)

.) g f x = (g (f x))

piel:

ungerade = not ▪ gerade