



# Praktische Informatik 3: Funktionale Programmierung

## Vorlesung 5 (15.11.2022): Funktionen Höherer Ordnung I

Christoph Lüth



Deutsches  
Forschungszentrum  
für Künstliche  
Intelligenz GmbH



Universität  
Bremen

Wintersemester 2022/23

# Fahrplan

- ▶ **Teil I: Funktionale Programmierung im Kleinen**
  - ▶ Einführung
  - ▶ Funktionen
  - ▶ Algebraische Datentypen
  - ▶ Typvariablen und Polymorphie
  - ▶ **Funktionen höherer Ordnung I**
  - ▶ Rekursive und zyklische Datenstrukturen
  - ▶ Funktionen höherer Ordnung II
- ▶ **Teil II: Funktionale Programmierung im Großen**
- ▶ **Teil III: Funktionale Programmierung im richtigen Leben**

# Inhalt

- ▶ Funktionen **höherer Ordnung**:
  - ▶ Funktionen als **gleichberechtigte Objekte**
  - ▶ Funktionen als **Argumente**
- ▶ Spezielle Funktionen: **map**, **filter**, **fold** und Freunde

## Lernziel

Wir verstehen, wie wir mit **map**, **filter** und **fold** wiederkehrende Funktionsmuster kürzer und verständlicher aufschreiben können, und wir verstehen, warum der Funktionstyp in  $\alpha \rightarrow \beta$  ein Typ wie jeder andere ist.

# I. Funktionen als Werte

# Funktionen Höherer Ordnung

## Slogan

“Functions are first-class citizens.”

- ▶ Funktionen sind **gleichberechtigt**: Ausdrücke wie **alle anderen**
- ▶ **Grundprinzip** der funktionalen Programmierung
- ▶ Modellierung **allgemeiner Berechnungsmuster**
- ▶ Kontrollabstraktion

# Ähnliche Datentypen der letzten Vorlesung

```
data Lager = LeeresLager
          | Lager Artikel Menge Lager
```

```
data Einkaufskorb = LeererKorb
                    | Einkauf Artikel Menge Einkaufskorb
```

```
data MyString = Empty
              | Char ::+ MyString
```

- ▶ ein **konstanter** Konstruktor
- ▶ ein **linear rekursiver** Konstruktor

# Ähnliche Datentypen der letzten Vorlesung

```
data Lager = LeeresLager
          | Lager Artikel Menge Lager
```

```
data Einkaufskorb = LeererKorb
                    | Einkauf Artikel Menge Einkaufskorb
```

```
data MyString = Empty
              | Char ::+ MyString
```

- ▶ ein **konstanter** Konstruktor
- ▶ ein **linear rekursiver** Konstruktor

Gelöst durch Polymorphie

## Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufskorb → Int
kasse LeererKorb = 0
kasse (Einkauf a m e) = cent a m + kasse e
```

```
inventur :: Lager → Int
inventur LeeresLager = 0
inventur (Lager a m l) = cent a m + inventur l
```

```
length :: MyString → Int
length Empty = 0
length (c :+ s) = 1 + length s
```

Gemeinsamkeiten:

- ▶ ein Fall pro Konstruktor
- ▶ **linearer** rekursiver Aufruf

## Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufskorb → Int
kasse LeererKorb = 0
kasse (Einkauf a m e) = cent a m + kasse e
```

```
inventur :: Lager → Int
inventur LeeresLager = 0
inventur (Lager a m l) = cent a m + inventur l
```

```
length :: MyString → Int
length Empty = 0
length (c :+ s) = 1 + length s
```

Gemeinsamkeiten:

- ▶ ein Fall pro Konstruktor
- ▶ **linearer** rekursiver Aufruf

**Nicht** durch Polymorphie gelöst

# Gesucht: Einheitlicher Rahmen

- Zwei ähnliche Funktionen:

```
toL :: String → String
toL []     = []
toL (c:cs) = toLower c : toL cs
```

```
toU :: String → String
toU []     = []
toU (c:cs) = toUpper c : toU cs
```

- Warum nicht **eine** Funktion ...

# Gesucht: Einheitlicher Rahmen

- Zwei ähnliche Funktionen:

```
toL :: String → String
toL []     = []
toL (c:cs) = toLower c : toL cs
```

```
toU :: String → String
toU []     = []
toU (c:cs) = toUpper c : toU cs
```

- Warum nicht **eine** Funktion ...

```
map f []     = []
map f (c:cs) = f c : map f cs
```

# Gesucht: Einheitlicher Rahmen

- Zwei ähnliche Funktionen:

```
toL :: String → String
toL []     = []
toL (c:cs) = toLower c : toL cs
```

```
toU :: String → String
toU []     = []
toU (c:cs) = toUpper c : toU cs
```

- Warum nicht **eine** Funktion und **zwei** Instanzen?

```
map f []     = []
map f (c:cs) = f c : map f cs
```

```
toL cs = map toLower cs
toU cs = map toUpper cs
```

- **Funktion f als Argument**

- Was hätte **map** für einen **Typ**?

# Funktionen als Werte: Funktionstypen

- ▶ Was hätte `map` für einen **Typ**?

```
map f []      = []
```

```
map f (c:cs) = f c : map f cs
```

- ▶ Was ist der Typ des ersten Arguments?

# Funktionen als Werte: Funktionstypen

- ▶ Was hätte `map` für einen **Typ**?

```
map f []      = []
map f (c:cs) = f c : map f cs
```

- ▶ Was ist der Typ des **ersten Arguments**?  $\alpha \rightarrow \beta$
- ▶ Was ist der Typ des **zweiten Arguments**?

# Funktionen als Werte: Funktionstypen

- ▶ Was hätte `map` für einen **Typ**?

```
map f []      = []
map f (c:cs) = f c : map f cs
```

- ▶ Was ist der Typ des **ersten Arguments**?  $\alpha \rightarrow \beta$
- ▶ Was ist der Typ des **zweiten Arguments**?  $[\alpha]$
- ▶ Was ist der **Ergebnistyp**?

# Funktionen als Werte: Funktionstypen

- ▶ Was hätte `map` für einen **Typ**?

```
map f []      = []
map f (c:cs) = f c : map f cs
```

- ▶ Was ist der Typ des **ersten Arguments**?  $\alpha \rightarrow \beta$
- ▶ Was ist der Typ des **zweiten Arguments**?  $[\alpha]$
- ▶ Was ist der **Ergebnistyp**?  $[\beta]$
- ▶ Alles **zusammengesetzt**:

# Funktionen als Werte: Funktionstypen

- ▶ Was hätte `map` für einen **Typ**?

```
map f []      = []
map f (c:cs) = f c : map f cs
```

- ▶ Was ist der Typ des **ersten Arguments**?  $\alpha \rightarrow \beta$
- ▶ Was ist der Typ des **zweiten Arguments**?  $[\alpha]$
- ▶ Was ist der **Ergebnistyp**?  $[\beta]$
- ▶ Alles **zusammengesetzt**:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
```

☞ Siehe Übung 5.1

## II. Map und Filter

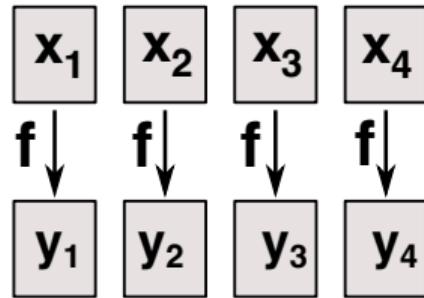
# Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
```

```
map f [] = []
```

```
map f (c:cs) = f c : map f cs
```



- ▶ Auswertung:  
`toL "AB"`

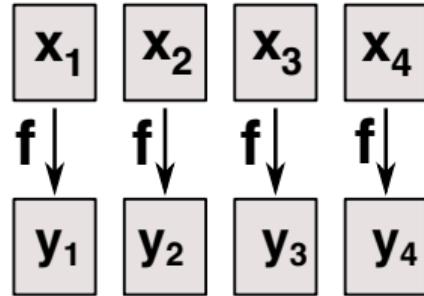
# Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
```

```
map f [] = []
```

```
map f (c:cs) = f c : map f cs
```



- ▶ Auswertung:

```
toL "AB"  $\rightarrow$  map toLower ('A':'B':[ ])
```

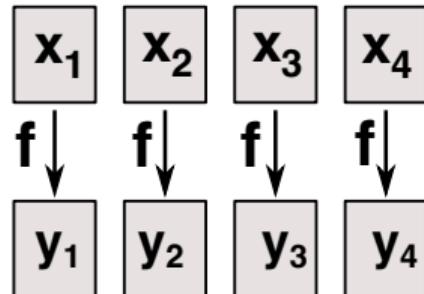
# Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
```

```
map f [] = []
```

```
map f (c:cs) = f c : map f cs
```



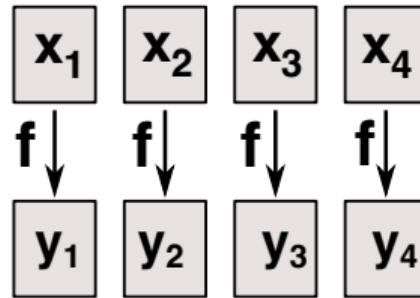
- ▶ Auswertung:

```
toL "AB" → map toLower ('A':'B':[])
→ toLower 'A': map toLower ('B':[])
```

# Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
map f []      = []
map f (c:cs) = f c : map f cs
```



- ▶ Auswertung:

```
toL "AB"  → map toLower ('A':'B':[])
                  → toLower 'A': map toLower ('B':[])
                  → 'a':map toLower ('B':[])
```

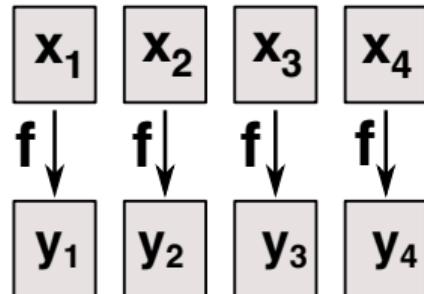
# Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
```

```
map f [] = []
```

```
map f (c:cs) = f c : map f cs
```



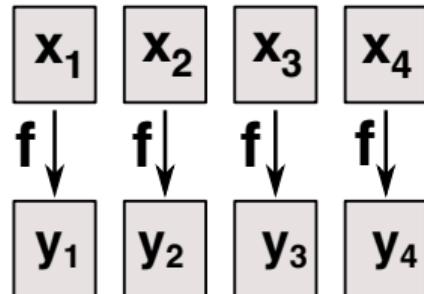
- ▶ Auswertung:

```
toL "AB" → map toLower ('A':'B':[])
      → toLower 'A': map toLower ('B':[])
      → 'a':map toLower ('B':[])
      → 'a':toLower 'B':map toLower []
```

# Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
map f []      = []
map f (c:cs) = f c : map f cs
```



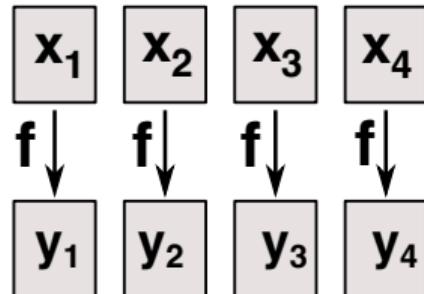
- ▶ Auswertung:

```
toL "AB"  → map toLower ('A':'B':[])
                → toLower 'A': map toLower ('B':[])
                → 'a':map toLower ('B':[])
                → 'a':toLower 'B':map toLower []
                → 'a':'b':map toLower []
```

# Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
map f []      = []
map f (c:cs) = f c : map f cs
```



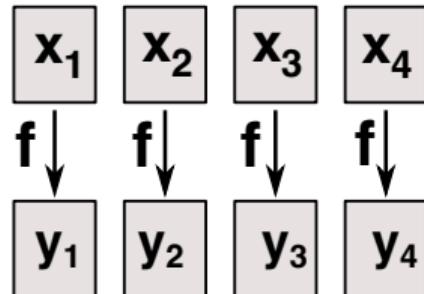
- ▶ Auswertung:

```
toL "AB"  → map toLower ('A':'B':[])
                → toLower 'A': map toLower ('B':[])
                → 'a':map toLower ('B':[])
                → 'a':toLower 'B':map toLower []
                → 'a':'b':map toLower []
                → 'a':'b':[]
```

# Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
map f []      = []
map f (c:cs) = f c : map f cs
```



- ▶ Auswertung:

```
toL "AB"  → map toLower ('A':'B':[])
                → toLower 'A': map toLower ('B':[])
                → 'a':map toLower ('B':[])
                → 'a':toLower 'B':map toLower []
                → 'a':'b':map toLower []
                → 'a':'b':[] ≡ "ab"
```

- ▶ **Funktionsausdrücke** werden **symbolisch** reduziert — keine Änderung der Auswertung

# Funktionen als Argumente: filter

- ▶ Elemente **filtern**: filter
- ▶ Signatur:

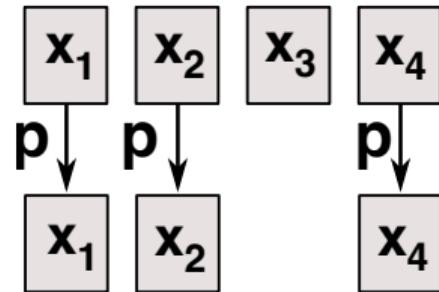
```
filter :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$   $[\alpha] \rightarrow [\alpha]$ 
```

- ▶ Definition

```
filter p []  = []
filter p (x:xs)
  | p x       = x: filter p xs
  | otherwise = filter p xs
```

- ▶ Beispiel:

```
digits :: String  $\rightarrow$  String
digits = filter isDigit
```



# Beispiel filter: Sieb des Erathostenes

- ▶ Für jede gefundene Primzahl  $p$  alle Vielfachen heraussieben:

```
sieve' :: [Integer] → [Integer]
sieve' [] = []
sieve' (p:ps) = p: sieve' (filterPs ps) where
    filterPs (q: qs)
        | q `mod` p ≠ 0 = q: filterPs qs
        | otherwise      = filterPs qs
```

- ▶ „Sieb“ filterPs: es werden alle  $q$  gefiltert mit  $mod\ q\ p \neq 0$

## Beispiel filter: Sieb des Erathostenes (1. Versuch)

- ▶ Es werden alle  $q$  **gefiltert** mit  $q \bmod p \neq 0$

```
siev3 :: [Integer] → [Integer]
siev3 [] = []
siev3 (p:ps) = p: siev3 (filter (filterMod p) ps) where
  filterMod p q = q `mod` p ≠ 0
```

- ▶ Damit Liste **aller** Primzahlen (NB: kleinste Primzahl ist 2):

```
prim3s :: [Integer]
prim3s = siev3 [2..]
```

## Beispiel filter: Sieb des Erathostenes (1. Versuch)

- ▶ Es werden alle  $q$  **gefiltert** mit  $mod\ q\ p \neq 0$

```
siev3 :: [Integer] → [Integer]
siev3 [] = []
siev3 (p:ps) = p: siev3 (filter (filterMod p) ps) where
  filterMod p q = q `mod` p ≠ 0
```

- ▶ Damit Liste **aller** Primzahlen (NB: kleinste Primzahl ist 2):

```
prim3s :: [Integer]
prim3s = siev3 [2..]
```

- ▶ **Unschön:** Definition der Hilfsfunktion **filterMod** wird uns „aufgezwungen“

## Beispiel filter: Sieb des Erathostenes

- ▶ Es werden alle  $q$  gefiltert mit  $q \bmod p \neq 0$
- ▶ Statt Definition einer **namenlose** (anonyme) Funktion  $\lambda q \rightarrow q \bmod p \neq 0$

```
sieve :: [Integer] → [Integer]
sieve [] = []
sieve (p:ps) = p: sieve (filter (λq → q `mod` p ≠ 0) ps)
```

- ▶ Damit Liste **aller** Primzahlen (NB: kleinste Primzahl ist 2):

```
primes :: [Integer]
primes = sieve [2..]
```

## Beispiel filter: Sieb des Erathostenes

- ▶ Es werden alle  $q$  gefiltert mit  $q \bmod p \neq 0$
- ▶ Statt Definition eine **namenlose** (anonyme) Funktion  $\lambda q \rightarrow q \bmod p \neq 0$

```
sieve :: [Integer] → [Integer]
sieve [] = []
sieve (p:ps) = p: sieve (filter (λq → q `mod` p ≠ 0) ps)
```

- ▶ Damit Liste **aller** Primzahlen (NB: kleinste Primzahl ist 2):

```
primes :: [Integer]
primes = sieve [2..]
```

- ▶ Primzahlzählfunktion  $\pi(n)$ :

```
pcf :: Integer → Int
pcf n = length (takeWhile (λm → m < n) primes)
```

Primzahltheorem:

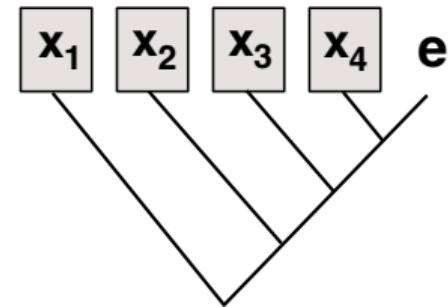
$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n / \log n} = 1$$

☞ Siehe Übung 5.2

# III. Strukturelle Rekursion

# Strukturelle Rekursion

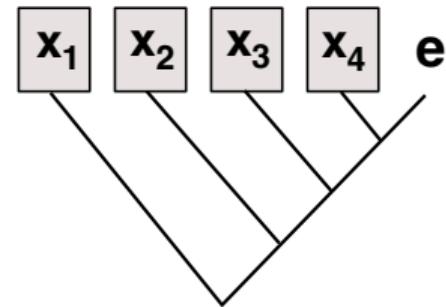
- ▶ **Strukturelle Rekursion:** gegeben durch
  - ▶ eine Gleichung für die leere Liste
  - ▶ eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)
- ▶ Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...
- ▶ Auswertung:



`sum [4,7,3]` →  
`concat [A, B, C]` →  
`length [4, 5, 6]` →

# Strukturelle Rekursion

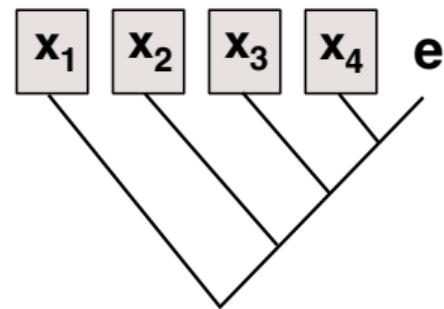
- ▶ **Strukturelle Rekursion:** gegeben durch
  - ▶ eine Gleichung für die leere Liste
  - ▶ eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)
- ▶ Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...
- ▶ Auswertung:



`sum [4,7,3]` →  $4 + 7 + 3 + 0$   
`concat [A, B, C]` →  
`length [4, 5, 6]` →

# Strukturelle Rekursion

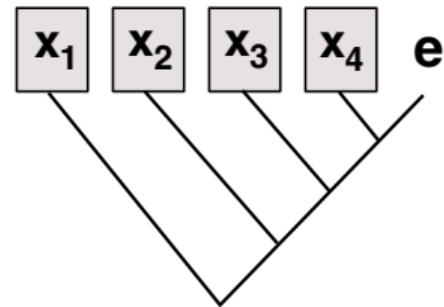
- ▶ **Strukturelle Rekursion:** gegeben durch
  - ▶ eine Gleichung für die leere Liste
  - ▶ eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)
- ▶ Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...
- ▶ Auswertung:



`sum [4,7,3]` → `4 + 7 + 3 + 0`  
`concat [A, B, C]` → `A ++ B ++ C ++ []`  
`length [4, 5, 6]` →

# Strukturelle Rekursion

- ▶ **Strukturelle Rekursion:** gegeben durch
  - ▶ eine Gleichung für die leere Liste
  - ▶ eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)
- ▶ Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...
- ▶ Auswertung:



`sum [4,7,3]` → `4 + 7 + 3 + 0`  
`concat [A, B, C]` → `A ++ B ++ C ++ []`  
`length [4, 5, 6]` → `1+ 1+ 1+ 0`

# Strukturelle Rekursion

## ► Allgemeines Muster:

$$\begin{aligned} f [] &= e \\ f (x:xs) &= x \otimes f \ xs \end{aligned}$$

## ► Parameter der Definition:

- Startwert (für die leere Liste)  $e :: \beta$
- Rekursionsfunktion  $\otimes :: \alpha \rightarrow \beta \rightarrow \beta$

## ► Auswertung:

$$f [x_1, \dots, x_n] = x_1 \otimes x_2 \otimes \dots \otimes x_n \otimes e$$

## ► **Terminiert** immer (wenn Liste endlich und $\otimes, e$ terminieren)

# Strukturelle Rekursion durch foldr

- ▶ **Strukturelle** Rekursion

- ▶ Basisfall: leere Liste
- ▶ Rekursionsfall: Kombination aus Listenkopf und Rekursionswert

- ▶ Signatur

```
foldr :: ( $\alpha \rightarrow \beta \rightarrow \beta$ )  $\rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$ 
```

- ▶ Definition

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

## Beispiele: foldr

- ▶ **Summieren** von Listenelementen.

```
sum  ::  [Int] → Int
sum xs = foldr (+) 0 xs
```

- ▶ **Flachklopfen** von Listen.

```
concat  ::  [[a]] → [a]
concat xs = foldr (++) [] xs
```

- ▶ **Länge** einer Liste

```
length  ::  [a] → Int
length xs = foldr (λx n → n + 1) 0 xs
```

## Beispiele: foldr

### ► Konjunktion einer Liste

```
and  ::  [Bool] → Bool
and xs = foldr (&&) True xs
```

### ► Konjunktion von Prädikaten

```
all  ::  ( $\alpha \rightarrow \text{Bool}$ ) →  $[\alpha]$  → Bool
all p xs = and (map p xs)
```

# Der Shoppe, revisited.

## ► Kasse alt:

```
kasse :: Einkaufskorb → Int
kasse (Ekwg ps) = kasse' ps where
  kasse' [] = 0
  kasse' (p: ps) = cent p + kasse' ps
```

## ► Kasse neu:

```
kasse' :: Einkaufskorb → Int
kasse' (Ek ps) = foldr (λp ps → cent p + ps) 0 ps
```

Besser:

```
kasse :: Einkaufskorb → Int
kasse (Ek ps) = sum (map cent ps)
```

# Der Shoppe, revisited.

- ▶ Inventur alt:

```
inventur :: Lager → Int
inventur (Lager ps) = inventur' ps where
  inventur' [] = 0
  inventur' (p: ps) = cent p + inventur' ps
```

- ▶ Suche nach einem Artikel neu:

```
inventur :: Lager → Int
inventur (Lager l) = sum (map cent l)
```

# Der Shoppe, revisited.

- ▶ Suche nach einem Artikel alt:

```
suche :: Artikel → Lager → Maybe Menge
suche art (Lager ps) = suche' art ps where
  suche' art (Posten lart  m: 1)
    | art == lart = Just m
    | otherwise    = suche' art l
  suche' art []    = Nothing
```

- ▶ Suche nach einem Artikel neu:

```
suche :: Artikel → Lager → Maybe Menge
suche a (Lager ps) =
  listToMaybe (map (λ(Posten _ m) → m)
    (filter (λ(Posten la _) → la == a) ps))
```

## Der Shoppe, revisited.

- ## ► Kassenbon formatieren [neu](#):

```
kassenbon :: Einkaufskorb → String
kassenbon ek@(Ek ps) =
  "Bob's Aulde Grocery Shoppe\n\n" ++
  "Artikel Menge Preis\n" ++
  "-----\n" ++
  concatMap artikel ps ++
  "\n-----\n" ++
  "Summe:" ++ formatR 31 (showEuro (kasse ek))
```

artikel :: Posten → String

# Iteration mit foldl

- foldr faltet von rechts:

$$\text{foldr } \otimes [x_1, \dots, x_n] \ e = x_1 \otimes x_2 \ (x_2 \otimes (\dots (x_n \otimes e)))$$

## Iteration mit foldl

- foldr faltet von rechts:

$$\text{foldr } \otimes [x_1, \dots, x_n] \ e = x_1 \otimes x_2 \ (x_2 \otimes (\dots (x_n \otimes e)))$$

- Warum nicht **andersherum**?

$$\text{foldl } \otimes [x_1, \dots, x_n] \ e = (((e \otimes x_1) \otimes x_2) \dots) \otimes x_n$$

# Iteration mit foldl

- foldr faltet von rechts:

$$\text{foldr } \otimes [x_1, \dots, x_n] \ e = x_1 \otimes x_2 \ (x_2 \otimes (\dots (x_n \otimes e)))$$

- Warum nicht **andersherum**?

$$\text{foldl } \otimes [x_1, \dots, x_n] \ e = (((e \otimes x_1) \otimes x_2) \dots) \otimes x_n$$

- Definition von foldl:

```
foldl :: ( $\alpha \rightarrow \beta \rightarrow \alpha$ )  $\rightarrow \alpha \rightarrow [\beta] \rightarrow \alpha$ 
```

```
foldl f a [] = a
```

```
foldl f a (x:xs) = foldl f (f a x) xs
```

- foldl ist ein **Iterator** mit Anfangszustand **e**, Iterationsfunktion  $\otimes$
- Entspricht einfacher Iteration (**for**-Schleife)

## Beispiel: rev

- ▶ Listen **umdrehen**:

```
rev1 :: [α] → [α]
rev1 []      = []
rev1 (x:xs) = rev1 xs ++ [x]
```

- ▶ Mit **foldr**:

```
rev2 :: [α] → [α]
rev2 = foldr (λx xs → xs ++ [x]) []
```

- ▶ Unbefriedigend: doppelte Rekursion  $O(n^2)$ !

## Beispiel: rev revisited

- ▶ Listenumkehr **endrekursiv**:

```
rev3 :: [α] → [α]
rev3 xs = rev0 xs [] where
  rev0 []      ys = ys
  rev0 (x:xs) ys = rev0 xs (x:ys)
```

- ▶ Listenumkehr durch falten **von links**:

```
rev4 :: [α] → [α]
rev4 = foldl (λxs x → x: xs) []
```

```
rev5 :: [α] → [α]
rev5 = foldl (flip (:)) []
```

- ▶ Nur noch **eine** Rekursion  $O(n)!$

## foldr vs. foldl

- $f = \text{foldr} \otimes e$  entspricht

$$f [] = e$$

$$f (x:xs) = x \otimes f \ xs$$

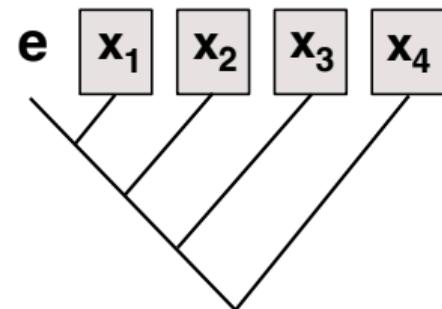
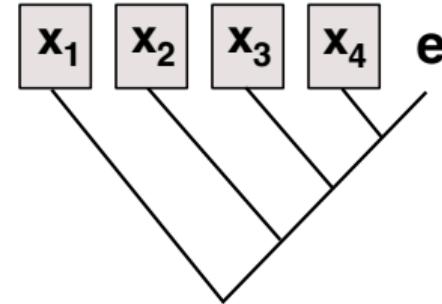
- **Nicht-strikt** in  $xs$ , z.B. `and`, `or`
- Konsumiert nicht immer die ganze Liste
- Auch für unendliche Listen anwendbar
- $f = \text{foldl} \otimes e$  entspricht

$$f \ xs = g \ e \ xs \text{ where}$$

$$g \ a [] = a$$

$$g \ a \ (x:xs) = g \ (a \otimes x) \ xs$$

- Effizient (endrekursiv) und **strikt** in  $xs$
- Konsumiert immer die ganze Liste
- Divergiert immer für unendliche Listen



# Wann ist $\text{foldl} = \text{foldr}$ ?

## Definition (Monoid)

$(\otimes, e)$  ist ein **Monoid** wenn

$$e \otimes x = x$$

(Neutrales Element links)

$$x \otimes e = x$$

(Neutrales Element rechts)

$$(x \otimes y) \otimes z = x \otimes (y \otimes z)$$

(Assoziativität)

## Theorem

Wenn  $(\otimes, e)$  **Monoid** und  $\otimes$  strikt, dann gilt für alle  $e, xs$

$$\text{foldl } \otimes e xs = \text{foldr } \otimes e xs$$

- ▶ Beispiele: `concat`, `sum`, `product`, `length`, `reverse`
- ▶ Gegenbeispiel: `all`, `any` (nicht-strikt)

# Übersicht: vordefinierte Funktionen auf Listen II

map	:: $(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$	— Auf alle Elemente anwenden
filter	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$	— Elemente filtern
foldr	:: $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$	— Falten von rechts
foldl	:: $(\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$	— Falten von links
mapConcat	:: $(\alpha \rightarrow [\beta]) \rightarrow [\alpha] \rightarrow [\beta]$	— map und concat
takeWhile	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$	— längster Prefix mit p
dropWhile	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$	— Rest von takeWhile
span	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow ([\alpha], [\alpha])$	— takeWhile und dropWhile
all	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow \text{Bool}$	— Argument gilt für alle
any	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow \text{Bool}$	— Argument gilt mind. einmal
elem	:: $(\text{Eq } \alpha) \Rightarrow \alpha \rightarrow [\alpha] \rightarrow \text{Bool}$	— Ist Element enthalten?
zipWith	:: $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow [\gamma]$	— verallgemeinertes zip

► Mehr: siehe `Data.List`

☞ Siehe Übung 5.3

# IV. Funktionen Höherer Ordnung

# Funktionen als Argumente: Funktionskomposition

## ► Funktionskomposition (mathematisch)

$(\circ) :: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$   
 $(f \circ g) \ x = f \ (g \ x)$

- Vordefiniert
- Lies:  $f$  nach  $g$
- Funktionskomposition **vorwärts**:

$(>.) :: (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$   
 $(f >.) g \ x = g \ (f \ x)$

- **Nicht** vordefiniert

## $\eta$ -Kontraktion

- ▶ “ $>.$ ” ist dasselbe wie
  - nur mit vertauschten Argumenten”
- ▶ Vertauschen der **Argumente** (vordefiniert):

```
flip :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow \beta \rightarrow \alpha \rightarrow \gamma$ 
```

```
flip f b a = f a b
```

## $\eta$ -Kontraktion

- “ $>.$ ” ist dasselbe wie
  - nur mit vertauschten Argumenten”
- Vertauschen der **Argumente** (vordefiniert):

```
flip :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow \beta \rightarrow \alpha \rightarrow \gamma$ 
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
( $>.$ ) :: ( $\alpha \rightarrow \beta$ )  $\rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$ 
(<.) = flip (o)
```

- **Da fehlt doch was?!**

## $\eta$ -Kontraktion

- “ $>.$ ” ist dasselbe wie  $\circ$  nur mit vertauschten Argumenten”
- Vertauschen der **Argumente** (vordefiniert):

```
flip :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow \beta \rightarrow \alpha \rightarrow \gamma$ 
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
( $>.$ ) :: ( $\alpha \rightarrow \beta$ )  $\rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$ 
( $>.$ ) = flip ( $\circ$ )
```

- **Da fehlt doch was?!** Nein:

$$(>.) f g a = \text{flip } (\circ) f g a \quad \equiv \quad (>.) = \text{flip } (\circ)$$

- Warum?  $\eta$ -Kontraktion

# Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

$$\alpha \rightarrow \beta \rightarrow \gamma \equiv \alpha \rightarrow (\beta \rightarrow \gamma)$$

- **Inbesondere**:  $(\alpha \rightarrow \beta) \rightarrow \gamma \neq \alpha \rightarrow (\beta \rightarrow \gamma)$
- Funktionsanwendung ist linksassoziativ:

$$f \ a \ b \equiv (f \ a) \ b$$

- **Inbesondere**:  $f \ (a \ b) \neq (f \ a) \ b$

# Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

$$\alpha \rightarrow \beta \rightarrow \gamma \equiv \alpha \rightarrow (\beta \rightarrow \gamma)$$

- **Inbesondere**:  $(\alpha \rightarrow \beta) \rightarrow \gamma \neq \alpha \rightarrow (\beta \rightarrow \gamma)$

- Funktionsanwendung ist linksassoziativ:

$$f \ a \ b \equiv (f \ a) \ b$$

- **Inbesondere**:  $f (a \ b) \neq (f \ a) \ b$
- **Partielle** Anwendung von Funktionen:

- Für  $f :: \alpha \rightarrow \beta \rightarrow \gamma$ ,  $x :: \alpha$  ist  $f \ x :: \beta \rightarrow \gamma$

- Beispiele:

- `map toLower :: String → String`
- `(3 ==) :: Int → Bool`
- `concat ∘ map (replicate 2) :: String → String`

# V. Andere Programmiersprachen

# Funktionen höherer Ordnung in C

- Implizit vorhanden: Funktionen = Zeiger auf Funktionen

```
extern list map(void *f(void *x), list l);
```

```
extern list filter(int f(void *x), list l);
```

- Keine direkte Syntax (e.g. namenlose Funktionen)
- Typsystem zu schwach (keine Polymorphie)
- Benutzung: `qsort` (C-Standard 7.20.5.2)

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

# Funktionen höherer Ordnung in Java

- **Java**: keine direkte Syntax für Funktionen höherer Ordnung
- Folgendes ist **nicht** möglich:

```
interface Collection {  
    Object fold(Object f(Object a, Collection c), Object a); }
```

- Aber folgendes:

```
interface Foldable { Object f (Object a); }  
  
interface Collection { Object fold(Foldable f, Object a); }
```

- Vergleiche **Iterator** aus Collections Framework (Java SE 6):

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next(); }
```

- Seit Java SE 8 (März 2014): Anonyme Funktionen (Lambda-Ausdrücke)

# Zusammenfassung

- ▶ Funktionen **höherer Ordnung**
  - ▶ Funktionen als **gleichberechtigte Objekte** und Argumente
  - ▶ Spezielle Funktionen höherer Ordnung: `map`, `filter`, `fold` und Freunde
- ▶ Formen der **Rekursion**:
  - ▶ Strukturelle Rekursion entspricht `foldr`
  - ▶ Iteration entspricht `foldl`
- ▶ Partielle Applikation,  $\eta$ -Äquivalenz, namenlose Funktionen
- ▶ Nächste Woche: Rekursive und zyklische Datenstrukturen