

# Praktische Informatik 3: Funktionale Programmierung

## Vorlesung 5 vom 30.11.2020: Funktionen Höherer Ordnung I

Christoph Lüth



Deutsches  
Forschungszentrum  
für Künstliche  
Intelligenz GmbH



Universität  
Bremen

Wintersemester 2020/21

## ▶ 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

# Ein 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 ...

# Ein 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
```

# Ein 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**?
  - ▶ Eine Funktion mit beliebigen Definitions- und Wertebereich:  $\alpha \rightarrow \beta$
- ▶ 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**?
  - ▶ Eine Funktion mit beliebigen Definitionsbereich und Wertebereich:  $\alpha \rightarrow \beta$
- ▶ Was ist der Typ des **zweiten Arguments**?
  - ▶ Eine Liste, auf deren Elemente die Funktion  $f$  angewandt wird:  $[\alpha]$
- ▶ 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**?
  - ▶ Eine Funktion mit beliebigen Definitions- und Wertebereich:  $\alpha \rightarrow \beta$
- ▶ Was ist der Typ des **zweiten Arguments**?
  - ▶ Eine Liste, auf deren Elemente die Funktion  $f$  angewandt wird:  $[\alpha]$
- ▶ Was ist der **Ergebnistyp**?
  - ▶ Eine Liste von Elementen aus dem Wertebereich von  $f$ :  $[\beta]$
- ▶ 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**?
  - ▶ Eine Funktion mit beliebigen Definitionsbereich und Wertebereich:  $\alpha \rightarrow \beta$
- ▶ Was ist der Typ des **zweiten Arguments**?
  - ▶ Eine Liste, auf deren Elemente die Funktion  $f$  angewandt wird:  $[\alpha]$
- ▶ Was ist der **Ergebnistyp**?
  - ▶ Eine Liste von Elementen aus dem Wertebereich von  $f$ :  $[\beta]$
- ▶ Alles **zusammengesetzt**:

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

# Was zum Selberdenken.

Die **konstante** Funktion ist

```
const ::  $\alpha \rightarrow \beta \rightarrow \alpha$   
const c _ = c
```

Übung 5.1: Was macht diese Funktion?

```
mystery xs = sum (map (const 1) xs)
```

# Was zum Selberdenken.

Die **konstante** Funktion ist

```
const ::  $\alpha \rightarrow \beta \rightarrow \alpha$   
const c _ = c
```

Übung 5.1: Was macht diese Funktion?

```
mystery xs = sum (map (const 1) xs)
```

Lösung: Betrachten wir eine Beispiel-Auswertung:

```
sum (map (const 1) []) ~> sum [] ~> 0  
sum (map (const 1) [True, False, True]) ~> sum [1,1,1] ~> 3  
sum (map (const 1) "foobaz") ~> sum ([1,1,1,1,1,1]) ~> 6
```

# Was zum Selberdenken.

Die **konstante** Funktion ist

```
const ::  $\alpha \rightarrow \beta \rightarrow \alpha$   
const c _ = c
```

Übung 5.1: Was macht diese Funktion?

```
mystery xs = sum (map (const 1) xs)
```

Lösung: Betrachten wir eine Beispiel-Auswertung:

```
sum (map (const 1) []) ~> sum [] ~> 0  
sum (map (const 1) [True, False, True]) ~> sum [1,1,1] ~> 3  
sum (map (const 1) "foobaz") ~> sum ([1,1,1,1,1,1]) ~> 6
```

Die mysteriöse Funktion berechnet die **Länge** der Liste `xs`!

## II. Map und Filter

# Funktionen als Argumente: map

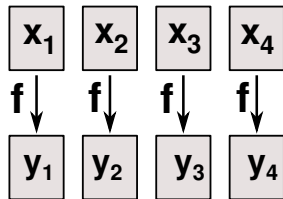
► `map` wendet Funktion auf alle Elemente an

► Signatur:

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

► Auswertung:

```
toL "AB"
```

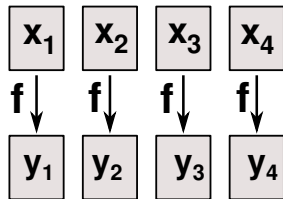


# Funktionen als Argumente: map

- ▶ `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

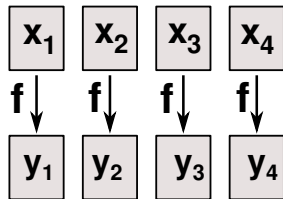
► `map` wendet Funktion auf alle Elemente an

► Signatur:

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

► Auswertung:

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





# Funktionen als Argumente: map

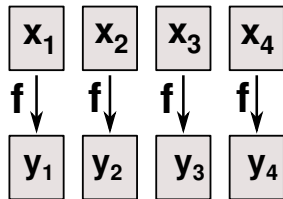
► `map` wendet Funktion auf alle Elemente an

► Signatur:

```
map :: (α → β) → [α] → [β]
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':[])
```



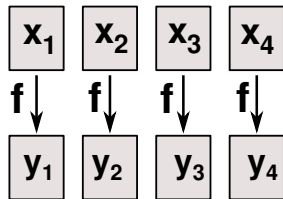
# Funktionen als Argumente: map

- ▶ `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"   $\rightarrow$  map toLower ('A':'B':[])  
            $\rightarrow$  toLower 'A': map toLower ('B':[])  
            $\rightarrow$  'a':map toLower ('B':[])  
            $\rightarrow$  'a':toLower 'B':map toLower []
```



# Funktionen als Argumente: map

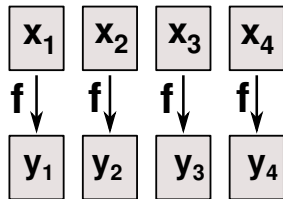
► `map` wendet Funktion auf alle Elemente an

► Signatur:

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

► Auswertung:

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



# Funktionen als Argumente: map

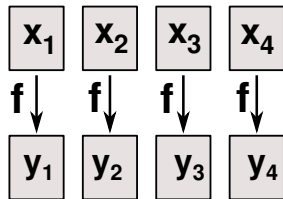
► `map` wendet Funktion auf alle Elemente an

► Signatur:

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

► Auswertung:

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

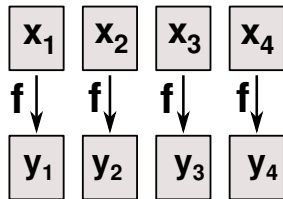


# Funktionen als Argumente: map

► `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"   $\rightarrow$  map toLower ('A':'B':[])  
            $\rightarrow$  toLower 'A': map toLower ('B':[])  
            $\rightarrow$  'a':map toLower ('B':[])  
            $\rightarrow$  'a':toLower 'B':map toLower []  
            $\rightarrow$  'a':'b':map toLower []  
            $\rightarrow$  'a':'b':[]  $\equiv$  "ab"
```

► **Funktionsausdrücke** werden symbolisch reduziert

► Keine Änderung

# Funktionen als Argumente: filter

- ▶ Elemente **filtern**: filter

- ▶ Signatur:

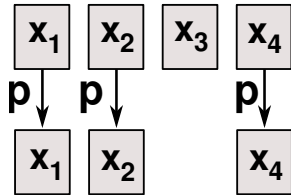
```
filter :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]
```

- ▶ 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 heraus sieve:

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

- „Sieb“: es werden alle  $q$  gefiltert mit  $\text{mod } q \ p \neq 0$

## Beispiel filter: Sieb des Erathostenes

- ▶ Es werden alle  $q$  gefiltert mit  $\text{mod } q \ p \neq 0$
- ▶ **Namenlose** (anonyme) Funktion  $\lambda q \rightarrow \text{mod } q \ p \neq 0$

```
sieve :: [Integer] → [Integer]
sieve (p:ps) = p: sieve (filter ( $\lambda q \rightarrow \text{mod } q \ p \neq 0$ ) ps)
```

- ▶ Damit (NB: kleinste Primzahl ist 2):

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



# Beispiel filter: Sieb des Erathostenes

- ▶ Es werden alle  $q$  gefiltert mit  $\text{mod } q \ p \neq 0$
- ▶ **Namenlose** (anonyme) Funktion  $\lambda q \rightarrow \text{mod } q \ p \neq 0$

```
sieve :: [Integer] → [Integer]
sieve (p:ps) = p: sieve (filter ( $\lambda q \rightarrow q \text{ 'mod' } p \neq 0$ ) ps)
```

- ▶ Damit (NB: kleinste Primzahl ist 2):

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

- ▶ Primzahlzählfunktion  $\pi(n)$ :

```
pcf :: Integer → Int
pcf n = length (takeWhile ( $\lambda m \rightarrow m < n$ ) primes)
```

Primzahltheorem:

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



## Jetzt seid ihr dran...

Für das Palindrom hatten wir eine Funktion `clean` definiert:

```
clean :: String → String
clean [] = []
clean (s:xs) | isAlphaNum s = toLower s : clean xs
              | otherwise    = clean xs
```

### Übung 5.2: Clean refactored

Wie sieht `clean` mit `map` und `filter` (und **ohne Rekursion**) aus?

## Jetzt seid ihr dran...

Für das Palindrom hatten wir eine Funktion `clean` definiert:

```
clean :: String → String
clean [] = []
clean (s:xs) | isAlphaNum s = toLower s : clean xs
              | otherwise    = clean xs
```

### Übung 5.2: Clean refactored

Wie sieht `clean` mit `map` und `filter` (und **ohne Rekursion**) aus?

Lösung:

```
clean' :: String → String
clean' xs = map toLower (filter isAlphaNum xs)
```

# III. Strukturelle Rekursion

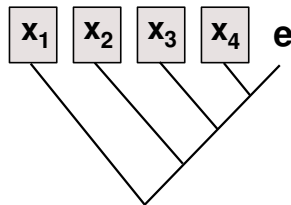
# Strukturelle Rekursion

- ▶ **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]`       $\rightarrow$

`concat [A, B, C]`       $\rightarrow$

`length [4, 5, 6]`       $\rightarrow$



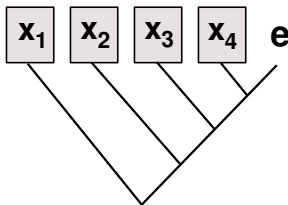
# Strukturelle Rekursion

- ▶ **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]`       $\rightarrow$     `4 + 7 + 3 + 0`

`concat [A, B, C]`     $\rightarrow$

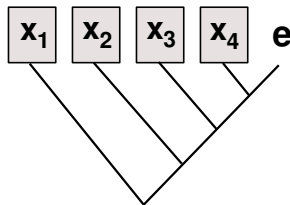
`length [4, 5, 6]`     $\rightarrow$



# 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:

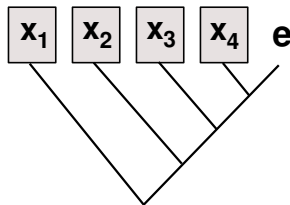
```
sum [4,7,3]      → 4 + 7 + 3 + 0
concat [A, B, C] → A ++ B ++ C ++ []
length [4, 5, 6] →
```



# 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:

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

## ► 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 ( $\lambda x\ n \rightarrow n + 1$ ) 0 xs
```

# Beispiele: foldr

## ► **Konjunktion** einer Liste

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

## ► **Konjunktion** von Prädikaten

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

# Der Shoppe, revisited.

## ► Kasse alt:

```
kasse :: Einkaufskorb → Int
kasse (Ekgw 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: l)
        | 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" ++
  "Summe:" ++ formatR 31 (showEuro (kasse ek))
```

```
artikel :: Posten → String
```



## Noch ein Beispiel: rev

- Listen **umdrehen**:

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

- Mit foldr:

```
rev2 :: [a] → [a]
rev2 = foldr (\x xs → xs ++ [x]) []
```

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

# 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 :: (α → β → α) → α → [β] → α
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 revisited

- ▶ Listenumkehr **endrekursiv**:

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

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

```
rev4 :: [a] → [a]
rev4 = foldl (\xs x → x: xs) []
```

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

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

## foldr vs. foldl

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

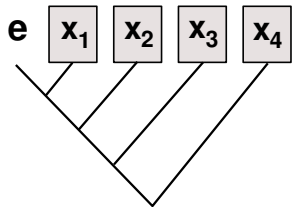
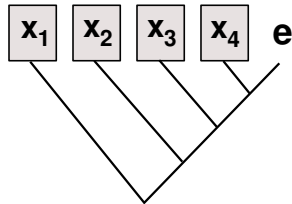
```
f []      = e
f (x:xs) = x  $\otimes$  f xs
```

- **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 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

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

► Mehr: siehe `Data.List`



# Jetzt seid ihr dran!

## Übung 5.3: elem selbstgemacht

Wie könnte die vordefinierte Funktion

`elem :: (Eq  $\alpha$ )  $\Rightarrow$   $\alpha \rightarrow [\alpha] \rightarrow \text{Bool}$`

definiert sein?

# Jetzt seid ihr dran!

## Übung 5.3: elem selbstgemacht

Wie könnte die vordefinierte Funktion

`elem :: (Eq  $\alpha$ )  $\Rightarrow$   $\alpha \rightarrow [\alpha] \rightarrow \text{Bool}$`

definiert sein?

Lösung: Eine Möglichkeit:

```
elem x xs = not (null (filter ( $\lambda y \rightarrow x == y$ ) xs))
```

# Jetzt seid ihr dran!

## Übung 5.3: elem selbstgemacht

Wie könnte die vordefinierte Funktion

`elem :: (Eq  $\alpha$ )  $\Rightarrow$   $\alpha \rightarrow [\alpha] \rightarrow \text{Bool}$`

definiert sein?

Lösung: Eine Möglichkeit:

```
elem x xs = not (null (filter ( $\lambda y \rightarrow x == y$ ) xs))
```

oder auch

```
elem x = not  $\circ$  null  $\circ$  filter (x ==)
```

# IV. Funktionen Höherer Ordnung

# Funktionen als Argumente: Funktionskomposition

## ► Funktionskomposition (mathematisch)

$$\begin{aligned}(\circ) &:: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma \\ (f \circ g) \ x &= f \ (g \ x)\end{aligned}$$

► Vordefiniert

► Lies:  $f$  nach  $g$

## ► Funktionskomposition **vorwärts**:

$$\begin{aligned}(>.) &:: (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma \\ (f > . g) \ x &= g \ (f \ x)\end{aligned}$$

► **Nicht** vordefiniert

## $\eta$ -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
```

## $\eta$ -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?!**

# $\eta$ -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 = flip ( $\circ$ ) f g a  $\equiv$  (>.>) = flip ( $\circ$ )
```

- ▶ Warum?



# $\eta$ -Äquivalenz und $\eta$ -Kontraktion

## $\eta$ -Äquivalenz

Sei  $f$  eine Funktion  $f : A \rightarrow B$ , dann gilt  $f = \lambda x. f\ x$

### ► In Haskell: $\eta$ -Kontraktion

- Bedingung: Ausdruck  $E :: \alpha \rightarrow \beta$ , Variable  $x :: \alpha$ ,  $E$  darf  $x$  nicht enthalten

$$\lambda x \rightarrow E\ x \quad \equiv \quad E$$

### ► Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$f\ x = E\ x \quad \equiv \quad f = E$$

### ► Hier:

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

# $\eta$ -Äquivalenz und $\eta$ -Kontraktion

## $\eta$ -Äquivalenz

Sei  $f$  eine Funktion  $f : A \rightarrow B$ , dann gilt  $f = \lambda x. f\ x$

### ► In Haskell: $\eta$ -Kontraktion

- Bedingung: Ausdruck  $E :: \alpha \rightarrow \beta$ , Variable  $x :: \alpha$ ,  $E$  darf  $x$  nicht enthalten

$$\lambda x \rightarrow E\ x \equiv E$$

### ► Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$f\ x = E\ x \equiv f = E$$

### ► Hier:

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

# $\eta$ -Äquivalenz und $\eta$ -Kontraktion

## $\eta$ -Äquivalenz

Sei  $f$  eine Funktion  $f : A \rightarrow B$ , dann gilt  $f = \lambda x. f\ x$

### ► In Haskell: $\eta$ -Kontraktion

- Bedingung: Ausdruck  $E :: \alpha \rightarrow \beta$ , Variable  $x :: \alpha$ ,  $E$  darf  $x$  nicht enthalten

$$\lambda x \rightarrow E\ x \equiv E$$

### ► Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$f\ x = E\ x \equiv f = E$$

### ► Hier:

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

# $\eta$ -Äquivalenz und $\eta$ -Kontraktion

## $\eta$ -Äquivalenz

Sei  $f$  eine Funktion  $f : A \rightarrow B$ , dann gilt  $f = \lambda x. f\ x$

### ► In Haskell: $\eta$ -Kontraktion

- Bedingung: Ausdruck  $E :: \alpha \rightarrow \beta$ , Variable  $x :: \alpha$ ,  $E$  darf  $x$  nicht enthalten

$$\lambda x \rightarrow E\ x \equiv E$$

### ► Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$f\ x = E\ x \equiv f = E$$

### ► Hier:

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

# 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`

# 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