Praktische Informatik 3: Funktionale Programmierung
Vorlesung 5 vom 15.11.2016: Funktionen Höherer Ordnung I

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II und Effizenzaspekte
- 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

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 Berechungsmuster
- Kontrollabstraktion

Ähnliche Datentypen der letzten Vorlesung

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

```
data Einkaufswagen = LeererWagen
                   | Einkauf Artikel Menge Einkaufswagen
```

```
data Path = Cons Id Path
          | Mt
```

```
data MyString = Empty
              | Cons Char MyString
```

- ein konstanter Konstruktor
- ein linear rekursiver Konstruktor

Ähnliche Datentypen der letzten Vorlesung

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

```
data Einkaufswagen = LeererWagen
                   | Einkaufswagen Artikel Menge Einkaufswagen
```

```
data Path
```

Gelöst durch Polymorphie

```
data MyString = Empty
              | Cons Char MyString
```

- ein konstanter Konstruktor
- ein linear rekursiver Konstruktor

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufswagen→ Int
kasse LeererWagen = 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
```

```
len :: MyString→ Int
len Empty        = 0
len (Cons c str) = 1+ len str
```

Gemeinsamkeiten:

- ein Fall pro Konstruktor
- linearer rekursiver Aufruf

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufswagen→ Int
kasse LeererWagen = 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
```

```
len :: MyString→ Int
len Empty        = 0
len (Cons c str) = 1+ len str
```

Gemeinsamkeiten:

- ein Fall pro Konstruktor
- linearer rekursiver Aufruf
- Nicht durch Polymorphie gelöst (keine Instanz einer Definition)

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 : toL 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 : toL 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 : toL 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 Definitions- und Wertebereich: $\alpha \rightarrow \beta$
- Was ist der Typ des zweiten Arguments?
  - Eine Liste, auf deren Elemente die Funktion f angewant 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 angewant 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 Definitions- und Wertebereich: $\alpha \rightarrow \beta$
- Was ist der Typ des zweiten Arguments?
  - Eine Liste, auf deren Elemente die Funktion f angewant wird: $[\alpha]$
- Was ist der Ergebnistyp?
  - Eine Liste von Elementen aus dem Wertebereich von f: $[\beta]$
- Alles zusammengesetzt:

```
map :: (α → β) → [α] → [β]
```

Map und Filter

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
map f []     = []
map f (x:xs) = f x : map f xs
```

- Auswertung:
  toL "AB"

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
map f []     = []
map f (x:xs) = f x : map f xs
```

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

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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 :: (α→ β)→ [α]→ [β]
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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 :: (α→ β)→ [α]→ [β]
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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 :: (α→ β)→ [α]→ [β]
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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 :: (α→ β)→ [α]→ [β]
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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 :: (α→ β)→ [α]→ [β]
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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

Funktionen als Argumente: `filter`

- Elemente filtern: `filter`
- Signatur:

```
filter :: (α→ Bool)→ [α]→ [α]
```

- Definition

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

- Beispiel:

```
letters :: String→ String
letters = filter isAlpha
```

Beispiel `filter`: Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben

Beispiel `filter`: Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben
  - Dazu: `filter` $(\lambda$`q`$\rightarrow$ `mod q p` $\neq 0)$ `ps`
  - Namenlose (anonyme) Funktion

Beispiel `filter`: Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben
  - Dazu: `filter (λq→ mod q p ≠ 0) ps`
  - Namenlose (anonyme) Funktion

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

- Alle Primzahlen:

```
primes :: [Integer]
primes = sieve [2..]     — Wichtig: bei 2 anfangen!
```

Beispiel `filter`: Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben
  - Dazu: `filter (λq→ mod q p ≠ 0) ps`
  - Namenlose (anonyme) Funktion

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

- Alle Primzahlen:

```
primes :: [Integer]
primes = sieve [2..]     — Wichtig: bei 2 anfangen!
```

- Die ersten n Primzahlen:

```
n_primes :: Int→ [Integer]
n_primes n = take n primes
```

Funktionen Höherer Ordnung

Funktionen als Argumente: Funktionskomposition

- Funktionskomposition (mathematisch)

```
(∘) :: (β→ γ) → (α→ β)→ α→ γ
(f ∘ g) x = f (g x)
```

  - Vordefiniert
  - Lies: f nach g

- Funktionskomposition vorwärts:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(f >.> g) x = g (f x)
```

  - Nicht vordefiniert!

η-Kontraktion

- Vertauschen der Argumente (vordefiniert):

```
flip :: (α→ β→ γ)→ β→ α→ γ
flip f b a = f a b
```

η-Kontraktion

- Vertauschen der Argumente (vordefiniert):

```
flip :: (α→ β→ γ)→ β→ α→ γ
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(>.>) = flip (∘)
```

- Da fehlt doch was?!

η-Kontraktion

- Vertauschen der Argumente (vordefiniert):

```
flip :: (α→ β→ γ)→ β→ α→ γ
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(>.>) = flip (∘)
```

- Da fehlt doch was?! Nein:

  `(>.>) = flip (∘)` $\equiv$ `(>.>) f g a = flip (∘) f g a`

- Warum?

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

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

- In Haskell: η-Kontraktion
  - Bedingung: Ausdruck E :: $\alpha \to \beta$, Variable x :: α, E darf x nicht enthalten

$$\lambda x \to \ \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{E}$$

- Spezialfall Funktionsdefinition (punktfreie Notation)

$$\mathsf{f}\ \mathsf{x} = \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{f} = \mathsf{E}$$

- Hier:

```
(>.>) f g a = flip (∘) f g a  ≡  (>.>) f g a = flip (∘) f g a
```

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

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

- In Haskell: η-Kontraktion
  - Bedingung: Ausdruck E :: $\alpha \to \beta$, Variable x :: α, E darf x nicht enthalten

$$\lambda x \to \ \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{E}$$

- Spezialfall Funktionsdefinition (punktfreie Notation)

$$\mathsf{f}\ \mathsf{x} = \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{f} = \mathsf{E}$$

- Hier:

```
(>.>) f g a = flip (∘) f g a  ≡  (>.>) f g  = flip (∘) f g
```

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

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

- In Haskell: η-Kontraktion
  - Bedingung: Ausdruck E :: $\alpha \to \beta$, Variable x :: α, E darf x nicht enthalten

$$\lambda x \to \ \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{E}$$

- Spezialfall Funktionsdefinition (punktfreie Notation)

$$\mathsf{f}\ \mathsf{x} = \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{f} = \mathsf{E}$$

- Hier:

```
(>.>) f g a = flip (∘) f g a  ≡  (>.>) f     = flip (∘) f
```

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

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

- In Haskell: η-Kontraktion
  - Bedingung: Ausdruck E :: $\alpha \to \beta$, Variable x :: α, E darf x nicht enthalten

$$\lambda x \to \ \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{E}$$

- Spezialfall Funktionsdefinition (punktfreie Notation)

$$\mathsf{f}\ \mathsf{x} = \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{f} = \mathsf{E}$$

- Hier:

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

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

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

  - Inbesondere: $(\alpha \to \beta) \to \gamma \neq \alpha \to (\beta \to \gamma)$

- Funktionsanwendung ist linksassoziativ:

$$\mathsf{f\ a\ b} \equiv (\mathsf{f\ a})\ \mathsf{b}$$

  - Inbesondere: $\mathsf{f\ (a\ b)} \neq (\mathsf{f\ a})\ \mathsf{b}$

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

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

  - Inbesondere: $(\alpha \to \beta) \to \gamma \neq \alpha \to (\beta \to \gamma)$

- Funktionsanwendung ist linksassoziativ:

$$\mathsf{f\ a\ b} \equiv (\mathsf{f\ a})\ \mathsf{b}$$

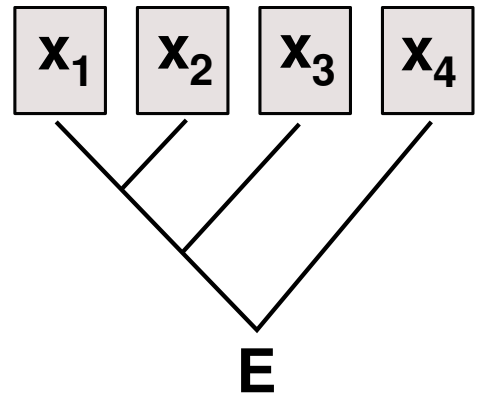
  - Inbesondere: $\mathsf{f\ (a\ b)} \neq (\mathsf{f\ a})\ \mathsf{b}$

- Partielle Anwendung von Funktionen:
  - Für $\mathsf{f} :: \alpha \to \beta \to \gamma$, $\mathsf{x} :: \alpha$ ist $\mathsf{f\ x} :: \beta \to \gamma$
- Beispiele:
  - $\mathsf{map\ toLower} :: \mathsf{String} \to \mathsf{String}$
  - $(3 ==) :: \mathsf{Int} \to \mathsf{Bool}$
  - $\mathsf{concat} \circ \mathsf{map\ (replicate\ 2)} :: \mathsf{String} \to \mathsf{String}$

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)



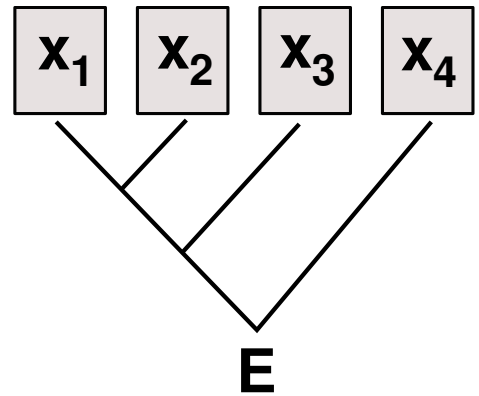
- Beispiel: kasse, inventur, sum, concat, length, $(+\!\!+)$, ...

- Auswertung:

$$\begin{aligned} &\text{sum}\ [4,7,3] && \rightarrow \\ &\text{concat}\ [A,\ B,\ C] && \rightarrow \\ &\text{length}\ [4,\ 5,\ 6] && \rightarrow \end{aligned}$$

Strukturelle Rekursion

- Strukturelle Rekursion: gegeben durch
  - eine Gleichung für die leere Liste
  - eine Gleichung für die nicht-leere Liste (mit einem rekursiven Aufruf)



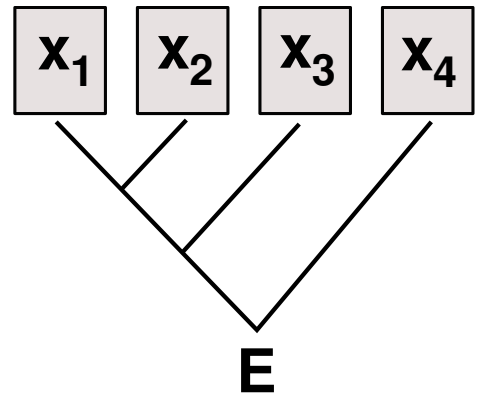
- Beispiel: kasse, inventur, sum, concat, length, $(+\!\!+)$, ...

- Auswertung:

$$\begin{array}{lll} \text{sum}\ [4,7,3] & \rightarrow & 4+7+3+0 \\ \text{concat}\ [A,\ B,\ C] & \rightarrow & \\ \text{length}\ [4,\ 5,\ 6] & \rightarrow & \end{array}$$

Strukturelle Rekursion

- Strukturelle Rekursion: gegeben durch
  - eine Gleichung für die leere Liste
  - eine Gleichung für die nicht-leere Liste (mit einem rekursiven Aufruf)



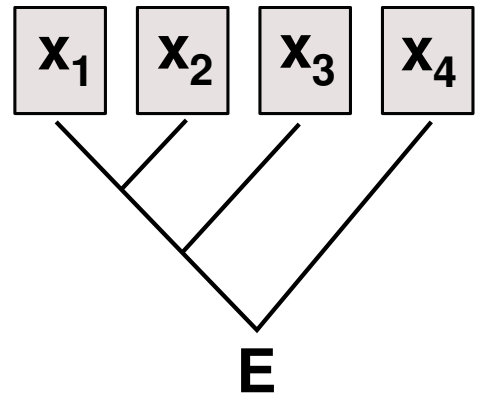
- Beispiel: kasse, inventur, sum, concat, length, $(+\!\!+)$, ...

- Auswertung:

$$
\begin{array}{lcl}
\text{sum } [4,7,3] & \rightarrow & 4 + 7 + 3 + 0 \\
\text{concat } [A,\ B,\ C] & \rightarrow & A +\!\!+ B +\!\!+ C +\!\!+ [\,] \\
\text{length } [4,\ 5,\ 6] & \rightarrow &
\end{array}
$$

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:

$$
\begin{array}{lcl}
\text{sum } [4,7,3] & \rightarrow & 4+7+3+0 \\
\text{concat } [A,\ B,\ C] & \rightarrow & A +\!\!+ B +\!\!+ C +\!\!+ [\,] \\
\text{length } [4,\ 5,\ 6] & \rightarrow & 1+1+1+0
\end{array}
$$

Strukturelle Rekursion

- Allgemeines Muster:

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

- Parameter der Definition:
  - Startwert (für die leere Liste) e :: β
  - Rekursionsfunktion $\otimes$ :: $\alpha \to \beta \to \beta$
- Auswertung:

$$f\ [x1, \ldots, xn] = x1 \otimes x2 \otimes \ldots \otimes xn \otimes e$$

- Terminiert immer (wenn Liste endlich und $\otimes, e$ terminieren)

Strukturelle Rekursion durch `foldr`

- Stukturelle Rekursion
  - Basisfall: leere Liste
  - Rekursionsfall: Kombination aus Listenkopf und Rekursionswert
- Signatur

```
foldr :: (α → β → β) → β → [α] → β
```

- 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 :: (α→ Bool)→ [α] → Bool
all p = and ∘ map p
```

Der Shoppe, revisited.

- Suche nach einem Artikel alt:

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

- Suche nach einem Artikel neu:

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

Der Shoppe, revisited.

- Kasse alt:

```
kasse :: Einkaufswagen → Int
kasse (Einkaufswagen []) = 0
kasse (Einkaufswagen (p: e)) = cent p + kasse (Einkaufswagen e)
```

- Kasse neu:

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

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

Der Shoppe, revisited.

- Kassenbon formatieren neu:

```
kassenbon :: Einkaufswagen→ String
kassenbon ew@(Einkaufswagen as) =
  "Bob's␣Aulde␣Grocery␣Shoppe\n\n"++
  "Artikel␣␣␣␣␣␣␣␣␣␣␣␣␣␣Menge␣␣␣␣␣␣Preis\n"++
  "————————————————————————————————————\n"++
  concatMap artikel as ++
  "====================================\n"++
  "Summe:"++ formatR 31 (showEuro (kasse ew))
```

```
artikel :: Posten→ String
artikel p@(Posten a m) =
   formatL 20 (show a) ++
   formatR 7  (menge m) ++
   formatR 10 (showEuro (cent p)) ++ "\n"
```

Noch ein Beispiel: rev

- Listen umdrehen:

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

- Mit fold:

```
rev' = foldr snoc []
```

```
snoc :: α → [α] → [α]
snoc x xs = xs ++ [x]
```

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

Iteration mit foldl

- foldr faltet von rechts:
  $$\texttt{foldr}\ \otimes\ [x1, \ldots, xn]\ e = x1 \otimes x2\ (x2 \otimes (\ldots (xn \otimes e)))$$
- Warum nicht andersherum?
  $$\texttt{foldl}\ \otimes\ [x1, \ldots, xn]\ e = (((e \otimes x1) \otimes x2) \ldots) \otimes xn$$
- 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 durch falten von links:

```
rev' xs = foldl (flip (:)) [] xs
```

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

foldr vs. foldl

- f = foldr ⊗ e entspricht

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

  - Kann nicht-strikt in xs sein, z.B. and, or
  - Konsumiert nicht immer die ganze Liste
  - Auch für zyklische Listen anwendbar

- f = foldl ⊗ e entspricht

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

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

Wann ist foldl = foldr?

Definition (Monoid)

$(\otimes, A)$ ist ein Monoid wenn

$$A \otimes x = x \qquad \text{(Neutrales Element links)}$$
$$x \otimes A = x \qquad \text{(Neutrales Element rechts)}$$
$$(x \otimes y) \otimes z = x \otimes (y \otimes z) \qquad \text{(Assoziativät)}$$

Theorem

Wenn $(\otimes, A)$ Monoid, dann für alle A, xs

$$foldl \otimes A\ xs = foldr \otimes A\ xs$$

- Beispiele: `length`, `concat`, `sum`
- Gegenbeispiele: `rev`, `all`

Übersicht: vordefinierte Funktionen auf Listen II

```
map       :: (α→ β)→ [α]→ [β]              — Auf alle anwenden
filter    :: (α→ Bool)→ [α]→ [α]           — Elemente filtern
foldr     :: (α→ β→ β)→ β→ [α]→ β          — Falten von rechts
foldl     :: (β→ α→ β)→ β→ [α]→ β          — Falten von links
mapConcat :: (α→ [β])→ [α]→ [β]            — map und concat
takeWhile :: (α→ Bool)→ [α]→ [α]           — längster Prefix mit p
dropWhile :: (α→ Bool)→ [α]→ [α]           — Rest von takeWhile
span      :: (α→ Bool)→ [α]→ ([α], [α])    — takeWhile und dropWhile
all       :: (α → Bool) → [α] → Bool       — p gilt für alle
any       :: (α → Bool) → [α] → Bool       — p gilt mind. einmal
elem      :: (Eq α) ⇒ α → [α] → Bool       — Ist Element enthalten?
zipWith   :: (α → β → γ) → [α] → [β] → [γ]
                                           — verallgemeinertes zip
```

- Mehr: siehe Data.List

Funktionen Höherer Ordnung in anderen Sprachen

Funktionen Höherer Ordnung: 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)

Funktionen Höherer Ordnung: C

- Implizit vorhanden: Funktionen = Zeiger auf Funktionen

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

extern list map1(void *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: C

- Implementierung von map
- Rekursiv, erzeugt neue Liste:

```
list map1(void *f(void *x), list l)
{
  return l== NULL ?
         NULL : cons(f(l-> elem), map1(f, l-> next));
}
```

- Iterativ, Liste wird in-place geändert (Speicherleck):

```
list map2(void *f(void *x), list l)
{
  list c;
  for (c= l; c!= NULL; c= c-> next) {
    c-> elem= f(c-> elem); }
  return l;
}
```

Zusammenfassung

- Funktionen höherer Ordnung
  - Funktionen als gleichberechtigte Objekte und Argumente
  - Partielle Applikation, η-Kontraktion, namenlose Funktionen
  - Spezielle Funktionen höherer Ordnung: map, `filter`, `fold` und Freunde
- Formen der Rekursion:
  - Strukturelle Rekursion entspricht `foldr`
  - Iteration entspricht `foldl`
- Nächste Woche: `fold` für andere Datentypen, Effizienz