

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 7 vom 14.12.2020: Funktionen Höherer Ordnung II — Jenseits der Liste

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

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

Heute

- ▶ Mehr über `map` und `fold`
- ▶ `map` und `fold` sind nicht nur für Listen
- ▶ Funktionen höherer Ordnung in anderen Programmiersprachen

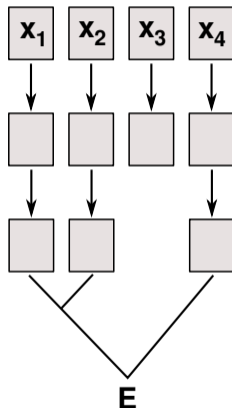
Lernziel

Wir verstehen, warum `map` und `fold` besonders sind, wie sie für andere Datentypen aussehen, und wann wir sie benutzen können.

I. Berechnungsmuster

map und filter als Berechnungsmuster

- ▶ `map`, `filter`, `fold` als Berechnungsmuster:
 - 1 Anwenden einer Funktion auf **jedes** Element der Liste
 - 2 möglicherweise **Filtern** bestimmter Elemente
 - 3 **Kombination** der Ergebnisse zu Endergebnis E
- ▶ Gut parallelisierbar, skalierbar
- ▶ Berechnungsmuster für große Datenmengen
 - ▶ Map/Reduce (Google), Hadoop



Listenkomprehension

- ▶ Besondere Notation: Listenkomprehension

```
[ f x | x ← as, g x ] ≡ map f (filter g as)
```

- ▶ Beispiel:

- ▶ Remember this?

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

- ▶ Sieht so besser aus:

```
suche :: Artikel → Lager → Maybe Menge
suche a (Lager ps) = listToMaybe [ m | Posten la m ← ps, la == a ]
```

Listenkomprehension mit mehreren Generatoren

- ▶ Anderes Beispiel: Primzahlzwillinge

```
twin_primes :: [(Integer, Integer)]  
twin_primes = [(x, y) | (x, y) ← zip primes (tail primes), x+2 == y]
```

- ▶ Mit mehreren Generatoren werden **alle Kombinationen** generiert:

```
idx :: [String]  
idx = [ a: show i | a ← ['a'.. 'z'], i ← [0.. 9]]
```

Beispiel I: Quicksort

- ▶ Quicksort per Listenkomprehension:

```
qsort1 :: Ord a => [a] -> [a]
qsort1 [] = []
qsort1 xs@(x:_) = qsort1 [y | y <- xs, y < x ] ++
                  [x0 | x0 <- xs, x0 == x ] ++
                  qsort1 [z | z <- xs, z > x ]
```

- ▶ Erstaunlich effizient



- ▶ Einfache Rekursion mit 3-Weg-Split nicht wesentlich effizienter, aber wesentlich länger

Beispiel I: Quicksort

- ▶ Quicksort per Listenkomprehension:

```
qsort1 :: Ord a => [a] -> [a]
qsort1 [] = []
qsort1 xs@(x:_) = qsort1 [y | y <- xs, y < x] ++
                  [x0 | x0 <- xs, x0 == x] ++
                  qsort1 [z | z <- xs, z > x]
```

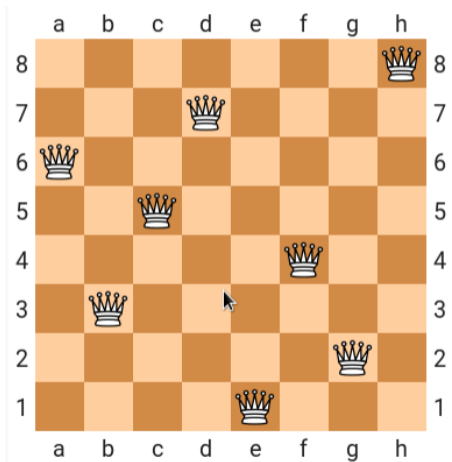
- ▶ Erstaunlich effizient



- ▶ Einfache Rekursion mit 3-Weg-Split nicht wesentlich effizienter, aber wesentlich länger
- ▶ Grund: Sortierte Liste wird nicht im ganzen aufgebaut

Beispiel II: 8-Damen-Problem

- Problem: Platziere 8 Damen sicher auf einem Schachbrett



Source: wikipedia

Beispiel II: n-Damen-Problem

- ▶ Position der Königinnen:

```
type Pos = (Int, Int)
type Board = [Pos]
```

- ▶ Rekursiv: Lösung für $n - 1$ Königinnen, n -te sicher dazu positionieren

```
queens :: Int → [Board]
queens n = qu n where
  qu :: Int → [Board]
  qu i | i == 0    = [[]] — Nicht [] !
       | otherwise = [ p++ [(i, j)] | p ← qu (i-1), j ← [1.. n],
                                     safe p (i, j)]
```

- ▶ Invariante: n -te Königin in n -ter Spalte

Beispiel II: n-Damen-Problem

- ▶ Wann ist eine Königin sicher?

```
safe :: Board → Pos → Bool
safe others nu = and [ not (threatens other nu) | other ← others ]
```

- ▶ Bedrohung: gleiche Zeile oder Diagonale

```
threatens :: Pos → Pos → Bool
threatens (i, j) (m, n) = (j == n) || (i+j == m+n) || (i-j == m-n)
```

- ▶ Diagonalen charakterisiert durch $y = a + x$ bzw. $y = a - x$ für konstantes a
- ▶ Gleiche Spalte ($i = m$) durch Konstruktion ausgeschlossen



Was zum Nachdenken

```
queens :: Int → [Board]
queens n = qu n where
  qu :: Int → [Board]
  qu i | i == 0    = [[]] — Nicht [] !
       | otherwise = [ p++ [(i, j)] | p ← qu (i-1), j ← [1.. n],
                                     safe p (i, j)]
```

Übung 7.1: Warum?

Wieso ist dort [[]] so wichtig? Was passiert, wenn wir [] zurückgeben?

Was zum Nachdenken

```
queens :: Int → [Board]
queens n = qu n where
  qu :: Int → [Board]
  qu i | i == 0    = [[]] — Nicht [] !
       | otherwise = [ p++ [(i, j)] | p ← qu (i-1), j ← [1.. n],
                                     safe p (i, j)]
```

Übung 7.1: Warum?

Wieso ist dort [[]] so wichtig? Was passiert, wenn wir [] zurückgeben?

Lösung:

- ▶ Mit [] gibt es **keine** Lösung, mit [[]] gibt es **eine, leere** Lösung für $i = 0$.
- ▶ Mit [] gäbe es **nie** eine Lösung für **alle** i .

II. Map und Fold: Jenseits der Listen

map als strukturerhaltende Abbildung

map ist die kanonische **strukturerhaltende Abbildung**

- ▶ Für `map` gelten folgende Aussagen:

$$\text{map id} = \text{id}$$

$$\text{map f} \circ \text{map g} = \text{map (f} \circ \text{g)}$$

$$\text{length} \circ \text{map f} = \text{length}$$

- ▶ Was davon ist spezifisch für Listen?
- ▶ Wie können wir das verallgemeinern?

map als strukturerhaltende Abbildung

map ist die kanonische **strukturerhaltende Abbildung**

- ▶ Für `map` gelten folgende Aussagen:

$$\text{map id} = \text{id}$$

$$\text{map } f \circ \text{map } g = \text{map } (f \circ g)$$

$$\text{length} \circ \text{map } f = \text{length}$$

- ▶ Was davon ist spezifisch für Listen?
- ▶ Wie können wir das verallgemeinern?

→ Typklassen?

map als strukturerhaltende Abbildung

`map` ist die kanonische **strukturerhaltende Abbildung**

- ▶ Für `map` gelten folgende Aussagen:

$$\text{map id} = \text{id}$$

$$\text{map } f \circ \text{map } g = \text{map } (f \circ g)$$

$$\text{length} \circ \text{map } f = \text{length}$$

- ▶ Was davon ist spezifisch für Listen?
- ▶ Wie können wir das verallgemeinern?

→ Konstruktorklassen!

Funktoren

- ▶ **Konstruktorklassen** sind Typklassen für Typkonstruktoren.
- ▶ Die Konstruktorklasse `Functor` für alle Typen mit einer stukturerhaltenden Abbildung:

```
class Functor f where
  fmap :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  f  $\alpha \rightarrow$  f  $\beta$ 
```

- ▶ Es sollte gelten (kann nicht geprüft werden):

$$\text{fmap id} = \text{id}$$

$$\text{fmap } f \circ \text{fmap } g = \text{fmap } (f \circ g)$$

- ▶ Infix-Synonym $\langle \$ \rangle$ für `fmap`

Instanzen von Functor

- ▶ Listen sind eine Instanz von `Functor`, aber es gibt `map` und `fmap`

Instanzen von Functor

- ▶ Listen sind eine Instanz von `Functor`, aber es gibt `map` und `fmap`
- ▶ `Maybe` ist eine Instanz von `Functor`:

```
instance Functor Maybe where
  fmap f (Just a) = Just (f a)
  fmap f Nothing  = Nothing
```

- ▶ Propagiert `Nothing` — oft sehr nützlich

Instanzen von Functor

- ▶ Listen sind eine Instanz von `Functor`, aber es gibt `map` und `fmap`
- ▶ `Maybe` ist eine Instanz von `Functor`:

```
instance Functor Maybe where
  fmap f (Just a) = Just (f a)
  fmap f Nothing  = Nothing
```

- ▶ Propagiert `Nothing` — oft sehr nützlich
- ▶ Tupel sind Instanzen von `Functor` im **zweiten** Argument, bspw:

```
instance Functor (a, ) where
  fmap f (a, b) = (a, f b)
```

foldr ist kanonisch

`foldr` ist die **kanonische strukturell rekursive** Funktion.

- ▶ Alle strukturell rekursiven Funktionen sind als Instanz von `foldr` darstellbar
- ▶ Insbesondere auch `map` und `filter`:

```
map f = foldr ((:). f) []
```

```
filter p = foldr ( $\lambda a as \rightarrow$  if p a then a:as else as) []
```

- ▶ Jeder algebraischer Datentyp hat ein `foldr`
- ▶ Nicht als Strukturklasse darstellbar (wie `Functor` und `fmap`)
 - ▶ Anmerkung: Typklasse `Foldable` schränkt Signatur von `foldr` ein

fold für andere Datentypen

fold ist universell

Jeder algebraische Datentyp T hat genau ein `foldr`.

- ▶ Kanonische Signatur für T :
 - ▶ Pro Konstruktor C ein Funktionsargument f_C
 - ▶ Freie Typvariable β für T
- ▶ Kanonische Definition:
 - ▶ Pro Konstruktor C eine Gleichung
 - ▶ Gleichung wendet f_C auf Argumente an (und `fold` rekursiv auf Argumente vom Typ T)

fold für andere Datentypen

- ▶ Beispiel:

```
data IL = Cons Int IL | Err String | Mt
```

- ▶ Das Fold dazu:

fold für andere Datentypen

- ▶ Beispiel:

```
data IL = Cons Int IL | Err String | Mt
```

- ▶ Das Fold dazu:

```
foldIL :: (Int → β → β) → (String → β) → β → IL → β  
foldIL f e a (Cons i il) = f i (foldIL f e a il)  
foldIL f e a (Err str)   = e str  
foldIL f e a Mt          = a
```

- ▶ Was ist das?

- ▶ Eine Art Listen von `Int` mit Fehlern („Ausnahmen“)
- ▶ Das zweite Argument von `foldIL` fängt aufgetretene Ausnahmen

fold für bekannte Datentypen

► Bool:

```
data Bool = False | True
```

```
foldBool ::  $\beta \rightarrow \beta \rightarrow$  Bool  $\rightarrow \beta$ 
```

```
foldBool a1 a2 False = a1
```

```
foldBool a1 a2 True  = a2
```

fold für bekannte Datentypen

- ▶ Bool: Fallunterscheidung:

```
data Bool = False | True
```

```
foldBool ::  $\beta \rightarrow \beta \rightarrow \text{Bool} \rightarrow \beta$ 
```

```
foldBool a1 a2 False = a1
```

```
foldBool a1 a2 True  = a2
```

- ▶ Maybe α :

```
data Maybe  $\alpha$  = Nothing | Just  $\alpha$ 
```

```
foldMaybe ::  $\beta \rightarrow (\alpha \rightarrow \beta) \rightarrow \text{Maybe } \alpha \rightarrow \beta$ 
```

```
foldMaybe b f Nothing = b
```

```
foldMaybe b f (Just a) = f a
```

fold für bekannte Datentypen

- ▶ Bool: Fallunterscheidung:

```
data Bool = False | True
```

```
foldBool ::  $\beta \rightarrow \beta \rightarrow \text{Bool} \rightarrow \beta$ 
```

```
foldBool a1 a2 False = a1
```

```
foldBool a1 a2 True  = a2
```

- ▶ Maybe α : Auswertung

```
data Maybe  $\alpha$  = Nothing | Just  $\alpha$ 
```

```
foldMaybe ::  $\beta \rightarrow (\alpha \rightarrow \beta) \rightarrow \text{Maybe } \alpha \rightarrow \beta$ 
```

```
foldMaybe b f Nothing = b
```

```
foldMaybe b f (Just a) = f a
```

- ▶ Als `maybe` vordefiniert

fold für bekannte Datentypen

► Tupel:

```
data (α, β) = (α, β)
```

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

fold für bekannte Datentypen

- ▶ Tupel: die `uncurry`-Funktion

```
data (α, β) = (α, β)
```

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

- ▶ Dazu gehört die Funktion `curry` (beide vordefiniert):

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

- ▶ Die beiden sind **invers**:

$$\text{uncurry} \circ \text{curry} = \text{id} \quad \text{curry} \circ \text{uncurry} = \text{id}$$

fold für bekannte Datentypen

► Natürliche Zahlen:

```
data Nat = Zero | Succ Nat
```

```
foldNat ::  $\beta \rightarrow (\beta \rightarrow \beta) \rightarrow \text{Nat} \rightarrow \beta$ 
```

```
foldNat e f Zero = e
```

```
foldNat e f (Succ n) = f (foldNat e f n)
```


fold für bekannte Datentypen

- ▶ Natürliche Zahlen: Iterator

```
data Nat = Zero | Succ Nat
```

```
foldNat ::  $\beta \rightarrow (\beta \rightarrow \beta) \rightarrow \text{Nat} \rightarrow \beta$ 
```

```
foldNat e f Zero = e
```

```
foldNat e f (Succ n) = f (foldNat e f n)
```

- ▶ Wendet Funktion f n -mal auf Startwert e an:

$$\text{foldNat } e \ f \ n = f^n(e)$$

- ▶ Konversion nach `Int`:

```
natToInt :: Nat  $\rightarrow$  Int
```

```
natToInt = foldNat 0 (1+)
```

Kurze Denkpause

Übung 7.2: Merkwürdige Zahlen

Wenn wir die natürlichen Zahlen mit einem Typ-Parameter versehen:

```
data FNat  $\alpha$  = FZero | FSucc  $\alpha$  (FNat  $\alpha$ )
```

Was ist die kanonische Funktion `foldFNat`, und welcher Datentyp ist das?

Kurze Denkpause

Übung 7.2: Merkwürdige Zahlen

Wenn wir die natürlichen Zahlen mit einem Typ-Parameter versehen:

```
data FNat  $\alpha$  = FZero | FSucc  $\alpha$  (FNat  $\alpha$ )
```

Was ist die kanonische Funktion `foldFNat`, und welcher Datentyp ist das?

Lösung:

```
foldFNat ::  $\beta \rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \text{FNat } \alpha \rightarrow \beta$ 
```

```
foldFNat e f FZero = e
```

```
foldFNat e f (FSucc a n) = f a (foldFNat e f n)
```

Das sind natürlich Listen, mit `foldr`:

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

fold für binäre Bäume

- ▶ Binäre Bäume:

```
data Tree  $\alpha$  = Mt | Node  $\alpha$  (Tree  $\alpha$ ) (Tree  $\alpha$ )
```

- ▶ Label **nur** in den Knoten

fold für binäre Bäume

- ▶ Binäre Bäume:

```
data Tree α = Mt | Node α (Tree α) (Tree α)
```

- ▶ Label **nur** in den Knoten

- ▶ Instanz von `fold`:

```
foldT :: β → (α → β → β → β) → Tree α → β  
foldT e f Mt = e  
foldT e f (Node a l r) = f a (foldT e f l) (foldT e f r)
```

- ▶ Instanz von `Functor`, kein (offensichtliches) Filter

```
instance Functor Tree where  
  fmap f Mt = Mt  
  fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)
```

Funktionen mit foldT

- ▶ Höhe des Baumes berechnen:

```
height :: Tree  $\alpha$   $\rightarrow$  Int  
height = foldT 0 ( $\lambda$ _ l r  $\rightarrow$  1+ max l r)
```

- ▶ Inorder-Traversion der Knoten:

```
inorder :: Tree  $\alpha$   $\rightarrow$  [ $\alpha$ ]  
inorder = foldT [] ( $\lambda$ a l r  $\rightarrow$  l++ [a]++ r)
```

- ▶ Enthält der Baum dieses Element?

```
isElem :: Eq  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$  Tree  $\alpha$   $\rightarrow$  Bool  
isElem a = foldT False ( $\lambda$ b l r  $\rightarrow$  a == b || l || r)
```

- ▶ Nich-Striktheit von || begrenzt Traversal

Kanonische Eigenschaften von foldT und fmap

- ▶ Auch hier gilt:

$$\text{foldT Mt Node} = \text{id}$$

$$\text{fmap id} = \text{id}$$

$$\text{fmap } f \circ \text{fmap } g = \text{fmap } (f \circ g)$$

- ▶ Gilt für **alle** Datentypen. Insbesondere gilt:

$$\text{fold } C_1 C_2 \dots C_n = \text{id}$$

Falten mit den Konstruktoren ergibt die Identität.

Variadische Bäume

- ▶ Das Labyrinth ist ein variadischer Baum:

```
data VTree  $\alpha$  = NT  $\alpha$  [VTree  $\alpha$ ]
```

- ▶ Auch hierfür `fold` und `map`:

Variadische Bäume

- ▶ Das Labyrinth ist ein variadischer Baum:

```
data VTree  $\alpha$  = NT  $\alpha$  [VTree  $\alpha$ ]
```

- ▶ Auch hierfür `fold` und `map`:

```
foldT :: ( $\alpha \rightarrow [\beta] \rightarrow \beta$ )  $\rightarrow$  VTree  $\alpha \rightarrow \beta$   
foldT f (NT a ns) = f a (map (foldT f) ns)
```

```
instance Functor VTree where  
  fmap f (NT a ns) = NT (f a) (map (fmap f) ns)
```

Suche im Labyrinth

- ▶ Tiefensuche via `foldT`

```
dfs1 :: VTree  $\alpha$   $\rightarrow$  [Path  $\alpha$ ]  
dfs1 = foldT add where  
  add a [] = [[a]]  
  add a ps = [ a:p | p  $\leftarrow$  concat ps]
```

- ▶ Problem:

- ▶ `foldT` terminiert **nicht** für **zyklische** Strukturen



Suche im Labyrinth

- ▶ Tiefensuche via `foldT`

```
dfs2 :: Eq  $\alpha$   $\Rightarrow$  VTree  $\alpha$   $\rightarrow$  [Path  $\alpha$ ]  
dfs2 = foldT add where  
  add a [] = [[a]]  
  add a ps = [a:p | p  $\leftarrow$  concat ps, not (a 'elem' p) ]
```

- ▶ Problem:

- ▶ `foldT` terminiert **nicht** für **zyklische** Strukturen
- ▶ Auch nicht, wenn `add` prüft ob `a` schon enthalten ist
- ▶ Pfade werden vom **Ende** konstruiert



Grenzen von foldr

- ▶ `foldr` traversiert die gesamte Struktur, konstruiert Ergebnis von nicht-rekursiven Konstruktoren her
- ▶ Nicht-Striktheit erlaubt zyklische Strukturen, wenn **lokal** Abbruch der Rekursion möglich
 - ▶ Beispiel: `all = foldr (&&) True`
 - ▶ Gegenbeispiel: Tiefensuche in zyklischen Strukturen, Breitensuche
- ▶ `foldl` ist **nicht** generalisierbar
 - ▶ Warum?

Grenzen von foldr

- ▶ `foldr` traversiert die gesamte Struktur, konstruiert Ergebnis von nicht-rekursiven Konstruktoren her
- ▶ Nicht-Striktheit erlaubt zyklische Strukturen, wenn **lokal** Abbruch der Rekursion möglich
 - ▶ Beispiel: `all = foldr (&&) True`
 - ▶ Gegenbeispiel: Tiefensuche in zyklischen Strukturen, Breitensuche
- ▶ `foldl` ist **nicht** generalisierbar
 - ▶ Warum? Nur für **linear rekursive** Typen

Andere Arten der Rekursion

- ▶ Andere rekursive Struktur über Listen
 - ▶ Quicksort: baumartige Rekursion
- ▶ Rekursion nicht (nur) über Listenstruktur:
 - ▶ `take`: Begrenzung der Rekursion

```
take :: Int -> [a] -> [a]
take n _      | n <= 0 = []
take _ []     = []
take n (x:xs) = x : take (n-1) xs
```

- ▶ Version mit `fold` divergiert für nicht-endliche Listen

Kurzes Gehirnjogging

Übung 7.3:

Wie sieht die Version von `take` mit `fold` aus (`foldl` oder `foldr`)?

Kurzes Gehirnjogging

Übung 7.3:

Wie sieht die Version von `take` mit `fold` aus (`foldl` oder `foldr`)?

Lösung:

► Mit `foldl`:

```
take :: Int -> [a] -> [a]
take i = foldl (\p a -> if length p < i then (p++[a]) else p) []
```

► Mit `foldr` und `zip`:

```
takez' i = map snd o zip [1.. i]
```

Geschummelt weil `zip` nicht mit `fold` implementiert werden kann

III. Anhang: Datentypen in anderen Programmiersprachen

Andere Programmiersprachen

- ▶ C — systemnah, schnell
- ▶ Java — objektorientiert, Systemsprache
- ▶ Python — Skriptsprache

Datentypen in C

- ▶ **C**: Produkte, Aufzählungen, keine rekursiven Typen
- ▶ Rekursion **nur** durch **Zeiger**
- ▶ Konstruktoren **nutzerimplementiert**
- ▶ Manuelle Speicherverwaltung (`malloc/free`)

Datentypen in Java

- ▶ Nachbildung durch Klassen
- ▶ Datentyp ist abstrakte Klasse, Konstruktoren sind Unterklassen dieser Klasse
- ▶ Volle Speicherverwaltung (mit garbage collection)

Datentypen in Python

- ▶ **Listen** und **Tupel** fest eingebaut
- ▶ Diverse Funktionen auf Listen
 - ▶ Methoden (**stateful**) vs. Funktionen
 - ▶ Bsp. `sort` vs. `sorted`
- ▶ Definition eigener Typen über Klassen
- ▶ Volle Speicherverwaltung (mit garbage collection)

Polymorphie in C

- ▶ Polymorphie in C: `void *`
- ▶ Pointer-to-void ist kompatibel mit allen anderen Pointer-Typen.
- ▶ Manueller Typ-Cast nötig
 - ▶ Vergl. `Object` in Java
- ▶ Extrem Fehleranfällig

Polymorphie in Java

- ▶ Polymorphie in **Java**: Methode auf alle Subklassen anwendbar
 - ▶ Manuelle **Typkonversion** nötig, fehleranfällig
- ▶ Neu ab Java 1.5: **Generics**
 - ▶ Damit **parametrische Polymorphie** möglich
 - ▶ **Nachteil**: Benutzung umständlich, weil keine Typherleitung
 - ▶ **Vorteil**: Typkorrektheit sichergestellt:
 - ▶ Allerdings: Typ-Parameter nur für Klassen.

Ad-Hoc Polymorphie in Java

- ▶ `interface` und `abstract class`
- ▶ Flexibler in Java: beliebig viele Parameter etc.
- ▶ Eingeschränkt durch Vererbungshierarchie
- ▶ Ähnliche Standardklassen
 - ▶ `toString`
 - ▶ `equals` und `==`, keine abgeleitete strukturelle Gleichheit

Polymorphie in Python

- ▶ In Python werden Typen zur **Laufzeit** geprüft (**dynamic typing**)
- ▶ **duck typing**: strukturell gleiche Typen sind gleich
- ▶ Polymorphie durch Klassen
- ▶ Statt Interfaces kennt Python **Mixins**
 - ▶ Abstrakte Klassen ohne Oberklasse

Funktionen höherer Ordnung in C

- ▶ Implizit vorhanden: Funktionen = Zeiger auf Funktionen

```
extern list map1(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 nmem, 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)

Funktionen höherer Ordnung in Python

- ▶ Python kennt map, filter, fold:

```
letters = map(chr, range(97, 123))
```

- ▶ Map auf Iteratoren definiert, nicht auf Listen

- ▶ Python kennt Listenkomprehension:

```
idx = [ x+ str(i) for x in letters for i in range(10) ]
```

- ▶ Python kennt Lambda-Ausdrücke:

```
num = map (lambda x: 3*x+1, range (1,10))
```

Zusammenfassung

- ▶ Einige Funktionen höherer Ordnung sind speziell:
 - ▶ `map` ist die strukturerhaltende Funktion
 - ▶ `fold` ist die strukturelle Rekursion über dem Typen
- ▶ Jeder Datentyp hat `map` und `fold`
- ▶ Konstruktorklassen sind Klassen für Typkonstruktoren
 - ▶ Beispiel `Functor`
- ▶ Listenkomprehension ist ein nützlicher, leichtgewichtiger syntaktischer Zucker für `map` und `filter`