

---

## Heutiges Thema

Nachträge, Ergänzungen und Weiterführendes zu

- Ad hoc Polymorphie (Überladen)
  - Typklassen
  - Vererbung (Einfachvererbung, Mehrfachvererbung)
  - Überschreiben
  - Polymorphie vs. ad hoc Polymorphie
  - Vorteile für die Programmierung
- Listen, Muster und Funktionen
  - Listen und Muster
  - Listenkomprehension
  - Listenkonstruktoren vs. Listenoperatoren

---

## Rückblick: Typklassenbeispiel (1)

Wir hatten angenommen, an der Größe interessiert zu sein von

- Listen und
- Bäumen

...wobei der Begriff "Größe" typabhängig gedacht war, z.B.

- Anzahl der
  - Elemente
  - *Tupelkomponenten* (als zusätzliches Beispiel) bei (*Tupel*-) Listen
- Anzahl der
  - Knoten
  - Blätter
  - Benennungen
  - ...bei Bäumen

---

## Rückblick: Typklassenbeispiel (2)

- *Wunsch*: ...eine Funktion `size`, die mit Argumenten der verschiedenen Typen aufgerufen werden kann und das Gewünschte leistet.
- *Lösung*: ...ad hoc Polymorphie mittels Typklassen

---

## Rückblick: Typklassenbeispiel (3)

Wir betrachteten folgende Baumvarianten...

```
data Tree a = Nil |
             Node a (Tree a) (Tree a)

data Tree1 a b = Leaf1 b |
               Node1 a b (Tree1 a b) (Tree1 a b)

data Tree2 = Leaf2 String |
            Node2 String Tree2 Tree2
```

...und den Haskell-Standardtyp für Listen.

---

## Rückblick: Typklassenbeispiel (4)

Haskells Typklassenkonzept erlaubte uns folgende Lösung:

```
class Size a where          -- Definition der Typklasse Size
  size :: a -> Int

instance Size (Tree a) where -- Instanzbildung fuer (Tree a)
  size Nil      = 0
  size (Node n l r) = 1 + size l + size r

instance Size (Tree1 a b) where -- Instanzbildung fuer (Tree1 a b)
  size (Leaf1 m)      = 1
  size (Node1 m n l r) = 2 + size l + size r

instance Size Tree2 where    -- Instanzbildung fuer Tree2
  size (Leaf2 m)      = length m
  size (Node2 m l r) = length m + size l + size r

instance Size [a] where
  size = length
```

---

## Rückblick: Typklassenbeispiel (5)

Mit diesen Definitionen galt für den Typ der Funktion size:

```
size :: Size a => a -> Int
```

...und die Funktion size ermöglichte wie gewünscht:

```
size Nil => 0
size (Node "asdf" (Node "jk" Nil Nil) Nil) => 2

size (Leaf1 "adf") => 1
size (Node1 "asdf" 3
      (Node1 "jk" 2 (Leaf1 17) (Leaf1 4))
      (Leaf1 21)) => 7

size (Leaf2 "abc") => 3
size (Node2 "asdf"
      (Node2 "jkertt" (Leaf2 "abc") (Leaf2 "ac"))
      (Leaf2 "xy")) => 17

size [5,3,45,676,7] => 5
size [True,False,True] => 3
```

---

## Als zusätzliches Beispiel (1)

...sei für Tupellisten "Größe" nicht durch die Anzahl der Listenelemente, sondern durch die Anzahl der Komponenten der tupelförmigen Listenelemente gegeben.

Lösung durch entsprechende Instanzbildung:

```
instance Size [(a,b)] where
  size = (*2) . length

instance Size [(a,b,c)] where
  size = (*3) . length
```

---

## Als zusätzliches Beispiel (2)

Wie bisher gilt für den Typ der Funktion size:

```
size :: Size a => a -> Int
```

...und wir erhalten wie erwartet und gewünscht:

```
size [(5,"Smith"),(4,"Hall"),(7,"Douglas")] => 6
size [(5,"Smith",45),(4,"Hall",37),(7,"Douglas",42)] => 9
```

---

## Anmerkungen

Sprechweisen:

- `(*2)`, `(*3)` sind im Haskell-Jargon *operator sections*.
- `“.”` bezeichnet in Haskell die aus der Mathematik bekannte Funktionskomposition:

Sei  $f : B \rightarrow C$  und  $g : A \rightarrow B$ , dann ist die *Funktionskomposition*  $(f \circ g) : A \rightarrow C$  definiert durch:

$$\forall a \in A. (f \circ g)(a) =_{df} f(g(a))$$

Damit bedeutet z.B.

```
((*2) . length) [(5,"Smith"),(4,"Hall"),(7,"Douglas")]
```

dasselbe wie

```
(*2) (length [(5,"Smith"),(4,"Hall"),(7,"Douglas")])
```

---

## Wermutstropfen (1)

Die Instanzbildungen

```
instance Size [a] where
  size = length
```

```
instance Size [(a,b)] where
  size = (*2) . length
```

```
instance Size [(a,b,c)] where
  size = (*3) . length
```

sind nicht gleichzeitig möglich.

---

## Wermutstropfen (2)

*Problem:* Überlappende Typen!

```
ERROR "test.hs:45" - Overlapping instances for class "Size"
*** This instance   : Size [(a,b)]
*** Overlaps with  : Size [a]
*** Common instance : Size [(a,b)]
```

*Konsequenz:*

- Für Argumente von Instanzen des Typs `[(a,b)]` (und ebenso des Typs `[(a,b,c)]`) ist die Überladung des Operators `size` nicht mehr auflösbar
- Wünschenswert wäre:

```
instance Size [a] without [(b,c)],[(b,c,d)] where
  size = length
```

*Beachte:* In dieser Weise in Haskell nicht möglich.

---

## Zusammenfassendes

...über die Funktion `size` und die Typklasse `Size`:

- die Typklasse `Size` stellt die Typspezifikation der Funktion `size` zur Verfügung
- jede Instanz der Typklasse `Size` muss eine instanzspezifische Implementierung der Funktion `size` zur Verfügung stellen
- Im Ergebnis ist die Funktion `size` wie auch z.B. die in Haskell vordefinierten Operatoren `+`, `*`, `-`, etc., oder die Relatoren `=`, `>`, `>=`, etc. *überladen*
- Synonym für *Überladen* ist *ad hoc Polymorphie*

---

## Mehr zu Typklassen

Anders als die Typklasse `Size` können Typklassen auch

- Spezifikationen mehr als einer Funktion bereitstellen
- Standardimplementierungen (engl. *default implementations*) für (alle oder einige) dieser Funktionen bereitstellen
- von anderen Typklassen *erben*

In der Folge betrachten wir dies an ausgewählten Beispielen von in Haskell vordefinierten Typklassen...

---

## Die vordefinierte Typklasse Eq (1)

Die in Haskell vordefinierte Typklasse `Eq`:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x==y)
  x == y = not (x/=y)
```

Die Typklasse `Eq` stellt

- Typspezifikationen von zwei Wahrheitswertfunktionen
- zusammen mit je einer Standardimplementierung

bereit.

---

## Die vordefinierte Typklasse Eq (2)

*Beachte:*

- die Standardimplementierungen sind für sich allein nicht ausreichend, sondern stützen sich wechselseitig aufeinander ab.

Trotz dieser Unvollständigkeit ergibt sich als Vorteil:

- Bei Instanzbildungen reicht es, entweder eine Implementierung für `(==)` oder für `(/=)` anzugeben. Für den jeweils anderen Operator gilt dann die vordefinierte Standard- (default) Implementierung.
- Auch für beide Funktionen können bei der Instanzbildung Implementierungen angegeben werden. In diesem Fall werden beide Standardimplementierungen *überschrieben*.

*Übung:* Vgl. dies z.B. mit Schnittstellendefinitionen und Definitionen abstrakter Klassen in Java. Welche Gemeinsamkeiten/Unterschiede gibt es?

---

## Beispiele von Instanzbildungen der Typklasse Eq (1)

Am Beispiel des Typs der Wahrheitswerte:

```
instance Eq Bool where
  (==) True True   = True
  (==) False False = True
  (==) _ _        = False
```

*Beachte:* Der Ausdruck "Instanz" im Haskell-Jargon ist überladen!

- Bislang: Typ `T` ist Instanz eines Typs `U` (z.B. Typ `[Int]` ist Instanz des Typs `[a]`)
- Jetzt zusätzlich: Typ `T` ist Instanz einer (Typ-) Klasse `C` (z.B. Typ `Bool` ist Instanz der Typklasse `Eq`)

---

## Beispiele von Instanzbildungen der Typklasse Eq (2)

Am Beispiel eines Typs für Punkte in der (x,y)-Ebene:

```
data Point = Point (Float,Float)

instance Eq Point where
  (==) (Point x) (Point y) = (fst(x)==fst(y)) &&
                             (snd(x)==snd(y))
```

*Bemerkung:* Mit feingranularen Mustern lässt sich die Implementierung einfacher und transparenter realisieren:

```
instance Eq Point where
  (==) (Point (x,y)) (Point (u,v)) = (x==u) && (y==v)
```

*Beachte:* Typ- und Konstrukturname (Point!) dürfen übereinstimmen.

---

## Beispiele von Instanzbildungen der Typklasse Eq (3)

Auch selbstdefinierte Typen können zu Instanzen vordefinierter Typklassen gemacht werden. Z.B. folgender Baumtyp zur Instanz der Typklasse Eq:

```
data Tree = Nil |
           Node Int Tree Tree

instance Eq Tree where
  (==) Nil Nil = True
  (==) (Node m t1 t2) (Node n u1 u2) = (m == n) &&
                                       (t1 == u1) &&
                                       (t2 == u2)
  (==) _ _ = False
```

---

## Beispiele von Instanzbildungen der Typklasse Eq (4)

Das Vorgenannte gilt selbstverständlich auch für selbstdefinierte polymorphe Typen wie folgende Beispiele zeigen:

```
data Tree1 a = Leaf1 a |
             Node1 a (Tree1 a) (Tree1 a)

data Tree2 a b = Leaf2 b |
               Node2 a b (Tree2 a b) (Tree2 a b)
```

---

## Beispiele von Instanzbildungen der Typklasse Eq (5)

```
instance (Eq a) => Eq (Tree1 a) where
  (==) (Leaf1 s) (Leaf1 t) = (s == t)
  (==) (Node1 s t1 t2) (Node1 t u1 u2) = (s == t) &&
                                       (t1 == u1) &&
                                       (t2 == u2)
  (==) _ _ = False

instance (Eq a, Eq b) => Eq (Tree2 a b) where
  (==) (Leaf2 q) (Leaf2 s) = (q == s)
  (==) (Node2 p q t1 t2) (Node2 r s u1 u2) = (p == r) &&
                                       (q == s) &&
                                       (t1 == u1) &&
                                       (t2 == u2)
  (==) _ _ = False
```

*Beachte:* Damit die Anwendbarkeit des Relators (==) auf Werte von Knotenbenennungen gewährleistet ist, muss die Instanzen der Typvariablen a und b selbst schon als Instanz der Typklasse Eq vorausgesetzt sein.

---

## Einschub zu Sprechweisen

```
instance (Eq a) => Eq (Tree1 a) where
  (==) (Leaf1 s) (Leaf1 t)           = (s == t)
  (==) (Node1 s t1 t2) (Node1 t u1 u2) = (s == t) &&
                                          (t1 == u1) &&
                                          (t2 == u2)
  (==) _ _                           = False
```

*Sprechweisen und Vereinbarungen:*

- `Tree1 a` ist Instanz der (gehört zur) Typklasse `Eq`, wenn `a` zu dieser Klasse gehört.
- Der Teil links von `=>` heißt *Kontext*.
- Rechts von `=>` dürfen ausschließlich Basistypen (z.B. `Int`), Typkonstruktoren beinhaltende Typen (z.B. `Tree a`, `[...]`) oder auf ausgezeichnete Typvariablen angewandte Tupel-typen (z.B. `(a,b,c,d)`) stehen.

---

## Beispiele von Instanzbildungen der Typklasse Eq (6)

Instanzbildungen sind flexibel...

Abweichend von der vorher definierten Gleichheitsrelation auf Bäumen vom Typ `(Tree2 a b)`, hätten wir den Gleichheitstest etwa auch so festlegen können, dass die Markierungen vom Typ `a` in inneren Knoten für den Gleichheitstest irrelevant sind:

```
instance (Eq b) => Eq (Tree2 a b) where
  (==) (Leaf2 q) (Leaf2 s)           = (q == s)
  (==) (Node2 _ q t1 t2) (Node2 _ s u1 u2)
                                          = (q == s) &&
                                          (t1 == u1) &&
                                          (t2 == u2)
  (==) _ _                           = False
```

Beachte, dass für Instanzen des Typs `a` jetzt nicht mehr Mitgliedschaft in der Typklasse `Eq` gefordert werden muss.

---

## Zusammenfassendes über den Relator (==) und die Typklasse Eq

Der Vergleichsoperator `(==)` ist...

- überladen (synonym: ad hoc polymorph), nicht echt polymorph
- in Haskell als Operation in der Typklasse `Eq` vorgegeben.
- damit anwendbar auf Werte aller Typen, die Instanz von `Eq` sind
- viele Typen sind bereits vordefinierte Instanz von `Eq`, z.B. alle elementaren Typen, Tupel und Listen über elementaren Typen
- auch selbstdefinierte Typen können zu Instanzen von `Eq` gemacht werden

---

## Spezielle Frage

Ist es vorstellbar, jeden Typ zu einer Instanz der Typklasse `Eq` zu machen?

De facto hieße das, den Typ des Vergleichsoperators `(==)` von

```
(==) :: Eq a => a -> a -> Bool
```

zu

```
(==) :: a -> a -> Bool
```

zu verallgemeinern.

---

## Zur Antwort (1)

Nein!

Der Grund ist im Kern folgender:

Anders als z.B. die Länge einer Liste, die eine vom konkreten Listentyp unabhängige Eigenschaft ist und deshalb eine (echt) polymorphe Eigenschaft ist und eine entsprechende Implementierung erlaubt

```
length :: [a] -> Int           -- echt polymorph
length [] = 0
length (_:xs) = 1 + length xs
```

ist Gleichheit eine typabhängige Eigenschaft, die eine typspezifische Implementierung verlangt.

Beispiel:

- unsere typspezifischen Implementierungen des Gleichheitstests auf Bäumen

---

## Zur Antwort (4)

In Haskell erfordert eine Umsetzung Instanzbildungen der Art:

```
instance Eq (Int -> Int) where
  (==) f g = ...
```

```
instance Eq (Int -> Int -> Int) where
  (==) f g = ...
```

Können wir die "Punkte" so ersetzen, dass wir einen Gleichheitstest für alle Funktionen der Typen  $(Int \rightarrow Int)$  und  $(Int \rightarrow Int \rightarrow Int)$  haben?

---

## Zur Antwort (3)

Warum ist nicht mehr möglich?

Im Sinne von Funktionen als *first class citizens* in funktionalen Sprachen wäre ein Gleichheitstest auch auf Funktionen sicher höchst wünschenswert.

Z.B.:

```
(==) fac fib           => False
(==) (\x -> x+x) (\x -> 2*x) => True
(==) (+2) doubleInc   => True
```

---

## Zur Antwort (5)

Leider nein!

Zwar läßt sich für konkret vorgelegte Funktionen Gleichheit fallweise (algorithmisch) entscheiden, generell aber gilt folgendes aus der Theoretischen Informatik bekannte negative Ergebnis:

**Theorem**

*Gleichheit von Funktionen ist unentscheidbar.*

---

## Zur Antwort (6)

*Erinnerung:*

“Gleichheit von Funktionen ist unentscheidbar” heißt informell, dass...

- es gibt keinen Algorithmus, der für zwei beliebig vorgelegte Funktionen stets nach endlich vielen Schritten entscheidet, ob diese Funktionen gleich sind oder nicht.

*Machen Sie sich klar, dass daraus in der Tat nicht folgt, dass Gleichheit zweier Funktionen nie (in endlicher Zeit) entschieden werden kann.*

---

## Schlussfolgerung (1)

...anhand der Beobachtungen am Gleichheitstest (==):

- ...offenbar können Funktion bestimmter Funktionalität nicht für jeden Typ angegeben werden, insbesondere lässt sich nicht für jeden Typ eine Implementierung des Gleichheitsrelators (==) angeben, sondern nur für eine Teilmenge aller möglichen Typen.
- ...die Teilmenge der Typen, für die das für den Gleichheitsrelator möglich ist, bzw. eine Teilmenge davon, für die das in einem konkreten Haskell-Programm tatsächlich gemacht wird, ist im Haskell-Jargon eine *Sammlung* (engl. *collection*) von Typen, eine sog. *Typklasse*.

---

## Schlussfolgerung (2)

Auch wenn es schön wäre, eine (echt) polymorphe Implementierung von (==) zu haben zur Signatur

```
(==) :: a -> a -> Bool
```

und damit analog zur Funktion zur Längenbestimmung von Listen

```
length :: [a] -> Int
```

...ist eine Implementierung in dieser Allgemeinheit für (==) nicht möglich!

Die Typklassen, für die eine Implementierung von (==) angegeben werden kann, sind in Haskell in der Typklasse Eq zusammengefasst.

---

## Typklassen und Vererbung (1)

*Vererbung* auf Typklassenebene...

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min           :: a -> a -> a
  compare            :: a -> a -> Ordering
  x <= y             = (x<y) || (x==y)
  x > y              = y < x
  ...
  compare x y
    | x == y   = EQ
    | x <= y   = LT
    | otherwise = GT
```



---

## Typklassen und Vererbung (2)

- Die (wie Eq vordefinierte) Typklasse Ord erweitert die Klasse Eq.
- Jede Instanz der Typklasse Ord muss Implementierungen für alle Funktionen der Klassen Eq und Ord bereitstellen.

Beachte:

- Ord stellt wie Eq für einige Funktionen bereits Standardimplementierungen bereit.
- Bei der Instanzbildung für weitere Typen reicht es deshalb, Implementierungen der Relatoren (==) und (<) anzugeben.
- Durch Angabe instanzspezifischer Implementierungen bei der Instanzbildung können diese Standardimplementierungen aber auch nach Wunsch überschrieben werden.

---

## Typklassen und Vererbung (3)

Auch *Mehrfachvererbung* auf Typklassenebene ist möglich, wie Haskells vordefinierte Typklasse Num zeigt:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate       :: a -> a
  abs, signum  :: a -> a
  fromInteger  :: Integer -> a    -- Typkonversionsfunktion!
  fromInt      :: Int -> a       -- Typkonversionsfunktion!

  x - y        = x + negate y
  fromInt      = ...
```

- ...vgl. dies auch mit Vererbungskonzepten objektorientierter Sprachen!

---

## Typklassen und Vererbung (4)

Überschreiben ererbter Funktionen am Beispiel der Instanz Point der Typklasse Eq:

- *Vererbung*:  
Für die Instanzdeklaration von Point zur Klasse Eq
- ```
instance Eq Point where
  Point (x,y) == Point (w,z) = (x==w) && (y==z)
```

erbt Point folgende Implementierung des Ungleichheitstests (/=) aus der Klasse Eq:

```
Point x /= Point y = not (Point x == Point y)
```

- *Überschreiben*:  
Die ererbte (Standard-) Implementierung von (/=) kann überschrieben werden, z.B. wie unten durch eine (geringfügig) effizientere Variante:

```
instance Eq Point where
  Point (x,y) == Point (w,z) = (x==w) && (y==z)
  Point x /= Point y        = if x/=w then True else y/=z
```

---

## Typklassen und Vererbung (5)

(Automatisch) *abgeleitete Instanzen* von Typklassen...

```
data Spielfarbe = Kreuz | Pik | Herz | Karo
                deriving (Eq,Ord,Enum,Show,Read)

data Tree a = Nil |
            Node a (Tree a) (Tree a)
            deriving (Eq,Ord)
```

- Algebraische Typen können durch Angabe einer *deriving*-Klausel als Instanzen vordefinierter Klassen automatisch angelegt werden.
- Intuitiv ersetzt die Angabe der *deriving*-Klausel die Angabe einer *instance*-Klausel.

---

## Typklassen und Vererbung (6)

So ist

```
data Tree a = Nil |
             Node a (Tree a) (Tree a) deriving Eq
```

gleichbedeutend zu:

```
data Tree a = Nil |
             Node a (Tree a) (Tree a)

instance Eq a => Eq (Tree a) where
  (==) Nil Nil                = True
  (==) (Node m t1 t2) (Node n u1 u2)
    = (m == n)  &&
      (t1 == u1) &&
      (t2 == u2)

  (==) _ _                = False
```

(Vgl. auch Aufgabenblatt 6)

---

## Typklassen und Vererbung (7)

Analog ist

```
data Tree2 a b = Leaf2 b1 |
                Node2 a b (Tree2 a b) (Tree2 a b) deriving Eq
```

gleichbedeutend zu:

```
data Tree2 a b = Leaf2 b1 |
                Node2 a b (Tree2 a b) (Tree2 a b)

instance (Eq a, Eq b) => Eq (Tree2 a b) where
  (==) (Leaf2 q) (Leaf2 s)      = (q == s)
  (==) (Node2 p q t1 t1) (Node2 r s u1 u2)
    = (p == r)  &&
      (q == s)  &&
      (t1 == u1) &&
      (t2 == u2)

  (==) _ _                = False
```

---

## Typklassen und Vererbung (8)

Möchten Sie hingegen, Gleichheit wie folgt realisiert wissen

```
data Tree2 a b = Leaf2 b1 |
                Node2 a b (Tree2 a b) (Tree2 a b)

instance (Eq a, Eq b) => Eq (Tree2 a b) where
  (==) (Leaf2 q) (Leaf2 s)      = (q == s)
  (==) (Node2 _ q t1 t1) (Node2 _ s u1 u2)
    = (q == s)  &&
      (t1 == u1) &&
      (t2 == u2)

  (==) _ _                = False
```

geht an obiger Instanzdeklaration kein Weg vorbei.

---

## Typklassen und Vererbung (9)

Mehr zu Typklassen, alles zu Funktionen vordefinierter Typklassen und über Grenzen und Einschränkungen etwa automatischer Ableitbarkeit von Typklassen...

- ...in jedem guten Buch über Haskell!

---

## Zum Abschluss dieses Themas

*Polymorphie* und *Überladen* auf Funktionen bedeuten...

- vordergründig  
... ein Funktionsname kann auf Argumente unterschiedlichen Typs angewendet werden.
- präziser
  - *Polymorphe Funktionen...*
    - \* ...haben eine einzige Implementierung, die für alle (zugelassenen/abgedeckten) Typen arbeitet (Bsp.: `length :: [a] -> Int`)
  - *Überladene Funktionen...*
    - \* ...arbeiten für Instanzen einer Klasse von Typen mit einer für jede Instanz spezifischen Implementierung (Bsp.: `size :: Size a => a -> Int`)

---

## Vorteile des Überladens von Operatoren

- Ohne Überladen ginge es nicht ohne ausgezeichnete Namen für alle Operatoren.
- Das gälte auch für die bekannten arithmetischen Operatoren, so wären insbesondere Namen der Art `+Int`, `+Float`, `*Int`, `*Float`, etc. erforderlich.
- Deren zwangweiser Gebrauch wäre nicht nur ungewohnt und unschön, sondern in der täglichen Praxis auch lästig.
- Haskells Angebot, hier Abhilfe zu schaffen und Operatoren zu überladen, ist das Konzept der *Typklassen*.

*Zum Selbststudium:* Andere Sprachen wie z.B. ML und Opal gehen hier einen anderen Weg und bieten andere Konzepte.

---

## Zum Abschluss für heute...

...noch etwas ganz anderes!

- Listen, Muster und Funktionen
- Listenkomprehension *at work*

...i.w. handelt es sich dabei um Nachträge und Randbemerkungen ("Vermischtes aus Haskell").

---

## Zurück zu Listen, Mustern und Funktionen darauf (1)

Einige Beispiele...

```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs) = x + sum xs
```

```
head :: [a] -> a
head (x:_) = x
```

```
tail :: [a] -> [a]
tail (_:xs) = xs
```

```
null :: [a] -> Bool
null []      = True
null (_:_) = False
```

---

## Zurück zu Listen, Mustern und Funktionen darauf (2)

*Muster, Wild Cards und Typvariablen...*

```
sign :: Integer -> Integer
sign x
  | x > 0  = 1
  | x == 0 = 0
  | x < 0  = -1

takeFirst :: Integer -> [a] -> [a]
takeFirst m ys = case (m,ys) of
  (0,_) -> []
  (_,[]) -> []
  (n,x:xs) -> x : takeFirst (n - 1) xs

ifThenElse :: Bool -> a -> a -> a
ifThenElse c t e = case c of True -> t
                             False -> e
```

---

Funktionale Programmierung (WS 2005/2006) / 7. Teil (09.11.2006) 45

---

## Somit erhalten wir als Fortschreibung...

...des Musterbegriffs: Muster können sein...

*Bislang:*

- *Werte* (z.B. 0, 'c', True)  
...ein Argument "passt" auf das Muster, wenn es vom entsprechenden Wert ist.
- *Variablen* (z.B. n)  
...jedes Argument passt.
- *Wild card* "\_"  
...jedes Argument passt.

*Jetzt zusätzlich:*

- *Konstruktormuster* (hier über Listen; z.B. [], (p:ps))
  - Eine Liste passt auf das Muster [], wenn sie leer ist.
  - Eine Liste passt auf (p:ps), wenn sie nicht leer ist und der Listenkopf auf p und der Listenrest auf ps passt.

*Hinweis:* Im Fall von (p:ps) reicht es, dass die Argumentliste nicht leer ist.

---

## Oft sehr nützlich

...das sog. as-Muster (mit @ gelesen als "as"):

*Beispiel:*

```
listTransform :: [a] -> [a]
listTransform l@(x:xs) = (x : l) ++ xs
```

Zum Vergleich ohne as-Muster...

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

...weniger elegant und weniger gut lesbar.

---

Funktionale Programmierung (WS 2005/2006) / 7. Teil (09.11.2006) 47

---

## Ein weiteres Beispiel

Auch Funktionsdeklarationen der Form...

```
binom :: (Integer,Integer) -> Integer
binom (n,k)
  | k==0 || n==k = 1
  | otherwise    = binom (n-1,k-1) + binom (n-1,k)
```

...sind Beispiele *musterbasierter* Funktionsdefinitionen.

---

Funktionale Programmierung (WS 2005/2006) / 7. Teil (09.11.2006) 48

---

## Zum Vergleich...

...mit Standardselektoren ohne Muster:

```
binom :: (Integer,Integer) -> Integer
binom p
  | snd(p)==0 || snd(p)==fst(p) = 1
  | otherwise = binom (fst(p)-1,snd(p)-1)
                + binom (fst(p)-1,snd(p))
```

...offenbar auch hier weniger elegant und weniger gut lesbar.

---

## Schlussfolgerung zwischendurch

Musterbasierte Funktionsdefinitionen sind (i.a.)...

- elegant und
- führen zu knappen, gut lesbaren Spezifikationen.

*Zu beachten aber ist:* Musterbasierte Funktionsdefinitionen...

- können zu subtilen Fehlern führen und
- erschweren (oft) Programmänderungen/-weiterentwicklungen (bis hin zur Tortur (vgl. Pepper [4]): denke etwa an das Hinzukommen eines weiteren Parameters)

---

## Listen, Listenkonstruktoren, Listenoperatoren

Wir kennen den vordefinierten Listentyp `String`:

```
Type String = [Char]
```

...und Beispiele gültiger Werte des Typs `String`, etwa:

```
['h','e','l','l','o'] == "hello"
"hello" ++ "world" -> "hello world" (+: vordef. Konkatenationsop.)
```

Wir hatten aber auch gesehen, dass Elementtypen weit komplexer sein dürfen, bis hin zu Funktionen (Funktionen als "first class citizens"):

- Listen von Listen  
[[2,4,23,2,5],[3,4],[],[56,7,6,]] :: [ [Int] ]
- Listen von Paaren  
[(3.14,42.0),56.1,51.3),(1.12,2.22)] :: [ Point ]
- ...
- Listen von Funktionen  
[fac, fib, fun91] :: [ Integer -> Integer ]

Ist die Zulässigkeit von `[fac, fib, fun91] :: [ Integer -> Integer ]` bemerkenswert?

---

## Erinnerung aus der Mathematik...

Eine Funktion  $f$  ist ein Tripel  $(D, W, G)$  mit einer Definitionsmenge (-bereich)  $D$ , einer Wertemenge (-bereich)  $W$  und einer rechtseindeutigen Relation  $G$  mit  $G \subseteq D \times W$ , dem sog. Funktionsgraphen von  $f$ .

Mithin...

Funktionen sind spezielle Relationen; spezielle Teilmengen *eines kartesischen Produkts*

Damit intuitiv naheliegend...

Listen von Funktionen ... "Listen von Listen von Paaren"

*Schlagwort...*

Funktionen als *"first class citizens"*

---

## Listenkomprehension (1)

...ein für funktionale Programmiersprachen charakteristisches Ausdrucksmittel.

- Listenkomprehension

...ein einfaches Beispiel:

[3\*n | n <- list] steht kurz für [3,6,9,12], wobei list vom Wert [1,2,3,4] vorausgesetzt ist.

↪ Listenkomprehension ist ein sehr elegantes und ausdruckskräftiges Sprachkonstrukt!

---

## Listenkomprehension (2)

Weitere Anwendungsbeispiele:

...wobei lst = [1,2,4,7,8,11,12,42] vorausgesetzt ist:

a) [ square n | n <- lst ] ⇒ [1,4,16,49,64,121,144,1764]

b) [ n | n <- lst, isPowOfTwo n ] ⇒ [1,2,4,8]

c) [ n | n <- lst, isPowOfTwo n, n>=5 ] ⇒ [8]

d) [ isPrime n | n <- lst ] ⇒

[False,True,False,True,False,True,False,False]

---

## Listenkomprehension (3)

e) addCoordinates :: [Point] -> [Float]

addCoordinates pLst = [ x+y | (x,y)<-pLst, (x>0||y>0) ]

addCoordinates [(0.0,0.5),(3.14,17.4),(-1.5,-2.3)] ⇒  
[0.5,20.54]

f) allOdd :: [Integer] -> Bool

allOdd xs = ( [ x | x<-xs, isOdd x ] == xs )

allEven :: [Integer] -> Bool

allEven xs = ( [ x | x<-xs, isOdd x ] == [] )

---

## Listenkomprehension (4)

g) grabCapVowels :: String -> String

grabCapVowels s = [ c | c<-s, isCapVowel c ]

isCapVowel :: Char -> Bool

isCapVowel 'A' = True

isCapVowel 'E' = True

isCapVowel 'I' = True

isCapVowel 'O' = True

isCapVowel 'U' = True

isCapVowel c = False

---

## Listenkomprehension “at work” (1)

...am Beispiel von “Quicksort”.

*Aufgabe:* Sortiere eine Liste  $L$  ganzer Zahlen aufsteigend.

*Lösung* mittels Quicksort:

- *Teile:* Wähle ein Element  $l$  aus  $L$  und partitioniere  $L$  in zwei (möglicherweise leere) Teillisten  $L_1$  und  $L_2$  so, dass alle Elemente von  $L_1$  ( $L_2$ ) kleiner oder gleich (größer) dem Element  $l$  sind.
- *Herrsche:* Sortiere  $L_1$  und  $L_2$  mit Hilfe rekursiver Aufrufe von Quicksort.
- *Zusammenführen der Teilergebnisse:* Trivial, die Gesamtliste entsteht durch Konkatenation der sortierten Teillisten.

---

## Listenkomprehension “at work” (2)

QuickSort: Eine typische Realisierung in Haskell...

```
quickSort :: [Int] -> [Int]
```

```
quickSort [] = []
quickSort (x:xs) =
    quickSort [ y | y<-xs, y<=x ] ++
        [x] ++ quickSort [ y | y<-xs, y>x ]
```

Beachte: Funktionsanwendung bindet stärker als Listenkonstruktion. Deshalb Klammerung des Musters  $x:xs$  in `quickSort (x:xs) = ...`

---

## Listenkomprehension “at work” (3)

Zum Vergleich: Eine typische imperative Realisierung von QuickSort...

```
quickSort (L,low,high)
  if low < high
  then splitInd = partition(L,low,high)
       quickSort(L,low,splitInd-1)
       quickSort(L,splitInd+1,high) fi

partition (L,low,high)
  l = L[low]
  left = low
  for i=low+1 to high do
    if L[i] <= l then left = left+1
                          swap(L[i],L[left]) fi od
  swap(L[low],L[left])
  return left
```

...und dem initialen Aufruf `quickSort(L,1,length(L))`.

---

## Rückblick auf Teil 1 der Vorlesung

Imperative vs. funktionale Programmierung – Ein Vergleich:

Gegeben ein Problem  $P$ .

**Imperativ:** Typischer Lösungsablauf besteht aus folgenden Schritten

1. Beschreibe eine(n) Lösung(s)weg)  $L$  für  $P$
2. Gieße  $L$  in die Form einer Menge von Anweisungen für den Rechner und organisiere dabei die Speicherverwaltung

**Funktional:**

- ...das “was” statt des “wie” in den Vordergrund stellen
- $\leadsto$  etwas von der Eleganz der Mathematik in die Programmierung bringen!

Quicksort: Ein eindrucksvolles Beispiel? Urteilen Sie selbst...

---

## Listenkonstruktoren vs. Listenoperatoren

Der Operator (:) ist Listenkonstruktor, (++) Listenoperator...

*Abgrenzung:* Konstruktoren führen zu eindeutigen Darstellungen, gewöhnliche Operatoren i.a. nicht.

*Veranschaulicht am Beispiel von Listen:*

```
42:17:4:[] == (42:(17:(4:[])))      -- eindeutig  
  
[42,17] ++ [] ++ [4] == [42,17,4] == [42] ++ [17,4] ++ []  
   -- nicht eindeutig
```

*Bemerkung:* (42:(17:(4:[]))) deutet an, dass eine Liste *ein* Objekt ist, erzwungen durch die Typstruktur. Anders in imperativen/objektorientierten Sprachen: Listen sind dort nur indirekt existent, nämlich bei "geeigneter" Verbindung von Elementen durch Zeiger.

---

## Vorschau auf die kommenden Aufgabenblätter...

Ausgabe des...

- fünften Aufgabenblatts: Mo, den 13.11.2006  
...Abgabetermine: Di, den 21.11.2006, und Di, den 28.11.2006, jeweils 15:00 Uhr
- sechsten Aufgabenblatts: Mo, den 20.11.2006  
...Abgabetermine: Di, den 28.11.2006, und Di, den 05.12.2006, jeweils 15:00 Uhr
- siebenten Aufgabenblatts: Mo, den 27.11.2006  
...Abgabetermine: Di, den 05.12.2006, und Di, den 12.12.2006, jeweils 15:00 Uhr

---

## Vorschau auf die nächsten Vorlesungstermine...

- Do, 16.11.2006: Keine Vorlesung, aber Besprechung von Aufgabenblatt 2 und 3 beginnend um 16:30 Uhr im Radinger-Hörsaal
- Do, 23.11.2006, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- Do, 30.11.2006, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal