# Einführung in die Funktionale Programmierung mit Haskell

## Typklassen und Polymorphie

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

23. Mai 2013

# Planung

- Freitag: Blatt 04 Lösung 2 ⇒ UniworX
- 30. & 31. Mai: keine Veranstaltung
- 5. Juni: Übung statt Vorlesung

Vorläufige Planung:

| Mi Vorlesung | Do Vorlesung | Fr Übung |
|---|---|---|
| 17. Apr 13 | 18. Apr 13 | 19. Apr 13 |
| 24. Apr 13 | 25. Apr 13 | 26. Apr 13 |
| 1. Mai 13 | 2. Mai 13 | 3. Mai 13 |
| 8. Mai 13 | 9. Mai 13 | 10. Mai 13 |
| 15. Mai 13 | 16. Mai 13 | 17. Mai 13 |
| 22. Mai 13 | 23. Mai 13 | 24. Mai 13 |
| 29. Mai 13 | 30. Mai 13 | 31. Mai 13 |
| 5. Jun 13 | 6. Jun 13 | 7. Jun 13 |
| 12. Jun 13 | 13. Jun 13 | 14. Jun 13 |
| 19. Jun 13 | 20. Jun 13 | 21. Jun 13 |
| 26. Jun 13 | 27. Jun 13 | 28. Jun 13 |
| 3. Jul 13 | 4. Jul 13 | 5. Jul 13 |
| 10. Jul 13 | 11. Jul 13 | 12. Jul 13 |
| 17. Jul 13 | 18. Jul 13 | 19. Jul 13 |

*Sollumfang:* 3+2
*Raumbuchung:* 4+2

42 Termine, davon
3 Feiertage
3 Übung nachholen
6 Entfällt (1x?)
1 Klausurtermin
= 20V + 12Ü

Klausur: 19.7. 10h

# Zusammenfassung: Datentypen



- Ein Typ (oder **Datentyp**) ist eine Menge von Werten
- Unter einer **Datenstruktur** versteht man einen Datentyp plus alle darauf verfügbaren Operationen
- Moderne Programmiersprachen ermöglichen, dass der Benutzer neue Typen definieren kann
- Datentypen können andere Typen als Parameter haben, **Konstruktoren** können als Funktionen betrachtetet werden
- **Records** erlauben Benennung dieser Typparameter
- Spezialfall **newtype**: nur 1 Konstruktor mit 1 Argument nur interessant zur Optimierung
- Datentypen können (wechselseitig) rekursiv definiert werden
- **Polymorphe Funktionen** können mit gleichem Code verschiedene Typen verarbeiten

# Datentypdeklaration



Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
  deriving (class_1, ..., class_l)
```

- Schlüsselwort `data`
- frischer Typname – beginnt wie immer mit Großbuchstaben
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen

# Parametrischer Polymorphismus (Wdh.)

Das **statische Typsystem** von Haskell prüft während dem Kompilieren alle Typen:

- Keine Typfehler und keine Typprüfung zur Laufzeit
- Kompiler kann besser optimieren

Dennoch können wir generischen Code schreiben:

```
reverse :: [a] -> [a]
reverse   []  = []
reverse (h:t) = reverse t ++ [h]
```

Wird von Haskell's Typsystem geprüft und für sicher befunden, da der **universell quantifizierte** Typ `a` keine Rolle spielt:

- Werte von Typ `a` werden herumgereicht, aber nicht inspiziert
- Code kann unabhängig von Typ `a` ausgeführt werden

`reverse` hat einen **Typparamter** und ist damit **polymorph**

# Polymorphe Gleichheit

Gleicheit ist ein leicht verständliches Konzept. Es wäre unhandlich, wenn wir für jeden Typ einen anderen Namen für dessen Vergleichsfunktion verwenden müssten.

Die Funktion `(==)` soll auf verschiedenen Typen funktionieren:

```
> 4 == 5
False
> 1.23 == 1 + 0.23
True
> 'a' == 'z'
False
> "Haskell" == 'H':'a':'s':'k':'e':'l':'l':[]
True
> [(1.3, "Ja")] == [(1.3, "Nein)]
False
```

# Polymorphe Gleichheit

Wie kann man Gleicheit für alle Typen implementieren?

```
(==) :: a -> a -> Bool
x == y =  ...?
```

↯ Geht nicht, da wir je nach Typ `a` anderen Code brauchen!
Z.B. Vergleich von zwei Listen komplizierter als von zwei Zeichen.

In vielen anderen Sprachen sind solche **überladenen** Operationen daher fest eingebaut. Eine Typprüfung zur Laufzeit entscheidet über den zu verwendenden Code ⇒ **Ad-hoc Polymorphismus.**

Problem: solche eingebauten Operation können dann oft nur schwerlich mit benutzerdefinierten Datentypen umgehen ☹

Haskell: Allgemeiner Mechanismus für ad-hoc Polymorphismus mit **Typklassen** und impliziten Dictionary Parametern

# Polymorphe Gleicheit

Welchen Typ hat (==)?

# Polymorphe Gleicheit

Welchen Typ hat `(==)`?

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

Lies “für alle Typen `a` aus der Typklasse `Eq`”, also
“für alle Typen `a`, welche wir vergleichen können”

# Polymorphe Gleicheit

Welchen Typ hat `(==)`?

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

Lies "für alle Typen `a` aus der Typklasse `Eq`", also
"für alle Typen `a`, welche wir vergleichen können"

- Polymorphismus der Vergleichsfunktion `(==)` ist eingeschränkt
- Zur Laufzeit wird ein Wörterbuch (Dictionary) *implizit* als weiteres Argument übergeben, welches für jeden Typ der **Typklasse** `Eq` den Code zum vergleichen bereithält

Verwenden wir `(==)` in einer Funktionsdefinition, so wird auch deren Typsignatur entsprechend eingeschränkt.

# Beispiel: Typklassen Anwenden

Verwendung der polymorphen Vergleichsfunktion in einer Funktionsdefinition, welche den ersten abweichenden Listenwert zweier Listen berechnet:

```
data Maybe a = Nothing | Just a -- zur Erinnerung
firstDiff (h1:t1) (h2:t2)
  | h1 == h2  = firstDiff t1 t2
  | otherwise = Just h2
firstDiff _ _ = Nothing

> firstDiff [1..10] [1,2,3,99,5,6]
Just 99
```

# Beispiel: Typklassen Anwenden

Verwendung der polymorphen Vergleichsfunktion in einer Funktionsdefinition, welche den ersten abweichenden Listenwert zweier Listen berechnet:

```
data Maybe a = Nothing | Just a -- zur Erinnerung
firstDiff (h1:t1) (h2:t2)
  | h1 == h2  = firstDiff t1 t2
  | otherwise = Just h2
firstDiff _ _ = Nothing

> firstDiff [1..10] [1,2,3,99,5,6]
Just 99
```

Die Typvariablen in der Typsignatur der Funktion `firstDiff` sind eingeschränkt quantifiziert:

```
> :type firstDiff
firstDiff :: Eq a => [a] -> [a] -> Maybe a
```

# Syntax Ad-hoc Polymorphismus

In Typsignaturen kann der Bereich eingeschränkt werden, über den eine Typvariable quantifiziert ist:

```
firstDiff :: Eq a => [a] -> [a] -> Maybe a
```

Die polymorphe Funktion `firstDiff` kann nur auf Werten von Typen angewendet werden, welche `Eq` instanziieren.

Es ist auch möglich, mehrere Einschränkungen anzugeben:

```
foo :: (Eq a, Read a) => [String] -> [a]
```

. . . und/oder mehrere Typvariablen einzuschränken:

```
bar :: (Eq a, Eq b, Read b, Ord c) => [(a,b)] -> [c]
```

Solche Typsignaturen kann GHC nicht nur inferieren, sondern wir können diese auch explizit in unserem Code hinschreiben und erzwingen.

# Übersicht Polymorphismus

## Parametrischer Polymorphismus

- Unabhängig vom tatsächlichen Typ wird immer der gleiche generische Code ausgeführt.
- Realisiert in Haskell über Typparameter: `a -> a`
- Haskell unterstützt auch erweiterte Formen, wie z.B. parametrischer Polymorphismus höheren Ranges.

## Ad-hoc Polymorphismus

- Abhängig vom tatsächlichen Typ zur Laufzeit wird spezifischer Code ausgeführt; Laufzeitprüfung notwendig.
- Realisiert in Haskell durch Einschränkung der Typparameter auf Typklassen: `Klasse a => a -> a`

## Subtyp Polymorphismus

- Mischform: Es kann der gleiche Code oder spezifischer Code ausgeführt werden.
- Irrelevant für uns, da GHC hat keinen direkten Subtyp-Mechanismus hat (wie z.B. Vererbung in Java).

# Typklassen

Eine **Typklasse** ist eine Menge von Typen, plus eine Menge von Funktionen, welche auf alle Typen der Klasse anwendbar sind.

Beispiel:

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

Auf jeden Typ `a`, der eine **Instanz** der Klasse `Eq` ist, können wir die beiden 2-stelligen Funktionen `(==)` und `(/=)` anwenden.

Die Standardbibliothek definiert diese Klasse `Eq` und zahlreiche Instanzen dazu:

```
Eq Bool      Eq Double
Eq Char      Eq Float    ...
Eq String    Eq Int
```

# Eq Typklasse

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

- Klassennamen schreiben wir wie Typen immer groß
- Die Typsignatur zeigt uns klar, `(==)` nur auf Werte des gleichen Typs anwendbar ist:
  Ausdruck `"Ugh" == [1..5]` ist nicht Typ-korrekt, obwohl `String` und `[Int]` beide Instanzen der Klasse `Eq` sind!
  Signatur `foo :: (Kl a, Kl b) => a -> b -> Bool`
  würde einen Ausdruck `"Ugh" ‘foo‘ [1..5]` erlauben,
  falls `String` und `[Int]` Instanzen der Klasse `Kl` wären.
- Klassendefinition kann auch Default-Funktionen definieren.
  Die können jedoch überschrieben werden.

⇒ Klassendefinition entsprechen grob Java-Interfaces

# Instanzen definieren

Man kann leicht neue Instanzen für eigene Datentypen definieren:

```
data Früchte = Apfel (Int,Int) | Birne Int Int

instance Eq Früchte where
  Apfel x1    == Apfel x2    = x1 == x2
  Birne x1 y1 == Birne x2 y2 = x1 == x2 && y1 == y2
  _           == _           = False
```

Damit können wir nun Äpfel und Birnen vergleichen:

```
> Apfel (3,4) == Apfel (3,4)
True
> Apfel (3,4) == Apfel (3,5)
False
> Apfel (3,4) /= Birne 3 4
True
```

Definition für /= verwendet Klassen Default-Implementierung, da wir diese nicht neu definiert haben.

# Instanzen ableiten

Man kann leicht neue Instanzen für eigene Datentypen definieren:

```
data Früchte = Apfel (Int,Int) | Birne Int Int

instance Eq Früchte where
  Apfel x1    == Apfel x2    = x1 == x2
  Birne x1 y1 == Birne x2 y2 = x1 == x2 && y1 == y2
  _           == _           = False
```

Diese Instanz Deklaration ist langweilig und ohne Überraschungen:

Bei gleichen Konstruktoren vergleichen wir einfach nur die Argumente der Konstruktoren.

GHC kann solche Instanzen mit `derving` automatisch generieren:

```
data Früchte = Apfel (Int,Int) | Birne Int Int
  deriving (Eq, Ord, Show, Read)
```

# Typklassen Instanzen

- Instanzdeklarationen müssen alle Funktionen der Klasse implementieren, für die es keine Default-Implementierung gibt
- Default-Implementierung dürfen überschrieben werden
- Viele Standard-Instanzen automatisch ableitbar
- Ein Typ kann mehreren Typklassen auf einmal angehören
- Aber nur *eine* Instanz pro Klasse definierbar

Beispiel: Wenn wir `Früchte` *manchmal* nur nach Preis vergleichen wollen, dann brauchen wir einen neuen Typ dafür:

```
newtype PreisFrucht = PF Früchte
instance Eq PreisFrucht where
  PF(Apfel (_,p1)) == PF(Apfel (_,p2)) = p1 == p2
  PF(Apfel (_,p1)) == PF(Birne p2 _  ) = p1 == p2
  PF(Birne p1 _  ) == PF(Apfel (_,p2)) = p1 == p2
  PF(Birne p1 _  ) == PF(Birne p2 _  ) = p1 == p2
```

# Show Typklasse

Die Typklasse `Show` erlaubt die Umwandlung eines Wertes in ein String, um den Wert zum Beispiel am Bildschirm anzuzeigen:

```
class  Show a  where
  show       :: a    -> String
  showList   :: [a] -> ShowS
  ...
```

Beispiel:

```
> show (Apfel (3,4))
"Apfel (3,4)"
> show (Birne 5 6)
"Birne 5 6"
```

Die Standardbibliothek definiert viele Instanzen dazu:

```
Show Bool       Show Double       Show Int
Show Char       Show Float        ...
```

# Show Typklasse

Wenn uns die automatisch abgeleitete `show` Funktion nicht gefällt, dann entfernen wir `Show` aus der `derving` Klausel der Datentypdeklaration für `Früchte`, und definieren die Instanz selbst:

```
instance Show Früchte where
  show (Apfel (z,p)) = "Apfel("++(show z)++")"
  show (Birne p z)   = "Birne("++(show z)++")"
```

Damit erhalten wir dann entsprechend:

```
> show (Apfel (3,4))
"Apfel(3)"
> show (Birne 5 6)
"Birne(6)"
```

# Read Typklasse

Das Gegenstück zu `Show` ist die Typklasse `Read`:

```
class Read a where
  read :: String -> a
  ...
```

Da der Typ `a` nur im Ergebnis vorkommt, müssen wir Haskell manchmal sagen, welchen Typ wir brauchen:

```
> read "7" :: Int
7
> read "7" :: Double
7.0
```

Innerhalb Programmes kann Haskell oft den Typ inferieren:

```
> [ read "True", 6==9, 'a'/='z' ]
[True,False,True] :: [Bool]
```

# Read Typklasse

Das Gegenstück zu `Show` ist die Typklasse `Read`:

```
class Read a where
  readsPrec :: Int -> ReadS a
  ...

read :: Read a => String -> a
```

- Kann ebenfalls automatisch abgeleitet werden
- Viele Instanzen in der Standardbibliothek definiert
- Wenn man die `Show`-Instanz selbst definiert,
  dann sollte man auch die `Read`-Instanz selbt definieren!
- Funktion `read` kann eine Ausnahme werfen

```
> read "7" :: Bool
*** Exception: Prelude.read: no parse
```

# Überladene Instanzen

Interssanterweise können wir auch gleich automatisch Paare und Listen von Früchten vergleichen:

```
> (Apfel (3,4), Birne 5 6) == (Apfel (3,4), Birne 5 6)
True

> [Apfel (3,4), Birne 5 6] /= [Apfel (3,4), Birne 5 6]
False
```

Dies liegt daran, dass die Deklaration von Typklassen und Instanzen auch selbst wieder überladen werden dürfen!

# Überladene Instanzen

Wenn wir jeweils zwei Werte von Typ `a` und zwei von Typ `b` vergleichen können, dann können wir auch Tupel des Typs `(a,b)` auf Gleichheit testen:

```
instance (Eq a,Eq b) => Eq (a,b) where
  (x1,y1) == (x2,y2) = (x1==x2) && (y1==y2)
```

Wenn wir jeweils zwei Werte von Typ `a` vergleichen können, dann können wir auch gleich eine Liste des Typs `[a]` vergleichen:

```
instance (Eq a) => Eq [a] where
  (x:xs) == (y:ys) = x == y  &&  xs == ys
  []     == []     = True
  _xs    == _ys    = False
```

# Überladene Instanzen

Wenn wir jeweils zwei Werte von Typ `a` und zwei von Typ `b` vergleichen können, dann können wir auch Tupel des Typs `(a,b)` auf Gleichheit testen:

```
instance (Eq a,Eq b) => Eq (a,b) where
  (x1,y1) == (x2,y2) = (x1==x2) && (y1==y2)
```

Wenn wir jeweils zwei Werte von Typ `a` vergleichen können, dann können wir auch gleich eine Liste des Typs `[a]` vergleichen:

```
instance (Eq a) => Eq [a] where
  (x:xs) == (y:ys) = x == y  &&  xs == ys
  []     == []     = True
  _xs    == _ys    = False
```

Erster Vergleich: `Eq a`

# Überladene Instanzen

Wenn wir jeweils zwei Werte von Typ `a` und zwei von Typ `b` vergleichen können, dann können wir auch Tupel des Typs `(a,b)` auf Gleichheit testen:

```
instance (Eq a,Eq b) => Eq (a,b) where
  (x1,y1) == (x2,y2) = (x1==x2) && (y1==y2)
```

Wenn wir jeweils zwei Werte von Typ `a` vergleichen können, dann können wir auch gleich eine Liste des Typs `[a]` vergleichen:

```
instance (Eq a) => Eq [a] where
  (x:xs) == (y:ys) = x == y  &&  xs == ys
  []     == []     = True
  _xs    == _ys    = False
```

Erster Vergleich: `Eq a`  Zweiter Vergleich: `Eq [a]` rekursiv!

# Ord Typklasse

Werte eines Typs der Typklasse `Ord` können wir ordnen und nach Größe sortieren:

```
data Ordering = LT | EQ | GT

class  (Eq a) => Ord a  where
  compare              :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min             :: a -> a -> a
```

- Alle Typen der Typklasse `Ord` müssen auch in `Eq` sein
- Kann automatisch abgeleitet werden
- Instanzen müssen mindestens `compare` oder `(<=)` angeben
- Viele praktische Funktion in der Standardbibliothek:

```
minimum, maximum :: Ord a => [a] -> a
sort             :: Ord a => [a] -> [a]
```

*Merke:* Typklassen können verschieden erweitert werden: abgeleitete Klassen oder durch auf Typklassen eingeschränkte Funktionen, die nicht in deren Klassendefinition enthalten sind!

# Enum Typklasse

Die Typklasse `Enum` ist für alle aufzählbaren Typen:

```
class  Enum a   where
  succ, pred     :: a -> a     -- Nachfolger/Vorgänger
  fromEnum       :: a -> Int   -- Ordnungszahl
  toEnum         :: Int -> a
  ...
  enumFromTo     :: a -> a -> [a]      -- [a..e]
  enumFromThenTo :: a -> a -> a -> [a] -- [a,s..e]
```

- Wenn alle Konstruktoren eines Datentyps keine Argumente haben, kann `Enum` automatisch abgeleitet werden
  Beispiel `data Day = Mon|Tue|Wed|Thu|Fri|Sat|Sun`
- Viele Instanzen in der Standardbibliothek vordefiniert
- `Enum` impliziert nicht automatisch `Ord`!

# Bounded Typklasse

Die Klasse aller beschränkten Typen:

```
class  Bounded a  where
  minBound, maxBound :: a

> maxBound :: Bool
True
> minBound :: Int
-9223372036854775808
```

- `Ord` ist keine Oberklasse von `Bounded`, da nicht jeder geordnete Typ ein größtes oder kleinstes Element haben muss
- Kann für alle Typen der Klasse `Enum` abgeleitetet werden
- Kann für Datentypen mit einem Konstruktor abgeleitet werden, wenn alle Argumente von Typen aus `Bounded` sind

# Num Typklasse

Auch die arithmetischen Funktion sind mit diesem Mechanismus überladen, es werden keine Sonderbehandlungen benötigt:

```
class Num a where
  (+), (*), (-) :: a -> a -> a
  negate :: a -> a
  abs    :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

- Wir könnten `(+)` leicht erweitern, um mit `Früchte` zu rechnen
- Kann nicht automatisch abgeleitet werden
- Mann kann bei der Instanzdeklarationen höchstens `negate` oder `(-)` weglassen

# Integral und Real Typklassen

Erweitert die Klasse `Num` um ganzzahlige Divison:

```
class  (Real a, Enum a) => Integral a where
  div :: a -> a -> a
  rem :: a -> a -> a
  mod :: a -> a -> a
  ...
  toInteger :: a -> Integer

class (Num a, Ord a) => Real a where
  toRational ::  a -> Rational
```

- Die Klasse `Integral` impliziert also `Real`, `Enum`, `Num`, `Ord`,
- `Real` enthält nur Umwandlungsfunktion zu Typ `Rational`

# Typklasse Fractional

Typklasse für alle Zahlen, welche volle Divison unterstützen:

```
class  (Num a) => Fractional a  where
  (/)          :: a -> a -> a
  recip        :: a -> a          -- Kehrwert
  fromRational :: Rational -> a
```

Typklasse für alle Fließkommazahlen:

```
class  (Fractional a) => Floating a  where
  pi             :: a
  exp, log, sqrt :: a -> a
  (**), logBase  :: a -> a -> a
  sin, cos, tan  :: a -> a
  ...
```

# Zahlen

- Es gibt noch weitere spezielle Zahlenklassen (z.B. `RealFrac`)
- Wichtige Funktionen zur Umwandlung von Zahlen sind:

```
fromIntegral :: (Integral a, Num b)    => a -> b
realToFrac   :: (Real a, Fractional b) => a -> b
```

- Viele numerische Funktion außerhalb von Typklassen definiert, welche so bequeme Verwendung ohne explizite Umwandlung ermöglichen:

```
(^)  :: (Num a, Integral b) =>          a -> b -> a
     -- verlangt positive Exponenten
(^^) :: (Fractional a, Integral b) => a -> b -> a
     -- erlaubt negative Exponenten
```

# Zusammenfassung

- Typklassen definieren ein **abstraktes Interface** bzw.
  Typklassen definieren polymorphe Funktionen für eine eingeschränkte Menge von Typen
  Parametrischer Polymorphismus $\Rightarrow$ Ad-hoc Polymorphismus
- Überladen benötigt keinen speziellen Mechanismus in Haskell
- Ein Typ kann mehreren Typklassen angehören
- Jeder Typ kann nur eine Instanz pro Klasse deklarieren
- Typklassen können mit Unterklassen erweitert werden
- Beliebige Funktionsdefinition können mit Typklassen eingeschränkt werden
- GHC versucht immer den **prinzipalen** (allgemeinsten) Typ einer Deklaration zu inferieren

# Module

Ein Haskell Programm besteht aus mehreren Modulen:

```
module Main where
  import Prelude
  ...
```

- Modulenamen werden immer groß geschrieben
- Pro Modul eine Datei mit gleich lautendem Namen
- Punkte in Modulnamen entsprechen Verzeichnisunterteilung:
  `ModA.SubA.MyMod` ist in Datei `ModA\SubA\MyMod.hs`
- Jedes Modul hat seinen eigenen Namensraum:
  `Mod1.foo` kann eine völlig andere Funktion wie `Mod2.foo` sein
- Standardbibliothek ist das Modul `Prelude`.
  Es gibt viele weitere nützliche Standard-Module:
  `Data.List`, `Data.Set`, `Data.Map`, `System.IO`, ...
- Haskell Module sind nicht parametrisiert

# Module

Ein Haskell Programm besteht aus mehreren Modulen:

```
module Main where
  import Prelude
  ...
```

- Modulenamen werden immer groß geschrieben
- Pro Modul eine Datei mit gleich lautendem Namen
- Punkte in Modulnamen entsprechen Verzeichnisunterteilung:
  `ModA.SubA.MyMod` ist in Datei `ModA\SubA\MyMod.hs`
- Jedes Modul hat seinen eigenen Namensraum:
  `Mod1.foo` kann eine völlig andere Funktion wie `Mod2.foo` sein
- Standardbibliothek ist das Modul `Prelude`.
  Es gibt viele weitere nützliche Standard-Module:
  `Data.List`, `Data.Set`, `Data.Map`, `System.IO`, ...
- Haskell Module sind nicht parametrisiert

# Module

Ein Haskell Programm besteht aus mehreren Modulen:

```
module Main where
  import Prelude
  ...
```

- Modulenamen werden immer groß geschrieben
- Pro Modul eine Datei mit gleich lautendem Namen
- Punkte in Modulnamen entsprechen Verzeichnisunterteilung:
  `ModA.SubA.MyMod` ist in Datei `ModA\SubA\MyMod.hs`
- Jedes Modul hat seinen eigenen Namensraum:
  `Mod1.foo` kann eine völlig andere Funktion wie `Mod2.foo` sein
- Standardbibliothek ist das Modul `Prelude`.
  Es gibt viele weitere nützliche Standard-Module:
  `Data.List, Data.Set, Data.Map, System.IO, ...`
- Haskell Module sind nicht parametrisiert

# Module

Ein Haskell Programm besteht aus mehreren Modulen:

```
module Main where
  import Prelude
  ...
```

- Modulenamen werden immer groß geschrieben
- Pro Modul eine Datei mit gleich lautendem Namen
- Punkte in Modulnamen entsprechen Verzeichnisunterteilung:
  `ModA.SubA.MyMod` ist in Datei `ModA\SubA\MyMod.hs`
- Jedes Modul hat seinen eigenen Namensraum:
  `Mod1.foo` kann eine völlig andere Funktion wie `Mod2.foo` sein
- Standardbibliothek ist das Modul `Prelude`.
  Es gibt viele weitere nützliche Standard-Module:
  `Data.List, Data.Set, Data.Map, System.IO, ...`
- Haskell Module sind nicht parametrisiert

# Module

Ein Haskell Programm besteht aus mehreren Modulen:

```
module Main where
  import Prelude
  ...
```

- Modulenamen werden immer groß geschrieben
- Pro Modul eine Datei mit gleich lautendem Namen
- Punkte in Modulnamen entsprechen Verzeichnisunterteilung:
  `ModA.SubA.MyMod` ist in Datei `ModA\SubA\MyMod.hs`
- Jedes Modul hat seinen eigenen Namensraum:
  `Mod1.foo` kann eine völlig andere Funktion wie `Mod2.foo` sein
- Standardbibliothek ist das Modul `Prelude`.
  Es gibt viele weitere nützliche Standard-Module:
  `Data.List, Data.Set, Data.Map, System.IO, ...`
- Haskell Module sind nicht parametrisiert

# Module

Ein Haskell Programm besteht aus mehreren Modulen:

```
module Main where
  import Prelude
  ...
```

- Modulenamen werden immer groß geschrieben
- Pro Modul eine Datei mit gleich lautendem Namen
- Punkte in Modulnamen entsprechen Verzeichnisunterteilung:
  `ModA.SubA.MyMod` ist in Datei `ModA\SubA\MyMod.hs`
- Jedes Modul hat seinen eigenen Namensraum:
  `Mod1.foo` kann eine völlig andere Funktion wie `Mod2.foo` sein
- Standardbibliothek ist das Modul `Prelude`.
  Es gibt viele weitere nützliche Standard-Module:
  `Data.List, Data.Set, Data.Map, System.IO, ...`
- Haskell Module sind nicht parametrisiert

# Export

Export und Import von Modulen kann eingeschränkt werden:

```
module Tree ( Tree(..), Oak(OakLeaf), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)
data Oak a  = OakLeaf | OakBranch (Tree a) (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right

hidden :: Tree a -> a
...
```

- `Tree` und alle seine Konstruktoren werden exportiert
- Von `Oak` wird nur Konstruktor `OakLeaf` exportiert
- Funktion `fringe` wird exportiert, aber `hidden` nicht

# Export

Export und Import von Modulen kann eingeschränkt werden:

```
module Tree ( Tree(..), Oak(OakLeaf), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)
data Oak a  = OakLeaf | OakBranch (Tree a) (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right

hidden :: Tree a -> a
...
```

- `Tree` und alle seine Konstruktoren werden exportiert
- Von `Oak` wird nur Konstruktor `OakLeaf` exportiert
- Funktion `fringe` wird exportiert, aber `hidden` nicht

# Export

Export und Import von Modulen kann eingeschränkt werden:

```
module Tree ( Tree(..), Oak(OakLeaf), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)
data Oak a  = OakLeaf | OakBranch (Tree a) (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right

hidden :: Tree a -> a
...
```

- `Tree` und alle seine Konstruktoren werden exportiert
- Von `Oak` wird nur Konstruktor `OakLeaf` exportiert
- Funktion `fringe` wird exportiert, aber `hidden` nicht

# Export

Export und Import von Modulen kann eingeschränkt werden:

```
module Tree ( Tree(..), Oak(OakLeaf), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)
data Oak a  = OakLeaf | OakBranch (Tree a) (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right

hidden :: Tree a -> a
...
```

- `Tree` und alle seine Konstruktoren werden exportiert
- Von `Oak` wird nur Konstruktor `OakLeaf` exportiert
- Funktion `fringe` wird exportiert, aber `hidden` nicht

# Import

```
module Import where
  import Prelude
  import MyModul -- definiert foo und bar

  myfun x = foo x + MyModul.bar x
  ...
```

- Wenn ein Modul importiert wird, werden alle exportierten Deklaration in dem importierenden Modul sichtbar.
- Man kann diese direkt verwenden, oder mit `ModulName.bezeichner` ansprechen

# Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head`, `init`, `last`, `tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen

Zum Beispiel gibt es `Map.map` und `map`, `Map.null` und `null`

Funktion `tail` z.B. daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

# Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head, init, last, tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen

  Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

# Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head, init, last, tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
  Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

# Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head, init, last, tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
  Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

# Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head, init, last, tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
  Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

# Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head, init, last, tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
  Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

# Module

Module erlauben also:

- Unterteilung des Namensraums
- Aufteilung von Code auf mehrere Dateien
- Unterstützen Modularisierung und Abstraktion durch das Verstecken von Implementierungsdetails
- Instanzdeklarationen für Typklassen werden immer exportiert und importiert

In GHCI importieren wir Module mit

```
> :module + Data.List Data.Set
```

und schließen Module mit

```
> :module - Data.List Data.Set
```