

# Fortgeschrittene funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

## Überblick für Heute:

- Organisatorisches & Überlebenstipps
- Wiederholung Haskell-Basics
- Thinking in Types
- Lazy Evaluation
- Problemlösen durch Zusammenstecken

# Organisatorisches & Überlebensstipps

## Organisatorisches: Veranstaltungen

Es gibt Vorlesungen (Freitags, 14-16 Uhr in V2-205)  
und Übungen (Montags, 12-14 & 18-20 Uhr in V2-221)

Teilnahme an den Übungen ist nicht verpflichtend, aber von Vorteil.

## Organisatorisches (2): Input / Output

Das Modul gibt es 5 (echte) Leistungspunkte.

Bürokratische Hürden  $\Rightarrow$  LP nur für *individuelle* Ergänzung

**Kriterium:** erfolgreicher Abschluss eines kleinen Programmierprojektes (Aufgabe TBA, Details in den Übungen)

## Organisatorisches (3): Personenkult

Wir, das sind Jonas Betzendahl und Stefan Dresselhaus.  
Mailadressen: {jbetzend,sdressel}@techfak...

Formal verantwortlich:

Dr. Alexander Sczyrba (asczyrba@techfak...)  
(für Fragen im Kontext der Fakultät und Beschwerden zu uns)

## Organisatorisches (4): Material

Aufgabenblätter, Foliensätze, Beispiele, Vorlagen und sonstige Unterlagen entweder im ekVV oder zum Selberklonen auf GitHub:

<https://github.com/FFPiHaskell>

Audio & Video - Mitschnitte auf YouTube, Näheres momentan ebenfalls TBA

## T & R (1): Haskell / GHC

Standard in dieser Vorlesung ist der  
Glasgow Haskell Compiler (GHC) ( $\geq$  v. 7.8, wo relevant)



## T & R (1): Haskell / GHC

Standard in dieser Vorlesung ist der  
Glasgow Haskell Compiler (GHC) ( $\geq$  v. 7.8, wo relevant)

Rundum-Glücklich-Paket für eigene Rechner: *Haskell Platform*  
<https://www.haskell.org/platform/>

## T & R (1): Haskell / GHC

Standard in dieser Vorlesung ist der  
Glasgow Haskell Compiler (GHC) ( $\geq$  v. 7.8, wo relevant)

Rundum-Glücklich-Paket für eigene Rechner: *Haskell Platform*  
<https://www.haskell.org/platform/>

Aktuellen GHC (7.10) kriegt ihr im GZI mit dem rcinfo-Paket `ghc`

## T & R (1): Haskell / GHC

Standard in dieser Vorlesung ist der  
Glasgow Haskell Compiler (GHC) ( $\geq$  v. 7.8, wo relevant)

Rundum-Glücklich-Paket für eigene Rechner: *Haskell Platform*  
<https://www.haskell.org/platform/>

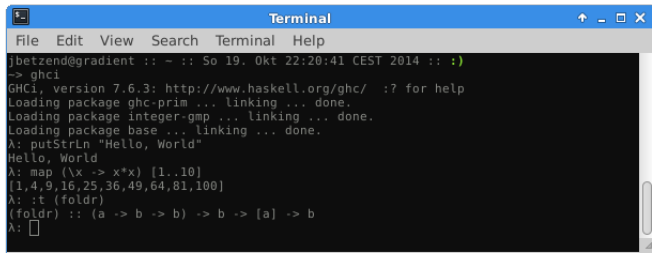
Aktuellen GHC (7.10) kriegt ihr im GZI mit dem rcinfo-Paket `ghc`

**Wichtig:**

Der Haskell-Interpreter Hugs wird von uns nicht unterstützt!

## T & R (2): GHCi

Der GHC hat auch eine interaktive Umgebung: GHCi.



```
Terminal
File Edit View Search Terminal Help
jbetzend@gradient :: ~ :: So 19. Okt 22:20:41 CEST 2014 :: :)
-> ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  ? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
λ: putStrLn "Hello, World"
Hello, World
λ: map (\x -> x*x) [1..10]
[1,4,9,16,25,36,49,64,81,100]
λ: :t (foldr)
(foldr) :: (a -> b -> b) -> b -> [a] -> b
λ: 
```

GHCi bietet auch ein REPL (Read - Evaluate - Print - Loop),  
*sehr* nützlich zum Entwickeln (ähnlich zu Hugs).

## T & R (3): Hackage

Die meisten Bibliotheken von Haskell wohnen auf *Hackage*:

<https://hackage.haskell.org/>

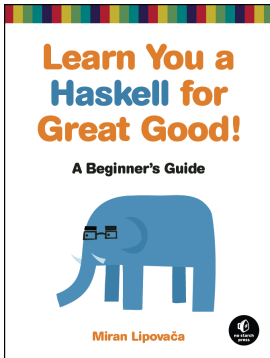
Dort findet ihr übersichtliche Zusammenfassungen der Bibliotheken, detaillierte Auflistungen der exportierten Funktionen und Datentypen und die jeweiligen Implementationen (!).

## T & R (4): cabal

Haskells `cabal` ist ein Programm zum erstellen, verpacken und installieren von Bibliotheken und Programmen:

- lokale Installation (keine sudo-Rechte notwendig)
- Zugriff auf Hackage
- Hilfe beim Erstellen von Paketen
- Dependency management
- ...

## T & R (5): LYAHFGG



Das Buch „Learn You A Haskell“ ist die beste<sup>TM</sup> Ressource um die ersten Schritte in Haskell zu lernen.

Ihr findet es online frei und kostenlos verfügbar hier:  
<http://learnyouahaskell.com/>

# Wiederholung Haskell-Basics



## Haskell auf einer Folie:

```
-- Only those elements that conform to the predicate
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs)
  | p x          = x : filter p xs
  | otherwise    =      filter p xs
```

## Haskell auf einer Folie:

```
-- Only those elements that conform to the predicate
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs)
    | p x        = x : filter p xs
    | otherwise  =      filter p xs
```

- Typsignaturen

## Haskell auf einer Folie:

```
-- Only those elements that conform to the predicate
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs)
    | p x        = x : filter p xs
    | otherwise  =      filter p xs
```

- Typsignaturen
- Pattern Matching

## Haskell auf einer Folie:

```
-- Only those elements that conform to the predicate
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs)
  | p x          = x : filter p xs
  | otherwise    =      filter p xs
```

- Typsignaturen
- Pattern Matching
- Polymorphismus

## Haskell auf einer Folie:

```
-- Only those elements that conform to the predicate
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs)
  | p x          = x : filter p xs
  | otherwise    =      filter p xs
```

- Typsignaturen
- Pattern Matching
- Polymorphismus
- higher order fun.

## Haskell auf einer Folie:

```
-- Only those elements that conform to the predicate
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs)
  | p x          = x : filter p xs
  | otherwise    =      filter p xs
```

- Typsignaturen
- Pattern Matching
- Polymorphismus
- higher order fun.
- Guards

## Haskell auf einer Folie:

```
-- Only those elements that conform to the predicate
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs)
    | p x        = x : filter p xs
    | otherwise  =      filter p xs
```

- Typsignaturen
- Pattern Matching
- Polymorphismus
- higher order fun.
- Guards
- Curryfizierung

## Haskell auf einer Folie:

```
-- Only those elements that conform to the predicate
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs)
    | p x        = x : filter p xs
    | otherwise  =      filter p xs
```

- Typsignaturen
- Pattern Matching
- Polymorphismus
- higher order fun.
- Guards
- Curryfizierung
- Anwendung von Funktionen:  
  f x y -- statt f(x,y)



# Thinking in Types

## Haskell auf etwas mehr als einer Folie:

Folgende Typen solltet ihr schon kennen...

`Int`, `Integer`, `Float`, `Double`, `Char`, `String`, `Bool` ...

## Haskell auf etwas mehr als einer Folie:

Folgende Typen solltet ihr schon kennen...

`Int`, `Integer`, `Float`, `Double`, `Char`, `String`, `Bool` ...

... außerdem gibt es Typkonstruktoren, die neue Typen machen ...

`[]`, `Tree`, `Maybe`, `Either`, `(,)` ...

## Haskell auf etwas mehr als einer Folie:

Folgende Typen solltet ihr schon kennen...

`Int`, `Integer`, `Float`, `Double`, `Char`, `String`, `Bool` ...

... außerdem gibt es Typkonstruktoren, die neue Typen machen ...

`[]`, `Tree`, `Maybe`, `Either`, `(,)` ...

... und so machen wir ganz neue Typen:

```
type List a = [a]
```

## Haskell auf etwas mehr als einer Folie:

Folgende Typen solltet ihr schon kennen...

`Int`, `Integer`, `Float`, `Double`, `Char`, `String`, `Bool` ...

... außerdem gibt es Typkonstruktoren, die neue Typen machen ...

`[]`, `Tree`, `Maybe`, `Either`, `(,)` ...

... und so machen wir ganz neue Typen:

```
type List a = [a]
```

```
newtype Sekunden = Sekunden Int
```

## Haskell auf etwas mehr als einer Folie:

Folgende Typen solltet ihr schon kennen...

`Int`, `Integer`, `Float`, `Double`, `Char`, `String`, `Bool` ...

... außerdem gibt es Typkonstruktoren, die neue Typen machen ...

`[]`, `Tree`, `Maybe`, `Either`, `(,)` ...

... und so machen wir ganz neue Typen:

```
type List a = [a]
```

```
newtype Sekunden = Sekunden Int
```

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

```
data [a] = [] | a : [a] -- algebraisch, rekursiv
```

## Problemstellung:

Was ist das Problem mit folgender Funktion?

```
quadrat :: a -> a  
quadrat x = x * x
```

## Problemstellung:

Was ist das Problem mit folgender Funktion?

```
quadrat :: a -> a  
quadrat x = x * x
```

→ Funktion (\*) könnte undefiniert für a sein (Funktionstypen)



## Problemstellung:

Was ist das Problem mit folgender Funktion?

```
quadrat :: a -> a  
quadrat x = x * x
```

- Funktion (\*) könnte undefiniert für a sein (Funktionstypen)
- Verschiedene Lösungsansätze:

## Problemstellung:

Was ist das Problem mit folgender Funktion?

```
quadrat :: a -> a  
quadrat x = x * x
```

- Funktion (\*) könnte undefiniert für a sein (Funktionstypen)
- Verschiedene Lösungsansätze:
  - „Local choice“, nur polymorphes Symbol (Abstraktionsverlust)

## Problemstellung:

Was ist das Problem mit folgender Funktion?

```
quadrat :: a -> a  
quadrat x = x * x
```

- Funktion (\*) könnte undefiniert für a sein (Funktionstypen)
- Verschiedene Lösungsansätze:
  - „Local choice“, nur polymorphes Symbol (Abstraktionsverlust)
  - Standardimplementationen für Gleichheit etc. (Laufzeitfehler)

## Haskell-Lösung: Typklassen

```
quadrat :: Num a => a -> a  
quadrat x = x * x
```

Polymorphismus beschränkt auf die Typen, die auch bestimmte Funktionen unterstützen.

## Haskell-Lösung: Typklassen

```
quadrat :: Num a => a -> a  
quadrat x = x * x
```

Polymorphismus beschränkt auf die Typen, die auch bestimmte Funktionen unterstützen.

*Abstrakte Definition:*

```
class Num a where  
    (+)    :: a -> a -> a  
    (*)    :: a -> a -> a  
    negate :: a -> a  
    ...
```

# Haskell-Lösung: Typklassen

```
quadrat :: Num a => a -> a  
quadrat x = x * x
```

Polymorphismus beschränkt auf die Typen, die auch bestimmte Funktionen unterstützen.

*Abstrakte Definition:*

```
class Num a where  
  (+)    :: a -> a -> a  
  (*)    :: a -> a -> a  
  negate :: a -> a  
  ...
```

*Konkrete Instanz:*

```
instance Num Int where  
  i + j    = plusInt i j  
  i * j    = mulInt  i j  
  negate i = negInt  i  
  ...
```

# Haskell-Lösung: Typklassen

```
quadrat :: Num a => a -> a  
quadrat x = x * x
```

Polymorphismus beschränkt auf die Typen, die auch bestimmte Funktionen unterstützen.

*Abstrakte Definition:*

```
class Num a where  
  (+)    :: a -> a -> a  
  (*)    :: a -> a -> a  
  negate :: a -> a  
  ...
```

*Konkrete Instanz:*

```
instance Num Int where  
  i + j    = plusInt i j  
  i * j    = mulInt  i j  
  negate i = negInt  i  
  ...
```

plusInt, mulInt und negInt  
an anderer Stelle definiert.

Es gibt in Haskell zwei Möglichkeiten, einen Typen einer Typklasse hinzuzufügen:



Es gibt in Haskell zwei Möglichkeiten, einen Typen einer Typklasse hinzuzufügen:

*Von Hand:*

```
class Show a where
  show :: a -> String

data Bool = False | True

instance Show Bool where
  show False = "False"
  show True  = "True"
```

Es gibt in Haskell zwei Möglichkeiten, einen Typen einer Typklasse hinzuzufügen:

*Von Hand:*

```
class Show a where
  show :: a -> String

data Bool = False | True

instance Show Bool where
  show False = "False"
  show True  = "True"
```

*Automatisch:*

```
class Show a where
  show :: a -> String

data Bool = False | True
  deriving Show
```

Es gibt in Haskell zwei Möglichkeiten, einen Typen einer Typklasse hinzuzufügen:

*Von Hand:*

```
class Show a where
  show :: a -> String

data Bool = False | True

instance Show Bool where
  show False = "False"
  show True  = "True"
```

*Automatisch:*

```
class Show a where
  show :: a -> String

data Bool = False | True
  deriving Show
```

⇒ Mehr zu Typklassen (inkl. Functor, Applicative, Monad)  
nächste Woche.

# Lazy Evaluation

-- ... *to be implemented*