

# 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/>

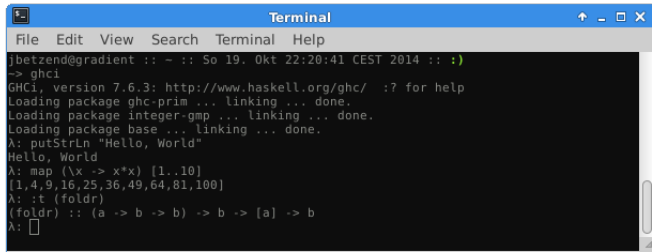
Aktuellen GHC (7.10) kriegt ihr im GZI mit dem rcinfo-Paket `ghc`

### 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/>

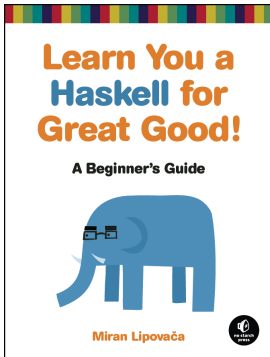
Dort findet ihr übersichtliche Zusammenfassungen der Bibliotheken, detaillierte Auflistungen der exportierten Funktionen und Datentypen und die jeweiligen Implementationen (!).

## 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
- Management von Abhängigkeiten
- Sandboxes
- ...

## 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://learnyouahaskell1.com/>

# Wiederholung Haskell-Basics



"Haskell is a purely functional programming language  
with strong and static types and lazy evaluation"

"Haskell is a **purely functional** programming language  
with **strong and static types** and **lazy evaluation**"

## 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
- Anwenden von Funktionen:

`f x y` -- statt `f(x,y)` wie z.B. in Java

# Thinking in Types

"Haskell is a purely functional programming language  
with **strong and static types** and lazy evaluation"

## 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)

# Haskells Lösung: Typklassen

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

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

# Haskells 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  
    ...
```



# Haskells 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  
  ...
```

# Haskells 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 (geht immer):*

```
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 (geht immer):*

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

data Bool = False | True

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

*Automatisch (geht meistens):*

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

data Bool = False | True
  deriving Show
```

Weitere Beispiele für Typklassen:

Weitere Beispiele für Typklassen:

```
class Eq a where
  (==) :: a -> a -> Bool  -- Minimale Definition für
  ...                    -- eine Instanz der Klasse
```

Weitere Beispiele für Typklassen:

```
class Eq a where
  (==) :: a -> a -> Bool    -- Minimale Definition für
  ...                       -- eine Instanz der Klasse

class Eq a => Ord a where
  (<=)      :: a -> a -> Bool    -- Minimale Definition
  compare  :: a -> a -> Ordering -- data Ordering=LT/EQ/GT
  ...
```



Weitere Beispiele für Typklassen:

```
class Eq a where
  (==) :: a -> a -> Bool    -- Minimale Definition für
  ...                       -- eine Instanz der Klasse

class Eq a => Ord a where
  (<=)      :: a -> a -> Bool    -- Minimale Definition
  compare  :: a -> a -> Ordering -- data Ordering=LT/EQ/GT
  ...

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

Weitere Beispiele für Typklassen:

```
class Eq a where
  (==) :: a -> a -> Bool    -- Minimale Definition für
  ...                      -- eine Instanz der Klasse

class Eq a => Ord a where
  (<=)      :: a -> a -> Bool    -- Minimale Definition
  compare :: a -> a -> Ordering -- data Ordering=LT/EQ/GT
  ...

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

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

# Purity

"Haskell is a **purely** functional programming language  
with strong and static types and lazy evaluation"

# Lazy Evaluation

"Haskell is a purely functional programming language  
with strong and static types and **lazy evaluation**"

# Problemlösen durch Zusammensetzen

"Haskell is a purely **functional** programming language  
with strong and static types and lazy evaluation"



In der Informatik ist „*devide and conquer*“ häufig ein guter Ansatz. In Haskell ist die Lösung zu einem größeren Problem ebenfalls oft das „Zusammenstecken“ von Lösungen kleinerer Teilprobleme.

In der Informatik ist „*devide and conquer*“ häufig ein guter Ansatz. In Haskell ist die Lösung zu einem größeren Problem ebenfalls oft das „Zusammenstecken“ von Lösungen kleinerer Teilprobleme.

```
-- function composition  
(.) :: (b -> c) -> (a -> b) -> a -> c  
(.) f g = \x -> f (g x)
```

In der Informatik ist „*devide and conquer*“ häufig ein guter Ansatz. In Haskell ist die Lösung zu einem größeren Problem ebenfalls oft das „Zusammenstecken“ von Lösungen kleinerer Teilprobleme.

```
-- function composition  
(.) :: (b -> c) -> (a -> b) -> a -> c  
(.) f g = \x -> f (g x)
```

**Beispielaufgabe:** Schreibe ein Programm das eine Zeile Text von stdin liest und die Wörter in umgekehrter Reihenfolge auf stdout wieder ausgibt.

```
int main()
{
    //the array to store the entered sentence
    char *text=(char *)malloc(100*sizeof(char));
    //used for storing words
    char *temp=(char *)malloc(10*sizeof(char));
    printf("Enter the line of text\n");
    gets(text); //use gets
    int ctr=0;
    // initialize words as one because there would
    // be at least one word
    int words=1;
    int row;

    while(text[ctr]!='\0')
        if(text[ctr++]==' ')
            //count number of words by counting spaces
            words++;

    //A 2-D array of words is made
    char **word=(char **)malloc(words*sizeof(char));
    for(row=0;row<words;row++)
        word[row]=(char *)malloc(10*sizeof(char));

    int len=0;
    row=0;

    while(len<strlen(text))
    {
        sscanf(text+len,"%s",temp); //scanf from appropriate
        strcpy(word[row++],temp); //copy the extracted word
        len=len+strlen(temp)+1; //scan the next word
    }

    char *swaptemp=(char *)malloc(10*sizeof(char));
    for(row=0;row<words/2;row++)
    {
        // swap the first with last second with second
        // last and so on
        strcpy(swaptemp,word[row]);
        strcpy(word[row],word[words-row-1]);
        strcpy(word[words-row-1],swaptemp);
    }

    strcpy(text,"");

    for(row=0;row<words;row++)
    {
        strcat(text,word[row]);
        strcat(text," "); //form the new text by conca
    }

    printf("%s",text);
    getch();
}
```

```
main :: IO ()
main = do putStrLn "Please enter text: "
        str <- getLine
        (putStrLn . unwords . reverse . words) str
```

```
main :: IO ()
main = do putStrLn "Please enter text: "
         str <- getLine
         (putStrLn . unwords . reverse . words) str

getLine  :: IO String
words    :: String -> [String]
map      :: (a -> b) -> [a] -> [b]
reverse  :: [a] -> [a]
unwords  :: [String] -> String
putStrLn :: String -> IO ()
```

```
main :: IO ()
main = do putStrLn "Please enter text: "
          str <- getLine
          (putStrLn . unwords . reverse . words) str

getLine  :: IO String
words    :: String -> [String]
map      :: (a -> b) -> [a] -> [b]
reverse  :: [a] -> [a]
unwords  :: [String] -> String
putStrLn :: String -> IO ()
```

⇒ Hohes Abstraktionslevel

## Zusammenfassung:

- Thinking in Types:



## Zusammenfassung:

- Thinking in Types:
  - Starke, statische Typen: Der Compiler ist dein Freund

## Zusammenfassung:

- Thinking in Types:
  - Starke, statische Typen: Der Compiler ist dein Freund
  - Seiteneffekte nur in IO

## Zusammenfassung:

- Thinking in Types:
  - Starke, statische Typen: Der Compiler ist dein Freund
  - Seiteneffekte nur in IO
- Funktionen sind „first class citizens“

## Zusammenfassung:

- Thinking in Types:
  - Starke, statische Typen: Der Compiler ist dein Freund
  - Seiteneffekte nur in IO
- Funktionen sind „first class citizens“
- Problemlösung durch Kombination von Funktionen

## Zusammenfassung:

- Thinking in Types:
  - Starke, statische Typen: Der Compiler ist dein Freund
  - Seiteneffekte nur in IO
- Funktionen sind „first class citizens“
- Problemlösung durch Kombination von Funktionen

Fragen?