

# Fortgeschrittene funktionale Programmierung in Haskell

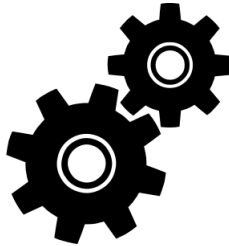
Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

## Überblick für Heute:

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

# Organisatorisches & Überlebenstipps



## 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/>

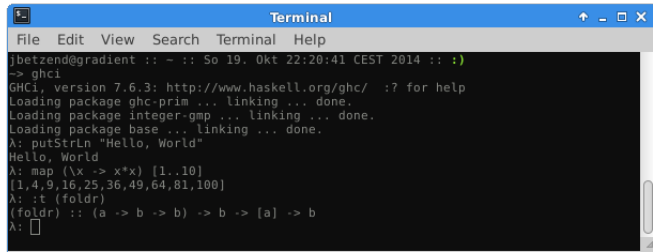
Aktuellen GHC (7.10) kriegt ihr im GZI mit dem rcinfo-Paket `ghc`

**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/>

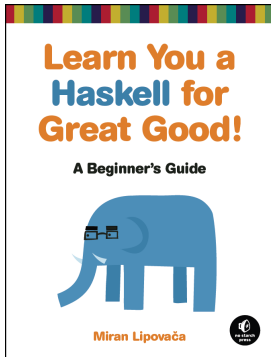
Dort findet ihr übersichtliche Zusammenfassungen der Bibliotheken, detaillierte Auflistungen der exportierten Funktionen und Datentypen und die jeweiligen Implementationen (!).

## 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
- ...

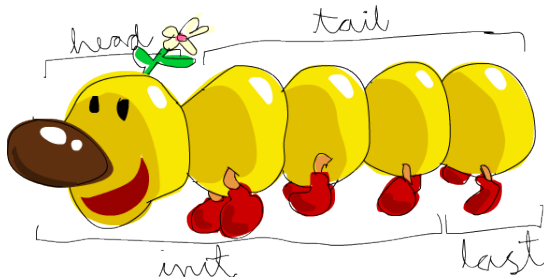
## T&R (5): LYAHFGG



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-Grundlagen





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

"Haskell is a **pure**, **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

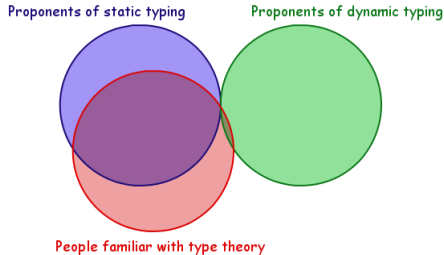
## Haskell auf einer Folie:

```
-- 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



Why static vs dynamic typing battles  
are rarely interesting

"Haskell is a pure, 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



**It's Pure Fun!**

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

Eine Funktion oder Methode wird „*pur*“ genannt, wenn sie sich auf eine mathematische Funktion reduzieren lässt. Das bedeutet:

Eine Funktion oder Methode wird „*pur*“ genannt, wenn sie sich auf eine mathematische Funktion reduzieren lässt. Das bedeutet:

- Keine Seiteneffekte  
(keinen Zustand ändern, keine Datei löschen etc.)

Eine Funktion oder Methode wird „*pur*“ genannt, wenn sie sich auf eine mathematische Funktion reduzieren lässt. Das bedeutet:

- Keine Seiteneffekte  
(keinen Zustand ändern, keine Datei löschen etc.)
- Keine destruktiven Updates von Variablen (immutability)

Eine Funktion oder Methode wird „*pur*“ genannt, wenn sie sich auf eine mathematische Funktion reduzieren lässt. Das bedeutet:

- Keine Seiteneffekte  
(keinen Zustand ändern, keine Datei löschen etc.)
- Keine destruktiven Updates von Variablen (immutability)

(Fast) alle Funktionen in Haskell sind pur.



Eine Funktion oder Methode wird „*pur*“ genannt, wenn sie sich auf eine mathematische Funktion reduzieren lässt. Das bedeutet:

- Keine Seiteneffekte  
(keinen Zustand ändern, keine Datei löschen etc.)
- Keine destruktiven Updates von Variablen (immutability)

(Fast) alle Funktionen in Haskell sind pur.

Stichwort *referential transparency*:

Ein Ausdruck ist *transparent*, wenn er jederzeit durch seinen Wert ersetzt werden kann, ohne dass sich das Verhalten des Programms ändert.

## *Referential Transparency: a case study*

```
-- pure  
zehn :: Int  
zehn = 5 + 5
```

## Referential Transparency: a case study

```
-- pure  
zehn :: Int  
zehn = 5 + 5
```

```
/* impure */  
public int zehn() {  
    return 5 + fuenf();  
}
```

```
private int fuenf() {  
    // oops...  
    raketenAbfeuern();  
    return 5;  
}
```

## Pure programming: The *pros* and *cons*:

## Pure programming: The *pros* and *cons*:

PRO:

## Pure programming: The *pros* and *cons*:

### PRO:

- Typsicherheit!  
Wir wissen per Compiler, dass  
passiert was wir wollen

## Pure programming: The *pros* and *cons*:

### PRO:

- Typsicherheit!  
Wir wissen per Compiler, dass passiert was wir wollen
- Modularität!  
Kein global state, Komponenten sind also sicher und isoliert

## Pure programming: The *pros* and *cons*:

### PRO:

- Typsicherheit!  
Wir wissen per Compiler, dass passiert was wir wollen
- Modularität!  
Kein global state, Komponenten sind also sicher und isoliert

### CON:



## Pure programming: The *pros* and *cons*:

### PRO:

- Typsicherheit!  
Wir wissen per Compiler, dass passiert was wir wollen
- Modularität!  
Kein global state, Komponenten sind also sicher und isoliert

### CON:

- Nutzlosigkeit!  
Ganz ohne Seiteneffekte haben wir keinen Grund, ein Programm auszuführen.  
  
Der Rechner wird warm, mehr passiert nicht.

## Pure programming: The *pros* and *cons*:

### PRO:

- Typsicherheit!  
Wir wissen per Compiler, dass passiert was wir wollen
- Modularität!  
Kein global state, Komponenten sind also sicher und isoliert

### CON:

- Nutzlosigkeit!  
Ganz ohne Seiteneffekte haben wir keinen Grund, ein Programm auszuführen.  
  
Der Rechner wird warm, mehr passiert nicht.

⇒ IO hat in Haskell einen eigenen Typen, damit offensichtlich ist, wo Seiteneffekte statt finden.

... can not unify (**Int**) with (**IO Int**) ...

## Pure programming: The *pros* and *cons*:

### PRO:

- Typsicherheit!  
Wir wissen per Compiler, dass passiert was wir wollen
- Modularität!  
Kein global state, Komponenten sind also sicher und isoliert

### CON:

- Nutzlosigkeit!  
Ganz ohne Seiteneffekte haben wir keinen Grund, ein Programm auszuführen.  
  
Der Rechner wird warm, mehr passiert nicht.

⇒ IO hat in Haskell einen eigenen Typen, damit offensichtlich ist, wo Seiteneffekte statt finden.

... can not unify (**Int**) with (**IO Int**) ...

⇒ Es gibt einen Unterschied zwischen *Wert* und *Berechnung*!

## Pure programming: The *dos* and *don'ts*:

## Pure programming: The *dos* and *don'ts*:

### DO:

```
-- All good! =)
main :: IO ()
main = do
  putStrLn "Hallo! Wie heißt du?"
  str <- getLine
  -- str :: String,
  -- trotzdem noch im IO-Typ
  putStrLn ("Hallo, " ++ show str)
```

## Pure programming: The *dos* and *don'ts*:

### DO:

```
-- All good! =)
main :: IO ()
main = do
  putStrLn "Hallo! Wie heißt du?"
  str <- getLine
  -- str :: String,
  -- trotzdem noch im IO-Typ
  putStrLn ("Hallo, " ++ show str)
```

### DON'T:

```
-- You're doing it wrong! >.<
pureInput :: String
pureInput = unsafePerformIO . getLine
```

## Pure programming: The *dos* and *don'ts*:

### DO:

```
-- All good! =)
main :: IO ()
main = do
  putStrLn "Hallo! Wie heißt du?"
  str <- getLine
  -- str :: String,
  -- trotzdem noch im IO-Typ
  putStrLn ("Hallo, " ++ show str)
```

### DON'T:

```
-- You're doing it wrong! >.<
pureInput :: String
pureInput = unsafePerformIO . getLine
```

⇒ Haskell's Herangehensweise an IO ist nicht für jeden intuitiv.  
Übung hilft allerdings. Mehr dazu in den Tutorien.

# Lazy Evaluation

*"Garbage collection means the programmer doesn't need to worry about the end of a value's life. Laziness means she doesn't need to worry about its beginning, either."*  
(Doaitse Swierstra)



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

*Lazy Evaluation* (a.k.a. call-by-need) bedeutet, einen Ausdruck erst dann zu berechnen, wenn er aktiv nachgefragt wird.

*Lazy Evaluation* (a.k.a. call-by-need) bedeutet, einen Ausdruck erst dann zu berechnen, wenn er aktiv nachgefragt wird.

Eine weitere Komponente ist das so genannte „sharing“ von einmal berechneten Werten, um unnötige Rechenzeit zu sparen.

*Lazy Evaluation* (a.k.a. call-by-need) bedeutet, einen Ausdruck erst dann zu berechnen, wenn er aktiv nachgefragt wird.

Eine weitere Komponente ist das so genannte „sharing“ von einmal berechneten Werten, um unnötige Rechenzeit zu sparen.

Das hat offensichtliche Vorteile:

*Lazy Evaluation* (a.k.a. call-by-need) bedeutet, einen Ausdruck erst dann zu berechnen, wenn er aktiv nachgefragt wird.

Eine weitere Komponente ist das so genannte „sharing“ von einmal berechneten Werten, um unnötige Rechenzeit zu sparen.

Das hat offensichtliche Vorteile:

- Gesteigerte Performance

*Lazy Evaluation* (a.k.a. call-by-need) bedeutet, einen Ausdruck erst dann zu berechnen, wenn er aktiv nachgefragt wird.

Eine weitere Komponente ist das so genannte „sharing“ von einmal berechneten Werten, um unnötige Rechenzeit zu sparen.

Das hat offensichtliche Vorteile:

- Gesteigerte Performance
  - Memoization

*Lazy Evaluation* (a.k.a. call-by-need) bedeutet, einen Ausdruck erst dann zu berechnen, wenn er aktiv nachgefragt wird.

Eine weitere Komponente ist das so genannte „sharing“ von einmal berechneten Werten, um unnötige Rechenzeit zu sparen.

Das hat offensichtliche Vorteile:

- Gesteigerte Performance
  - Memoization
- Unendliche Datenstrukturen

*Lazy Evaluation* (a.k.a. call-by-need) bedeutet, einen Ausdruck erst dann zu berechnen, wenn er aktiv nachgefragt wird.

Eine weitere Komponente ist das so genannte „sharing“ von einmal berechneten Werten, um unnötige Rechenzeit zu sparen.

Das hat offensichtliche Vorteile:

- Gesteigerte Performance
  - Memoization
- Unendliche Datenstrukturen
- Faule Kontrollstrukturen



# Beispiele:

# Beispiele:

Unendliche Datenstrukturen:

```
fibs :: [Int]
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

# Beispiele:

Unendliche Datenstrukturen:

```
fibs :: [Int]
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

```
fibs = [0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597 ...
```

## Beispiele:

Unendliche Datenstrukturen:

```
fibs :: [Int]
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

```
fibs = [0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597 ...
```

Konstruktstrukturen als eigene Abstraktion (statt primitiv):

## Beispiele:

Unendliche Datenstrukturen:

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fibs = [0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597 ...
```

Konstruktstrukturen als eigene Abstraktion (statt primitiv):

```
-- lazy, as by default

foo :: Int -> Int -> Int
foo x y = 2 * x
```

## Beispiele:

Unendliche Datenstrukturen:

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fibs = [0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597 ...
```

Konstruktoren als eigene Abstraktion (statt primitiv):

```
-- lazy, as by default

foo :: Int -> Int -> Int
foo x y = 2 * x

ghci> foo 5 undefined
10
```

## Beispiele:

Unendliche Datenstrukturen:

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fibs = [0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597 ...]
```

Konstruktstrukturen als eigene Abstraktion (statt primitiv):

```
-- lazy, as by default           {-# LANGUAGE BangPatterns #-}

foo :: Int -> Int -> Int          bar :: Int -> Int -> Int
foo x y = 2 * x                  bar x !y = 2 * x

ghci: foo 5 undefined
10
```

## Unendliche Datenstrukturen:

Kontrollstrukturen als eigene Abstraktion (statt primitiv):

-- lazy, as by default	{-# LANGUAGE BangPatterns #-}
foo :: Int -> Int -> Int	bar :: Int -> Int -> Int
foo x y = 2 * x	bar x !y = 2 * x
ghci> foo 5 undefined	ghci> bar 5 undefined
10	*** Exception: Prelude.undefined



## Beispiele:

Unendliche Datenstrukturen:

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fibs = [0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597 ...
```

Konstruktstrukturen als eigene Abstraktion (statt primitiv):

<pre>-- lazy, as by default</pre>	<pre>{-# LANGUAGE BangPatterns #-}</pre>
<pre>foo :: Int -&gt; Int -&gt; Int foo x y = 2 * x</pre>	<pre>bar :: Int -&gt; Int -&gt; Int bar x !y = 2 * x</pre>
<pre>ghci: foo 5 undefined 10</pre>	<pre>ghci: bar 5 undefined *** Exception: Prelude.undefined</pre>

Die Unterschiede können bei Funktionen mit Seiteneffekten subtil aber gefährlich werden.

Lazy Evaluation kann auch genau *nicht* das sein, was man braucht:

- File - IO (Datei schließen, bevor Inhalt gelesen wurde)

Lazy Evaluation kann auch genau *nicht* das sein, was man braucht:

- File - IO (Datei schließen, bevor Inhalt gelesen wurde)
- Performance (Trotz Analyse durch den Compiler)

Lazy Evaluation kann auch genau *nicht* das sein, was man braucht:

- File - IO (Datei schließen, bevor Inhalt gelesen wurde)
- Performance (Trotz Analyse durch den Compiler)
- Exception Handling

Lazy Evaluation kann auch genau *nicht* das sein, was man braucht:

- File - IO (Datei schließen, bevor Inhalt gelesen wurde)
- Performance (Trotz Analyse durch den Compiler)
- Exception Handling
- ...

Lazy Evaluation kann auch genau *nicht* das sein, was man braucht:

- File - IO (Datei schließen, bevor Inhalt gelesen wurde)
- Performance (Trotz Analyse durch den Compiler)
- Exception Handling
- ...

Für diese Fälle gibt es Möglichkeiten, den Default zu umgehen (*strictness annotation*, siehe *BangPatterns*). Für Code mit hohen Anforderungen bzgl. Performance ist das quasi unumgänglich.

Lazy Evaluation kann auch genau *nicht* das sein, was man braucht:

- File - IO (Datei schließen, bevor Inhalt gelesen wurde)
- Performance (Trotz Analyse durch den Compiler)
- Exception Handling
- ...

Für diese Fälle gibt es Möglichkeiten, den Default zu umgehen (*strictness annotation*, siehe *BangPatterns*). Für Code mit hohen Anforderungen bzgl. Performance ist das quasi unumgänglich.

Es herrscht Uneinigkeit darüber, was der Standard sein sollte.

# Problemlösen durch Zusammensetzen





"Haskell is a pure, **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 "Bitte Text eingeben: "  
        str <- getLine  
        (putStrLn . unwords . reverse . words) str
```

```
main :: IO ()
main = do putStrLn "Bitte Text eingeben: "
        str <- getLine
        (putStrLn . unwords . reverse . words) str

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

```
main :: IO ()
main = do putStrLn "Bitte Text eingeben: "
          str <- getLine
          (putStrLn . unwords . reverse . words) str

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

⇒ Thinking in Types



```
main :: IO ()
main = do putStrLn "Bitte Text eingeben: "
          str <- getLine
          (putStrLn . unwords . reverse . words) str

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

⇒ Thinking in Types

⇒ Hohes Abstraktionslevel

# Zusammenfassung:

## 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; Wert vs. Berechnung

## Zusammenfassung:

- Thinking in Types:
  - Starke, statische Typen: Der Compiler ist dein Freund
  - Seiteneffekte nur in IO; Wert vs. Berechnung
  - call-by-need & sharing
- Funktionen sind „first class citizens“

## Zusammenfassung:

- Thinking in Types:
  - Starke, statische Typen: Der Compiler ist dein Freund
  - Seiteneffekte nur in IO; Wert vs. Berechnung
  - call-by-need & sharing
- Funktionen sind „first class citizens“
- Problemlösung durch Kombination von Funktionen

Fragen?