

# Tutorium 01: Haskell Basics

---

Paul Brinkmeier

21. Oktober 2019

Tutorium Programmierparadigmen am KIT

# Heutiges Programm

---

- Haskell installieren
- Wiederholung der Vorlesung
- Aufgaben zu Haskell

# Organisatorisches

---

- paul.brinkmeier@fsmi.uni-karlsruhe.de
  - Feedback
  - Fragen zur Orga
- <https://github.com/pbrinkmeier/pp-tut>
  - Folien
  - Codebeispiele
- Bitte Laptop o.Ä. mitbringen

- ProPa hat keinen Übungsschein
- $\rightsquigarrow$  ÜBs zur eigenen Übung!
- Abgabe per Praktomat
- [https://praktomat.cs.kit.edu/pp\\_2019\\_WS/tasks](https://praktomat.cs.kit.edu/pp_2019_WS/tasks)
- Nicht-Code-Abgaben:
  - Briefkasten im Infobau-UG
  - Oder per Praktomat
  - Zur Not per Mail

- 24.03.2020, 11:00 bis 13:00
- Papier-Materialien dürfen mitgebracht werden!
- $\rightsquigarrow$  Skript, Mitschriebe, „Formelsammlung“

# Haskell

---



```
$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/
Prelude> putStrLn "Hello, World!"
Hello, World!
```

- Populärster, von der VL verwendeter Haskell-Compiler: GHC
- Interaktive Haskell-Shell: `ghci`
- Installation:
  - Windows: Installer von Haskell-Website
  - Linux: Je nach Distro `haskell-platform` oder `ghc` installieren
  - macOS: `ghcup`

```
module Maths where

add x y = x + y
sub x y = x - y

tau = 2 * pi

circumference r = tau * r
```

- Ein Haskell-Programm ist eine Folge von Funktionsdefinitionen
- Funktionen müssen keine Argumente haben

```
$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/
Prelude> :l Maths.hs
[1 of 1] Compiling Maths ( Maths.hs, interpreted )
Ok, one module loaded.
*Maths> tau
6.283185307179586
*Maths> :t tau
tau :: Double
```

- `ghci` ist ein sog. „Read-Eval-Print-Loop“
- `:l` — Modul aus Datei laden
- `:t` — Typ eines Ausdrucks abfragen

```
module Maths where

add x y = x + y
sub x y = x - y

tau = 2 * pi

circumference r = tau * r
```

- Unterschied zu C-ähnlichen: Keine Klammern/Kommata, =
- Funktionsaufruf: Selbe Syntax

# Basistypen

Wert	Typ in Java	Typ in Haskell
"Hello, World!"	String	

# Basistypen

Wert	Typ in Java	Typ in Haskell
"Hello, World!"	String	String
'x'	char	

# Basistypen

Wert	Typ in Java	Typ in Haskell
"Hello, World!"	String	String
'x'	char	Char
5	int	

# Basistypen

Wert	Typ in Java	Typ in Haskell
"Hello, World!"	String	String
'x'	char	Char
5	int	Int
9999999999999999999999999999	BigInteger	



# Basistypen

Wert	Typ in Java	Typ in Haskell
"Hello, World!"	String	String
'x'	char	Char
5	int	Int
9999999999999999999999999999999	BigInteger	Integer
3.1415927	float	

# Basistypen

Wert	Typ in Java	Typ in Haskell
"Hello, World!"	String	String
'x'	char	Char
5	int	Int
9999999999999999999999999999999	BigInteger	Integer
3.1415927	float	Float
3.141592653589793	double	

# Basistypen

Wert	Typ in Java	Typ in Haskell
"Hello, World!"	String	String
'x'	char	Char
5	int	Int
9999999999999999999999999999	BigInteger	Integer
3.1415927	float	Float
3.141592653589793	double	Double
[Tt]rue, [Ff]alse	boolean	

# Basistypen

Wert	Typ in Java	Typ in Haskell
"Hello, World!"	String	String
'x'	char	Char
5	int	Int
999999999999999999999999999999	BigInteger	Integer
3.1415927	float	Float
3.141592653589793	double	Double
[Tt]rue, [Ff]alse	boolean	Bool
$\frac{1}{3}$	X	

# Basistypen

[illegible]

```
module Maths where

add x y = x + y
sub x y = x - y

tau = 2 * pi

circumference r = tau * r
```

- Schreibt ein Modul `FirstSteps` mit folgenden Funktionen:
  - `double x` — verdoppelt `x`
  - `dSum x y` — verdoppelt `x` und `y` und summiert die Ergebnisse
  - `area r` — Fläche eines Kreises mit Radius `r`
  - `sum3 a b c` — Summiert `a`, `b` und `c`
  - `sum4 a b c d` — Summiert `a`, `b`, `c` und `d`

- `sum3`, `sum4`, `sumX` zu schreiben ist irgendwie doof

# Listen

- `sum3`, `sum4`, `sumX` zu schreiben ist irgendwie doof
- Lösung des Problems: Listen
- `[ a ]` ist der Typ einer Liste, deren Elemente von Typ `a` sind
- $\rightsquigarrow$  Listen sind homogen, nur eine Art von Element

```
module Lists where
```

```
sumL :: [ Int ] -> Int
```

```
sumL [] = 0
```

```
sumL (first : rest) = first + (sumL rest)
```

- Ein Ausdruck des Typs `[ a ]` hat genau einen von zwei Werten:
  - `[]` — die leere Liste
  - `(h : t)` — Erstes El. + Rest, mit `h :: a` und `t :: [ a ]`



- **Funktionen sind Werte**
- $\rightsquigarrow$  Funktionen haben einen Typ
- Allgemeine Form:  $x \rightarrow y$
- Beispiel: `length :: [ a ] -> Int`
  - Java: `Function<List<A>, Integer> length;`
  - C: `int (*strlen)(char *str);`

## Funktionstypen, mehrere Argumente

```
module Maths where

add x y = x + y
sub x y = x - y

tau = 2 * pi

circumference r = tau * r
```

- Funktionen sind vom Typ  $x \rightarrow y$
- Welchen Typ hat denn dann add?

## Funktionstypen, mehrere Argumente

```
module Maths where

add x y = x + y
sub x y = x - y

tau = 2 * pi

circumference r = tau * r
```

- Funktionen sind vom Typ  $x \rightarrow y$
- Welchen Typ hat denn dann `add`?  
 $\rightsquigarrow \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$
- $\rightarrow$  ist rechtsassoz.  
 $\rightsquigarrow a \rightarrow a \rightarrow a \Leftrightarrow a \rightarrow (a \rightarrow a)$

- Haskell-Funktionen sind „ge-Curry-d“
- D.h.: Jede Funktion hat exakt ein Argument
- Funktionen mit (logisch gesehen) mehreren Argumenten geben solange Funktionen zurück, bis sie ausreichend „versorgt“ sind

```
Prelude> add x y = x + y
Prelude> :t add
add :: Num a => a -> a -> a
Prelude> :t pi
pi :: Floating a => a
Prelude> max x y = if x > y then x else y
Prelude> :t max
max :: Ord p => p -> p -> p
```

- Typklassen übernehmen ähnliche Aufgaben wie Generics in Java
- Bspw. enthält `Ord` alle Typen `a`, auf denen eine Ordnung definiert ist
  - Java:

```
Prelude> add x y = x + y
Prelude> :t add
add :: Num a => a -> a -> a
Prelude> :t pi
pi :: Floating a => a
Prelude> max x y = if x > y then x else y
Prelude> :t max
max :: Ord p => p -> p -> p
```

- Typklassen übernehmen ähnliche Aufgaben wie Generics in Java
- Bspw. enthält `Ord` a alle Typen a, auf denen eine Ordnung definiert ist
  - Java: `Comparable<A>`

## Fallunterscheidung: if-then-else

```
module MaxIf where
```

```
max' x y = if x > y then x else y
```

- Einfachste Form der Fallunterscheidung
- if <Bedingung> then <WertA> else <WertB>

## Fallunterscheidung: if-then-else

```
module MaxIf where
```

```
max' x y = if x > y then x else y
```

- Einfachste Form der Fallunterscheidung
- if <Bedingung> then <WertA> else <WertB>
- Das ist nichts anderes als der ternäre Operator in den C-ähnlichen
  - <Bedingung> ? <WertA> : <WertB>



## Fallunterscheidung: Guard-Notation

```
module MaxGuard where
```

```
max' x y
```

```
  | x > y      = x
```

```
  | otherwise = y
```

- „Guard“-Notation
- Wird einfach von oben nach unten abgearbeitet
- Oft kürzer als `if a then x else if b then y else z`

## Fallunterscheidung: Guard-Notation

```
module MaxGuard where
```

```
max' x y
```

```
  | x > y      = x
```

```
  | otherwise = y
```

- „Guard“-Notation
- Wird einfach von oben nach unten abgearbeitet
- Oft kürzer als `if a then x else if b then y else z`
- `otherwise == True`

## Fallunterscheidung: Pattern Matching

```
module Bool where

xor False False = False
xor True  True  = False
xor _     _     = True
```

- Statt Variablen einfach Werte in den Funktionskopf setzen
- Mehrere Funktionsdefinitionen möglich
- Funktioniert nicht immer (bspw. bei `max`)
- Hier nützlich: `_` ignoriert Argument

# Eingebaute Funktionen

- Für Num a:
  - (+), (-), (\*), (^)
- Für Integral a:
  - div, mod
- Für Floating a:
  - (/)
- Für Bool:
  - (&&), (||)
- Für Eq a:
  - (==), (/=)
- Für Ord a:
  - (<), (<=), (>), (>=), min, max

- Für `[ a ]`:
  - `(++) :: [ a ] -> [ a ] -> [ a ]`
  - `(!!) :: [ a ] -> Int -> a`
  - `head, last :: [ a ] -> a`
  - `null :: [ a ] -> Bool`
  - `take, drop :: Int -> [ a ] -> [ a ]`
  - `length :: [ a ] -> Int`
  - `reverse :: [ a ] -> [ a ]`
  - `elem :: a -> [ a ] -> Bool`

- Für `[ a ]`:
  - `map :: (a -> b) -> [ a ] -> [ b ]`

- Für `[ a ]`:
  - `map :: (a -> b) -> [ a ] -> [ b ]`
  - `filter :: (a -> Bool) -> [ a ] -> [ a ]`

- Für `[ a ]`:
  - `map :: (a -> b) -> [ a ] -> [ b ]`
  - `filter :: (a -> Bool) -> [ a ] -> [ a ]`
  - `foldl :: (b -> a -> b) -> b -> [ a ] -> b`



- Für `[ a ]`:
  - `map :: (a -> b) -> [ a ] -> [ b ]`
  - `filter :: (a -> Bool) -> [ a ] -> [ a ]`
  - `foldl :: (b -> a -> b) -> b -> [ a ] -> b`
- Für Funktionen:
  - `(.) :: (a -> b) -> (b -> c) -> (a -> c)`
  - `($) :: (a -> b) -> a -> b`

Schreibt ein Modul Tut01 mit:

- `fac n` — Berechnet Fakultät von `n`
- `fib n` — Berechnet `n`-te Fibonacci-Zahl
- `fibs n` — Liste der ersten `n` Fibonacci-Zahlen
- `fibsTo n` — Liste der Fibonacci-Zahlen bis `n`
- `productL 1` — Berechnet das Produkt aller Einträge von `1`
- `odds` — (Unendliche) Liste aller ungeraden natürlichen Zahlen
- `evens` — (Unendliche) Liste aller geraden natürlichen Zahlen
- `squares 1` — Liste der Quadrate aller Einträge von `1`