

# Tutorium 01: Haskell Basics

---

Paul Brinkmeier

10. November 2020

Tutorium Programmierparadigmen am KIT

# Organisatorisches

---

- `pp-tut@pbrinkmeier.de`
  - Für Feedback und Fragen
- <https://github.com/pbrinkmeier/pp-tut>
  - Folien
  - Codebeispiele
  - Liste von Klausuraufgaben
- ~~Bitte Laptop o.Ä. mitbringen~~
  - Bis auf weiteres: Onlinetutorium

- ProPa hat keinen Übungsschein
- $\rightsquigarrow$  ÜBs zur eigenen Übung!
- Abgabe per Praktomat
- [https://praktomat.cs.kit.edu/pp\\_2020\\_WS/tasks](https://praktomat.cs.kit.edu/pp_2020_WS/tasks)
- Nicht-Code-Abgaben:
  - ~~Briefkasten im Infobau-UG~~
  - Per Praktomat
  - ~~Zur Not~~ per Mail

- Termin: ??.??..2020, vermutlich April/Mai
- Papier-Materialien dürfen mitgebracht werden!
- $\rightsquigarrow$  Skript, Mitschriebe, „Formelsammlung“

# Heutiges Programm

---

- Haskell installieren
- Wiederholung des Vorlesungs
- Aufgaben zu Haskell

# Haskell

---



```
$ ghci
GHCI, version 8.8.4: 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 8.8.4: 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
- :r — Modul neu 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 Sprachen: Keine Klammern/Kommata, =
- Leerzeichen als Syntax für „Funktionsaufruf“

# 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

[illegible]



# Basistypen

Wert	Typ in Java	Typ in Haskell
"Hello, World!"	String	String
'x'	char	Char
5	int	Int
9999999999999999999999999999	BigInteger	Integer
3.1415927	float	

# Basistypen

[illegible]

# Basistypen

[illegible]

# 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)` — Element + 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.  
 $\rightsquigarrow a \rightarrow a \rightarrow a \equiv a \rightarrow (a \rightarrow a)$

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

```
add3 x y z = x + y + z
<=> add3 = \x -> \y -> \z -> x + y + z

add3 15          = \y -> \z -> 15 + y + z
add3 15 10       =      \z -> 15 + 10 + z
add3 15 10 17    =              15 + 10 + 17
```

## 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 C-ähnlichen Sprachen:
  - <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

## Cheatsheet: Listen

- `[]`, `(:)`
- `(++) :: [a] -> [a] -> [a]`
- `head :: [a] -> a`
- `tail :: [a] -> [a]`
- `null :: [a] -> Bool`
- `length :: [a] -> Int`
- `isIn :: [a] -> a -> Bool`
- `elem :: a -> [a] -> Bool`
- `minimum, maximum :: Ord a => [a] -> a`
- `reverse :: [a] -> [a]`
- `take, drop :: Int -> [a] -> [a]`
- Endrekursion, Akkumulatortechnik

# Cheatsheet: Basics

- `(==) :: Eq a => a -> a -> Bool`
- `(<), (<=), (>), (>=) :: Ord a => a -> a -> Bool`
- `min, max :: Ord a => a -> a -> a`
- `type String = [Char]`
- Syntax:
  - `if ... then ... else`
  - `case ... of ...`
  - Guard-Notation, Pattern-Matching
  - Lambda-Notation
  - `where` vs. `let`
- Anonyme Funktionen

# Cheatsheet: Funktionen höherer Ordnung

- Currying, Unterversorgung
- $\lambda$ -Abstraktion, gebundene/freie Variablen
- $(.)$ ,  $\text{comp} :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$
- $\text{iter} :: (t \rightarrow t) \rightarrow \text{Integer} \rightarrow (t \rightarrow t)$
- $\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
- $\text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
- $\text{sum}$ ,  $\text{product} :: [\text{Int}] \rightarrow \text{Int}$  (per Fold implementiert)

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`

Schreibt ein Module Digits mit:

- `digits :: Int -> [Int]` — Liste der Stellen einer positiven Zahl.

Bspw.:

```
digits 42 == [4, 2]
digits 101 == [1, 0, 1]
digits 1024 == [1, 0, 2, 4]
digits 0 == [0]
digits -5 == error "need positive number"
```