

Tutorium 01: Haskell Basics

Paul Brinkmeier

2. November 2022

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
 - Für Mitarbeit im Tutorium:
<https://pad.pbrinkmeier.de/pp-tut>
 - ... und natürlich zum Experimentieren

- ProPa hat keinen Übungsschein
- \leadsto ÜBs zur eigenen Übung!
- Abgabe:
 - https://praktomat.cs.kit.edu/pp_2022_WS/tasks
 - Kasten im Infobau-UG
 - Wenn ihr die Abgabe verpasst habt auch per Mail

- Termin: 31.03.2022
- Papier-Materialien dürfen mitgebracht werden!
- \leadsto Skript, Mitschriebe, „Formelsammlung“

Heutiges Programm

- Haskell installieren
- Wiederholung der Vorlesung
- Aufgaben zu Haskell

Haskell

- Learn You a Haskell (learnyouahaskell.com)
 - Vorlesungsstoff \subseteq erste zehn Kapitel
 - Zehntes Kapitel enthält gut erklärtes „Mini-Projekt“
- 99 Haskell Problems (wiki.haskell.org)
 - Sammlung von Aufgaben mit Lösung
 - Großer Teil zu Listen
- Hoogle (hoogle.haskell.org): Dokumentation

```
$ ghci
GHCi, version 8.8.4: http://www.haskell.org/ghc/
Prelude> putStrLn "Hello, World!"
Hello, World!
```

- Von der VL verwendeter Haskell-Compiler: GHC
- Interaktive Haskell-Shell: ghci
- Installation via ghcup (haskell.org/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 Auflistung von Definitionen
- Definitionen können Konstanten (`tau`) oder Funktionen sein
- Funktionsargument schreibt man in Haskell *ohne Klammern*:
 $f\ x\ y\ z \equiv f(x, y, z)$

```
$ 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
9999999999999999999999999999999	BigInteger	Integer
3.1415927	float	

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	


```
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
- \leadsto 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**
- \leadsto 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 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 dann `add`?
 $\leadsto \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$
- \rightarrow ist rechtsassoz.
 $\leadsto a \rightarrow a \rightarrow a \equiv a \rightarrow (a \rightarrow a)$

- 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

Aufgabe: Summen

```
module Series where
```

```
squareSum [] = 0
```

```
squareSum (x:xs) = x^2 + squareSum xs
```

`squareSum xs` berechnet $\sum_{x \in xs} x^2$. Schreibt folgende Funktionen:

- `cubeSum xs`: $\sum_{x \in xs} x^3$
- `mysterySum xs`: $\sum_{x \in xs} \frac{1}{(4x+1)(4x+3)}$ (wofür ist das gut?)

Beispiele zum Testen:

- `cubeSum [0..10]` = 3025
- `mysterySum [0..10]` = 0.38702019080795513

Aufgabe: Summen mit Funktionen höherer Ordnung

- squareSum xs: $\sum_{x \in \text{xs}} x^2$
- cubeSum xs: $\sum_{x \in \text{xs}} x^3$
- mysterySum xs: $\sum_{x \in \text{xs}} \frac{1}{(4x+1)(4x+3)}$ (Konvergiert gg. $\frac{\pi}{8}$)

Hier gibt es ein gemeinsames Muster:

Aufgabe: Summen mit Funktionen höherer Ordnung

- squareSum xs: $\sum_{x \in \text{xs}} x^2$
- cubeSum xs: $\sum_{x \in \text{xs}} x^3$
- mysterySum xs: $\sum_{x \in \text{xs}} \frac{1}{(4x+1)(4x+3)}$ (Konvergiert gg. $\frac{\pi}{8}$)

Hier gibt es ein gemeinsames Muster:

$$\sum_{x \in \text{xs}} f(x)$$

Schreibt eine Funktion funcSum f xs, die dieses Muster umsetzt.
Schreibt damit neue Versionen von squareSum, cubeSum und mysterySum!

Beispiel: funcSum ($\backslash x \rightarrow x$) [0..10] = 55

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, List comprehension*

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*

- *Currying, Unterversorgung*
- *λ -Abstraktion, gebundene/freie Variablen*
- $(.)$, $\text{comp} :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$
- $\text{iter} :: (t \rightarrow t) \rightarrow \text{Integer} \rightarrow (t \rightarrow t)$
- *Funktionen sind Werte*

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 1` — Ungerade Zahlen in `1`
- `evens 1` — Gerade Zahlen in `1`
- `squares 1` — Liste der Quadrate aller Einträge von `1`

Schreibt ein Modul 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"
```

Tipps für Blatt 1

- Überlegt euch was die Funktionen ausgeben müssen, bspw.
`pow1 9 3 == pow2 9 3 == pow3 9 3 == 729,`
`root 2 100 == 10, ...`
- `error "Nachricht"` ist nützlich für Fehlerfälle
- Für `root`: Visualisiert die Intervallhalbierung auf Papier
- Für `isPrime`: Hier ist List comprehension praktisch
- Für `mergeSort`: Alle Basisfälle abdecken! Bspw.
`mergeSort [] == [], mergeSort [42] == ?`