

# Haskell

## Compiler

---

Prof. Dr. Oliver Braun

Letzte Änderung: 27.01.2020 19:45

- <https://www.haskell.org/>
- rein funktionale Sprache
- funktionale Programmierung == Programmieren mit Werten
- imperative Programmierung == Programmieren mit Zustandsänderungen
- in Haskell gibt es nur Werte == unveränderlich
  - es gibt keine Variablen!
  - es gibt keine Zuweisung!
- keine side-effects sondern z.B. IO-actions die Werte sind

- keine OOP
- keine Klassen, nur Funktionen!
- (mathematische) Funktionen bilden Werte auf andere Werte ab
- KEINE METHODEN!!!
  - eine Methode ist eine Prozedur oder eine Funktion die ein Member einer Klasse ist

- statisch typisiert
- keine impliziten oder expliziten Casts
  - aber Funktionen, die Werte umwandeln können,

statt

```
(int) 3.1415 // Java
```

einfach

```
toInteger 3.1415 -- Haskell
```

- Typinferenz
  - Typen müssen nicht angegeben werden, sondern können vom Haskell Typechecker berechnet (inferiert) werden

# Haskell ist lazy

- berechnet wird nur was wirklich benötigt wird
  - Bedarfsauswertung (lazy evaluation)
  - im Gegensatz zu eager evaluation
- bekannt aus Java/C/.. bei booleschen Ausdrücken

```
if (true || bla = blub + 19 * 27) // Java o.ä.
```

...

```
if (x.hasMore() && y = x.getMore()) // typisches C/C++
```

...

- in Haskell ist das der Default, z.B. wird hier `fibonacci 100000000` **nie** berechnet

```
fst (100, fibonacci 100000000) -- fst (a,b) = a
```

- unendliche Listen können genutzt werden, z.B. die ersten `n` ungeraden

## Wer nutzt Haskell? (Auswahl)

- Alcatel-Lucent: Software Radio Systems in soft realtime
- AT&T: Network Security Überwachung
- Chordify
- viele Banken (Credit Suisse, Deutsche Bank, ...) zur Analyse
- Ericsson AB: digital signal processing
- Facebook: Manipulation von PHP-Code, Zwischenschicht, ...
- Galois, Inc. kritische Software, Embedded Systems, HaskellVM
- Google: IT-Infrastruktur-Management
- Intel: Research on multicore parallelism at scale
- Microsoft: Bond (production serialization system)

```
main = putStrLn "Hello World!"
```

- der “Wert” `main` hat den Typ `IO ()`
  - `()` entspricht `void` in Java... und wird Unit gesprochen
  - `IO` ist wie ein Container und eine Annotation gleichzeitig
  - würde in Java so geschrieben: `IO<void>`
- Signatur steht üblicherweise einfach über Implementierung

```
main :: IO ()
```

```
main = putStrLn "Hello World!"
```

- “richtige” IDE fehlt (aber wird auch nicht so sehr benötigt)
- Plugins für IDEs und Editoren: <https://wiki.haskell.org/IDEs>
  - Empfehlung: [Atom](#) mit Haskell-Plugins
- [The Haskell Cabal](#):
  - Common Architecture for Building Applications and Libraries
- [The Haskell Tool Stack](#)
- [Haskell Platform](#)
  - Haskell with batteries included
  - enthält den **Glasgow Haskell Compiler**, Stack, Cabal, ...
  - ist auf den Laborrechnern unter Windows installiert, coden dann z.B. mit UltraEdit



- Haskell Code kann sich sehr imperativ anfühlen:

```
main = do
  putStrLn "What is your name?"
  name ← getLine
  putStrLn ("Hello " ++ name ++ "!")
```

- Achtung: Das Layout (die Einrückung) hat eine Semantik und ist wichtig (ähnlich Python)
  - deshalb ist eine sehr große Fehlerquelle das Mischen von Leerzeichen und Tabulatoren
- es gibt auch **print**, aber das verwandelt einen Wert erst in einen String

- Haskell wird in Maschinencode kompiliert
- kompilierter Code kann in einen Interpreter geladen werden
- gesteuert werden kann alles über Cabal und Stack, z.B.
  - automatisches Herunterladen und Installieren von Dependencies
  - Testen, Benchmarking, ...
  - Doku mit Haddock erzeugen

# Funktionen in Haskell

- add in Java

```
int add(int a, int b) {  
    return a + b;  
}
```

- add in Haskell

```
add :: Integer → Integer → Integer  
add a b = a + b
```

- die Typsignatur kann auch weg gelassen werden
- Haskell hat **keine** Parameterlisten
  - Parameter werden *currysiert*, durch Leerzeichen getrennt, hintereinander geschrieben
- kein **return**

## Doubles addieren

- damit **Doubles** addiert werden können, ändern wir die Typsignatur

```
add :: Double → Double → Double
```

```
add a b = a + b
```

- jetzt können aber nur noch **Doubles** addiert werden!
- `add 1 2` funktioniert, weil `1` und `2` gültige Literale für **Doubles** sind

- wir definieren `add` ohne Typsignatur

```
add a b = a + b
```

- plötzlich können `Integer` und `Doubles` addiert werden!?!
- der Typ muss in Haskell aber immer eindeutig sein!

- Lösung: In Haskell können mehrere Typen zu einer **Typklasse** zusammen gefasst werden
- Beispielsweise enthält u.a. die Typklasse **Num** die Typen **Integer** und **Double**
- und der Typinferenzmechanismus inferiert den allgemeinsten Typ

```
add :: Num a => a -> a -> a
```

```
add a b = a + b
```

- damit ist **add** quasi automatisch für alle Typen der Typklasse **Num** überladen

- eigentlich hat jede Haskell-Funktion eigentlich nur einen Parameter

```
add a b = a + b
```

- `add` hat einen Parameter `a` und gibt als Ergebnis eine Funktion zurück
- `add 1 2` ist eigentlich `(add 1) 2`
- Funktionen können auch partiell angewendet werden:

```
addFive = add 5
```

- mit einem  $\lambda$  kann der Parameter auf die andere Seite des Gleichheitszeichens geschrieben werden

- ohne  $\lambda$ :

```
inc x = x + 1
```

- mit  $\lambda$ :

```
inc = \x → x + 1
```

- add mit  $\lambda$ s

```
add a = \b → a + b
```

```
add = \a → \b → a + b
```

```
add = \a b → a + b
```



- Arithmetisch

```
3 + 2 * 6 / 3 == 3 + ((2*6)/3)
```

- Logik

```
True || False == True
```

```
True && False == False
```

```
True == False == False
```

```
True /= False == True
```

- Potenzieren

```
x^n      n muss Word, Int oder Integer sein
```

```
x**y     y muss Float oder Double
```

- `Integer` ist beliebig groß (so wie `BigInt` bei Java)

```
4^78
```

```
91343852333181432387730302044767688728495783936
```

- und sogar eingebaute Rationale Zahlen ;-)

```
$ ghci
```

```
....
```

```
Prelude> :m Data.Ratio
```

```
Data.Ratio> (11 % 15) * (5 % 3)
```

```
11 % 9
```

<code>[]</code>	<code>=</code>	leere Liste
<code>[1,2,3]</code>	<code>=</code>	Liste von Zahlen
<code>["foo","bar","baz"]</code>	<code>=</code>	Liste von Strings
<code>1:[2,3]</code>	<code>=</code>	<code>[1,2,3]</code> , ( <code>:</code> ) vorne ein Element anfügen
<code>1:2:[]</code>	<code>=</code>	<code>[1,2]</code>
<code>[1,2] ++ [3,4]</code>	<code>=</code>	<code>[1,2,3,4]</code> , ( <code>++</code> ) konkatenieren
<code>[1,2,3] ++ ["foo"]</code>	<code>=</code>	ERROR String $\neq$ Integral
<code>[1..4]</code>	<code>=</code>	<code>[1,2,3,4]</code>
<code>[1,3..10]</code>	<code>=</code>	<code>[1,3,5,7,9]</code>
<code>[2,3,5,7,11..100]</code>	<code>=</code>	ERROR! I am not so smart!
<code>[10,9..1]</code>	<code>=</code>	<code>[10,9,8,7,6,5,4,3,2,1]</code>

- alle Elemente müssen den selben Typ haben
- kein Subtyping wie in Java!

Strings sind Listen von Zeichen:

```
'a' :: Char
```

```
"a" :: [Char]
```

```
"" = []
```

```
"ab" = ['a','b'] = 'a':"b" = 'a':['b'] = 'a':'b':[]
```

```
"abc" = "ab"++"c"
```

- Listen von Zeichen reichen uns.
- in “Real-World-Code” ist das zu langsam, Lösung:
  - `Data.Text` oder `Data.ByteString`
  - überladene String-Literale

- der Typ eines Tupels ist  $(a, b)$ .
- die Komponenten eines Tupels können verschiedene Typen haben
- Beispiele

```
(2, "foo")
```

```
(3, 'a', [2,3])
```

```
((2, "a"), "c", 3)
```

```
fst (x,y) == x
```

```
snd (x,y) == y
```

```
fst (x,y,z) == ERROR: fst :: (a,b) -> a
```

```
snd (x,y,z) == ERROR: snd :: (a,b) -> b
```

## Infix- und Präfixnotation

- der Operator  $\wedge$  ist infix notiert

```
square :: Num a => a -> a
```

```
square x = x^2
```

- durch die Verwendung von Klammern wird er zu eine Funktion, die präfix geschrieben wird:

```
square' x = (^) x 2
```

- er kann auch partiell angewendet und geklammert präfix geschrieben werden:

```
square'' x = (^2) x
```

- schließlich kann auch das x entfernt werden ( $\eta$ -Reduktion):

```
square''' = (^2)
```

## Infix- und Präfixnotation (2)

- umgekehrt werden Funktionen präfix geschrieben, z.B.

```
elem 3 [1..10]
```

- durch Verwendung von Backticks können “zweistellige” Funktionen aber auch Infix geschrieben werden, z.B.

```
3 `elem` [1..10]
```

```
19 `mod` 3
```

- das ist oft flüssiger lesbar

## Keine Kontrollstrukturen

- nachdem eine Funktion aus nur einem Ausdruck besteht, gibt es keine Kontrollstrukturen
- nachdem es keinen veränderlichen Zustand gibt, kann es auch keine Schleifen geben
  - Probleme rekursiv lösen
- es gibt ein `if` das dem ternären Operator `?:` entspricht, z.B.

```
absolute :: (Ord a, Num a) => a -> a
absolute x = if x >= 0 then x else -x
```

Achtung: Es gibt kein `if` ohne `else`!

- Fallunterscheidungen können aber auch mit sog. Guards programmiert werden:

```
absolute' x
```



## Pattern Matching

- kann als Verallgemeinerung von `switch-case` gesehen werden
- statt

```
sumOfTuple :: (Int, Int) → Int
sumOfTuple tuple = fst tuple + snd tuple
```

kann das Tupel (oder eine andere Datenstruktur) gleich links vom = zerlegt werden:

```
sumOfTuple (x,y) = x + y
```

- genauso mit Listen (oder Strings)

```
firstElementOrNull :: [Integer] → Integer
firstElementOrNull [] = 0 -- leere Liste
    -- erstes Element : Restliste
firstElementOrNull (x:xs) = x
```

## Pattern Matching zur Fallunterscheidung und rechts

- verschiedene Pattern werden zeilenweise ausprobiert, das erste das “matcht” wird genutzt:

```
helloWorldI18N :: String → String
helloWorldI18N "de" = "Hallo Welt!"
helloWorldI18N "en" = "Hello world!"
helloWorldI18N lang = "Unknown language: " ++ lang
```

- mit `case of` kann Pattern Matching an einer beliebigen Stelle genutzt werden

```
helloWorldI18N :: String → String
helloWorldI18N lang = case lang of
    "de" → "Hallo Welt!"
    "en" → "Hello world!"
    -    → "???"
```

```
greetI18N :: String → String → String
greetI18N "" lang = case lang of
    "de" → "Kein Name?"
    "en" → "No name?"
greetI18N name lang = "H" ++ (case lang of
    "de" → "a"
    "en" → "e"
    _    → "?") ++ "llo " ++ name
```

## Zwischenergebnisse definieren

- entweder mit `let ... in` oder mit `where`

```
weirdCalculation :: Int → Int → Int
weirdCalculation a b =
    let c = 4 * a
        d = 23 `mod` b + 2
    in c - d
```

oder

```
weirdCalculation :: Int → Int → Int
weirdCalculation a b = c - d
    where c = 4 * a
          d = 23 `mod` b + 2
```

- Achtung: alle Zeilen in dem `let` oder `where` müssen in der selben Spalte beginnen