

Hallo Haskell

Funktionale Programmierung

Prof. Dr. Oliver Braun

Letzte Änderung: 15.10.2018 07:06

- **der** Haskell-Compiler ist der [GHC](#)
- Sie [installieren](#) ihn als Teil der [Haskell Platform](#)
- oder Sie installieren nur [Stack](#) und lassen ihn anschließend von Stack installieren
- in den Laboren ist die Haskell Platform unter Windows installiert
- das Arbeiten mit Interpreter und Compiler, sowie Entwicklungstools, lernen Sie im ersten Praktikumstermin

- die **Pre**lude ist eine Bibliothek von Standardfunktionen
- sobald Sie den GHCi starten werden diese automatisch geladen
- auch in jedem Modul sind diese verfügbar
- den Inhalt der **Pre**lude finden Sie zusammen mit der Info über das **base**-Package unter <https://www.stackage.org/package/base>
 - **Pre**lude aus **base** in der Version 4.11.1.0 z.B. unter <https://www.stackage.org/haddock/lts-12.10/base-4.11.1.0/Prelude.html>

Hello World!

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

- Aufruf als beliebige Funktion aus dem GHCi:

```
Prelude> :l Main.hs
```

```
...
```

```
*Main> main
```

```
Hello World
```

- Ausführen als Script in der Shell/Cmd:

```
$ runhaskell Main.hs
```

```
Hello World!
```

- Compilieren und Ausführen

```
$ ghc Main
```

```
$ ./Main
```

```
sayHello :: String → IO ()  
sayHello x = putStrLn ("Hallo, " ++ x ++ "!")
```

- Aufruf im GHCi:

```
Prelude> :l Main.hs  
...  
*Main> sayHello "Oliver Braun"  
Hallo Oliver Braun!
```

- alles in Haskell ist
 - ein **Ausdruck** (*expression*) oder
 - eine **Deklaration** (*declaration*)
- Ausdrücke
 - werden berechnet und haben ein Ergebnis (*evaluated*)
- Deklarationen
 - geben einem Ausdruck einen Namen

- ein Ausdruck ist in Normalform wenn er nicht weiter evaluiert werden kann
- Beispiel
 - die Normalform von $1 + 1$ ist 2
 - d.h. der Ausdruck $1 + 1$ ist **reduzierbar** (*reducible*)
 - ein reduzierbarer Ausdruck (*reducible expression*) heißt auch **redex**

- Funktionen sind Ausdrücke
- Funktionen in Haskell entsprechen Funktionen in der Mathematik
- eine Funktion ist ein Ausdruck der auf ein Argument angewendet werden kann und immer ein Ergebnis liefert
- weil Funktionen ausschließlich aus Ausdrücken bestehen, reduzieren Sie bei den selben Argumenten immer zum selben Ergebnis
- wie im λ -Kalkül haben Funktionen immer nur ein Argument
- wenn es so aussieht als ob mehrere Argumente verwendet werden, wird tatsächlich eine Reihe von Funktionen, die immer nur ein Argument haben, hintereinander geschaltet
 - das heißt **currying**, benannt nach *Haskell B. Curry*


```
triple x = x * 3
```

- `triple` ist der Name der Funktion die wir definieren
 - **muss** mit einem Kleinbuchstaben beginnen!
- `x` ist der Parameter der Funktion
- `=` wird genutzt um Werte und Funktionen zu definieren.
 - **Achtung:** Das ist kein Zuweisungsoperator, weil es in Haskell keine Zuweisung gibt. Das `=` entspricht dem `=` in der Mathematik
- hinter dem `=` steht der Rumpf der Funktion: **ein** Ausdruck

Groß- und Kleinschreibung!

- Haskell verwendet Layout (ähnlich Python) zur Strukturierung
- Groß- und Kleinschreibung von Bezeichnern hat eine Semantik und ist zwingend \rightsquigarrow Compilerfehler!
- Funktionsbezeichner und Bezeichner von Variablen **müssen** mit einem Kleinbuchstaben beginnen
 - üblicherweise zusätzlich Verwendung von *Camelcase*
 - ' ist ein zulässiges Zeichen in einem Bezeichner (nicht erstes Zeichen)
 - es ist üblich Varianten von Funktionen mit ' zu unterscheiden,
 - z.B. `sum`, `sum'`, `sum''`
 - wie in der Mathematik üblich
 - gesprochen *prime*, also *sum-prime* und *sum-prime-prime*

- Haskell wertet die Ausdrücke **nicht-strikt** (*nonstrict*) aus
 - wird auch **lazy evaluation** oder **Bedarfsauswertung** genannt
- Sie kennen das von booleschen Ausdrücken in Java
- d.h. ein Ausdruck wie `triple 2` wird zunächst gar nicht ausgewertet
 - erst wenn der Wert benötigt wird (z.B. weil er ausgegeben werden soll), wird folgendermaßen ausgewertet:

`triple 2` \rightsquigarrow `2 * 3` \rightsquigarrow `6`

Weak Head Normal Form (WHNF)

- Haskell wertet Ausdrücke nicht komplett aus
- Beispiel: $f\ x = (1, 2 + x)$
- wenden wir nun f auf 2 an, bleibt es zunächst unausgewertet
- wird die Auswertung benötigt, z.B. durch den Ausdruck $fst\ (f\ 2)$, so wird $f\ 2$ nur folgendermaßen ausgewertet

$$f\ 2 \rightsquigarrow (1, 2 + 2)$$

- damit kann dann ausgewertet werden:

$$fst\ (f\ 2) \rightsquigarrow fst\ (1, 2 + 2) \rightsquigarrow 1$$

- d.h. $2 + 2$ wird in dem Fall **niemals** berechnet
- $(1, 2 + 2)$ ist die WHNF von $f\ 2$

- in Haskell können Sie eigene Funktionen und eigene Operatoren definieren
- Funktionsbezeichner werden präfix, Operatoren infix geschrieben
 - z.B. `add 2 3` oder `2 + 3`
- es ist aber auch möglich Funktionsbezeichner wie Operatoren zu nutzen:
 - z.B. `2 `add` 3`
 - das ist üblich, weil oft schöner/natürlicher
- und auch Operatoren können vorangestellt werden:
 - z.B. `(+) 2 3`
 - das ist notwendig, z.B. für partielle Anwendung

- für Operatoren ist sinnvollerweise eine Assoziativität und ein Vorrang definiert (sonst müssten wir immer Klammern)
 - im GHCi können wir das einfach abfragen:

```
Prelude> :info (*)
...
infixl 7 *
Prelude> :info (+) (-)
...
infixl 6 +
Prelude> :info (^)
infixr 8 ^
```

- **infixl** steht für Links-, **infixr** für Rechtsassoziativität
- für den Vorrang gilt, je größer die Zahl zwischen 0 und 9 umso höher der Vorrang

```
module Fun where
```

```
x = 10 * 5 + y
```

```
myResult = x * 5
```

```
y = 10
```

- Modulnamen **müssen** mit einem Großbuchstaben beginnen
 - weiter Camelcase
- Modul **Fun** in Datei **Fun.hs**
- ohne expliziten Modulname \Rightarrow implizit **Main**
 - **main**-Funktion muss im Modul **Main** sein
 - Dateiname kann abweichen

Layout und Tabs (2)

- Haskell verwendet Layout statt geschweiften Klammern u.ä. um den Code zu strukturieren
- Grundregel: Alles was weiter eingerückt ist, als der Beginn eines Ausdrucks gehört dazu
 - Beispiel:

```
f x =  
  x  
    +  
  2  
g x = x *  
  x
```

- an anderen Stellen **muss** man exakt gleich weit einrücken
- **Verwenden Sie KEINE Tabs in Haskell Source Code Files**
- konfigurieren Sie am Besten Ihren Editor so, dass er Tabs automatisch durch Leerzeichen ersetzt

Arithmetische Operatoren

- `+`, `-`, `*`
- `/` nur für Gleitpunktzahlen
- `div` und `mod` nur für ganze Zahlen
- `rem` und `quot` nur für ganze Zahlen
- `div` rundet ab, `rem` rundet in Richtung 0
- Beispiele:

```
Prelude> div (-5) 2
```

```
-3
```

```
Prelude> mod (-5) 2
```

```
1
```

```
Prelude> quot (-5) 2
```

```
-2
```

```
Prelude> rem (-5) 2
```

```
-1
```

- Gesetze

$$(\text{quot } x \ y) * y + (\text{rem } x \ y) = x$$

$$(\text{div } x \ y) * y + (\text{mod } x \ y) = x$$

- zunächst kein Problem

```
Prelude> -42
```

```
-42
```

- aber innerhalb eines Ausdrucks

```
Prelude> 84 + -42
```

```
<interactive>:29:1: error:
```

```
  Precedence parsing error
```

```
    cannot mix '+' [infixl 6] and prefix '-' [infixl 6]
```

```
    in the same infix expression
```

- Lösung

```
Prelude> 84 + (-42)
```

```
42
```

- manche Ausdrücke müssen geklammert werden, z.B.

```
Prelude> (\x → x^2) 2 + 2
```

```
6
```

```
Prelude> (\x → x^2) (2 + 2)
```

```
16
```

- statt $(\backslash x \rightarrow x^2)$ kann übrigens kürzer einfach $(^2)$ geschrieben werden, also

```
Prelude> (^2) (2 + 2)
```

```
16
```

- das heißt *sectioning*

- es gibt eine Funktion

```
Prelude> :info ($)
($) :: (a -> b) -> a -> b
infixr 0 $
```

- mit der Definition

```
f $ a = f a
```

- damit z.B. möglich

```
(^2) $ 2 + 2
(2^) $ (+2) $ 3*2
```

Lokale Definitionen mit **where**

- Beispiele

```
printInc n = print plusTwo
  where plusTwo = n + 2
```

```
f x = y + z
  where y = 2^x * x
        z = y + x
```

- in **f** müssen die Definitionen von **y** und **z** exakt untereinander stehen
- komplexere Ausdrücke sollten zur Übersichtlichkeit in kleinere aufgeteilt werden
- wenn Teilausdrücke mehrfach in einem Ausdruck vorkommen, sollten diese nur einmal geschrieben und **nur einmal berechnet** werden

- statt `where` kann auch `let ... in` verwendet werden:

```
printInc n = let plusTwo = n + 2
              in print plusTwo
```

```
f x =
  let y = 2^x * x
      z = y + x
  in y + z
```

- in `f` müssen die Definitionen exakt untereinander stehen
- es gibt (später) noch einen anderen Unterschied zwischen `where` und `let ... in`