# 2 Haskell

## 2.1 Einführung

### Literatur

*Haskell-Webseite*
`https://www.haskell.org/`

*Haskell-Wiki*
`https://wiki.haskell.org/`

*Hoogle*
`https://hoogle.haskell.org/`

Graham Hutton
*Programming in Haskell*,
Cambridge University Press, 2016.

Richard Bird
*Thinking Functional with Haskell*,
Cambridge University Press, 2015.

Miran Lipovaca
*Learn You a Haskell for Great Good!: A Beginner's Guide*,
No Starch Press, 2011.
Online verfügbar: `http://learnyouahaskell.com/`

### Funktionale Programmierung

- ist eine Methode Programme zu erstellen aus Funktionen und deren Anwendung und nicht aus Anweisungen und deren Ausführung.
- benutzt einfache mathematische Notationen, die es erlauben Probleme eindeutig, kurz und präzise zu beschreiben.
- hat eine einfache mathematische Basis, die die Anwendung von Methoden der Algebra auf die Eigenschaften von Programmen unterstützt.

(*frei übersetzt nach Richard Bird, Thinking Functional with Haskell*)

Es gibt verschiedene funktionale Programmiersprachen, eine davon ist **Haskell** (benannt nach dem amerikanischen Logiker Haskell B. Curry), die in dieser Veranstaltung eingesetzt wird.

### GHC

**Glasgow Haskell Compiler (GHC)** ist der state-of-the-art, open-source Haskell-Compiler und bietet eine interaktive Umgebung für Entwicklung und Test.

Im Pool sind die Programmm **`ghc`** und **`ghci`** verfügbar.

GHCi ist eine interaktive Haskell-Umgebung. Haskell-Ausdrücke können direkt eingegeben werden, werden ausgewertet und das Ergebnis wird ausgegeben.

Außerdem ermöglicht GHCi das Kompilieren und Laden von Quelltext, um ihn zu testen, sowie das Einbinden von Modulen und das Ausgeben von Informationen über Funktionen, Typklassen, Datentypen und Module.

Startet man **`ghci`** erhält man folgende Ausgabe.

```
$ ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/
:? for help
Prelude>
```

Das Prompt **`Prelude`** gibt an, dass die Standardbibliothek mit grundlegenden Funktionen, sowie vordefinierten Typen und Werten geladen ist.

GHCi kann man als interaktive Testumgebung benutzen.

```
Prelude> 7+5
12
Prelude> sqrt 30.25
5.5
```

Beendet wird GHCi mit **`:quit`** oder **`:q`**.

```
Prelude> :q
Leaving GHCi.
```

## 2.2 Funktionen und Operatoren

### Funktion

In der Mathematik ist eine **Funktion** eine Relation (Beziehung) zwischen zwei Mengen, die jedem Element (Funktionsargument) der einen Menge (Definitionsbereich) genau ein Element (Funktionswert) der anderen Menge (Wertebereich) zuordnet.

Ist $X$ der Definitionsbereich und $Y$ der Wertebereich, dann schreibt man

- $f : X \to Y$ für die Funktion (Deklaration)
- $f(x)$ für den Funktionswert aus $Y$, der dem Funktionsargument $x$ aus $X$ von der Funktion zugeordnet wird (Definition).

Die Funktion $f$ bildet ein Argument $x$ aus $X$ auf einen Wert $f(x)$ aus $Y$ ab.

### Haskell Typen

In Haskell spricht man nicht von Mengen, sondern von **Typen**.

Typen sind Mengen von Elementen mit bestimmten Eigenschaften.

Beispiele

| | |
|---|---|
| `Float` | Gleitkomma-Zahlen mit einfacher Genauigkeit |
| `Double` | Gleitkomma-Zahlen mit doppelter Genauigkeit |
| `Int` | beschränkte ganze Zahlen |
| `Integer` | unbeschränkte ganze Zahlen |
| `Bool` | Wahrheitswerte (`True`, `False`) |
| `Char` | Aufzählungstyp (nicht negative ganze Zahlen), dessen Werte Zeichen repräsentieren |
| `[`*Type*`]` | (beliebig lange) Listen mit Werten vom Typ *Type* z.B. `[Bool]` |
| `String` | Zeichenketten, ist Platzhalter für `[Char]` |
| `(`*TypeA*`,` *TypeB*`)` | Paare (2-Tupel) mit Typen *TypeA* und *TypeB* z.B. `(String, Int)` |
| *TypeA* `->` *TypeB* | Funktionen mit *TypeA* als Definitionsbereich und *TypeB* als Wertebereich z.B. `Float -> Float` |

Bemerkung

- Der Typ **Int** umfasst mindestens das Intervall $[-2^{29}, 2^{29} - 1]$.
- Die Zahlenformate **Float** und **Double** entsprechen in Zahlenbereich und Genauigkeit mindestens den Gleitkommazahlen des IEEE-754-Standard.
- Die Werte des Aufzählungstyps **Char** repräsentieren Unicode-Zeichen (ISO/IEC 10646). Das ist eine Erweiterung der Latin-1 (ISO 8859-1) Zeichenmenge (die ersten 256 Zeichen), die wiederum eine Erweiterung der ASCII Zeichenmenge (die ersten 128 Zeichen) ist.

### Haskell Funktion

Eine Funktion **f** bildet ein Argument vom Typ **X** auf einen Wert vom Typ **Y** ab.

In Haskell-Notation wird diese Deklaration wie folgt ausgedrückt.

```
f :: X -> Y
```

Beispiele

```
sqrt    :: Float -> Float
first   :: (String, Int) -> String
second  :: (String, Int) -> Int
not     :: Bool -> Bool
and     :: [Bool] -> Bool
logBase :: Float -> Float -> Float
```

In Haskell kann man

```
f x
```

für die Anwendung der Funktion **f** auf das Argument **x** schreiben.

Beispiele

```
sqrt 25.0
not True
and [True, False, True]
logBase 2 10
```

Bemerkung

- `logBase` bildet ein Argument vom Typ `Float` auf eine Funktion ab. Diese Funktion wiederum bildet ein `Float`-Argument auf einen `Float`-Funktionswert ab.
- Das entspricht dem, was man aus der Mathematik kennt, die Funktionen $\log_2$ und ln entsprechem dem Wert der Haskell Funktionsaufrufe `logBase 2` und `logBase e`.
- `e` ist in diesem Fall eine konstante Funktion.

```
e :: Float
```

### Haskell Operatoren

Ein Spezialfall von Funktionen sind Operatoren.

Mathematisch ist ein Operator eine Funktion

$$op : X \to [Y \to Z]$$

mit

$$[Y \to Z] = \{f \mid f : Y \to Z\}$$

für dessen Anwendung sowohl die Präfix- als auch die Infix-Schreibweise

$$\underbrace{op(x)}_{\in [Y \to Z]}(y) \;=\; x \; op \; y \;\in Z \quad \text{für alle } x \in X, y \in Y$$

verwendet werden kann.

In Haskell wird aus einer Funktions- eine Operatordeklaration, indem der Bezeichner in rund Kammern eingeschlossen wird.

```
(op) :: X -> Y -> Z
```

Die Anwendung des Operators kann entweder präfix (Operator in runden Klammern) oder infix erfolgen.

```
(op) x y
x op y
```

Bemerkung

- In Haskell gelten die Zeichen `!#$%&*+./<=>?@^|-~:`) als Symbole, aber der Unterstrich _ ist kein Symbol.
- Bezeichner für Operatoren dürfen ausschließlich Symbole enthalten.
- Funktionsnamen dürfen keine Symbole enthalten, nur Zeichen, Ziffern und den Unterstrich.

Wichtige Operatoren:

| | |
|---|---|
| `+`, `-`, `*`, `/` | Arithmetik |
| `==`, `/=` | Gleichheit/Ungleichheit |
| `<`, `<=`, `>`, `>=` | Vergleiche |
| `&&`, `\|\|` | Verknüpfung von Wahrheitswerten |

Beispiele

```
7+5
(*) 6 7
(3+4)*8
3 /= 5
3 >= 5
True || False
(&&) True False
```

Bemerkung

Durch runde Klammern in Ausdrücken kann man die Auswertungsreihenfolge beeinflussen.

### Funktionen/Operatoren definieren

In der Mathematik besteht eine Funktionsdeklaration aus Angabe von Definitions- und Wertebereich.

Die Funktionsdefinition beschreibt wie jedes Element des Definitionsbereichs auf ein Element des Wertebereiches abgebildet wird.

In der Regel werden dazu bereits vorher definierte Operatoren und Funktionen verwendet.

$$f : \mathbb{Z} \to \mathbb{Z}$$
$$f(x) = x^2 + x$$

In Haskell macht man das genauso. Man kann z.B. in **prelude** definierte Operatoren und Funktionen verwenden.

```
f :: Int -> Int
f x = x * x + x
```

Bemerkung

Das Standard-Modul **prelude** wird per default in alle Haskell-Module importiert und enthält viele nützliche Operatoren, Funktionen und Definitionen.

Die Möglichkeit Funktionen direkt durch andere Funktionen zu definieren besteht in Haskell ebenfalls.

$$f : \mathbb{R} \to \mathbb{R}$$
$$g : \mathbb{R} \to \mathbb{R}$$
$$f = g$$

Beispiel

```
log2 :: Float -> Float
log2 = logBase 2
```

Erinnerung

**logBase** bildet ein Argument vom Typ **Float** auf eine Funktion **Float->Float** ab.

Die Deklaration eines Operators **op** erfolgt immer über die Deklaration der zugehörigen Funktion **(op)**.

Bei der Definition hat man die Wahl zwischen Infix- und Präfixdarstellung.

Beispiel

Implikation über dem booleschen Körper.

$$\Rightarrow : \ \mathbb{F}_2 \to [\mathbb{F}_2 \to \mathbb{F}_2]$$
$$a \Rightarrow b \ = \ \neg a \vee b$$

Präfix

```
(==>) :: Bool -> Bool -> Bool
(==>) a b = not a || b
```

Infix

```
(==>) :: Bool -> Bool -> Bool
a ==> b = not a || b
```

### Konstante Funktionen

Konstante Funktionen werden durch explizite Angabe des Werts definiert.

```
-- euler number
e :: Float
e = 2.71828182845904523536028874
```

Oder mit Hilfe bereits vorher definierter Funktionen (z.B. aus `prelude`).

```
-- euler number
e :: Float
e = exp 1
```

Bemerkung

`--` leitet einen Zeilenkommentar ein, d.h. bis zum Ende der Zeile wird alles Nachfolgende ignoriert.

### GHCi Kommandos

GHCi verarbeitet Eingaben zeilenweise, um mehrzeilige Eingaben zu verarbeiten, z.B. um Funktionen, Operatoren, etc. zu definieren, kann man mit dem Kommando `:{` einen *multiline block* öffnen und mit `:}` schließen.

```
Prelude> :{
Prelude| f :: Int -> Int
Prelude| f x = x * x + x
Prelude| :}
Prelude> f 5
30
```

Das Kommando **:type** oder **:t** gibt Auskunft über den Typ des nachfolgenden Ausdrucks.

```
Prelude> :t 'A'
'A' :: Char
Prelude> :t sqrt
sqrt :: Floating a => a -> a
```

**'A'** ist eine konstante Funktion, die eine Wert vom Typ **Char** zurückliefert.

**sqrt** ist nicht für einen bestimmten Typ definiert, sondern eine Funktion **a -> a**, wobei für den Typ **a** gelten muss, dass es sich um einen Gleitkomma-Typ handelt.

Tatsächlich ist es etwas komplizierter. **Floating** ist eine *Typklasse*. Typklassen werden später noch behandelt.

Eine Methode um selbst definierte Funktionen, Operatoren, etc. in GHCi zu benutzen ist, die Definitionen in einer Datei, üblicherweise mit der Endung **.hs**, zu speichern und in GHCi zu laden.

Eine Datei wird geladen, indem der Dateiname beim Starten von **ghci** auf der Kommandozeile übergeben wird.

Alternativ kann man im laufenden GHCi mit den Kommandos

```
:load [file]
:l [file]
```

die Datei *file* laden.

Die Kommandos

```
:reload
:r
```

laden die zuletzt geladene Datei, z.B. nach Änderungen, erneut.

```
> cat intro.hs
-- parabola
f :: Int -> Int
f x = x * x + x

-- logarithm to base 2
log2 :: Float -> Float
log2 = logBase 2
```

```
-- implication
(==>) :: Bool -> Bool -> Bool
a ==> b = not a || b

-- euler number
e :: Float
e = exp 1
```

```
> ghci intro.hs
...
ghci> f 5
30
ghci> log2 8
3.0
ghci> log2 0.25
-2.0
ghci> True ==> False
False
ghci> e
2.7182817
ghci> :q
```

```
> ghci
...
ghci> :l intro.hs
...
ghci> f 5
30
...
ghci> :q
```

## 2.3 Pattern Matching

In den Beispielen zu Funktions- und Operatordefinition wurde bereits *pattern matching* (Mustererkennung) verwendet.

Die Funktion

```
f :: Int -> Int
f x = x * x + x
```

benutzt das *pattern* **x** und der Operator

```
(==>) :: Bool -> Bool -> Bool
a ==> b = not a || b
```

die *pattern* `a` und `b` als Platzhalter für die Argumente, mit denen Funktion/-Operator aufgerufen werden.

Die verwendeten (allgemein gültigen) *pattern* können jeweils jeden beliebigen Wert des Definitionsbereichs repräsentieren. D.h. für jede gültige Anwendung der Funktion passen (*match*) diese *pattern* und die Funktion mit ihren Argumenten wird durch den Ausdruck, der dem passenden *pattern* zugeordnet ist, ersetzt.

Für die Auswertung eines Ausdrucks werden Funktionen und Argumente solange ersetzt, bis ein Ausdruck erreicht ist, der nur noch aus einem Element eines Typs (`Int`, `Float`, `(Int, Float)`, `[Char]` etc.) besteht.

Beispiel

```
f :: Int -> Int
f x = x * x + x
```

```
f 5
5 * 5 + 5
25 + 5
30
```

```
(==>) :: Bool -> Bool -> Bool
a ==> b = not a || b
```

```
True ==> False
not True || False
False || False
False
```

Bemerkungen

- Für Haskell ist alles Funktion/Operator, z.B. auch die arithmetischen Operatoren. Das Prinzip, die Argumente aus dem Definitionsbereich durch Funktionen/Operatoren auf einen Wert des Wertebereiches abzubilden, bleibt erhalten. Nur wird der Wert, durch den Funktion/-Operator und Argumente ersetzt werden, nicht durch *pattern matching* o.Ä. bestimmt, sondern durch Berechnungen des Prozessors.
- Haskell benutzt *lazy evaluation* um eine möglichst effiziente Auswertung zu erreichen, z.B. soll eine Funktion mit identischen Argumenten nur einmal ausgewertet werden.

*Pattern* können auch weniger allgemeingültig sein, bis hin zu *pattern*, die nur einen Wert des Definitionsbereich repräsentieren.

Eine Funktions-/Operatordefinition kann mehrere *pattern* enthalten. Jedem *pattern* muss eine Ausdruck zugeordnet werden, durch den Funktion/Operator und Argumente, bei passenden *pattern*, ersetzt werden. Im einfachsten Fall ist das ein Wert des Wertebereichs.

```
f :: ...
f [pattern_1] = [expr_1]
f [pattern_2] = [expr_2]
f ...
```

Die *pattern* werden von oben nach unten ausgewertet, d.h., dass erste passende *pattern* bestimmt den Ausdruck, durch den die Funktions-/Operatoranwendung ersetzt wird.

Beispiele

Negation der Aussagenlogik

```
neg :: Bool -> Bool
neg False = True
neg True  = False
```

Konjunktion der Aussagenlogik (Version 1)

```
(<&>) :: Bool -> Bool -> Bool
(<&>) False False = False
(<&>) False True  = False
(<&>) True  False = False
(<&>) True  True  = True
```

Konjunktion der Aussagenlogik (Version 2)

```
(<&>) :: Bool -> Bool -> Bool
False <&> False = False
False <&> True  = False
True  <&> False = False
True  <&> True  = True
```

Konjunktion der Aussagenlogik (Version 3)

```
(<&>) :: Bool -> Bool -> Bool
(<&>) True True  = True
(<&>) a b = False
```

Konjunktion der Aussagenlogik (Version 4)

```
(<&>) :: Bool -> Bool -> Bool
True <&> True  = True
_ <&> _ = False
```

Bemerkung

Den Unterstrich _ nennt man ein *wildcard* oder *don't care pattern*.

## 2.4 Alternativen

### Alternativen

Betrachten wir folgende Definition des Betrags einer ganzen Zahl.

$$\text{abs}(x) = \begin{cases} -x & \text{wenn } x < 0 \\ x & \text{sonst} \end{cases}$$

Um diese Definition umzusetzen, bietet sich ein **bedingter Ausdruck** an.

**if** *[test]* **then** *[expr_if]* **else** *[expr_else]*

Abhängig vom *[test]* (ein Ausdruck der nach **Bool** ausgewertet wird) nimmt der Ausdruck für

- *[test]* **== True** den Wert von *[expr_if]* an.
- *[test]* **== False** den Wert von *[expr_else]* an.

```
absolute :: Int -> Int
absolute x = if x < 0 then -x else x
```

In Haskell sind **bewachte Gleichungen (guarded equations)** möglich, mit denen sich Alternativen, insbesondere mehr als zwei, sehr übersichtlich realisieren lassen.

```
f :: X -> Y -> Z
f x y
  | [guard_1] = [expr_1]
  | [guard_2] = [expr_2]
  | ...
  | [guard_n] = [expr_n]
  | otherwise = [expr_default]
```

Die Wächter *[guard_1]* ... *[guard_n]* sind Test (**Bool**-Ausdrücke), die von Oben nach Unten ausgewertet werden.

Der Wert der Funktion ist der erste Ausdruck *[expr_i]* für dessen Wächter *[guard_i]* **== True** gilt.

Es gilt **otherwise == True**.

Beispiel

Exclusives Oder der Aussagenlogik

```
-- XOR
(<+>) :: Bool -> Bool -> Bool
(<+>) a b
  | a == b    = False
  | otherwise = True
```

Vergleichsfunktion

$$\text{compare}(x, y) = \begin{cases} +1 & \text{wenn } x > y \\ -1 & \text{wenn } x < y \\ 0 & \text{sonst} \end{cases}$$

```
comp :: Int -> Int -> Int
comp x y
  | x > y     = 1
  | x < y     = -1
  | otherwise = 0
```

## 2.5 Rekursion

### Rekursion

Definitionen von Folgen lassen sich häufig **rekursiv** sehr kompakt angeben.

Beispiel

Die rekursive Näherung der Quadratwurzel einer positiven Zahl `a` nach Heron.

$$\begin{aligned} x_0 &= a \\ x_n &= (x_{n-1} + a/x_{n-1})/2 \qquad \text{für } n > 0 \end{aligned}$$

Daraus lässt sich eine rekursive Funktion zur Berechnung des $n$-ten Folgenglieds ableiten.

$$\begin{aligned} &\text{heron} :: \mathbb{N} \times \mathbb{R}^+ \to \mathbb{R}^+ \\ &\text{heron}(n, a) = \begin{cases} \Big(\text{heron}(n-1, a) + a/\text{heron}(n-1, a)\Big)/2 & \text{für } n > 0 \\ a & \text{sonst} \end{cases} \end{aligned}$$

Diese rekursive Definiton lässt sich in Haskell direkt umsetzen.

```
heronA :: (Int, Double) -> Double
heronA (n, a)
  | n > 0     = (heronA ((n-1), a) + a / heronA ((n-1), a))/2
  | otherwise = a
```

Häufig sinnvoller ist aber die Sichtweise, bei der die Funktion heron mit Argument $n$ auf eine Funktion $\text{heron}_n$ abbildet, die das $n$-te Folgenglied zum Argument $a$ berechnet.

$$\begin{aligned} &\text{heron}_n :: \mathbb{R}^+ \to \mathbb{R}^+ \\ &\text{heron}_n(a) = \begin{cases} \Big(\text{heron}_{n-1}(a) + a/\text{heron}_{n-1}(a)\Big)/2 & \text{für } n > 0 \\ a & \text{sonst} \end{cases} \end{aligned}$$

```
heronB :: Int -> Double -> Double
heronB n a
  | n > 0     = (heronB (n-1) a + a / heronB (n-1) a)/2
  | otherwise = a
```

Haskell bietet nicht die Möglichkeit Speicherplätze zu reservieren, deshalb können in Funktionen auch keine Speicherplätze für lokale Variablen bereitgestellt werden, z.B. zur Speicherung von Zwischenergebnissen.

Mit der Klausel **where** können Platzhalter für lokale Definitionen, die nur im Kontext einer Funktionsdefinition gültig sind, angelegt werden.

Insbesondere zur Strukturierung und Verbesserung der Lesbarkeit von Definitionen ist **where** nützlich.

Beispiel

```
heronC :: Int -> Double -> Double
heronC n a
  | n > 0     = (x + a/x)/2
  | otherwise = a
  where x = heronC (n-1) a
```

Mit der Klausel **where** können mehrere lokale Definitionen vorgenommen werden.

```
f :: ...
f ...
  | [guard_1] = [expr_1]
  | [guard_2] = [expr_2]
  | ...
  | [guard_n] = [expr_n]
  | otherwise = [expr_default]
  where
    [expr_where_1]
    [expr_where_2]
    ...
    [expr_where_n]
```

Beispiel

Fibonacci-Folge

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \text{ für } n \geq 2$$

```
fibA :: Int -> Int
fibA n
  | n == 0    = 0
  | n == 1    = 1
  | otherwise = x + y
  where
    x = fibA (n-1)
    y = fibA (n-2)
```

Bei der Fibonacci-Folge bietet sich stattdessen die Verwendung passender *pattern* an.

```
fibB :: Int -> Int
fibB 0 = 0
fibB 1 = 1
fibB n = fibB (n-1) + fibB (n-2)
```

Bemerkung. Beide Versionen sind nicht korrekt definiert, denn es werden auch ungültige (negative) Argumente akzeptiert.

Fehler, z.B. ungültige Argumente, kann man mit der Funktion **`error`** aus **`Prelude`** behandeln.

Die Funktion **`error`** wird mit einem **`String`** (Zeichenfolge eingeschlossen in doppelte Hochkommata `"`) als Argument aufgerufen. Die Auswertung der aktuellen Funktion wird unterbrochen und eine Ausnahme (*exception*) ausgelöst, die den **`String`** ausgibt.

Beispiel

```
fibC :: Int -> Int
fibC 0 = 0
fibC 1 = 1
fibC n
  | n < 0     = error "illegal argument"
  | otherwise = fibC (n-1) + fibC (n-2)
```

Bemerkung.

Auf das Auslösen und Behandeln von Ausnahmen wird nicht weiter eingegangen.

Da Operatoren und Funktionen in Haskell im wesentlichen identisch sind, ist auch die rekursive Definition von Operatoren möglich.

Beispiel

Näherung der Quadratwurzel nach Heron als Operator.

```
(<##>) :: Int -> Double -> Double
0 <##> a = a
n <##> a
  | n > 0     = (x + a/x)/2
  | otherwise = error "illegal argument"
  where x = (n-1) <##> a
```

### Wiederholungen

Viele Programmiersprachen bieten meist mehrere Konstrukte zur Wiederholung von Codeblöcken an, sogenannte Schleifen.

**Schleifen gibt es in Haskell nicht**.

Wiederholungen müssen durch Rekursion realisiert werden.