# Typdeklarationen

Es gibt in Haskell bereits primitive Typen:

- *Integer*: ganze Zahlen, z.B. `1289736781236`
- *Int*: ganze Zahlen mit Computerarithmetik, z.B. `123`
- *Double*: Fließkommazahlen, z.B. `3.14159`
- *String*: Zeichenketten, z.B. `"Hello, World"`
- *Char*: ein Zeichen, z.B. `'A'`

Wir können auch neue Typen deklarieren.

# Notation

Hat eine Funktion ein Argument muß dieses nicht geklammert werden.

Wir schreiben also lieber

$$\begin{aligned} \mathit{factorial}\ 1 &= 1 \\ \mathit{factorial}\ n &= n * \mathit{factorial}\ (n-1) \end{aligned}$$

Die $(n-1)$ muß geklammert sein, weil sonst implizit so geklammert würde:

$$(n * (\mathit{factorial}(n))) - 1$$

# Tupel

Es gibt in Haskell *Tupel*:

Beispiele: $(1, 2)$, $(\texttt{"John"}, \texttt{"Doe"})$, $(17, \texttt{"plus"}, 4)$

Eine Funktion hat streng genommen immer nur ein Argument.

Mehrstellige Funktionen können durch Tupel „simuliert“ werden:

$$sum1 :: (Integer, Integer) \rightarrow Integer$$
$$sum1\ (x, y) = x + y$$

Man bevorzugt aber solche Funktionen:

$$sum2 :: Integer \rightarrow Integer \rightarrow Integer$$
$$sum2\ x\ y = x + y$$

Technisch gesehen ist *sum2* eine Funktion, die eine Zahl auf eine andere Funktion abbildet, die dann eine Zahl auf eine Zahl abbildet.

So werden aber praktisch mehrstellige Funktionen umgesetzt.

Was steckt hinter *sum2*?

$$sum2 :: Integer \to Integer \to Integer$$
$$sum2\ x\ y\ =\ x + y$$

Tatsächlich ist $sum2\ y$ eine Funktion.

Insbesondere ist $sum2\ 5$ die Funktion $x \mapsto x + 5$.

Wir können $(sum2\ 5)(3)$ schreiben.

Die Deklaration $f\ =\ sum2\ 5$ ist auch möglich.

So können wir eine Exponentialfunktion definieren:

$$power3 :: Integer \to Integer \to Integer$$
$$power3\ n\ 0\ =\ 1$$
$$power3\ n\ m\ =\ n * power3\ n\ (m - 1)$$

Folgendes Programm ist ineffizient, weil $fib(x-1)$ und $fib(x-2)$ *beide* aufgerufen werden.

$$
\begin{aligned}
&fib :: Integer \rightarrow Integer \\
&fib\ 0 = 0 \\
&fib\ 1 = 1 \\
&fib\ x = fib\ (x-1) + fib\ (x-2)
\end{aligned}
$$

Die Laufzeit ist exponentiell.

Die Ausführung könnte durch Caching von bereits berechneten *fib*s automatisch optimiert werden.

Dies macht der Compiler aber leider nicht.

In funktionalen Sprachen können wir nicht einfach Werte „speichern“.

Um $F_n$ zu berechnen, benötigen wir $F_{n-1}$ *und* $F_{n-2}$.

Lösung: Statt nur $F_n$ berechnen wir das Paar $P_n = (F_n, F_{n+1})$.

Jetzt läßt sich $P_{n+1}$ aus $P_n$ allein berechnen.

```
fibpair :: Integer → (Integer, Integer)
fibpair 0 = (0, 1)
fibpair n = (b, a + b)
  where (a, b) = fibpair (n − 1)
```

Durch eine **where**-Anweisung können wir verhindern, $fibpair(n-1)$ zweimal aufzurufen. Durch **where**-Anweisungen können wir Zwischenergebnissen einen Namen geben.

# Listen

Listen sind in funktionalen Sprachen allgemein eine wichtige Datenstruktur.

In Haskell sind Listen vordefiniert:

1. $[\,]$ ist die leere Liste.
2. $x : xs$ ist die Liste deren Kopf aus $x$ besteht, gefolgt von den Elementen aus der Liste $xs$.

Beispiel: $1 : 2 : 3 : 4 : 5 : [\,]$

Abkürzung:

$[a, b, c, d]$ ist eine Abkürzung für $a : b : c : d : [\,]$.

Typnamen:

$[Integer]$ ist der Typ „Liste von *Integer*s“.

# Arbeiten mit Listen

So verdoppeln wir die Einträge einer Liste:

```
verdoppeln :: [Integer]→[Integer]
verdoppeln [ ]  =  [ ]
verdoppeln (x : xs)  =  2 ∗ x  :  verdoppeln xs
```

So können wir die Reihenfolge umdrehen:

```
umdrehen :: [Integer]→[Integer]
umdrehen [ ]  =  [ ]
umdrehen (x : xs)  =  (umdrehen xs) ++ [x]
```

Der Operator $++$ konkateniert zwei Listen.

# Beispiel: Mergesort

Ein einfaches Sortierverfahren heißt *Mergesort*.

Es funktioniert so:

1. Teile die Liste in zwei etwa gleich lange Listen *l*1 und *l*2.
2. Sortiere *l*1 und *l*2.
3. Mische *l*1 und *l*2 in eine sortierte Liste.

Wir implementieren daher drei Funktionen *teile*, *mische* und *mergesort*.

Die folgende Funktion verteilt Elemente einer Liste immer abwechselnd auf zwei neue Listen.

```
teile :: [Integer]→([Integer], [Integer])
teile [ ] = ([ ],[ ])
teile (x : [ ]) = ([x], [ ])
teile (x : y : [ ]) = ([x], [y])
teile (x : y : xs) = (x : r1, y : r2)
  where (r1, r2) = teile xs
```

```
mische :: [Integer]→[Integer]→[Integer]
mische [ ] l  =  l
mische l [ ]  =  l
mische (x : xs) (y : ys)  =
  if x  <  y
  then x  :  mische xs (y : ys)
  else y  :  mische (x : xs) ys
```

```
mergesort :: [Integer]→[Integer]
mergesort [ ]  =  [ ]
mergesort (x : [ ])  =  [x]
mergesort l  =  mische (mergesort l1) (mergesort l2)
  where (l1, l2)  =  teile l
```