Tag 7

„Pattern Matching“ und eigene Datentypen

Heute werden wir eine Technik kennenlernen, die dafür sorgt, daß wir sehr viel übersichtlichere und kürzere Programme schreiben können.
Als Überleitung auf diese Technik, die den Namen „pattern matching“ trägt, blicken wir zunächst einmal ein wenig voraus und schauen uns an, wie man in Haskell eigene Datentypen definieren kann.
Auch diesmal wieder bildet der Quellcode zur Lektion ein eigenes Modul:

```
> module Tag7 where
```

Wir beginnen, wie so oft, mit einem Beispiel:

```
> data Numerus = Singular | Plural
```

Obige Zeile führt einen neuen Datentyp `Numerus` ein. Dieser ist *verschieden* von allen anderen Datentypen, d.h. ein Wert vom Typ `Numerus` gehört keinem anderen Datentyp an. Wie sehen nun Werte dieses Datentyps aus? Es gibt zwei Möglichkeiten, angedeutet durch den senkrechten Strich `|`, den man am besten als „oder“ liest: entweder es ist ein `Singular` oder es ist ein `Plural`. Jede Möglichkeit definiert einen sogenannte `Konstruktor` für den Datentyp. Konstruktoren sind spezielle Funktionen, die dazu dienen, Werte des jeweiligen Datentyps zu erzeugen. Wenn wir das Modul laden, können wir die Typen der Konstruktoren erfragen:

```
Tag7> :t Singular
Numerus

Tag7> :t Plural
Numerus

Tag7> :t [Singular,Singular,Plural]
[Numerus]
```

Wir sehen, daß `Singular` und `Plural` selber Werte des Typs `Numerus` sind, und daß wir mit ihnen komplexere Werte aufbauen können, indem wir sie zum Beispiel in eine Liste stecken.
Warum sollten wir einen Typ wie `Numerus` deklarieren wollen? Wir könnten doch auch ganz einfach `0` und `1` nehmen, um Singular und Plural zu unterscheiden, oder `False` und `True`, oder gar die Strings `"Singular"` und `"Plural"`. Erstes Argument: Spezielle Typen dienen der Selbstdokumentation des geschriebenen Codes. Wie wir gesehen haben, ist das Definieren von Typen nicht aufwendig. Eine Zeile! Dafür können wir dann mit „sprechenden“ Namen wie `Singular` oder `Plural` arbeiten, wenn unsere Anwendung tatsächlich etwas damit zu tun hat, und wir können diese Werte von anderen, „echten“ Wahrheitswerten wie etwa Resultaten von Vergleichen unterscheiden. Zweites Argument: Nicht nur wir können den Code besser verstehen, wenn wir einen eigenen Typ verwenden, auch der Compiler kann das! Wie bereits erwähnt, `Numerus`

ist ein Typ, der von anderen Typen verschieden ist. Wir laufen also nicht Gefahr, einen `Numerus` „aus Versehen“ in einer `if`-Abfrage als Bedingung einzusetzen, weil der Compiler das abfangen würde, während wir, wenn wir `Bool` als Representation nähmen, davor nicht sicher wären. Gegen `Integer` oder `String` als Representation spricht zudem, daß diese Typen viel mehr Werte umfassen, und wir jedesmal, wenn wir eine Berechnung auf der Basis von einem Numerus durchführen wollen, sicherstellen müssen, daß der ihn representierende Integer oder String tatsächlich die richtige Form hat (also `0` oder `1` ist bzw. `"Singular"` oder `"Plural"`).
Wenn wir nun jedoch tatsächlich eine Funktion schreiben wollen, die etwas unter Verwendung eines `Numerus` macht, geraten wir in Schwierigkeiten. Nehmen wir an, wir wollen eine (stark vereinfachte) Funktion schreiben, die die dritte Personsform eines Verbs ermittelt. Diese bekommt den Stamm des Verbs als String und einen `Numerus`, und sie liefert am Ende die korrekte Form, wiederum als String, also:

```
> drittePerson :: String -> Numerus -> String
```

Falls Singular gefragt ist, wollen wir ein „t“ an den Stamm hängen, andernfalls ein „en“. Aber wir haben keine Möglichkeit, den Wert vom Typ `Numerus` zu analysieren. Hier kommt „pattern matching“ ins Spiel: Das `case`-Konstrukt in Haskell ermöglicht es uns, verschiedene Konstruktoren eines Typs voneinander zu unterscheiden:

```
> drittePerson stamm num =
>     case num of
>       { Singular -> stamm ++ "t"
>       ; Plural   -> stamm ++ "en"
>       }
```

Sowohl `case` als auch `of` sind Schlüsselworte, die den Ausdruck einrahmen, der analysiert werden soll. Danach folgen mehrere Fälle, die jeweils aus einem „Muster“ und einem Ausdruck bestehen. Muster und Ausdruck sind dabei wiederum jeweils durch `->` voneinander getrennt. In diesem Fall gibt es also zwei Fälle mit den Mustern `Singular` und `Plural`. Haskell wählt den ersten Fall, dessen Muster auf den analysierten Wert paßt (daher der Name „pattern matching“). (Wir verwenden ab jetzt den Anglizismus „Pattern“ für das Wort Muster.)
Damit verhält sich die Funktion nun wie erwartet, aber natürlich nicht perfekt:

```
Tag7> drittePerson "geh" Singular
"geht"

Tag7> drittePerson "lauf" Plural
"laufen"

Tag7> drittePerson "lauf" Singular
"lauft"
```

Wie Haskell uns schon erlaubt, Lambda-Abstraktionen in die Deklaration zu integrieren und uns ein explizites Verwenden von `\` zu ersparen, ist es auch möglich, das `case`-Konstrukt in die Funktionsdeklaration direkt mit aufzunehmen, indem wir einfach mehrere Funktionsdefinitionen schreiben, eine für jeden Fall:

```
> zweitePerson :: String -> Numerus -> String
> zweitePerson stamm Singular = stamm ++ "t"
> zweitePerson stamm Plural   = stamm ++ "en"
```

Jeder Variablenname, der in einer Definition gebunden wird, kann also alternativ durch ein Pattern ersetzt werden. Eine Funktionsdefinition kann aus beliebig vielen Teilen bestehen. Der erste Fall, der paßt (und nur der erste), wird ausgeführt.
Zurück zu den Datentypen: Wir führen nun einen zweiten Datentyp ein, einen, mit dem man binäre Bäume repräsentieren kann, die an den Blättern Elemente haben. Wie schon bei Listen wollen wir, daß alle Elemente vom gleichen Typ sind, aber der Typ soll ansonsten nicht näher bestimmt sein.

```
> data BinTree a = Node (BinTree a) (BinTree a) | Leaf a
```

Wieder gibt es zwei Möglichkeiten (durch den einen vertikalen Strich getrennt), und die Konstruktoren in diesem Falle heißen `Node` und `Leaf`. Diesmal haben jedoch sowohl der Datentyp selbst als auch die Konstruktoren Argumente. Der Baum-Datentyp ist parametrisiert mit dem Typ seiner Elemente, repräsentiert durch die Typvariable `a`. Ein Baum kann *entweder* eine binäre Verzweigung (`Node`) sein, die aus zwei Unterbäumen besteht (daher die beiden Argumente von `Node`, beide wieder vom Typ `BinTree a`), *oder* er besteht lediglich aus einem Blatt, und da wir gesagt haben, daß die Blätter Elemente enthalten sollten, hat das Blatt ein Argument vom Elementtyp `a`. Wieder können wir den Interpreter zur Bestätigung nach den Typen fragen:

```
Tag7> :t Node
forall a. BinTree a -> BinTree a -> BinTree a

Tag7> :t Leaf
forall a. a -> BinTree a

Tag7> :t Node (Node (Leaf "a") (Leaf "b")) (Leaf "42")
BinTree [Char]
```

Wir sehen, daß beide Konstruktoren in der Tat Binärbäume erzeugen und daß die Argumente der Konstruktoren in der Typdeklaration in Funktionsargumente umgewandelt werden. Die dritte Eingabe zeigt einen Beispielbaum mit Strings als Elemente in den Blättern.
Wenn wir nun eine Funktion per „pattern matching“ auf binären Bäumen definieren, dann können wir für die Argumente der Konstruktoren wieder Variablen einführen:

```
> countLeaves :: BinTree a -> Integer
> countLeaves (Leaf x)   = 1
> countLeaves (Node l r) = countLeaves l + countLeaves r
```

Obige Funktion zählt die Anzahl Blätter in einem Baum (mit beliebigem Elementtyp). Falls der Baum ein Blatt ist, dann hat er genau ein Blatt. Falls er eine Verzweigung ist, dann zählen wir die Blätter in beiden Unterbäumen und summieren sie. Die Variablen `l` und `r` sind Bestandteil des zweiten Patterns und werden an die Unterbäume gebunden.

Aufgabe 1

Schreibe eine Funktion `flattenBinTree :: BinTree a -> [a]`, die alle Blätter „von links nach rechts“ in einer Liste aufsammelt.

```
Flatten> flattenBinTree (Node (Node (Leaf "a") (Leaf "b")) (Leaf "42"))
["a","b","42"]
it :: [[Char]]
```

Die meisten Haskell-Datentypen, die wir bislang kennengelernt haben, haben syntaktische Besonderheiten. Dennoch schadet es nicht, sie unter dem Gesichtspunkt zu betrachten, daß man sie als eigene Datentypen definieren wollte.
Beginnen wir mit Wahrheitswerten. Diese sind in der Tat einfach als

```
data Bool = False | True
```

definiert. Folglich sind `False` und `True` auch Konstruktoren und können in Patterns verwendet werden. Womit sich auch die Frage, die sich vielleicht schon mancher gestellt hat, klärt: Konstruktoren beginnen immer mit einem Großbuchstaben, genau wie Typnamen. Es ist sogar erlaubt, daß Konstruktoren und Typnamen sich überschneiden, es kann also einen Konstruktor geben, der genauso heißt wie ein Typ. Trotzdem sind es zwei verschiedene Dinge: ein Konstruktor ist eine Funktion und als solches ein Ausdruck und *hat* einen Typ. Ein Typ hingegen *ist* ein Typ.
Einzelne Zeichen (vom Typ `Char`) sind im Grunde nichts weiter als eine Aufzählung aller möglichen Zeichen, etwa so

```
data Char = ...
          | '!' | '"' | '#' | '$' | '%' | ...
          | 'A' | 'B' | 'C' | 'D' | 'E' | ...
          | 'a' | 'b' | 'c' | 'd' | 'e' | ...
          | ...
```

Im Prinzip also spielen die einzelnen Zeichen allesamt die Rolle von ziemlich vielen, aber immer noch endlich vielen Konstruktoren. Und in der Tat – auch Zeichen können als Pattern verwendet werden.
Bei Integers ist es ähnlich. Hier ist die Anzahl der Werte prinzipiell unbegrenzt, aber auch sie können als Typ mit unendlich vielen Konstruktoren – alle ohne Argumente – aufgefaßt werden:

```
data Integer = ... | -2 | -1 | 0 | 1 | 2 | ...
```

Und daher sind auch Integer-Literale in der Rolle von Konstruktoren und können in Patterns auftreten. Die Fakultätsfunktion aus den Übungen der letzten Funktion kann wie folgt geschrieben werden:

```
> factorial :: Integer -> Integer
> factorial 0 = 1
> factorial n = n * factorial (n-1)
```

Listen könnte man wie folgt mit Hilfe von `data` definieren:

```
data [a] = [] | a : [a]
```

Dies ist syntaktisch nicht erlaubt, entspricht aber genau dem Verhalten von Listen. Der Listentyp hat einen Parameter, der zwischen die eckigen Klammern geschrieben wird. Er hat zwei Konstruktoren, nämlich die leere Liste `[]` und den Cons-Operator `:`. Letzterer hat zwei Argumente, wie bekannt:

```
Tag7> :t []
forall a. [a]

Tag7> :t (:)
forall a. a -> [a] -> [a]
```

Und natürlich können auch diese Konstruktoren in Patterns auftreten. Die bereits bekannten Funktionen head tail und null zum Selektieren des Kopfes und Restes einer Liste bzw. zum Überprüfen, ob eine Liste leer ist, lassen sich mit Hilfe von „pattern matching“ auf Listen einfach definieren (wir wählen leicht modifizierte Namen, um unsere Varianten von den vordefinierten unterscheiden zu können):

```
> head' :: [a] -> a
> head' (x:xs) = x

> tail' :: [a] -> [a]
> tail' (x:xs) = xs

> null' :: [a] -> Bool
> null' [] = True
> null' xs = False
```

Aufgabe 2

Schreibe eine Funktion `inssort :: [Integer] -> [Integer]`, die eine Liste von ganzzahligen Werten nach dem „insertion sort“-Verfahren aufsteigend sortiert. (Jedes andere Sortierverfahren ist auch in Ordnung.) Benutze „pattern matching“. Schreibe dazu zunächst eine Hilfsfunktion `ins :: Integer -> [Integer] -> [Integer]`, die eine Zahl in eine bereits sortierte Liste an der richtigen Stelle einfügt.

```
Inssort> ins 6 [1,4,7,10]
[1,4,6,7,10]

Inssort> inssort [7,1,6,4,10]
[1,4,6,7,10]
```

Zum Abschluß dieser schon viel zu langen Lektion betrachten wir nochmal die Tupel. Hier kann man sich vorstellen, daß jeder Tupeltyp (also für jede Tupellänge) durch ein eigenes `data` definiert ist:

```
data (a,b)     = (a,b)
data (a,b,c)   = (a,b,c)
data (a,b,c,d) = (a,b,c,d)
...
```

Jeder Tupeltyp hat nur einen Konstruktor, mit spezieller Syntax für die Argumente, die „in den Konstruktor eingebettet“ werden. Außerdem haben Tupeltypen eine aufsteigende Anzahl von Parametern. Letzlich bedeutet das, daß auch die Tupelkonstruktoren in Patterns auftreten können. Die Funktionen `fst` und `snd` zum Selektieren der Komponenten, und auch die bereits in den Aufgaben implementierte Funktion `swap` zum Vertauschen der Komponenten lassen sich so definieren (wir wählen wiederum leicht modifizierte Namen):

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

> snd' :: (a,b) -> b
> snd' (x,y) = y

> swap' :: (a,b) -> (b,a)
> swap' (x,y) = (y,x)
```

Lernziele Tag 7

- In Haskell können mit dem `data`-Schlüsselwort auf einfache Weise neue Datentypen definiert werden. (Die genauen Möglichkeiten werden wir aber noch im Detail besprechen.)
- Haskell erlaubt „pattern matching“ mittels `case`-Konstrukt. Verschiedene Fälle können unterschieden werden durch verschiedene Pattern. Pattern bestehen aus einer Mischung aus Variablennamen und *Konstruktoren*. Wenn der zu analysierende Wert vom richtigen Konstruktor ist, dann werden die Variablennamen an die Argumente des Konstruktors gebunden.
- Auch in Funktionsdefinitionen können statt Variablennamen für die Argumente Pattern benutzt werden.
- Auch die Haskell-Basistypen haben Konstruktoren und können deshalb in Patterns verwendet werden.