Heinrich-Hertz-Gymnasium
Berlin-Friedrichshain
Fachbereich Informatik
R. Dorn

Darstellung der Operatoren in Haskell

Der Grundgedanke von Haskell ist der aller funktionaler Sprachen, nämlich das Auswerten von Ausdrücken, die meist rekursiv definiert werden. Dabei kann in den Definitionen (rechte Seiten) von einer ganzen Reihe vor definierter Funktionen Gebrauch gemacht werden. Wie der eine oder andere schon herausgefunden hat, haben wir bisher überwiegend Funktionen nachimplementiert, die in der Standardumgebung bereits existieren. Es lohnt sich also die vordefinierten Operationen zu kennen. Hier ein kleiner Überblick.

Datentypen und wichtige Funktionen:

1. Ganzzahlen

Es gibt den Datentyp Int und Integer. Dabei hat Int einfache Genauigkeit, d.h. es stehen für die Zahlen 32 bit zur Verfügung. Integer dagegen hat theoretisch unendliche Genauigkeit. Für ganz normale Fälle reicht also Int.

```
+ :: Int -> Int -> Int          (Addition)
* :: Int -> Int -> Int          (Multiplikation)
^ :: Int -> Int -> Int          (Exponentation)
- :: Int -> Int -> Int          (Subtraktion, Infix)
- :: Int -> Int                 (Vorzeichenwechsel, Prefix)

div :: Int -> Int -> Int        (Division)
mod :: Int -> Int -> Int        (Divisionsrest)

abs :: Int -> Int               (Absolutbetrag)
negate :: Int -> Int            (Vorzeichenwechsel)

signum :: Float -> Int          (Vorzeichenfunktion)
signum x | x > 0 = 1
         | x == 0 = 0
         | x < 0 = -1

> :: Int -> Int -> Bool         (echt größer)
>= :: Int -> Int -> Bool        (größer gleich)

<= :: Int -> Int -> Bool        (kleiner gleich)
< :: Int -> Int -> Bool         (echt kleiner)

== :: Int -> Int -> Bool        (gleich)
/= :: Int -> Int -> Bool        (ungleich)
```

Heinrich-Hertz-Gymnasium
Berlin-Friedrichshain
Fachbereich Informatik
R. Dorn

2. Gleitkommazahlen

Diese werden mit Float (einfache Genauigkeit) bzw. Double (doppelte Genauigkeit) deklariert.

```
'+' :: Float -> Float -> Float      (Addition)
'*' :: Float -> Float -> Float      (Multiplikation)

sqrt :: Float -> Float
sin :: Float -> Float
```

Weiterhin gibt es pi, exp, log, cos, tan, asin, acos, atan uvm.

3. Char

Der Typ Char steht für einzelne Zeichen. Man nimmt den einfachen Akzent, um einen Char zu kennzeichnen: ’a’

`ord :: Char -> Int` (Umwandlung von Char nach Int, Beispiel: `ord 'A'` (liefert 65))
`chr :: Int -> Char` (Umwandlung von Int nach Char, Beispiel `chr 65` (liefert 'A'))

4. Bool

Vom Typ Bool sind die Wahrheitswerte True und False, die in keiner Programmiersprache fehlen dürfen.

```
&& :: Bool -> Bool -> Bool    (logisches und)
|| :: Bool -> Bool -> Bool    (logisches oder)
not :: Bool -> Bool           (logisches nicht)
```

5. Listen

Setzt man einen Typ zwischen zwei eckige Klammern erhält man einen Typ ”Liste vom Typ“. Beispiel: `[Int]` ist eine Liste von ganzen Zahlen und `[Char]` ist eine Liste von Zeichen. In Haskell schreibt man eine konkrete Liste so: `[1,2,3]`. Eine andere Möglichkeit, Listen zu erzeugen ist der Listenkonstruktor. Mit dem : fügt man ein Element vorne in eine Liste ein: `1:[2,3]` wird zu `[1,2,3]`. Dabei muss das einzufügende Element den selben Typ haben, wie die Elemente, die schon in der Liste sind! `'a':[2,3]` führt zu einem Fehler.
Man kann übrigens `String` statt `[Char]` schreiben und dann die Schreibweise "Hallo" benutzen.

`length :: [a] -> Integer`	(liefert die Anzahl der Elemente in gegebener Liste)
`head :: [a] -> a`	(liefert erstes Element)
`tail :: [a] -> [a]`	(liefert die Liste ohne das Kopfelement)
`++ :: String -> String -> String`	(Concat)
`== :: String -> String -> Bool`	(gleich)
`/= :: String -> String -> Bool`	(ungleich)

6. Tupel

Tupel schreibt man in der Form $(Wert_1,Wert_2,...,Wert_n)$, z.B. `(Int,Char,[Int])`. Dies bezeichnet ein Tupel mit einer ganzen Zahl an erster Stelle, einem Zeichen an zweiter Stelle und einer Liste von ganzen Zahlen an dritter Stelle. Ein Tupel dieses Typs könnte so aussehen: `(3,'x',[10,9,8])`
Mit Pattern Matching bezeichnet man das Verfahren, mit welchem der Interpreter bei einem Funktionsaufruf feststellt, welchen Funktionsrumpf er benutzen soll. Um also einen Funktionsaufruf auszuwerten geht der Interpreter die Funktionsköpfe von oben nach unten durch, bis er einen findet, der von der Struktur her zum Argument des Aufrufes passt.

`fst :: (a,b) -> a`	(liefert erstes Element ("first"))
`snd :: (a,b) -> b`	(das zweite ("second"))

7. Funktionen

In Haskell sind auch Funktionen Objekte, mit denen man umgehen kann wie mit jedem anderen Typ auch: Funktionen können Argument oder Rückgabewert einer anderen Funktion haben. Den Funktionstyp deklariert man mit ->.

Wird weiter ergänzt!!