Christoph Denzler
Daniel Kröni

Arbeitsblatt: Pattern Matching

Pattern Matching (dt. Mustererkennung) kann eingesetzt werden um Fallunterscheidungen auszudrücken und gleichzeitig die Lesbarkeit des Codes zu erhöhen. Wir haben gesehen, dass Patterns Konstanten, Namen und Strukturen (Listen/Tuples) beinhalten können. Hier ein paar Beispiele:

Konstanten:

```
sayNumber :: (Integral a) => a -> String
sayNumber 0 = "Nothing"
sayNumber 1 = "One"
sayNumber 2 = "Another one"
sayNumber n = "Many"
```

Hier wurde n als Platzhalter verwendet. Der Name wird sonst nirgends gebraucht. Statt einen Namen kann auch _ verwendet werden (auch Wildcard Pattern genannt), falls der Wert nicht weiter von Bedeutung ist:

```
shibboleth :: String -> Bool
shibboleth "password" = True
shibboleth _          = False
```

Listen:

```
listLength :: [a]  -> String
listLength []       = "empty list"
listLength (x:[])   = "one element"
listLength (x:y:[]) = "two elements"
listLength _        = "long list"

foo :: [Int] -> Int
foo [1,2,3] = 1+2+3
foo [4,x,7] = 4+ x*7
foo [] = 0
foo z = -1
```

Hier wird gezeigt, wie in einem Muster ein Name verwendet wird. Passt das Muster (eine List mit drei Elementen, dessen erstes Element 4 und dessen letztes Element 7 ist) dann wird das mittlere Element an den gegebenen Namen gebunden. Rechts des Gleichheitszeichens kann dieser Name dann in den Berechnungen verwendet werden.

Tuples:

```
switchNonZero :: (Int, Int) -> (Int, Int)
switchNonZero (0, y) = (0, y)
switchNonZero (x, 0) = (x, 0)
switchNonZero (x, y) = (y, x)

bar :: (Integer, String) -> String
bar (0, "hello") = "world"
bar (0, x) = x
bar (x, "foo") = "bar"
bar (x, y) = "who cares?"
```

Tuples und Listen werden falls nötig “aufgebrochen” um ihre Strukturen erkennen zu können.
Üben Sie nun selbst ein wenig! Drehen Sie bitte das Blatt um.

Aufgabe 1:
Implementieren Sie eine Funktion switchFirstTwo:

```
switchFirstTwo :: [a] -> [a]
```

Diese Funktion tauscht die ersten beiden Elemente der Liste aus. Falls die Liste weniger als zwei Elemente enthält, bleibt sie unverändert. Lösen Sie die Aufgabe mittels Pattern Matching.

Aufgabe 2:
Zweidimensionale Vektoren können als Paar von Ints repräsentiert werden:

```
type Vec = (Int, Int)
```

a) Implementieren Sie die Funktion `addVec` um zwei Vektoren zu addieren. Lösen Sie auch diese Aufgabe mit Patternmatching.

   ```
   addVec :: Vec -> Vec -> Vec
   ```

   Zur Erinnerung: Vektoren werden komponentenweise addiert.

   $$\begin{pmatrix} a_x \\ a_y \end{pmatrix} + \begin{pmatrix} b_x \\ b_y \end{pmatrix} = \begin{pmatrix} a_x + b_x \\ a_y + b_y \end{pmatrix}$$

b) Jetzt implementieren Sie eine optimierte Variante dieser Funktion indem bei 0 Komponenten erst gar keine Addition ausführt wird.

   ```
   addVecOpt :: Vec -> Vec -> Vec
   ```

   Hinweis: Möglicherweise ist eine Hilfsfunktion zur optimierten Addition zweier Ints nützlich:

   ```
   addOpt :: Int -> Int -> Int
   ```