Übung 5: Operatoren, Currying, Funktionskomposition

1. Aufgabe: Operatoren

Operatoren haben eine definierte Bindungsstärke (Präzedenz). Diese sagt, dass z.B. folgender Ausdruck 1 + 2 * 3 so geklammert wird 1 + (2 * 3). Das * bindet als stärker als das + wie Sie das von der Mathematik her kennen.
Wenn nun derselbe Operator mehrmals aufeinander angewendet wird 1 o 2 o 3 stellt sich die Frage, auf welche Seite die Klammern zu interpretieren sind. Bei der Addition kommt das nicht drauf an (Addition ist assoziativ) (1 + 2) + 3 == 1 + (2 + 3) aber z.B. bei der Division macht es einen Unterschied: (1/2)/3 = 1/6 aber 1/(2/3) = 1.5 .

Im Unterricht haben Sie gesehen, dass für Operatoren mittels Fixity-Deklaration die Bindungsstärke und Bindungsrichtung definiert werden kann.

Hier ist ein Beispiel:
```
(|+|) :: Int -> Int -> Int
a |+| b = abs a + abs b
infixl 6 |+|
```

Mit dieser Definition wird also 1 |+| 2 |+| 3 so geklammert (1 |+| 2) |+| 3 .

a) Präzedenzen

Finden Sie für folgende Operatoren jeweils die Bindungsstärke und Bindungsrichtung. Füllen Sie die Operatoren dann nach ihrer Präzedenz sortiert in eine Tabelle:
Mathematische Operatoren: `+ - * / ^` Bool'sche Operatoren: `|| &&`
Vergleichs Operatoren: `== /= < > <= >=` Listen Operatoren: `++ :`
Funktionskomposition: `.`

Verwenden Sie GHCi:
```
Prelude> :info (+)
class Num a where
  (+) :: a -> a -> a
  ...     -- Defined in `GHC.Num'
infixl 6 +
```

Operator	**Präzedenz**
((f a) b) links	10
	9
	8
	7
(+) links	6
	5
	4
	3
	2
	1
	0

b) Zeigen Sie mit Klammern an wie Haskell die folgenden Ausdrücke interpretieren würde:

```
1 + 2 ^ 3 == 6 && 3 / 4 < 12 || snd (1,True)

(3:) [] == map (*5)[2 ^ 4 ^ 6]
```

2. Aufgabe: Currying

Funktionen in Haskell nehmen immer nur ein Argument und geben nur ein Resultat zurück. Häufig benötigen Sie aber Funktionen, die mehr als ein Argument benötigen um ein Resultat zu berechnen. Als Beispiel diene die Funktion mit dem Namen add, die zwei Ints addiert. Hier ist die Definition:

```
add :: Int -> Int -> Int
add a b = a + b
```

Was bedeutet jetzt aber die Typdeklaration? Der '->' (Funktionstyp) ist rechtsassoziativ. Wenn Sie zwei Pfeile nacheinander haben, müssen Sie mit den Klammern von rechts anfangen:

```
add :: (Int -> (Int -> Int))
```

Die Funktion add nimmt also einen Parameter (das erste Int) und gibt eine Funktion vom Typ (Int -> Int) zurück. Zurück kommt also genau die Funktion, die den zweiten Parameter erwartet um dann damit das Resultat zu berechnen.

Sie können diese Funktion aber auch anders formulieren. Statt "zwei Parameter" können Sie ein Paar von Ints als einen Parameter verlangen:

```
add' :: (Int,Int) -> Int
```

Diese Funktion nimmt ein Tuple mit zwei Int Komponenten als Argument. Sie kann so aufgerufen werden:

```
add' (1,2)
```
wobei das "(1,2)" als Paar mit zwei Ints zu verstehen ist.

a) curry

Implementieren Sie die Funktion curry'.

```
curry' :: ((a,b) -> c) -> a -> b -> c
```

Sie nimmt als Argument eine uncurried Funktion und erzeugt daraus eine curried Funktion.
Beispiel:
```
curry' add' :: Int -> Int -> Int
```

b) uncurry

Implementieren Sie die Funktion uncurry'.

```
uncurry' :: (a -> b -> c) -> (a,b) -> c
```

Sie nimmt als Argument eine curried Funktion und erzeugt daraus eine uncurried Funktion.
Beispiel:
```
uncurry' add :: (Int,Int) -> Int
```

3. Aufgabe: flip

Die Reihenfolge der Parameter einer Funktion legt fest in welcher Abfolge die Argumente übergeben werden müssen. Als Beispiel diene die Funktion replicate:

```
replicate :: Int -> a -> [a]
```

Sie nimmt als erstes Argument einen Int und als zweites Argument ein Wert von einem beliebigen Typ a. Das erste Argument definiert wie häufig das zweite Argument in der resultierenden Liste wiederholt vorkommt.

```
replicate 3 "hatschi" ~> ["hatschi", "hatschi", "hatschi"]
```

Wenn Sie beim Aufruf der Funktion alle Argument übergeben, ist es unproblematisch in welcher Reihenfolge die Parameter definiert wurden. Wenn die replicate Funktion so definiert wäre:

```
replicate' :: a -> Int -> [a]
```

würden Sie beim Aufruf einfach die Argumente entsprechend in der anderen Reihfolge übergeben.

```
replicate' "hatschi" 3 ~> ["hatschi", "hatschi", "hatschi"]
```

Problematisch wird es dann, wenn Sie die Funktion nur partiell anwenden wollen. Wenn die Reihenfolge der Parameter passt, haben Sie Glück und können das elegant hinschreiben:

```
replicateThreeTimes = replicate 3
replicateThreeTimes "hatschi" ~> ["hatschi", "hatschi", "hatschi"]
```

Wenn Sie aber nur das zweite Argument übergeben wollen, haben Sie ein Problem:

```
replicateHatschi = replicate "hatschi"
   Couldn't match expected type `Int' with actual type `[Char]'
```

Definieren Sie die Funktion flip'. Als Argument nimmt sie eine Funktion mit zwei Parametern und gibt die Funktion mit gekehrten Parametern zurück. Die folgende Definition sollte typechecken:

```
replicateHatschi = flip' replicate "hatschi"
```

a) Schreiben Sie den Typ der Funktion flip':

```
flip' ::
```

b) Implementieren Sie die Funktion einmal mit Lambda Expressions (add = \x y -> x+y) und einmal mit Parameter Definitionssyntax (add x y = x + y).

4. Aufgabe: Funktionskomposition

Gegeben sind folgende zwei Funktionen:

```
f :: a -> b
     g :: b -> c
```

Wenn Sie die Typen der Funktionen (und die Einrückung) anschauen, stellen Sie fest, dass der Wert, der aus f als Resultat raus kommt, gleich wieder als Input für das g verwendet werden kann.

Haskell bietet einen vordefinierten Operator (.), der verwendet werden kann um zwei Funktionen zu einer Funktion zu vereinen:

```
g.f  :: a -> c
```

ist die Komposition der Funktionen g und f und kann als "g nach f" gelesen werden.

Gegeben sind nun folgende Funktionen:

```
f :: Int  -> Int
g :: Int  -> Bool
h :: a    -> Int
i :: Bool -> (Int,Int)
```

a) Gesucht sind alle zulässigen zweier Kombinationen und deren Typ.
Hinweis: Malen Sie jeweils ein kleines Diagramm falls Sie Mühe haben das im Kopf zu machen. Das kommt mit der Zeit.

Beispiel:

```
f.f :: Int -> Int
```

b) Wie kann man die Funktionen f mit der Funktion i verknüpfen, wenn zusätzlich noch `fst` und `snd` zur Verfügung stehen?

```
fst :: (a,b) -> a
snd :: (a,b) -> b
```