

Alexander Becker (*alexander2.becker@tu-dortmund.de*)  
Sascha Mücke (*sascha.muecke@tu-dortmund.de*)

Wintersemester 2016/2017

# Freiwillige Zusatzübungen zu Funktionaler Programmierung

**Ausgabe:** 19. Januar 2017, **Abgabe:** keine

**Aufgabe Z.1** Implizite Klammerung von Funktionen und Typen  
Ergänzen Sie die impliziten Klammern folgender Funktionstypen.

```
(*) :: Num a => a -> a -> a  
(.) :: (b -> c) -> (a -> b) -> a -> c  
filter :: (a -> Bool) -> [a] -> [a]  
add5Ints :: Int -> Int -> Int -> Int -> Int -> Int
```

Ergänzen Sie die impliziten Klammern folgender Funktionen.

```
(*) 3 4  
(.) f g  
filter even [1..10]  
add5Ints 1 2 3 4 5  
map (\x -> \y -> \z -> x:y:z:[]) . filter even  
\x -> \y -> g x $ f . h y
```

**Aufgabe Z.2** Typinferenz

Bestimmen Sie die Funktionstypen folgender Funktionen so allgemein wie möglich.

```
\x -> x + 1  
filter id  
sum . filter odd  
map ($ 'x')  
map (\x -> Wrap $ (even . length) x)  
flip $ flip bexp2store x
```

Zur Bestimmung verwenden Sie:

```
bexp2store :: BExp a -> BStore a -> Store a -> Bool  
data WrapBool = Wrap Bool
```

**Aufgabe Z.3** Listen

- Schreiben sie eine Funktion, die die unendliche Liste aller geraden Fibonacci-Zahlen zurückgibt.
- Schreiben Sie eine Funktion, die überprüft, ob eine gegebene Liste ein Palindrom ist.
- Schreiben Sie eine Funktion, die alle Elemente der Collatzfolge enthält, beginnend bei einem Startwert  $x$ .
- Implementieren Sie den Quicksort-Algorithmus auf Listen.  
**Hinweis:** Nutzen Sie als Pivotelement das erste Element der Liste.

- Schreiben Sie eine Funktion, die die Quadrate aller Zahlen von 1 bis  $x$  enthält.
- Schreiben Sie eine Funktion, die die unendliche Liste der pythagoreischen Tripel erzeugt. Nutzen sie dazu Listenkompensation.
- **Hinweis:** Ein pythagoreisches Tripel  $(x, y, z)$  ist ein Tripel für das  $x^2 + y^2 = z^2$  gilt.
- Schreiben Sie eine Funktion, die aus einer eingegeben Int-Liste alle Vielfachen von 7 entfernt. Nutzen Sie dafür die Funktion *filter*.
- Schreiben Sie die Funktion *updateEntries* unter Verwendung von *foldl*, die die Werte einer Int-Liste addiert mit den Werten der Int-Listen, die jeweils an selber Stelle stehen.

Beispiel:

```
updateEntries :: [Int] -> [[Int]] -> [Int]
updateEntries [1,4,2] [[1,0,3], [0, -2, 1]] = [2,2,6]
```

- Schreiben Sie die Funktion *compare3*, die drei Listen als Eingabe erwartet. Verglichen werden dann jeweils die Elemente der Listen, die an selber Stelle stehen. Gilt  $x \leq y \leq z$ , soll die Ergebnisliste an selber Stelle True enthalten, ansonsten False.

Beispiel:

```
compare3 :: Eq a => [a] -> [a] -> [a] -> [Bool]
compare3 [1,2,3] [2,3,4] [5,1,5] = [True, False, True]
```

#### Aufgabe Z.4 Datentypen und Typklassen

- Schreiben Sie einen Datentyp *List a* analog zum Haskell-Standardtyp für Listen.
- Definieren Sie eine *Eq*-Instanz für *List a*.
- Definieren Sie eine *Ord*-Instanz für *List a*, sodass folgendes Verhalten erreicht wird:

```
"hallo" <= "hello" = True
[1,2,3,4,5] <= [1,2,3,5,5] = True
[1,3,4,5,6] <= [1,2,3,4,5] = False
```

- Definieren Sie eine *Show*-Instanz für *List a* analog zu dem standard Listendatentyp.

Gegeben seien folgende Datentypen:

```
data PosNat = One | Succ' PosNat
data Int' = Zero' | Plus PosNat | Minus PosNat
data Q = Div Int' PosNat
```

- Definieren Sie eine *Eq*-Instanz für *Q*.
- Definieren Sie eine *Ord*-Instanz für *Q*.
- Definieren Sie eine *Show*-Instanz für *Q*.

#### Aufgabe Z.5 Typen von Ausdrücken

Gehen Sie für die gegebenen Ausdrücke folgendermaßen vor:

- Ergänzen Sie die impliziten Klammern!
- Geben Sie an, ob der Ausdruck auswertbar ist oder ob ein Typfehler vorliegt!

- Ist der Ausdruck auswertbar, geben Sie seinen gesamten Typ und die Typen der verwendeten Variablen ( $f$ ,  $x$ ,  $y$ ,  $z$ ,  $v$ ) an!
- Ist der Ausdruck nicht auswertbar, begründen Sie kurz, wieso!

1. `\f -> f + 3`
2. `\x -> \y -> x y 2`
3. `zipWith and`
4. `foldr (:) [0]`
5. `foldl (:) [0]`
6. `\u v -> u $ 2 * v - 3 * u`
7. `flip replicate () 20`
8. `\f -> \x -> f x && x 5`
9. `concat $ map (\x -> [[x]]) [1..100]`
10. `filter (<= 10) "Hello_world!"`

### Aufgabe Z.6 (Lazy) Auswertung von Ausdrücken

Werten Sie die folgenden Ausdrücke schrittweise aus. Wenn nötig, nutzen Sie das Prinzip der *Lazy Evaluation*! Die Funktion `col` sei definiert als

```
col :: Int -> Int
col = if even n then n `div` 2 else 3*n + 1
```

1. `(\u -> u 'E') (negate . fromEnum)`
2. `reverse $ take 3 $ iterate pred 900`
3. `takeWhile (< 20) $ iterate col 18`
4. `(\y -> y (+)) $ (\g p -> p 3 $ g 2 2) (*)`
5. `foldr (++) [] [[1,2],[3,4],[5,6]]`

### Aufgabe Z.7 Multimenge

Gegeben sei folgende Datenstruktur **Multiset** `a`, die eine Multimenge modelliert. Die Liste `items` ist die Menge der enthaltenen Werte, die Funktion `count` gibt an, wie oft ein Element in der Multimenge enthalten ist (z.B. `count ms 'G' == 4`, also 'G' ist viermal in der Multimenge `ms` enthalten).

```
data Multiset a = Multiset {
    count :: a -> Int,
    items :: [a]
}
```

Folgende Bedingungen sollen für **Multiset** `a` immer gelten:

- `count ms x >= 0` für alle `x :: a` und `ms :: Multiset a`
- Wenn `count ms x == 0`, dann gilt `not $ x `elem` items ms`
- Wenn `count ms x > 0`, dann gilt `x `elem` items ms`
- `items ms` enthält jedes Element nur einmal

Implementieren Sie folgende Funktionen unter Beachtung der o.g. Invarianten!

1. `empty :: Multiset a` - eine leere Multimenge
2. `size :: Multiset a -> Int` - gibt die Anzahl der Elemente einer Multimenge zurück, insbesondere werden mehrfach auftretende Elemente auch mehrfach gezählt

3. **add, remove :: Eq a => Int -> a -> Multiset a -> Multiset a** - der Aufruf `add n x` fügt  $n$ -mal das Element  $x$  zu einer Multimenge hinzu bzw. entfernt  $n$  davon
4. **fromList :: Eq a => [a] -> Multiset a** - überführt eine Liste sinnvoll in eine Multimenge
5. **fromMultiset :: Multiset a -> [a]** - überführt eine Multimenge sinnvoll in eine Liste
6. **unionMS, meetMS, diffMS :: Eq a => Multiset a -> Multiset a -> Multiset a** - bildet die Vereinigung (bzw. den Schnitt, die Differenz) zweier Multimengen analog zur Mengenoperation (bzw. `meet`, `diff`)
7. **mapMS :: Eq b => (a -> b) -> Multiset a -> Multiset b** - analog zu `map` auf Listen. Hinweis: Die Funktion  $f$  bei einem Aufruf `mapMS f` ist i.A. *nicht* injektiv!

### Aufgabe Z.8 Instanziierung von Typklassen

Gegeben sei die folgende Typklasse **Stack**, die typische Operationen eines Kellerspeichers bereitstellt.

```
class Stack s where
  empty  :: s a
  push   :: a -> s a -> s a
  pop    :: s a -> Maybe (a, s a)
  peek   :: s a -> Maybe a
```

1. Schreiben Sie eine Stack-Instanz für `[]`.
2. Schreiben Sie eine Stack-Instanz für **Colist**.  
Zur Erinnerung: `Colist a = Colist { split :: Maybe (a, Colist a) }`
3. Schreiben Sie eine Funktion **fromStack :: (Eq a, Stack s) => s a -> Multiset a**, die alle Elemente eines Stacks in eine Multimenge legt.
4. Schreiben Sie eine Funktion **reverseStack :: Stack s => s a -> s a**, die die Reihenfolge der Elemente eines Stacks umkehrt.
5. Schreiben Sie eine Funktion **pops :: Stack s => Int -> s a -> ([a], s a)**, sodass `pops n` die ersten  $n$  Elemente von einem Stack entfernt und in einer Liste zusammen mit dem Reststack zurückgibt. Ist  $n$  größer als die Anzahl  $m$  der Elemente auf dem Stack, sollen die  $m$  Elemente zusammen mit dem leeren Stack zurückgegeben werden, insbesondere soll bei einem leeren Stack die Rückgabeliste leer sein.