technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2016/2017

Übungen zu Funktionaler Programmierung
Übungsblatt 4

Ausgabe: 4.11.2016, **Abgabe:** 11.11.2016

Aufgabe 4.1 (3 Punkte) Implementieren Sie folgende Funktionen in Haskell und geben Sie die Typen der Funktionen an.

1. $collatz(n) = \begin{cases} n/2, & \text{falls n gerade} \\ 3n+1, & \text{falls n ungerade} \end{cases}$

   Sie können die Haskell-Funktion **`div`** und **`even`** benutzen.

2. $f(n) = \begin{cases} 2 * n, & \text{falls n} < 10 \\ n + 30, & \text{sonst} \end{cases}$

3. $g(n) = \begin{cases} n + 10, & \text{falls n gerade} \\ g(n+3), & \text{falls n ungerade} \end{cases}$

   Sie können die Haskell-Funktion **`even`** benutzen.

Lösungsvorschlag Diese Funktionen lassen sich alternativ auch mit `if then else` lösen.

```
collatz :: Int -> Int
collatz n
  | even n    = div n 2
  | otherwise = 3 * n + 1

f :: Int -> Int
f n
  | n < 10    = 2 * n
  | otherwise = n + 30

g :: Int -> Int
g n
  | even n   = n + 10
  |otherwise = g (n + 3)
```

Aufgabe 4.2 (3 Punkte) Implementieren Sie folgende Listenfunktionen in Haskell und geben Sie die Typen der Funktionen an. Die Typen sollten möglichst allgemein sein.

1. `flatten` macht aus einer Liste von Listen eine einfache Liste.
   `flatten [[3,4,5], [], [2,3]]` ⇝ `[3,4,5,2,3]`
   `flatten [[[1,2], [3,4]], [[5]], [[6,7]]]` ⇝ `[[1,2],[3,4],[5],[6,7]]`

2. `toTupel` wandelt zwei Listen in eine Liste von Tupeln.
   `toTupel [1,2,3] [10,15,20,25,30]` ⇝ `[(1,10),(2,15),(3,20)]`

3. `takeUpTo` soll aus einer Liste so lange Elemente ausgeben, bis das Vergleichselement gefunden wurde.
   `takeUpTo 5 [1,6,5,3,8,5]` $\rightsquigarrow$ `[1,6]` Mit dem Operator **`(/=) :: Eq a => a -> a -> Bool`** kann auf Ungleichheit geprüft werden.

Lösungsvorschlag

1. Entspricht der Funktion `concat`.

```
flatten :: [[a]] -> [a]
flatten (x:xs) = x ++ flatten xs
flatten []     = []
```

2. Entspricht der Funktion `zip`.

```
toTupel :: [a] -> [b] -> [(a,b)]
toTupel (x:xs) (y:ys) = (x,y) : toTupel xs ys
toTupel _ _           = []
```

3. Entspricht `takeUpTo x = takeWhile (/=x)`.

```
takeUpTo :: Eq a =>  a -> [a] -> [a]
takeUpTo comp (x:xs)
  = if comp /= x then x : takeUpTo comp xs else []
takeUpTo _ [] = []
```

Aufgabe 4.3 (3 Punkte) Formen Sie folgende Funktionen, die Schleifen enthalten, in *endrekursive* Funktionen um.

1. Eine Potenzfunktion, die mit einer Multiplikation arbeitet.

```
int power(int base, int expo) {
   int state = 1;
   while (expo > 0) {
       state = state * base;
       expo = expo - 1;
   }
   return state;
}
```

2. Eine Funktion die alle Zahlen in einem Feld summiert. Benutzen Sie in Haksell eine Liste anstelle des Feldes.

```
int summe(int[] ls) {
    int state = 0;
    int i = 0;
    while (i < ls.length) {
        state = state + ls[i];
        i = i + 1;
    }
    return state;
}
```

Lösungsvorschlag

```
power :: Int -> Int -> Int
power base expo = loop 1 expo
  where
    loop state expo
      = if expo > 0 then loop (state * base) (expo - 1) else state

summe :: [Int] -> Int
summe ls = loop 0 ls
  where
    loop state (x:xs) = loop (state + x) xs
    loop state []     = state
```

Aufgabe 4.4 (3 Punkte) Werten Sie folgende Haskell-Ausdrücke schrittweise aus.

1. `take 2 $ tail [1,2,3,4,5]`
2. `head $ drop 2 [5,4,3,2,1]`

Lösungsvorschlag

1. `take 2 $ tail [1,2,3,4,5]`
   ⇝ `take 2 [2,3,4,5]`
   ⇝ `2 : take 1 [3,4,5]`
   ⇝ `2 : 3 : take 0 [4,5]`
   ⇝ `2 : 3 : []`
   ⇝ `[2,3]`
2. `head $ drop 2 [5,4,3,2,1]`
   ⇝ `head $ drop 1 [4,3,2,1]`
   ⇝ `head $ drop 0 [3,2,1]`
   ⇝ `head [3,2,1]`
   ⇝ `3`