Programmierung WS 18
Hausaufgaben - Blatt 8

Julian Giesen (MNR 388487)
Levin Gäher (MNR 395035)
Gruppe 12

HA 2

f (gT (tN 1) (f [ ]))
f (gT (1 : tN 0) (f [ ]))
f (gT (1 : [ ]) (f [ ]))
f (gT (1 : [ ]) (0))
f (1 : gT ([ ]) (0))
f (1 : [ ])
1

HA 4

a)

Die Gleichung stimmt nicht. Der : Operator fügt die Liste links des Operators in die Liste rechts des Operators ein. Der ++ Operator hängt die Liste links des Operators an die Liste rechts des Operators an.
$((x : []) : []) : [] = [[[Int]]]$ 1 Element vom Typ $[[Int]]$
$((x : []) : []) + +[] = [[Int]]$ 1 Element vom Typ $[Int]$

b)

Die Gleichung Stimmt. Beide Listen enthalten $m + 2$ Elemente vom Typ Int.

c)

Der Ausdruck $(x : y)$ ist syntaktisch inkorrekt, da y keine Liste ist. Der Ausdruck

$$(x : (y : (z : [])))$$

produziert eine Liste, die 3 Elemente vom Typ Int enthält.

d)

Die Gleichung stimmt. Beide Listen enthalten 2 Elemente vom Typ $[Int]$.

e)

Die Gleichung stimmt nicht. Der linke Ausdruck fügt durch den : Operator die Liste $[x, y]$ vom Typ $[Int]$ in die Liste $[[z]]$ vom Typ $[[Int]]$ ein, jedoch ohne x, y und z in einer Liste zu vereinen. Die neue Liste enthält zwei Elemente vom Typen $[Int]$. Der rechte Ausdruck fügt mit dem Operator : zunächst y in die Liste $[z]$, dann x in die Liste $[y, z]$ und dann die Liste $[x, y, z]$ in die leere Liste [] ein. Die resultierende Liste enthält somit ein Element vom Typen $[Int]$.

HA 6

```
fib :: Int -> Int
fib n  | n < 1   = 0
       | n == 1   = 1
       | otherwise = fib(n-2) + fib(n-1)

pow :: Int -> Int -> Int
pow a b | b == 0  = 1
       | otherwise = a * pow a (b-1)

modulo :: Int -> Int -> Int
modulo a b | a == 0    = 0
           | a < 0     = a + b
           | otherwise = modulo (a - b) b

isDiv :: Int -> Int -> Bool
isDiv a b  | modulo a b == 0 = True
           | otherwise       = False

sumUp :: [Int] -> Int
sumUp []   = 0
sumUp (x:xs) = x + sumUp xs

multLists :: [Int] -> [Int] -> [Int]
multLists (x:[]) (y:[]) = [x*y]
multLists (x:xs) (y:ys) = [x*y] ++ multLists xs ys
multLists _ _ = []

binRep :: Int -> (Int, [Int])
binRep n   | n == 0   = (0, [0])
           | n > 1    = (1, binRepHelper n)
           | n < 1    = (-1, binRepHelper ((-1)*n))

binRepHelper :: Int -> [Int]
binRepHelper n | n == 0 = []
               | otherwise = binRepHelper (div n 2) ++ [modulo n 2]
```