

Algorithmen und Programmierung

6. Aufgabenblatt

Aufgabe 6.1

a)

Signaturen

```
f :: Int -> String
g :: String -> Int
h :: Int -> Int
```

Beispiele

```
f x = [chr x]
g xs = foldr1 (+) (map ord xs)
h x = x*x
```

Sinnvolle Ausdrücke

```
*Programme> :t g.h.f
error
*Programme> :t h.f.g
error
*Programme> :t f.g.h
error

*Programme> :t f.h.g
f.h.g :: [Char] -> [Char]
*Programme> :t g.f.h
g.f.h :: Int -> Int
*Programme> :t h.g.f
h.g.f :: Int -> Int
```

b)

- a formaler Parameter (Variable) oder Funktion(-saufruf)
- `a` Infixoperator
- ´a´ ´a´ mit ´ ist nicht definiert.
- ´a´ wäre ein Char
- "a" Zeichenkette/String, bestehend aus dem einzelnen
 Buchstaben a.
- (a,2) Tupel mit einem formalen Parameter beliebigen Typs
 und einer 2 als Elemente.
- [a] Liste eines beliebigen Typs a oder Liste mit einem
 Element in Musterschreibweise.
- [a,b] Liste mit zwei Elementen in Musterschreibweise.
 Elemente a und b müssen vom selben Typ sein.
- (a,b) Tupel mit zwei formalen Parametern beliebigen Typs.

c)

Wir gehen davon aus, dass keine Annahmen über die Eigenschaften

(kommutativ, assoziativ) von op getroffen wurden!

- (1) **falsch**, weil zuerst b und c mit op verknüpft werden.
richtig, falls op assoziativ
 - (2) **falsch**, weil die Reihenfolge der Argumente a b vertauscht ist und die Kommutativität nicht vorausgesetzt wurde.
richtig, falls op kommutativ
 - (3) **richtig**, da op a b zuerst ausgewertet wird und dann op c auf das Ergebnis angewandt wird
 - (4) **falsch**, weil die Reihenfolge der Argumente c und $(op\ a\ b)$ vertauscht wurde und die Kommutativität nicht vorausgesetzt wurde.
richtig, falls op kommutativ
 - (5) **richtig**, $op\ (op\ a\ b)\ c$
- (3) und (5) bedeuten, ohne Aussagen über op treffen zu müssen, dasselbe wie $(a\ op\ b)\ op\ c$

d)

$[]:xs = xs$

Damit die linke Seite korrekt ist, muss xs eine Liste von Listen sein. Die linke Seite enthält im Gegensatz zur rechten Seite eine leere Liste mehr und ist deswegen **für alle xs falsch**.

$[]:xs = [[],xs]$

Damit die linke Seite korrekt ist, muss xs eine Liste von Listen sein. Auf der rechten Seite steht jedoch eine Liste die Listen von Listen enthält. Damit ist dieser Ausdruck **für alle xs falsch**.

$xs:xs = [[],xs]$

Die linke Seite kann nie einen korrekten Ausdruck ergeben, weil das zweite Argument des Listenkonstruktors immer eine Listenebene mehr haben muss als das erste Argument. Damit ist dieser Ausdruck **für alle xs falsch**.

$xs:[] = [xs]$

Dieser Ausdruck ist **für alle xs korrekt**.

Aufgabe 6.2

a)

Idee:

vergleiche alle Elemente mit dem ersten Element der Liste und falte die Wahrheitswerte mit $\&\&$

```

alleGleich :: Eq a => [a] -> Bool
alleGleich []      = True
alleGleich (x:xs) = foldr (&&) True [x==y | y <- xs]

-- Ohne ZF:
alleGleich2 :: Eq a => [a] -> Bool
alleGleich2 []      = True
alleGleich2 (x:xs) = foldr (f x) True xs
                    where f x y z = z && x==y

```

Testläufe:

```

*Programme> alleGleich [1..10]
False
*Programme> alleGleich [x | x <- [1], y <- [1..10]]
True
*Programme> alleGleich2 [1..10]
False
*Programme> alleGleich2 [x | x <- [1], y <- [1..10]]
True

```

b)

Idee:

prüfe für jedes x das Prädikat p und falte die Wahrheitswerte mit ||

```

exists :: Eq a => (a -> Bool) -> [a] -> Bool
exists p xs = foldr (||) False [p x | x <- xs]

-- Ohne ZF:
exists2 :: Eq a => (a -> Bool) -> [a] -> Bool
exists2 p xs = foldr (f) False xs
              where f x y = p x || y

```

Testläufe:

```

*Programme> exists (==2) [1..10]
True
*Programme> exists (==0) [1..10]
False
*Programme> exists2 (==2) [1..10]
True
*Programme> exists2 (==0) [1..10]
False

```

c)

Idee:

Für alle x aus xs: prüfe, ob x noch in xs enthalten ist und wenn ja, verwerfe x

```

unique' :: Eq a => [a] -> [a]
unique' xs = foldr help [] xs
          where
            -- verwerfe x, wenn noch in xs enthalten
            help a xs | exists (==a) xs = xs
                    | otherwise       = a:xs

```

Aufgabe 6.3

Framework

```
import Char

data Bit = O | I
    deriving (Show, Eq)

type Reg = [Bit]

-- Polynomrechnung nach dem Hornerschema:

horner :: Num a => a -> [a] -> a
horner x xs = foldl (horn x) 0 xs
    where horn :: Num a => a -> a -> a -> a
          horn x e b = e*x + b
```

a) Dezimalzahl dargestellt als String in Dezimalzahl umwandeln

```
-- wandelt einen String in eine Dezimalzahl um
decString2Dec :: String -> Int
decString2Dec xs = (horner 10 (map toNumber xs))
    where toNumber c | '0' <= c && c <= '9'
                    = ord c - ord '0'
                  | otherwise = error "No dec number"
```

Testlauf

```
*Programme> decString2Dec "123456789"
123456789
*Programme> decString2Dec "987654321"
987654321
*Programme> decString2Dec "A9"
*** Exception: No dec number
```

b) Binärzahl dargestellt als String in Dezimalzahl umwandeln

```
-- Wandelt eine Zahl in Registerdarstellung in
-- eine Dezimalzahl um
binString2Dec :: String -> Int
binString2Dec xs = (horner 2 (map toNumber xs))
    where toNumber c | c=='0' = 0
                    | c=='1' = 1
                    | otherwise = error "No bin number"
```

Testlauf

```
*Programme> binString2Dec "IO"
2
*Programme> binString2Dec "II"
3
*Programme> binString2Dec "IIAA"
*** Exception: No bin number
*Programme> binString2Dec "II "
*** Exception: No bin number
```

Hexadezimalzahl dargestellt als String in Dezimalzahl umwandeln

```
-- Wandelt einen String aus Hexadezimalzahlen in
-- eine Dezimalzahl um
hexString2Dec :: String -> Int
hexString2Dec xs = (horner 16 (map toNumber xs))
    where toNumber c | ('0'<=c && c<='9')= ord c - ord '0'
                  | ('A'<=c && c<='F')=10+ord c - ord 'A'
                  | otherwise = error "No hex number"
```

Testlauf

```
*Programme> hexString2Dec "F"
15
*Programme> hexString2Dec "FF"
255
*Programme> hexString2Dec "XX"
*** Exception: No hex number
```

c) Dezimalzahl in Binärzahl umwandeln

```
-- Dezimalzahl in Binärzahl umwandeln
dec2Bin :: Int -> [Int]
dec2Bin xs = dec2Bin2 16 xs

-- Dezimalzahl in Register der Länge n schreiben
-- Leere Stellen werden mit 0 aufgefüllt
dec2Bin2 :: Int -> Int -> [Int]
dec2Bin2 n xs = nullen ++ binNumber
    where binNumber = (reverse . dec2Bin3) xs
          nullen = replicate (n-length binNumber) 0

-- Dezimalzahl in Binärzahl umwandeln
dec2Bin3 :: Int -> [Int]
dec2Bin3 0 = []
dec2Bin3 n = (mod n 2):(dec2Bin3 (div n 2))
```

Testlauf

```
*Programme> dec2Bin2 8 10
[0,0,0,0,1,0,1,0]
*Programme> dec2Bin2 4 3
[0,0,1,1]

*Programme> dec2Bin 3
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1]
*Programme> dec2Bin 10
[0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0]
```

Aufgabe 6.4

Verschieben

Aufgabe 6.5

a) `nBitAdd` verwendet den Volladdierer von Blatt 4.

Für die eingegeben Zahlen x und y wird zuerst der 2er Logarithmus der Zahlen berechnet, um die benötigte Länge l der Register zu bestimmen. Anschließend werden die Zahlen in 2 Registern der Länge l addiert. Wenn es keine Überlauf gab, wird das Ergebnisregister wieder in eine Dezimalzahl umgewandelt.

Das Addierwerk von der letzten Übung

```
-- Addierwerk
bAdder xs ys = reverse ( bAdderH (reverse xs) (reverse ys) 0 )

-- Endrekursiver Addierer
bAdderH :: [Bit] -> [Bit] -> Bit -> [Bit]
bAdderH [] [] I = error "Ueberlauf" -- letzter Übertrag
bAdderH [] [] O = [] -- letzter Übertrag
bAdderH (x:xs) (y:ys) cin = sout:(bAdderH xs ys cout)
    where
        ( sout , cout ) = bVollAdder x y cin
bAdderH _ _ _ = error "Laengen sind nicht gleich"
```

Hilfsfunktionen

```
-- Da fromInt nicht mehr in der prelude enthalten ist
fromInt2 :: Num a => Int -> a
fromInt2 = fromInteger . toInteger

-- Berechnet die benötigte Länge der Register
getLength x y = max (ceiling (logBase 2 (x)))
                    (ceiling (logBase 2 (y))) + 1

-- Hilfsfunktion: Konvertiert ein Register in einen String
registerToString :: Reg -> String
registerToString xs = map fromBit xs
    where fromBit c | c==I = 'I'
                  | c==O = 'O'
```

Addierer von 2 Dezimalzahlen in Register

```
-- Addier 2 Dezimalzahlen
-- Dezimalzahlen werden zuerst in Register der Länge l
-- geschrieben und dann addiert
bIntToRegisterAdder :: Int -> Int -> Int -> Reg
bIntToRegisterAdder x y l = bAdder (map toBit (dec2Bin2 l x) )
    (map toBit (dec2Bin2 l y) )
    where toBit c | c==0 = O
                  | c==1 = I
```

Der Addierer

```
-- Addiert zwei Zahlen durch Register-Addition
nBitAdd :: Int -> Int -> Int
nBitAdd x y
    | length strSumme <= reglen = binString2Dec strSumme
    | otherwise = error "Ueberlauf"
    where
        -- benötigte Länge der Register
        -- berechnet durch den 2er Logarithmus
```

```
reglen = getLength (fromInt2 x) (fromInt2 y)

-- Konvertiere Register in String
strSumme :: String
strSumme = registerToString summe

-- Berechne die Summe 2er Zahlen in Register
summe :: Reg
summe = bIntToRegisterAdder x y reglen
```

Testlauf

```
*Programme> nBitAdd 10 10
20
*Programme> nBitAdd (2^15) (2^15)
*** Exception: Ueberlauf
*Programme> nBitAdd (2^15) (2^14)
49152
```

Bei der Addition von 2^x und 2^x gibt es einen Überlauf. Das ist logisch, da die Zahl 2^x die Registerlänge $x+1$ hat und im $x+1$ -ten Bit eine 1 steht. Wenn ich die Zahl zu sich selbst addiere gibt es einen Übertrag in das $x+2$ -te Bit. Das ist der Überlauf.

Patrick Schäfer et al.