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
Präsenzblatt 1

Aufgabe 1.1

Installieren Sie die Haskell-Platform (`http://www.haskell.org/platform/`) auf ihrem Rechner. Stellen Sie dabei sicher, dass `ghc` und `ghci` zu ihrer Pfadvariablen hinzugefügt sind.

1. Öffnen Sie den Texteditor Ihrer Wahl und tippen Sie folgendes Programm ab:

```
f :: Int -> Int -> Int -> Int
f x y z = x + y + z * z
```

2. Speichern Sie das Programm in einer Datei mit der Endung `.hs`. Den Pfad zu der Datei nennen wir im Folgenden `file.hs`.

3. Öffnen Sie die Kommandozeile und laden Sie die Datei mit dem interaktiven Modus des GHC (GHCi genannt), wie folgt: `ghci file.hs`
   Sie sollten nun die folgende Ausgabe erhalten:

```
[1 of 1] Compiling Main             ( file.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

4. Rufen Sie nun die Funktion `f` auf, indem Sie zum Beispiel `f 1 2 3` eingeben und mit ENTER bestätigen. Das Ergebnis wird ausgegeben und Sie können weitere Funktionsaufrufe auswerten lassen.

Folgende Kommandos des GHCi haben sich als nützlich erwiesen:

- `:load file` (kurz `:l`) lädt die Datei `file` in den GHCi.
- `:reload` (kurz `:r`) lädt die aktuelle Datei neu ein. Nachdem Änderungen an dem Quelltext vorgenommen wurden, kann die aktuelle Datei mit `:r` leicht neu geladen werden.
- `:type ausdruck` (kurz `:t`) zeigt den Typ des Ausdruckes `ausdruck` an, z.B. `:t f` oder `:t f 1 2 3`.
- `:help` (kurz `:h`) öffnet die Hilfe mit weiteren nützlichen Befehlen.
- `:quit` (kurz `:q`) beendet den GHCi.

Aufgabe 1.2

Die folgende Aufgabe enthält eine Reihe von fehlerhaften Haskell-Ausdrücken. Ziel dieser Aufgabe ist, dass Sie sich mit den Fehlermeldungen des GHCi vertraut machen. Laden Sie dazu die Datei aus Aufgabe 1 und interpretieren Sie die folgenden Ausdrücke mit dem GHCi. Versuchen Sie die Fehlermeldungen nachzuvollziehen.

1. `f 3 1 True`

2. `f 4 3 2 1`
3. `f 3 2 1`
4. `foo 3 2 1`

Aufgabe 1.3

Gegeben sei die Funktion `addFive` und die konstante Funktion `one`:

```
addFive :: Int -> Int
addFive x = x + 5

one :: Int
one = 1
```

1. Definieren Sie eine Konstante `k :: Int`, deren Auswertung die Zahl `11` liefert. Nutzen Sie dabei nur die Funktionen `addFive` und `one`.
2. Definieren Sie eine Funktion `addTen :: Int -> Int`, die eine ganze Zahl als Parameter erhält und die Summe aus Zehn und der übergebenen Zahl berechnet. Nutzen Sie dafür die Funktion `addFive` und den Dollar-Operator.
3. Definieren Sie eine Funktion `addTenComposition :: Int -> Int`, die semantisch äquivalent zu der Funktion `addTen` ist, jedoch die Funktion `addFive` und die Funktionskomposition nutzt.

Aufgabe 1.4

Werten Sie den folgenden Ausdruck unter Angabe von Zwischenergebnissen aus:

$$(\lambda y.(\lambda x.x + 1)\ (4 + y))\ 6$$

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 2

Ausgabe: 21.10.2016, **Abgabe:** 28.10.2016

Aufgabe 2.1 (2 Punkte) Folgende Polynomfunktionen sind vereinfacht dargestellt.

a. $5x^2 + 14x + 6$

b. $56x^3 + 9x$

Formen Sie diese in vollständige Lambda-Ausdrücke um.

Lösungsvorschlag Die Variabel muss zusammen mit einem λ vorangestellt werden. Zusätzlich müssen die Multiplikationszeichen wieder eingefügt werden.

a. $\lambda x.5 * x^2 + 14 * x + 6$

b. $\lambda x.56 * x^3 + 9 * x$

Aufgabe 2.2 (8 Punkte) Folgende Lambda-Ausdrücke sind gegeben.

a. $(\lambda x.x + 3)(23)$

b. $(\lambda x.x + 1)(5, 3)$

c. $(\lambda(x, y).x * 2 + y)(5, 2)$

d. $(\lambda(x, y, z).x * y + (100 * z) + x)(10)$

e. $(\lambda f.f(5))(\lambda x.x + 2)$

1. Geben Sie an, welche Ausdrücke sich nicht auswerten lassen. Begründen Sie ihre Antwort.

   Lösungsvorschlag

   b. Das Muster matcht zwar, aber der Ausdruck $(5, 3) + 1$ kann nicht ausgewertet werden. Es handelt sich um einen Typfehler.

   d. Kann nicht ausgewertet werden, weil das Muster nicht matcht.

2. Werten Sie die übrigen Ausdrücke schrittweise aus.

   Lösungsvorschlag Jeweils 2 Punkte

   a. $(\lambda x.x + 3)(23) \rightsquigarrow 23 + 3 \rightsquigarrow 26$

   c. $(\lambda(x, y).x * 2 + y)(5, 2) \rightsquigarrow 5 * 2 + 2 \rightsquigarrow 10 + 2 \rightsquigarrow 12$

   e. $(\lambda f.f(5))(\lambda x.x + 2) \rightsquigarrow (\lambda x.x + 2)(5) \rightsquigarrow 5 + 2 \rightsquigarrow 7$

Aufgabe 2.3 (2 Punkte) Geben Sie die Typen folgender Haskell-Funktionen an.

a. `f1 = \x -> x + 1`

b. `f2 = \(x, y) -> x * y`

Dabei sind die Typen `1 :: Int`, `(+) :: (Int, Int) -> Int` und `(*) :: (Int, Int) -> Int` gegeben.

Lösungsvorschlag

a. Gegeben: `1 :: Int` und `(+) :: (Int, Int) -> Int`
Daraus folgt:
`x :: Int` $\wedge$ `x + 1 :: Int`
$\Longrightarrow$ `\x -> x + 1 :: Int -> Int`
$\Longrightarrow$ `f1 :: Int -> Int`

b. Gegeben: `(*) :: (Int, Int) -> Int`
Daraus folgt:
`x :: Int` $\wedge$ `y :: Int` $\wedge$ `x * y :: Int`
$\Longrightarrow$ `\(x, y) -> x * y :: (Int, Int) -> Int`
$\Longrightarrow$ `f2 :: (Int, Int) -> Int`

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 3

Ausgabe: 28.10.2016, **Abgabe:** 4.11.2016

Aufgabe 3.1 (3 Punkte) Schreiben Sie die Funktionen **`curry3`** und **`uncurry3`** für dreistellige Funktionen bzw. Tripel (`(a,b,c)`). Geben Sie auch die Typen an.

Lösungsvorschlag

```
curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
curry3 f a b c = f (a, b, c)

uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d
uncurry3 f (a, b ,c) = f a b c
```

Aufgabe 3.2 (3 Punkte) Fügen Sie die impliziten Klammern in folgende Haskell-Ausdrücke ein.

1. `x * y * z + 3`
2. `add3 1 2 3`
3. `f $ g . h x`

Lösungsvorschlag

1. `((x * y) * z) + 3`
2. `((add3 1) 2) 3`
3. `f $ (g . (h x))`

Aufgabe 3.3 (3 Punkte) Werten Sie folgende Haskell-Ausdrücke schrittweise aus.

1. `(\n -> 5 * n) $ (\m -> 1 + m) 4`
2. `(\f -> (f . f) 5) (+3)`

Lösungsvorschlag

1. `(\n -> 5 * n) $ (\m -> 1 + m) 4`
   `↝ (5 * ((\m -> 1 + m) 4))`
   `↝ (5 * (1 + 4))`
   `↝ (5 * 5)`
   `↝ 25`

2. `(\f -> (f . f) 5) (+3)`
   `⇝ ((+3) . (+3)) 5`
   `⇝ (+3) ((+3) 5)`
   `⇝ (+3) 8`
   `⇝ 11`

Aufgabe 3.4 (3 Punkte) Geben Sie die geforderten Typen an oder ob es sich um einen Typfehler handelt.

1. Gegeben:
   `(5+) a`
   `(+) :: Int -> Int -> Int`

   Gesucht sind die Typen von **`a`** und **`(5+) a`**.

2. Gegeben:
   `(==) :: a -> a -> Bool`

   Gesucht ist der Typ von **`True == 5`**.

3. Gegeben:
   `f $ g . h`

   Gesucht sind die Typen von **`f`**, **`g`** und **`h`**.

Lösungsvorschlag

1. `a :: Int`
   `(5+) a :: Int`

2. Typfehler

3. `f :: (a -> c) -> d`
   `g :: b -> c`
   `h :: a -> b`

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`

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 5

Ausgabe: 11.11.2016, **Abgabe:** 18.11.2016

Aufgabe 5.1 (4 Punkte) Lösen Sie die folgenden Aufgaben mithilfe der Funktion **`map`** oder **`zipWith`** (Funktionslifting).

1. **`double :: [Int] -> [Int]`** multipliziert alle Ganzzahlen in einer Liste mit zwei.
   Beispiel: `double [1,2,3]` ⇝ `[2,4,6]`
2. **`maxima :: [Int] -> [Int] -> [Int]`** vergleicht die Werte aus zwei Listen und gibt eine Liste mit den Maxima zurück.
   Beispiel: `maxima [10,3,5,23] [1,14,6,10]` ⇝ `[10,14,6,23]`
3. **`funs :: Int -> [Int]`** erzeugt eine Liste mit drei Werten. Dem Nachfolger der Eingabe, dem Doppeltem der Eingabe und die Eingabe hoch 2.
   Beispiel: `funs 3` ⇝ `[4,6,9]`
4. **`toUnicode :: String -> [Int]`** wandelt einen `String` in eine Liste der Unicode-Codierungen der einzelnen Zeichen um. Mit der Funktion `fromEnum` kann ein Zeichen (`Char`) in seine Codierung gewandelt werden.
   Beispiel: `toUnicode "%hello!"` ⇝ `[37,104,101,108,108,111,33]`

Lösungsvorschlag

```
double :: [Int] -> [Int]
double = map (*2)

maxima :: [Int] -> [Int] -> [Int]
maxima = zipWith (\x y -> if x < y then y else x)
-- alternativ:
-- maxima = zipWith max

funs :: Int -> [Int]
funs i = map ($ i) [(+1), (*2), (^2)]

toUnicode :: String -> [Int]
toUnicode = map fromEnum
```

Aufgabe 5.2 (4 Punkte) Werten Sie folgende Haskell-Ausdrücke schrittweise aus.

1. `foldl (-) 5 [1, 3]`
2. `foldr (-) 5 [1, 3]`

Lösungsvorschlag

```
-- foldl (-) 5 [1, 3] = (5-1)-3
foldl (-) 5 [1, 3]
~> foldl (-) ((-) 5 1) [3]
~> foldl (-) 4 [3]
~> foldl (-) ((-) 4 3) []
~> foldl (-) 1 []
~> 1

-- foldr (-) 5 [1, 3] = (1-(3-5))
foldr (-) 5 [1, 3]
~> foldr (-) 5 [1, 3]
~> (-) 1 $ foldr (-) 5 [3]
~> (-) 1 $ (-) 3 $ foldr (-) 5 []
~> (-) 1 $ (-) 3 $ 5
~> (-) 1 ((-) 3 5)
~> (-) 1 (-2)
~> 3
```

Aufgabe 5.3 (4 Punkte) Definieren Sie folgende Funktionen mithilfe der Listenkomprehension.

1. `inBoth :: [Int] -> [Int] -> [Int]` gibt nur die Werte aus der ersten Liste aus, die auch in der zweiten Liste vorkommen.
   Beispiel: `inBoth [1,2,3,4] [1,4,5,3,4]` $\rightsquigarrow$ `[1,3,4]`

2. `map2 :: (a -> b -> c) -> [a] -> [b] -> [c]` wendet einen binären Operator auf jeder Kombination von Werten aus zwei Eingabelisten.
   Beispiel: `map2 (+) [1,2] [10,20,30]` $\rightsquigarrow$ `[11,21,31,12,22,32]`

3. `divisors :: Int -> [Int]` gibt eine Liste aller Teiler des Eingabewertes.
   Beipsiel: `divisors 12` $\rightsquigarrow$ `[1,2,3,4,6,12]`

4. `solutions :: [(Int, Int, Int)]` enthält Tripel $(x, y, z) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, welche die Gleichung $3x^2 + 2y + 1 = z$ lösen. Nehmen Sie für x, y und z nur Werte von 0 bis 100.

Lösungsvorschlag

```
inBoth :: [Int] -> [Int] -> [Int]
inBoth xs ys = [ x | x <- xs, x `elem` ys]

map2 :: (a -> b -> c) -> [a] -> [b] -> [c]
map2 f xs ys = [f x y | x <- xs, y <- ys]

divisors :: Int -> [Int]
divisors n = [ m | m <- [1..n], n `mod` m == 0 ]

solutions :: [(Int, Int, Int)]
solutions = [ (x,y,z)
  | z <- [0..100]
  , y <- [0..100]
  , x <- [0..100]
  , 3*x^2 + 2*y + 1 == z
  ]
```

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 6

Ausgabe: 18.11.2016, **Abgabe:** 25.11.2016

Aufgabe 6.1 (4 Punkte)

1. Werten Sie folgenden Haskell-Ausdruck mit unendlicher Liste *lazy* aus. Werten Sie schrittweise immer nur den Teil aus, bei dem ein Ergebnis benötigt wird, um weiter auszuwerten zu können.
   `iterate (*2) 3 !! 2`

2. Schreiben Sie die Liste `solutions :: [(Int, Int, Int)]` aus Aufgabe 5.3 so um, dass sie *alle* Lösung der Gleichung $3x^2 + 2y + 1 = z$ enthält. Die Liste wird dadurch unendlich lang. Beispiel: `solutions !! 700` $\rightsquigarrow$ `(7,38,224)`

Lösungsvorschlag

1. ```
   iterate (*2) 3 !! 2
   ⇝ (3:iterate (*2) ((*2) 3)) !! 2
   ⇝ (iterate (*2) ((*2) 3)) !! 1
   ⇝ (((*2) 3):iterate (*2) ((*2) ((*2) 3))) !! 1
   ⇝ (iterate (*2) ((*2) ((*2) 3))) !! 0
   ⇝ (((*2) ((*2) 3)):iterate (*2) ((*2) ((*2) ((*2) 3)))) !! 0
   ⇝ ((*2) ((*2) 3))
   ⇝ (*2) 6
   ⇝ 12
   ```

2. Um eine Endlosschleife zu vermeiden, darf nur die erste Liste in der Komprehension unendlich sein. Da für $x, y > z$ keine Lösungen mehr existieren können, ist dies eine sinnvolle Einschränkung.

   ```
   solutions :: [(Int , Int , Int )]
   solutions = [ (x,y,z)
     | z <- [0..] , y <- [0..z] , x <- [0..z]
     , 3*x^2 + 2*y + 1 == z]
   ```

Aufgabe 6.2 (2 Punkte)

1. Erweitern Sie die vorgestellten Datentypen für Zahlen um einen Datentyp für rationale Zahlen. Basieren Sie den Datentyp nur auf den anderen Datentypen für Zahlen (`Nat, Int', PosNat`).

2. Definieren Sie eine Konstante $c = -\frac{1}{2}$ für Ihren Datentyp.

Lösungsvorschlag
   ```

```
data Rat = Rat Int' PosNat

c :: Rat
c = Rat (Minus One) (Succ' One)
```

Aufgabe 6.3 (4 Punkte) Definieren Sie folgende Haskell-Funktionen.

1. `indexNat :: [a] -> Nat -> a`, wie **`(!!)`** für den Datentyp **`Nat`** anstatt **`Int`**.
2. `colistTake :: Int -> Colist a -> Colist a`, wie **`take`** nur für **`Colist a`** anstatt **`[a]`**.
3. `getX :: Point -> Double` liest die x-Koordinate eines Punktes aus.
4. `setX :: Double -> Point -> Point` setzt die x-Koordinate eines Punktes auf den angegebenen Wert.

Die Datentypen `Point` und `Colist` sind wie folgt definiert:

```
data Point = Point Double Double
data Colist a = Colist (Maybe (a, Colist a))
-- Liste 5:3:[] als Colist:
example = Colist (Just (5, Colist (Just (3, Colist Nothing))))
```

Lösungsvorschlag

```
indexNat :: [a] -> Nat -> a
indexNat (a:_) Zero = a
indexNat (_:s) (Succ n) = indexNat s n

colistTake :: Int -> Colist a -> Colist a
colistTake 0 _              = Colist Nothing
colistTake n (Colist (Just (a,s)))
  | n > 0 = Colist (Just (a, colistTake (n-1) s))
colistTake _ (Colist Nothing) = Colist Nothing

getX :: Point -> Double
getX (Point x _) = x

setX :: Double -> Point -> Point
setX x (Point _ y) = Point x y
```

Aufgabe 6.4 (2 Punkte)

1. Zeigen Sie, dass der Typ `[()]` isomorph zu `Nat` ist. Schreiben Sie zwei Funktionen **`to :: [()] -> Nat`** und **`from :: Nat -> [()]`**, welche die Bedingungen `to . from = id` und `from . to = id` erfüllen.

Lösungsvorschlag

```
to :: [()] -> Nat
to (():xs) = Succ (to xs)
to []      = Zero

from :: Nat -> [()]
from (Succ n) = () : from n
from Zero     = []
```

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 7

Ausgabe: 25.11.2016, **Abgabe:** 2.12.2016

Aufgabe 7.1 (3 Punkte)

1. Definieren Sie den Typ `Person` mit den Attributen (Destruktoren) `name`, `familyName`, `age`. Benutzen Sie sinnvolle Typen für die Attribute.
2. Schreiben Sie die Funktionen `getX` und `setX` aus der Aufgabe 6.3 so um, dass ausschließlich Destruktoren verwendet werden.

Der Datentyp `Point` sei jetzt wie folgt definiert:

```
data Point = Point{ x:: Double, y :: Double }
```

Lösungsvorschlag

```
data Person = Person{ name :: String, familyName :: String, age :: Int }

getX :: Point -> Double
getX = x

setX :: Double -> Point -> Point
setX x pt = pt{x = x}
```

Aufgabe 7.2 (3 Punkte)

1. Stellen Sie den Ausdruck $3x^2 + 2y + 1$ als Element vom Typ `Exp String` da.
2. Schreiben Sie die Listenkomprehension `solutions :: [(Int,Int,Int)]` um. Machen Sie gebrauch von `exp2store`.

Lösungsvorschlag

1.
```
expr :: Exp String
expr = Sum [3 :* (Var "x" :^ 2), 2 :* Var "y", Con 1]

solutions :: [(Int,Int,Int)]
solutions = [(x,y,z) | z <- [0..], x <- [0..z], y <- [0..z]
            , exp2store expr (st x y) == z
            ]
  where
    st x y "x" = x
    st x y "y" = y
```

Aufgabe 7.3 (3 Punkte) Schreiben Sie eine Funktion `simplify :: Exp x -> Exp x`, welche arithmetische Ausdrücke vereinfacht. Dabei sollen folgende Gleichungen zur Vereinfachung genutzt werden:

$$0 + x = x \qquad x^0 = 1$$
$$0 * x = 0 \qquad x^1 = x$$
$$1 * x = x$$

Lösungsvorschlag

```
simplify :: Exp x -> Exp x
-- Potenzen
simplify (e :^ 0) = Con 1
simplify (e :^ 1) = simplify e
simplify (e :^ i) = simplify e :^ i
-- Summen
simplify (Sum []) = Con 0
simplify (Sum es) = Sum $ filter (conEqInt 0) $ map simplify es
-- Produkte
simplify (Prod []) = Con 1
simplify (Prod es)
  | any (conEqInt 0) simplified = Con 0
  | otherwise  = Prod $ filter (conEqInt 1) simplified
  where simplified = map simplify es
simplify (i :* e) = simplMul (i :* simplify e)
  where
    simplMul (0 :* _) = Con 0
    simplMul (_ :* Con 0) = Con 0
    simplMul (1 :* e) = e
    simplMul (i :* Con 1) = Con i
    simplMul e = e
-- Sonstiges
simplify (e1 :- e2) = simplify e1 :- simplify e2
simplify e = e

conEqInt :: Int -> Exp x -> Bool
conEqInt i (Con c) = i == c
conEqInt _ _       = False
```

Aufgabe 7.4 (3 Punkte) Schreiben Sie eine Funktion `bexp2store`, welche sich ähnlich wie `exp2store` verhält. Anstelle arithmetischer Ausdrücke sollen boolesche Ausdrücke vom Typ `BExp x` ausgewertet werden. Diese Funktion benötigt zwei Variablenbelegungen. Eine für boolesche Ausdrücke und die andere für arithmetische Ausdrücke.

Benutzen Sie folgende Typen:

```
type BStore x = x -> Bool
bexp2store :: BExp x -> BStore x -> Store x -> Bool
```

Lösungsvorschlag

```
bexp2store :: BExp x -> BStore x -> Store x -> Bool
bexp2store True_ _ _ = True
bexp2store False_ _ _ = False
bexp2store (BVar x) bst _ = bst x
```

```
bexp2store (Or bs) bst st = or $ map (\x -> bexp2store x bst st) bs
bexp2store (And bs) bst st = and $ map (\x -> bexp2store x bst st) bs
bexp2store (Not bs) bst st = not $ bexp2store bs bst st
bexp2store (e1 := e2) _ st = exp2store e1 st == exp2store e2 st
bexp2store (e1 :<= e2) _ st = exp2store e1 st <= exp2store e2 st
```

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 8

Ausgabe: 2.12.2016, **Abgabe:** 9.12.2016 - 12:00 Uhr

Aufgabe 8.1 (3 Punkte) Schreiben Sie eine Klasse für eine überladene Additionsfunktion. Instanziieren Sie die Klasse für `Nat` und `PosNat`.

Lösungsvorschlag

```
class Addition a where
  add :: a -> a -> a

instance Addition Nat where
  add Zero n     = n
  add (Succ n) m = Succ (add n m)

instance Addition PosNat where
  add One n      = Succ' n
  add (Succ' n) m = Succ' (add n m)
```

Aufgabe 8.2 (6 Punkte)

1. Schreiben Sie eine Instanz der Klasse `Eq` für den Typ `Nat`.

2. Schreiben Sie eine Instanz der Klasse `Ord` für den Typ `Nat`. Es ist ausreichend den Operator `(<=)` zu definieren.

3. Schreiben Sie eine Instanz der Klasse `Enum` für den Typ `Nat`. Es ist ausreichend die Funktionen `toEnum` und `fromEnum` zu definieren.
   Beispiel:
   `take 3 $ map fromEnum [Zero .. ]` ⇝ `[0,1,2]`

4. Schreiben Sie eine Instanz der Klasse `Show` für den Typ `Nat`. Nehmen Sie an, die Klasse sei wie folgt definiert:

   ```
   class Show a where
     show :: a -> String
   ```

   Die Definition der `show`-Funktion soll Ausgaben ähnlich denen vom Typ `Int` geben:

   ```
   show Zero ~> "0"
   show (Succ Zero) ~> "1"
   show (Succ (Succ (Succ Zero))) ~> "3"
   ```

5. Schreiben Sie eine Instanz der Klasse `Num` für den Typ `Nat`. Nutzen Sie folgende Vorgabe:

```
instance Num Nat where
  negate = undefined
  abs n  = n
  signum Zero = Zero
  signum n    = Succ Zero
  fromInteger = toEnum . fromInteger
```

Sie müssen lediglich die fehlenden Operatoren `(+)` und `(*)` definieren (Stichwort: Peano-Axiome).

6. Ändern Sie den Typ der unendlichen Liste `solutions` in `[(Nat,Nat,Nat)]`. Die Lösung soll auf Aufgabe 6.1.2 basieren, und weiterhin als Listenkomprehension definiert werden. Die Funktion `exp2store` wird nicht benötigt. Sie dürfen dabei alle zuvor definierten Klasseninstanzen nutzen.

Lösungsvorschlag

```
instance Eq Nat where
  Zero == Zero     = True
  Succ n == Succ m = n == m
  _ == _           = False

instance Ord Nat where
  Succ n <= Succ m = n <= m
  Zero <= _        = True
  _ <= _           = False

instance Enum Nat where
  toEnum 0          = Zero
  toEnum n | n > 0  = Succ (toEnum (n - 1))
  fromEnum Zero     = 0
  fromEnum (Succ n) = fromEnum n + 1

instance Show Nat where
  show = show . fromEnum

instance Num Nat where
  Zero + n     = n
  (Succ n) + m = Succ (n + m)
  Zero * n     = Zero
  (Succ n) * m = m + (n * m)
  negate = undefined
  abs n  = n
  signum Zero = Zero
  signum n    = Succ Zero
  fromInteger = toEnum . fromInteger

solutions :: [(Nat,Nat,Nat)]
solutions = [ (x,y,z)
  | z <- [0..] , y <- [0..z], x <- [0..z], 3*x^2 + 2*y + 1 == z ]
```

Aufgabe 8.3 (3 Punkte) Definieren Sie folgende Haskell-Funktionen:

1. Symmetrische Differenz: $A \Delta B$.

2. `height :: Bintree a -> Int` berechnet die Höhe eines binären Baumes.

3. `isBalanced :: Bintree a -> Bool` testet, ob ein binärer Baum ausbalanciert ist.

Lösungsvorschlag

```
symDiff :: Eq a => [a] -> [a] -> [a]
symDiff a b = (a `diff` b) `union` (b `diff` a)
-- oder: symDiff a b = (a `union` b) `diff` (a `meet` b)

height :: Bintree a -> Int
height (Fork _ l r) = 1 + max (height l) (height r)
height Empty = 0

isBalanced :: Bintree a -> Bool
isBalanced (Fork _ l r) = (difference <= 1)
  && isBalanced l
  && isBalanced r
  where difference = abs (height l - height r)
isBalanced Empty = True
```

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 9

Ausgabe: 9.12.2016, **Abgabe:** 16.12.2016 - 12:00 Uhr

Aufgabe 9.1 (3 Punkte) Schreiben Sie Traversierungsfunktionen für `Bintree`.

1. Die Funktion `preorderB` soll die Knoten des Baumes als Liste in Hauptreihenfolge (pre-order) ausgeben.
2. Die Funktion `postorderB` soll die Knoten des Baumes als Liste in Nebenreihenfolge (post-order) ausgeben.

Lösungsvorschlag

```
preorderB :: Bintree a -> [a]
preorderB (Fork a l r) = a : preorderB l ++ preorderB r
preorderB Empty                = []

postorderB :: Bintree a -> [a]
postorderB (Fork a l r) = postorderB l ++ postorderB r ++ [a]
postorderB Empty        = []
```

Aufgabe 9.2 (3 Punkte) Bestimmen Sie die Typen der folgenden Ausdrücke. Die Aufgabe soll mithilfe von Typinferenzregeln gelöst werden.

1. `zipWith (+)`
2. `\x -> \f -> f x`

Lösungsvorschlag

1.
$$\frac{\texttt{zipWith :: (a->a->a) -> [a] -> [a] -> [a]}, \texttt{(+) :: Num a => a->a->a}}{\texttt{zipWith (+) :: Num a => [a] -> [a] -> [(a,a)]}}$$

2.
$$\frac{\texttt{x :: a}, \quad \dfrac{\texttt{f :: a -> b}, \quad \dfrac{\texttt{f :: a -> b}, \texttt{x :: a}}{\texttt{f x :: b}}}{\texttt{\f -> f x :: (a -> b) -> b}}}{\texttt{\x -> \f -> f x :: a -> (a -> b) -> b}}$$

Aufgabe 9.3 (6 Punkte)

1. Modellieren folgende Eigenschaften mit Datentypen.
   - Eine Bank führt eine Liste von Konten.

- Ein Konto hat einen Kontostand und einen Kunden als Besitzer.
- Für einen Kunden werden die Daten Vorname, Name und Adresse (`String`) gespeichert.

2. Legen Sie eine Beispielbank an mit mindestens zwei Konten.
3. Definieren Sie folgende Funktionen. Benutzen Sie für `ID` den Typ `Int` (`type ID = Int`).

   `credit :: Int -> ID -> Bank -> Bank` Addiert den angegebenen Betrag auf das angegebene Konto. Hinweis: Schauen Sie sich die Funktion `updList` an.

   `debit :: Int -> ID -> Bank -> Bank` Subtrahiert den angegebenen Betrag von dem angegebenen Konto.

   `transfer :: Int -> ID -> ID -> Bank -> Bank` Überweist den angegebenen Betrag vom ersten Konto auf das zweite.

Lösungsvorschlag

```
type ID = Int
data Bank = Bank [Account] deriving Show
data Account = Account { balance :: Int, owner :: Client } deriving Show
data Client = Client
  { name :: String
  , surname :: String
  , address :: String
  } deriving Show

own1, own2 :: Client
own1 = Client "Max" "Mustermann" "Musterhausen"
own2 = Client "John" "Doe" "Somewhere"

acc1, acc2 :: Account
acc1 = Account 100 own1
acc2 = Account 0 own2

bank :: Bank
bank = Bank [acc1, acc2]

credit :: Int -> ID -> Bank -> Bank
credit amount id (Bank ls)
  = Bank (updList ls id entry{ balance = oldBalance + amount})
  where
    entry = ls !! id
    oldBalance = balance entry

debit :: Int -> ID -> Bank -> Bank
debit amount = credit (-amount)

transfer :: Int -> ID -> ID -> Bank -> Bank
transfer amount id1 id2 = debit amount id1 . credit amount id2
```

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 10

Ausgabe: 16.12.2016, **Abgabe:** 9.1.2017 - 12:00 Uhr

Aufgabe 10.1 (3 Punkte) Definieren Sie folgende Funktionen als Baumfaltungen (`foldBtree` bzw. `foldTree`) und geben Sie den Typ an. Wählen Sie den Typ möglichst allgemein.

`product_` Produkt aller Zahlen in einem Baum mit beliebigen Ausgrad (`Tree`).

`inorderB` Gibt die Knoten eines Binärbaumes (`Bintree`) als Liste in symmetrischer Reihenfolge (in-order) wieder.

Lösungsvorschlag

```
product_ :: Num a => Tree a -> a
product_ = foldTree id (*) 1 (*)

inorderB :: Bintree a -> [a]
inorderB = foldBtree [] (\a l r -> l ++ [a] ++ r)
```

Aufgabe 10.2 (3 Punkte)

1. Schreiben Sie die Faltung `foldNat` für den Typen `Nat`.
2. Schreiben Sie eine Funktion `toInt` die Werte vom Typ `Nat` in entsprechende Werte vom Typ `Int` wandelt. Benutzen Sie dazu die Faltung `foldNat`.

Lösungsvorschlag

```
foldNat :: val -> (val -> val) -> Nat -> val
foldNat val _ Zero = val
foldNat val f (Succ n) = f (foldNat val f n)

toInt :: Nat -> Int
toInt = foldNat 0 (+1)
```

Aufgabe 10.3 (3 Punkte) Geben Sie die Kommandosequenz für die Auswertung `execute (exp2code expr) ([],vars)` an. Zu jedem Kommando soll auch der Stapelinhalt (Stack) nach der Ausführung angegeben werden. Dabei sei der Ausdruck `expr` und die Belegungsfunktion `vars` wie folgt definiert:

```
expr :: Exp String
expr = Sum [3 :* Var "x", Con 5]

vars :: Store String
vars "x" = 2
```

Mit `getResult (execute (exp2code expr) ([],vars))` kann das Ergebnis 11 angezeigt werden. Diese Hilfsfunktion wird für die Bearbeitung der Aufgabe nicht benötigt.

```
getResult :: State x -> Int
getResult = head . fst
```

Lösungsvorschlag

```
Kommandos:     Stapel:
Push 3         [3]
Load "x"       [2,3]
Mul 2          [6]
Push 5         [5,6]
Add 2          [11]
```

Aufgabe 10.4 (3 Punkte) Schreiben Sie eine überladene `hash`-Funktion, welche einen Hash vom Typ `Int` erzeugt. Instanziieren Sie die Funktion sinnvoll für die Typen `Nat`, `[a]` und `Tree a`.

Lösungsvorschlag

```
class Hash a where
  hash :: a -> Int

instance Hash Nat where
  hash Zero = 31
  hash (Succ n) = 31 + hash n

instance Hash a => Hash [a] where
  hash [] = 961
  hash (a:as) = 31 * (31 + hash a) + hash as

instance Hash a => Hash (Tree a) where
  hash (V a) = 31 * (31 + hash a)
  hash (F a ls) = 31 * (31 + hash a) + hash ls
```

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 11

Ausgabe: 13.1.2016, **Abgabe:** 20.1.2017 - 12:00 Uhr

Aufgabe 11.1 (4 Punkte)

1. Zeigen Sie, dass die Rekursionsgleichung `fib` eine Funktion definiert. Definieren Sie dazu eine Schrittfunktion Φ analog zu der auf Folie 127.

2. Beweisen Sie durch Induktion, dass $\text{lfp}(\Phi)$ keine natürliche Zahl auf $\bot$ abbildet.

Lösungsvorschlag

1.

$$\begin{aligned}\Phi : (\mathbb{N} \to \mathbb{N}_\bot) &\to (\mathbb{N} \to \mathbb{N}_\bot)\\ f &\mapsto \lambda n.\text{if } n > 1 \text{ then } f(n-1) + f(n-2) \text{ else } 1\end{aligned}$$

2. Induktionsvoraussetzung: $\text{lfp}(\Phi)(n) \neq \bot$ und $\text{lfp}(\Phi)(m) \neq \bot$ für alle $m < n$.
Induktionsanfang:

$$\begin{aligned}&\text{lfp}(\Phi)(0)\\ =&\Phi(\text{lfp}(\Phi))(0)\\ =&(\lambda n.\text{if } n > 1 \text{ then } \text{lfp}(\Phi)(n-1) + \text{lfp}(\Phi)(n-2) \text{ else } 1)(0)\\ =&1 \neq \bot\end{aligned}$$

$$\begin{aligned}&\text{lfp}(\Phi)(1)\\ =&\Phi(\text{lfp}(\Phi))(1)\\ =&(\lambda n.\text{if } n > 1 \text{ then } \text{lfp}(\Phi)(n-1) + \text{lfp}(\Phi)(n-2) \text{ else } 1)(1)\\ =&1 \neq \bot\end{aligned}$$

Induktionsschritt:

$$\begin{aligned}&\text{lfp}(\Phi)(n+1)\\ =&\Phi(\text{lfp}(\Phi))(n+1)\\ =&(\lambda n.\text{if } n > 1 \text{ then } \text{lfp}(\Phi)(n-1) + \text{lfp}(\Phi)(n-2) \text{ else } 1)(n+1)\\ =&\text{lfp}(\Phi)(n) + \text{lfp}(\Phi)(n-1)\\ &(\textit{Induktionsvoraussetzung})\\ \neq&\bot\end{aligned}$$

Aufgabe 11.2 (4 Punkte) Definieren Sie folgende Haskell-Funktionen.

1. `isCyclic :: Eq a => Graph a -> Bool` erkennt, ob ein Graph zyklisch ist. Sie können hier den transitiven Abschluss nutzen.
   Beispiele:
   `isCyclic graph1` ⇝ `True`
   `isCyclic graph2` ⇝ `False`

2. `undirected :: Eq a => Graph a -> Graph a` macht aus einem Graphen einen ungerichteten Graphen. Ein ungerichteter Graph lässt sich als gerichteter Graph darstellen, indem für jede Kante eine Kante in die Gegenrichtung eingefügt wird.
   Beispiel: `undirected graph1` ⇝
   1 -> [2,3,4]; 2 -> [1,6]; 3 -> [1,4,6,5]; 4 -> [1,3,6]; 5 -> [3,5,6]; 6 -> [2,4,5,3]

Lösungsvorschlag

```
isCyclic :: Eq a => Graph a -> Bool
isCyclic graph = or (map self nodes) where
  G nodes closure = closureF graph
  self node = node `elem` closure node

undirected :: Eq a => Graph a -> Graph a
undirected (G nodes sucs) = G nodes sucs' where
  sucs' n = sucs n `union` [n' | n' <- nodes, n `elem` sucs n']
```

Aufgabe 11.3 (4 Punkte) Definieren Sie folgende partielle Funktionen mithilfe des Maybe-Datentyps.

1. `safeDiv :: Int -> Int -> Maybe Int` berechnet die Division zweier Ganzzahlen. Berücksichtigt jedoch die Teilung durch 0.

2. `safeSqrt :: Int -> Maybe Int` berechnet die Wurzel für positive Ganzahlen.
   Hilfsfunktion:

   ```
   intsqrt :: Int -> Int
   intsqrt = floor . sqrt . fromIntegral
   ```

3. `f :: Int -> Int -> Int -> Maybe Int` berechnet folgende Funktion:

$$f(x, y, z) = \frac{\sqrt{x}}{\sqrt{\frac{y}{z}}}$$

   Lösen Sie die Aufgabe mit den Funktionen `safeDiv` und `safeSqrt`. Sie können die Funktionen aus dem Modul `Data.Maybe` zu Hilfe nehmen. Dazu müssen Sie die Zeile „`import Data.Maybe`“ in Ihre .hs-Datei einfügen. Detaillierte Informationen zu `Data.Maybe` finden Sie mit Hoogle (https://www.haskell.org/hoogle/).

Lösungsvorschlag

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)

safeSqrt :: Int -> Maybe Int
safeSqrt x
  | x < 0     = Nothing
  | otherwise = Just (intsqrt x)
```

```
f :: Int -> Int -> Int -> Maybe Int
f x y z = if isNothing par1 || isNothing par2
  then Nothing
  else safeDiv (fromJust par1) (fromJust par2)
  where
    par1 = safeSqrt x
    par2 = maybe Nothing safeSqrt (safeDiv y z)
```

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 12

Ausgabe: 20.1.2016, **Abgabe:** 27.1.2017 - 12:00 Uhr

Aufgabe 12.1 (2 Punkte) Gegeben sei folgende Listenkomprehension:

```
solutions :: [(Int , Int , Int )]
solutions = [ (x,y,z) | z <- [0..] , y <- [0..z] , x <- [0..z]
            , 3*x^2 + 2*y + 1 == z]
```

1. Überführen Sie die Listenkomprehension in die do-Notation.
2. Überführen Sie die do-Notation in monadische Operatoren und Funktionen(»=,»,`return`).

Lösungsvorschlag

1. do-Notation

```
solutions' :: [(Int , Int , Int )]
solutions' = do
   z <- [0..]
   y <- [0..z]
   x <- [0..z]
   guard $ 3*x^2 + 2*y + 1 == z
   return (x,y,z)
```

2. »=-Notation

```
solutions'' :: [(Int , Int , Int )]
solutions'' =
  [0..] >>= \z ->
  [0..z] >>= \y ->
  [0..z] >>= \x ->
  (guard $ 3*x^2 + 2*y + 1 == z) >>
  return (x,y,z)
```

Aufgabe 12.2 (2 Punkte) *Hinweis: Für diese Aufgabe muss Abschnitt 7.1* Maybe- und Listenmonaden *gelesen werden.*
Definieren Sie die Haskell-Funktion `f` vom Typ `Int -> Int -> Int -> Maybe Int`. Wie in Aufgabe 11.3 soll diese mit den Funktionen `safeDiv` und `safeSqrt` gelöst werden und die Gleichung $f(x,y,z) = \frac{\sqrt{x}}{\sqrt{\frac{y}{z}}}$ erfüllen. Diesmal soll jedoch sonst nur die Monadeneigenschafft von `Maybe` genutzt werden.

```
intsqrt :: Int -> Int
intsqrt = floor . sqrt . fromIntegral

safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)

safeSqrt :: Int -> Maybe Int
safeSqrt x
  | x < 0     = Nothing
  | otherwise = Just (intsqrt x)
```

Lösungsvorschlag

```
f :: Int -> Int -> Int -> Maybe Int
f x y z = do
  yz <- safeDiv y z
  sqrt1 <- safeSqrt x
  sqrt2 <- safeSqrt yz
  safeDiv sqrt1 sqrt2
```

Aufgabe 12.3 (4 Punkte) *Hinweis: Für diese Aufgabe muss Abschnitt 7.10* Schreibermonaden *gelesen werden.*
Schreiben Sie einen Algorithmus, welche die Operation $2*(3+1)+5$ ausführt und dabei jeden Schritt protokolliert.
Gehen Sie dazu schrittweise vor.

1. Schreiben Sie die Funktionen `addW` und `multW`, welche zwei Ganzzahlen addieren bzw. multiplizieren und den Vorgang protokollieren. Definieren Sie die Funktionen mithilfe der `do`-Notation und der Funktion `tell`. Die Funktion `tell` schreibt einen beliebigen `String` in das Protokoll.
   Vorgaben:

   ```
   type Writer s a = (s,a)

   tell :: s -> Writer s ()
   tell s = (s,())

   addW, multW :: Int -> Int -> Writer String Int
   ```

2. Definieren Sie das Programm `progW` vom Typ `Writer String Int`, welches die Operation $2*(3+1)+5$ ausführt. Nutzen Sie auch hier die `do`-Notation.
   Beispiel: `progW` $\rightsquigarrow$ `("3+1 = 4\n2*4 = 8\n8+5 = 13\n",13)`

Lösungsvorschlag

1. Addition und Multiplikation

   ```
   addW x y = do
     tell $ show x ++ "+" ++ show y ++ "␣=␣" ++ show r ++ "\n"
     return r
     where r = x + y

   multW x y = do
     tell $ show x ++ "*" ++ show y ++ "␣=␣" ++ show r ++ "\n"
     return r
     where r = x * y
   ```

2. $2 * (3 + 1) + 5$

```
progW :: Writer String Int
progW = do
  r1 <- addW 3 1
  r2 <- multW 2 r1
  addW r2 5
```

Aufgabe 12.4 (4 Punkte) *Hinweis: Für diese Aufgabe muss Abschnitt 7.9* Lesermonaden *gelesen werden.*
Schreiben Sie die Funktion `bexp2store` aus Aufgabe 7.4 so um, dass sie Gebrauch von der Lesermonade macht. Es gelten folgende Typen:

```
type BStore x = x -> Bool
bexp2store :: BExp x -> Store x -> BStore x -> Bool
```

Lösungsvorschlag

```
bexp2store True_ _ = return True
bexp2store False_ _ = return False
bexp2store (BVar x) _ = ($x)
bexp2store (Or bs) st = do
  is <- mapM (\x -> bexp2store x st) bs
  return (or is)
bexp2store (And bs) st = do
  is <- mapM (\x -> bexp2store x st) bs
  return (and is)
bexp2store (Not bs) st = do
  i <- bexp2store bs st
  return (not i)
bexp2store (e1 := e2) st
  = return (exp2store e1 st == exp2store e2 st)
bexp2store (e1 :<= e2) st
  = return (exp2store e1 st <= exp2store e2 st)
```

Alternativ kann auch der Stil aus Aufgabe 12.3 verwendet werden.

```
type Reader s a = s -> a

ask :: Reader s s
ask = id

bexp2store' ::  BExp x -> Store x -> Reader (BStore x) Bool
bexp2store' (BVar x) _ = do
  bst <- ask
  return (bst x)
```

Die restlichen Muster (Pattern) von `bexp2store'` seien exakt gleich definiert wie bei `bexp2store`.

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 13

Ausgabe: 27.1.2016, **Abgabe:** 3.2.2017 - 12:00 Uhr

Hinweis: *Für die Bearbeitung des Übungsblattes müssen die Folien 188–201 gelesen werden.*

Aufgabe 13.1 (3 Punkte) Implementieren Sie eine Funktion

```
filtM :: MonadPlus m => (a -> Bool) -> [a] -> m a,
```

welche die Funktion `filter` verallgemeinert. Die Funktion `filtM` soll sich bei einer Festlegung des Rückgabetyps auf eine Liste von Werten wie `filter` verhalten:

```
filtM (<1) [1,-2,3,-4,5] :: [Int] ⇝ [-2,-4]
```

Darüber hinaus soll `filtM` aber beispielsweise auch für einen in `Maybe` eingebetteten Wert funktionieren und dann den ersten Wert in der Liste zurückgeben, der das Prädikat erfüllt:

```
filtM (<1) [1,-2,3,-4,5] :: Maybe Int ⇝ Just (-2)
```

Gibt es keinen Wert, für den das Prädikat gilt, gibt die Funktion für den Rückgabetyp `Maybe Int` den Fehlerwert `Nothing` zurück.

Lösungsvorschlag

```
filtM :: MonadPlus m => (a -> Bool) -> [a] -> m a
filtM _ [] = mzero
filtM p (x:xs)
  | p x       = return x `mplus` filtM p xs
  | otherwise = filtM p xs
```

Aufgabe 13.2 (4 Punkte) Gegeben sei folgender Code:

```
type PointMethod = Trans (Double,Double)

getX, getY :: PointMethod Double
getX = T $ \(x,y) -> (x,(x,y))
getY = T $ \(x,y) -> (y,(x,y))

setX, setY :: Double -> PointMethod ()
setX x = T $ \(_,y) -> ((),(x,y))
setY y = T $ \(x,_) -> ((),(x,y))
```

Definieren Sie folgende Haskell-Funktionen mithilfe der Transitionsmonade. Konstruktor (`T`) und Destruktor (`runT`) der Transitionsmonade dürfen *nicht* benutzt werden. Machen Sie gebrauch von der `do`-Notation und den oben definierten gettern und settern.

1. Die Funktion `pointSwap` vom Typ `PointMethod ()` soll die x-Koordinate und die y-Koordinate eines Punktes vertauschen.
   Beispiel: `runT pointSwap (3,16)` ⇝ `((),(16.0,3.0)`

2. Die Funktion `pointDistance` vom Typ `(Double, Double) -> PointMethod Double` soll die Distanz zu einem anderen Punkt angeben. Die Funktion kann analog zu der Funktion `distance` gelöst werden.
   Beispiel:
   `runT (pointDistance (0,0)) (3,16)` $\rightsquigarrow$ `(16.278820596099706,(3.0,16.0))`

3. Die Funktion `prog` vom Typ `PointMethod Double` soll folgende Java-Methode simulieren:

```
double prog() {
    double x, y;
    x = getX();
    y = getY();
    pointSwap();
    x = pointDistance(x,y);
    pointSwap();
    return x;
}
```

   Beispiel: `runT prog (3,16)` $\rightsquigarrow$ `(18.384776310850235,(3.0,16.0))`

Lösungsvorschlag

1. `pointSwap`

```
pointSwap :: PointMethod ()
pointSwap = do
  x <- getX
  y <- getY
  setX y
  setY x
```

2. `pointDistance`

```
pointDistance :: (Double, Double) -> PointMethod Double
pointDistance (x2, y2) = do
  x1 <- getX
  y1 <- getY
  return $ sqrt $ (x2-x1)^2 + (y2-y1)^2
```

3. `prog`

```
prog :: PointMethod Double
prog = do
  x <- getX
  y <- getY
  pointSwap
  x <- pointDistance (x,y)
  pointSwap
  return x
```

Aufgabe 13.3 (2 Punkte) Schreiben Sie eine Funktion `main` vom Typ `IO ()`, die

1. eine Eingabe aus der Konsole ausliest,
2. in Großbuchstaben wandelt
3. und in eine Datei speichert.

Für die Aufgabe darf die Funktion `toUpper` aus dem Modul `Data.Char` benutzt werden. Kompilieren Sie die Datei mit dem Befehl „**`ghc <dateiname.hs>`**“. Dadurch wird eine ausführbare Datei generiert. Testen Sie diese. Als Lösung ist der Quellcode der Funktion anzugeben.

Lösungsvorschlag

```
main :: IO ()
main = do
  putStrLn "Eingabe:␣"
  str <- getLine
  writeFile "out.txt" (map toUpper str)
```

Aufgabe 13.4 (3 Punkte) Gegeben sei folgende rekursive Berechnung der *i*-ten Catalan-Zahl:

```
catalan :: Int -> Int
catalan 0 = 1
catalan n = sum $ map (\i -> catalan i * catalan (n-1-i)) [0..n-1]
```

Definieren Sie eine Funktion `catalanDyn` vom Typ `Int -> Int`, welche das Problem mithilfe der dynamischen Programmierung löst.

Lösungsvorschlag

```
catalanDyn :: Int -> Int
catalanDyn n = catalanArr ! n where

  catalanArr :: Array Int Int
  catalanArr = mkArray (0,n) cata

  cata :: Int -> Int
  cata 0 = 1
  cata n = sum
    $ map (\i -> catalanArr ! i * catalanArr ! (n-1-i)) [0..n-1]
```