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]
```