04IN1023: Grundlagen der funktionalen Programmierung

Klausur SoSe 2023

Universität Koblenz, FB4
Prof. Dr. Ralf Lämmel,
M.Sc., M.Ed. Hakan Aksu
17 August 2023

Name, Vorname	
Matrikel-Nr.	
Email	.....@uni-koblenz.de
Studiengang	☐ BSc Inf ☐ BSc CV ☐ .....
Prüfungsversuch	☐ 1 ☐ 2 ☐ 3 ☐ 4
Prüfungsordnung	☐ $>= PO2019$ ☐ $< PO2019$

Hiermit bestätige ich, dass ich zur Klausur angemeldet und zugelassen bin.
Eine falsche Angabe wird als Täuschungsversuch gewertet.

Unterschrift: _____

Korrekturabschnitt

Aufgabe	**Punkte (0-2)**
1	
2	
3	
4	
5	
6	
7	
8	
9	

1 “Simple Algorithmen”

Implementieren Sie die Funktion *replaceEO*, die einen *String* einliest. Im Text soll jedes ’E’ durch die Zahl 3 und jedes ’O’ durch die Zahl 8 ersetzt werden.

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
        replaceEO "HALLO WELT!" ~?= "HALL8 W3LT!",
        replaceEO "FUNKTIONALE PROGRAMMIERUNG"
        ~?= "FUNKTI8NAL3 PR8GRAMMI3RUNG"
    ])

replaceEO :: String -> String
```

Reference solution

```
replaceEO [] = []
replaceEO (x:xs) = if x=='E' then ['3'] ++ replaceEO xs
                   else if x=='O' then ['8'] ++ replaceEO xs
                   else [x] ++ replaceEO xs
```

2 “Suchen und Sortieren”

Implementieren Sie die Funktion *iSortTupel*, die ein Array mit Integer-Tupeln und einer Integer-Zahl einliest und ein Array mit Integer-Tupeln zurückliefert. Die Funktion soll mithilfe des Insertionsort-Algorithmus die Tupel aufsteigend gemäß ihrer Summe sortieren. Sollte ein Wert aus dem Tupel echt größer dem Wert aus der eingelesenen Zahl sein, so soll dieser weggelassen werden.

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
        iSortTupel [(10, 20), (30, 5), (15, 8), (2, 7)] 25 ~?= [(2,7),(15,8),(10,20)],
        iSortTupel [(5, 1), (4, 4), (4, 3) , (2, 7)] 6 ~?= [(5,1),(4,3),(4,4)]
    ])

iSortTupel :: [(Int, Int)] -> Int -> [(Int, Int)]
```

Reference solution

```
iSortTupel [] _ = []
iSortTupel ((x, y):xs) maxVal
    | x > maxVal || y > maxVal = iSortTupel xs maxVal
    | otherwise = insertTupel (x, y) (iSortTupel xs maxVal)
    where
        insertTupel :: (Int, Int) -> [(Int, Int)] -> [(Int, Int)]
        insertTupel tupel [] = [tupel]
        insertTupel (a, b) ((c, d):ys)
            | a + b <= c + d = (a, b) : (c, d) : ys
            | otherwise = (c, d) : insertTupel (a, b) ys
```

3 “Simple Datenmodelle”

Deklarieren Sie einen vereinfachten Datentypen zur Verwaltung von zwei Arten von Formen. Eine Form kann entweder ein Kreis mit einem Radius oder ein Rechteck mit zwei Seiten sein. Der Radius und die Seiten sind jeweils Gleitkommazahlen.

Reference solution

```
data Form = Kreis Radius | Rechteck Seite Seite
type Radius = Float
type Seite = Float
```

4 “Abstrakte Datentypen”

Definieren Sie einen abstrakten Datentypen für eine Druckerwarteschlange mit dem Namen *PrinterQueue* und den Funktionen *add*, *next* und *removeNext* - illustriert wie folgt:

```
> add "D3" (Node "D1" (Node "D2" Empty))
Node "D1" (Node "D2" (Node "D3" Empty))

> next (Node "D1" (Node "D2" (Node "D3" Empty)))
Just "D1"

> removeNext (Node "D1" (Node "D2" (Node "D3" Empty)))
Node "D2" (Node "D3" Empty)
```

Reference solution

```
module PrinterQueue (PrinterQueue, add, next, removeNext) where

data PrinterQueue = Empty | Node String PrinterQueue

add :: String -> PrinterQueue -> PrinterQueue
add item Empty = Node item Empty
add item (Node x rest) = Node x (add item rest)

next :: PrinterQueue -> Maybe String
next Empty = Nothing
next (Node item _) = Just item

removeNext :: PrinterQueue -> PrinterQueue
removeNext Empty = Empty
removeNext (Node _ rest) = rest
```

5 “Funktionale Datenstrukturen”

Gesucht wird ein ADT MindMap mit Operationen *createNode*, *addChild* und *addParent*. Die Funktion *createNode* erzeugt das erste MindMap-Element mit einem Begriff. Die Funktion *addChild* fügt einen untergeordneten Knoten hinzu. Die Funktion *addParent* fügt einen übergeordneten Knoten hinzu.

Gegeben sind die Signaturen der Funktionen

```
createNode :: String -> MindMap

addChild :: String -> MindMap -> MindMap

addParent :: String -> MindMap -> MindMap
```

Gegeben sind einige Anwendungsbeispiele

```
> createNode "W1"
Node "W1" []

> addChild "W11" (Node "W1" [])
Node "W1" [Node "W11" []]
> addChild "W12" (Node "W1" [Node "W11" []])
Node "W1" [Node "W12" [],Node "W11" []]

> addParent "W0" (Node "W1" [Node "W12" [],Node "W11" []])
Node "W0" [Node "W1" [Node "W12" [],Node "W11" []]]
```

Definieren Sie den Datentypen *MindMap* und implementieren Sie die Funktionen `createNode`, `addChild` und `addParent`

Reference solution

```
data MindMap = Node String [MindMap]
    deriving (Show)

createNode :: String -> MindMap
createNode term = Node term []

addChild :: String -> MindMap -> MindMap
addChild t1 (Node t2 c) = Node t2 (Node t1 [] : c)

addParent :: String -> MindMap -> MindMap
addParent t1 (Node t2 c) = Node t1 [Node t2 c]
```

6 “Funktionen höherer Ordnung”

Gegeben ist eine Liste von Integer-Tupeln. Schreiben Sie eine Funktion namens *doubleEven*, die diese Liste entgegennimmt und die geraden Zahlen in jedem Tupel verdoppelt. Ungerade Zahlen sollen in der Ergebnisliste unverändert bleiben.

Implementieren Sie die Funktion doubleEven unter Verwendung von *map*.

Hinweise:

```
map :: (a -> b) -> [a] -> [b]

tests :: Test
tests = TestLabel "Klausur" (TestList [
        doubleEven [(2, 5), (4, 6), (7, 1), (3, 8)] ~?= [(4, 5), (8, 12), (7, 1), (3, 16)],
        doubleEven [(1, 3), (5, 9), (2, 4)] ~?= [(1, 3), (5, 9), (4, 8)]
    ])

doubleEven :: [(Int, Int)] -> [(Int, Int)]
```

Reference solution

```
doubleEven = map doubleEvenTupel
  where
    doubleEvenTupel :: (Int, Int) -> (Int, Int)
    doubleEvenTupel (x, y) = (dEven x, dEven y)

    dEven :: Int -> Int
    dEven n = if even n then n * 2 else n
```

Alternative:

```
doubleEven :: [(Int, Int)] -> [(Int, Int)]
doubleEven = map (\(x, y) -> (dEven x, dEven y))
  where
    dEven n = if even n then n * 2 else n
```

7 “Typ-Klassen Polymorphismus”

Erstellen Sie eine Typklasse namens Shape für geometrische Formen. Die Typklasse soll eine Funktion *area* definieren, die die Fläche einer geometrischen Form berechnet.

Implementieren Sie Instanzen der Typklasse Shape für die folgenden geometrischen Formen:

Rechteck (Rectangle):
- Ein Rechteck wird durch seine Breite und Höhe dargestellt.
- Die Fläche eines Rechtecks kann mit der Formel Fläche = Breite x Höhe berechnet werden.

Kreis (Circle):
- Ein Kreis wird durch seinen Radius dargestellt.
- Die Fläche eines Kreises kann mit der Formel Fläche = π x Radius x Radius berechnet werden (wobei π etwa 3.14159 ist).

Schreiben Sie die Definitionen für die Typklasse *Shape* und die Instanzen für *Rectangle* und *Circle*, um die gewünschte Funktionalität zu erreichen.

Anwendungsbeispiel:

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
        area (Rectangle 5.0 3.0) ~?= 15.0,
        area (Circle 2.0) ~?= 12.56636 ])
```

Reference solution

```
class Shape a where
    area :: a -> Double

data Rectangle = Rectangle Double Double
data Circle = Circle Double

instance Shape Rectangle where
    area (Rectangle width height) = width * height

instance Shape Circle where
    area (Circle radius) = pi * radius * radius
```

8 “Functors & Friends”

Gegeben ist ein Datentyp für zwei nichtleere Listen:

```
data DoubleBag a = DoubleBag {
  firstBag :: [a],
  secondBag :: [a]
} deriving (Eq, Show)
```

Gesucht werden Functor- und Foldable-Instanzen für *DoubleBag* – illustriert wie folgt:

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
    fmap (+2) (DoubleBag [1,2,3] [2,6,0] ~?= DoubleBag [3,4,5] [4,8,2]),
    foldr (+) 0 (DoubleBag [1,2,3] [2,6,0] ~?= 14)
  ])
```

Reference solution

```
instance Functor DoubleBag
  where
    fmap f (DoubleBag first second) = DoubleBag (fmap f first) (fmap f second)

instance Foldable DoubleBag
  where
    foldr f z (DoubleBag first second) = foldr f z (first ++ second)
```

9 “Monaden”

Gegeben ist eine Liste von Waggons mit einer Ladebezeichnung und dessen Gewicht. Implementieren Sie eine Funktion calculateSumWeight, die die Summe der Gewichte der Waggons berechnet und gleichzeitig eine Liste von Nachrichten generiert, die für jeden Waggon die Nachricht enthält, was und wieviel geladen wurde. Verwenden Sie dafür die Writer-Monade.

Gegeben ist folgender Datentyp mit folgenden Funktionen:

```
type Wagon = (String, Float)

getName :: Wagon -> String
getName (x,_) = x

getWeight :: Wagon -> Float
getWeight (_,x) = x
```

Gegeben ist folgendes Anwendungsbeispiel:

```
testWagon :: [Wagon]
testWagon = [ ("Kies", 900.0)
            , ("Sand", 500.0)
            , ("Kohle", 600.0)
            , ("Benzin", 700.0)
            ]

> calculateSumWeight testWagon
WriterT (Identity (2700.0,["Wagon-Inhalt: Kies Gewicht: 900.0 kg",
  "Wagon-Inhalt: Sand Gewicht: 500.0 kg",
  "Wagon-Inhalt: Kohle Gewicht: 600.0 kg",
  "Wagon-Inhalt: Benzin Gewicht: 700.0 kg"]))
```

Implementieren Sie die Funktion:

```
calculateSumWeight :: [Wagon] => Writer [String] Float
```

Reference solution

```
calculateSumWeight [] = do return 0
calculateSumWeight (x:xs) = do
p1 <= calculateSumWeight xs
tell ["Wagon-Inhalt: " ++ (getName x) ++ " Gewicht: " ++
  (getWeight x) ++ " kg geladen"]
return ((getWeight x) + p1)
```

Alternative:

```
calculateSumWeight WagonList = do
    let weights = map snd WagonList
        sumWeights = sum weights
    tell $ map (\(name, weight) -> "Wagon-Inhalt: " ++ name ++
        " Gewicht: " ++ weight ++ " kg")
        WagonList
    return sumWeights
```