

04IN1023: Grundlagen der funktionalen Programmierung

Klausur SoSe 2020

Universität Koblenz-Landau, FB4

Prof. Dr. Ralf Lämmel,

M.Sc., M.Ed. Hakan Aksu

28 July 2020

Name, Vorname	
Matrikel-Nr.	
Email@uni-koblenz.de
Studiengang	<input type="checkbox"/> BSc Inf <input type="checkbox"/> BSc CV <input type="checkbox"/>
Prüfungsversuch	<input type="checkbox"/> 1 <input type="checkbox"/> 2 <input type="checkbox"/> 3

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 *countSentences*, die einen String einliest und die Sätze zählt. Sätze enden mit einem Punkt und einem anschließendem Leerzeichen. Die einzige Ausnahme ist das Ende des Strings: Dort reicht ein Punkt aus.

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
    countSentences "" ~?= 0,
    countSentences "Ein Satz." ~?= 1,
    countSentences "Kein Satz!" ~?= 0,
    countSentences "Erster Satz. Zweiter Satz." ~?= 2,
    countSentences "Das ist ein Satz. Dieses Smily -.- ist kein Satz." ~?= 2,
    countSentences "A. B. C. D..." ~?= 4,
])

countSentences :: String -> Int
```

Reference solution

```
countSentences [] = 0
countSentences (x:[]) = if x == '.'
    then 1
    else 0
countSentences (x:y:xs) = if x == '.' && y == ' '
    then 1 + countSentences xs
    else countSentences (y:xs)
```

2 “Suchen und Sortieren”

Implementieren Sie die Funktion *getMinMax*, die einen Integer-Array einliest und das kleinste und größte Element in einem 2-Tupel (min,max) zurückgibt.

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
  getMinMax [] ~?= Nothing,
  getMinMax [5,2] ~?= Just (2,5),
  getMinMax [4,8,1,5] ~?= Just (1,8)
])

getMinMax :: [Int] -> Maybe (Int,Int)
```

Reference solution

```
getMinMax [] = Nothing
getMinMax (x:xs) = Just (searchMin x xs, searchMax x xs)
  where
    searchMin min [] = min
    searchMin min (x:xs) = if x < min then searchMin x xs else searchMin min xs
    searchMax max [] = max
    searchMax max (x:xs) = if x > max then searchMax x xs else searchMax max xs
```

3 “Simple Datenmodelle”

Deklarieren Sie einen vereinfachten Datentypen zur Verwaltung von Fahrern und Fahrzeugen in einer Fahrschule. Ein *Fahrlehrer* hat einen Namen vom Typ String und eine Liste von zugewiesenen Fahrzeugen. Die Fahrzeuge können LKW's, PKW's oder Motorräder sein. LKW's haben eine ID als Integer-Wert und das Gesamtgewicht als Float-Wert. PKW's haben eine ID als Integer-Wert und die Getriebeart (Schalt/Automatik) als Boolean-Wert. Motorräder haben eine ID als Integer-Wert und die Hubraumgröße als Integer-Wert.

Reference solution

```
data Fahrlehrer = Fahrlehrer Name [Fahrzeug]
data Fahrzeug = LKW ID Float | PKW ID Boolean | Motorrad ID Integer
type ID = Integer
```

4 “Funktionale Datenstrukturen”

Gegeben ist folgender Stack-ADT:

data *MyStack a = Empty | Push a (MyStack a)*

empty :: MyStack a
empty = Empty

push :: a -> MyStack a -> MyStack a
push = Push

pop :: MyStack a -> Maybe (MyStack a, a)
pop Empty = Nothing
pop (Push x s) = Just (s, x)

Implementieren Sie zusätzlich die Funktion *getN*. Diese Funktion gibt die nächsten *n* Elemente in einem Stack zurück.

Beispiel:

> getN 2 (Push 9 \$ Push 8 \$ Push 7 \$ Push 6 \$ Empty)
Push 9 (Push 8 Empty)

getN :: Int -> MyStack a -> MyStack a

Reference solution

getN :: Int -> MyStack a -> MyStack a
getN _ Empty = Empty
getN 0 (Push x s) = Empty
getN n (Push x s) = Push x (getN (n-1) s)

5 “Funktionen höherer Ordnung”

Implementieren Sie die Funktion *sumWordsLength*, die als Parameter einen String-Array nimmt. Die Längen der Strings sollen zusammenaddiert und zurückgegeben werden. Es ist nicht erlaubt einen lokalen Scope zu benutzen (keine Hilfsfunktionen und -variablen). Anonyme Funktionen sind erlaubt!

Hinweise:

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

```
tests :: Test
```

```
tests = TestLabel "Klausur" (TestList [  
    sumWordsLength [] ~?= 0,  
    sumWordsLength ["Hallo", "Welt"] ~?= 9,  
    sumWordsLength ["Grundlagen", "der", "funktionalen", "Programmierung"] ~?= 39  
])
```

```
sumWordsLength :: [String] -> Int
```

Reference solution

```
sumWordsLength xs = foldr (\x y -> (y + (length x))) 0 xs
```

Alternative:

```
sumWordsLength xs = foldr (+) 0 (map (\x -> length x) xs)
```

6 “Typ-Klassen Polymorphismus”

Implementieren Sie zwei Instanzen der Klasse MyGeometrie für die Datentypen Kreis (Circle) und Rechteck (Rectangle).

```
data Circle = Circle Float
data Rectangle = Rectangle Float Float
```

```
class MyGeometry a
  where
    area :: a -> Float
    circumference :: a -> Float
```

Folgend sind die Formeln für die Fläche (area) und den Umfang (circumference):

- Kreisfläche: πr^2
- Kreisumfang: $2\pi r$
- Rechtecksfläche: ab
- Rechtecksumfang: $2a + 2b$

Hier noch einige Hinweise:

```
sampleC1 = Circle 3
sampleR1 = Rectangle 1 2
```

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
  area sampleC1 ~?= 28.274334,
  area sampleR1 ~?= 2,
  circumference sampleC1 ~?= 18.849556,
  circumference sampleR1 ~?= 6
])
```

```
Prelude> pi
3.141592653589793
```

Reference solution

```
instance MyGeometry Circle
  where
    area (Circle r) = pi * r ^ 2
    circumference (Circle r) = 2 * pi * r

instance MyGeometry Rectangle
  where
    area (Rectangle a b) = a * b
    circumference (Rectangle a b) = 2 * a + 2 * b
```

7 “Functors & Friends”

Betrachten Sie die folgende Datenstruktur für Binäre Bäume.

```
data BTree a = Leaf a | Node a (BTree a) (BTree a)
```

Implementieren Sie die entsprechende Instanz für die Typklasse *Functor* (nur *fmap*).

```
sample1 :: BTree Int
```

```
sample1 = Node 5 (Leaf 1) (Node 8 (Node 4 (Leaf 6) (Leaf 9)) (Leaf 2))
```

```
sample2 :: BTree String
```

```
sample2 = Node "Hallo" (Leaf "Welt") (Leaf "!")
```

```
tests :: Test
```

```
tests = TestLabel "Klausur" (TestList [
```

```
  fmap (+1) sample1 ~?= Node 6 (Leaf 2) (Node 9 (Node 5 (Leaf 7) (Leaf 10)) (Leaf 3)),
```

```
  fmap (length) sample2 ~?= Node 5 (Leaf 4) (Leaf 1)
```

```
])
```

Reference solution

```
instance Functor BTree
```

```
  where
```

```
    fmap f (Leaf x) = Leaf (f x)
```

```
    fmap f (Node x y z) = Node (f x) (fmap f y) (fmap f z)
```

8 “Unparsing & Parsing”

Implementieren Sie die Funktion `unparseKuchen`, die ein Kuchen als Parameter nimmt und mithilfe der Bibliothek `HughesPJ` in folgende Form überführt:

- Zu Beginn steht das Schlüsselwort 'Rezept:'
- Die Zutaten werden untereinander aufgelistet und mit einem Semikolon (;) getrennt. (Nach der letzten Zutat folgt kein Semikolon!)
- Die Zubereitung wird nach den Zutaten aufgelistet.

```
data Kuchen = Rezept Zutaten Zubereitung
```

```
type Zutaten = [(String,Float)]
```

```
type Zubereitung = String
```

```
sample1 :: Kuchen
```

```
sample1 = Rezept
```

```
  [("Mehl",500.0),("Eier",4.0),("Zucker",1.5)]
```

```
  "Mische alles zusammen. Bei 180 Grad – Umluft – 60 Minuten backen!"
```

```
show (unparseKuchen sample1) liefert folgende Ausgabe:
```

```
Rezept:
```

```
Mehl 500.0;
```

```
Eier 4.0;
```

```
Zucker 1.5
```

```
Mische alles zusammen. Bei 180 Grad – Umluft – 60 Minuten backen!
```

```
unparseKuchen :: Kuchen -> Doc
```

Reference solution

```
unparseKuchen :: Kuchen -> Doc
```

```
unparseKuchen (Rezept x y) = text "Rezept:"
```

```
    $$ unparseZutaten x
```

```
    $$ unparseZubereitung y
```

```
unparseZutaten :: Zutaten -> Doc
```

```
unparseZutaten [] = text ""
```

```
unparseZutaten (t:[]) = unparseZutat t
```

```
unparseZutaten (t:ts) = unparseZutat t <> semi $$ (unparseZutaten ts)
```

```
unparseZutat :: (String, Float) -> Doc
```

```
unparseZutat (x,y) = text x <+> float y
```

```
unparseZubereitung :: Zubereitung -> Doc
```

```
unparseZubereitung x = text x
```

9 “Monaden”

Folgende Datenstruktur dokumentiert die bekannten Vorfahren einer Person;

```
data Person = Unknown | Known String Person Person
```

```
getName :: Person -> String
getName Unknown = "Unknown"
getName (Known x _) = x
```

Implementieren Sie die Funktion *evalM*, die eine Person als Parameter nimmt. Die Funktion soll mithilfe des Writer-Monaden zum Einen die Anzahl der bekannten Vorfahren (inkl. sich selbst) bestimmen und zum Anderen die einzelnen Verhältnisse (wie unten im Beispiel) loggen.

```
sample1 :: Person
sample1 = Known "Hans"
          (Known "Peter"
            Unknown
            (Known "Brigitte" Unknown Unknown))
          (Known "Ute"
            (Known "Wilhelm" Unknown Unknown)
            (Known "Sabrina" Unknown Unknown))
```

```
> evalM sample1
WriterT (Identity (6,["Person: Brigitte Vater: Unknown Mutter: Unknown",
"Person: Peter Vater: Unknown Mutter: Brigitte",
"Person: Wilhelm Vater: Unknown Mutter: Unknown",
"Person: Sabrina Vater: Unknown Mutter: Unknown",
"Person: Ute Vater: Wilhelm Mutter: Sabrina",
"Person: Hans Vater: Peter Mutter: Ute"])))
```

```
evalM :: Person -> Writer [String] Int
```

Reference solution

```
evalM (Unknown) = do
  return 0

evalM (Known x y z) = do
  p1 <- evalM y
  p2 <- evalM z
  tell ["Person: " ++ x ++ " Vater: " ++ (getName y) ++ " Mutter: " ++ (getName z)]
  return (1 + p1 + p2)
```