04IN1023: Grundlagen der funktionalen Programmierung

Klausur WiSe 2019/20

Universität Koblenz-Landau, FB4
Prof. Dr. Stefan Bosse, M.Sc. Marcel Heinz
25 October 2019

Name, Vorname	
Matrikel-Nr.	
Email	.....@uni-koblenz.de
Studiengang	☐ BSc Inf ☐ BSc CV ☐ .....
Prüfungsversuch	☐ 1 ☐ 2 ☐ 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	
10	

Exam Manual

1. Wenn Sie Fragen zu folgenden Punkten haben, können Sie diese während der Prüfung stellen. Fragen werden aber nur persönlich beantwortet.

2. Es gibt 10 Aufgaben mit jeweils 0-2 Punkten. 0 bedeutet fehlt oder falsch; 1 bedeutet wohl angemessen, aber wesentlich unvollständig oder teils falsch; 2 bedeutet angemessen und im Wesentlichen vollständig und korrekt.

3. Noten werden wie folgt berechnet: 0-8: 5; 9: 4; 10: 3,7; 11: 3.3; 12: 3; 13: 2,7; 14: 2,3; 15: 2; 16: 1.7;17: 1.3; 18-20: 1

4. Die Prüfung dauert 1 Stunde. So kann man mehr als 5 Minuten pro Aufgabe verbringen. Alle Aufgaben erfordern nur sehr wenige Codezeilen: 1-5 in der Referenzlösung. bermäig langer Code kann eine reduzierte Punktzahl erhalten. Wenn Text benötigt wird, gilt ein 140-Zeichen-Limit.

5. Die allgemeinen Themen für die Prüfung orientieren sich am Inhalt der Vorlesung und der bungen.

6. Man sollte - systematisch - darauf vorbereitet sein, dass sich der Text der Aufgaben auf die (Software-)Konzepte bezieht, die in der Vorlesung eingeführt wurden. Definitionen der Konzepte werden nicht abgefragt, aber ein grundlegendes Verständnis der Konzepte wird vorausgesetzt und ist entscheidend für das Bestehen der Prüfung.

7. Detailliertes Bibliothekswissen (wie zum Beispiel Kombinatoren aus den Bibliotheken zum Parsen oder formatiertes Drucken) wird nicht vorausgesetzt; Relevante Hinweise werden bereitgestellt, wenn Bibliotheken verwendet werden sollen. Vertrautheit mit Haskells Prelude wird jedoch angenommen -soweit es in der Vorlesung und den bungen behandelt wurde.

1 “Simple Algorithmen”

Implementieren sie die Funktion *split*, die einen String und einen Char einliest. Der String wird nun überall dort getrennt, wo der Char auftaucht. Der Char existiert damit nicht mehr in den Strings der zurückgegebenen Liste.

```
split :: String -> Char -> [String]

tests :: Test
tests = TestLabel "splitTests" (TestList [
        split "Hello World" ' ' ~?= ["Hello", "World"],
        split "Hello World " ' ' ~?= ["Hello", "World",""],
        -- multiple spaces:
        split "Hello  World " ' ' ~?= ["Hello","","World",""],
        split " Hello World" ' ' ~?= ["","Hello", "World"],
        split "Hello World" ' ' ~?= "Hello   World"
    ])
```

2 “Simple Datenmodelle”

Deklarieren sie einen vereinfachten Datentypen zur Verwaltung von einem sozialen Netzwerk. Dort soll eine Liste von registrierten Personen verwaltet werden. Jede Person hat einen öffentlich einsehbaren Namen, ein Alter und einen Benutzernamen. Des weiteren kann jede Person Beiträge schreiben, die von anderen Personen wiederum als Lesezeichen markiert werden können. Zu Beiträgen sind in der Datenbank nur der Benutzername des Autors und die Benutzernamen der Lesezeichen-Abonnenten erfasst.

3 “Unit testing”

Wie würden sie die Funktion *signum* testen, die für eine Zahl zurückgibt, ob sie positiv, negativ oder Null ist. Geben sie drei möglichst unterschiedliche Testfälle an.

$$data\ Sign = Pos \mid Neg \mid Zero\ deriving\ (Show, Eq)$$
$$signum\ ::\ Num\ a\ \Rightarrow\ a\ \rightarrow\ Sign$$

4 “Parameter Polymorphismus”

Leiten Sie für folgende Funktionen die Typsignatur und für Konstanten den Datentyp schrittweise von oben nach unten Zeile für Zeile ab. Zeigen Sie die einzelnen Schritte der Typableitung und die Typsignaturen der Funktionen und Ausdrücke (auch Teilausdrücke wie $f\ hd$ in Zeile 2). Die Datentypen dürfen vereinfacht dargestellt werden (d.h., ‘Num‘, ‘Ord a‘, ‘Bool‘, ‘[a]‘, für polymorphe/nicht bestimmte Typen ‘a,b,c,..‘).

seqcheck [] f = []
seqcheck (hd:tl) f = if f hd then hd : (seqcheck tl f) else (seqcheck tl f)
seqcheckPositive l = seqcheck l (\x −> x > 0)
seqcheckTrue l = seqcheck l (\x −> x)
l1 = seqcheckPositive [1,2,3,(−1)]
l2 = seqcheckTrue [True,False,True]

Hier einige Hilfen:

> :t (>)
(>) :: Ord a => a −> a −> Bool

> :t 1
1 :: Num a => a

> :t [1,2,3]
[1,2,3] :: Num t => [t]

Beginnen sie also mit:

```
Zeile 1:  seqcheck [] f = []
-- Auf Basis von ([]::?), (f::?), siehe Funktionskoerper ([]::?)
-- Bestimmen sie Typen nur anhand der bisherigen Zeilen.
```

5 “Funktionen höherer Ordnung”

Implementieren sie die Funktion *times*, die als Parameter eine Anzahl n, eine Funktion und einen Wert nimmt. Mit *times* soll nun die Funktion n-Mal mit dem Wert ausgeführt werden.

```
times :: Int -> (a -> a) -> a

tests :: Test
tests = TestLabel "timesTests" (TestList [
        times 5 (+1) 0 ~?= 5,
        times 21 not True ~?= False,
        times 3 (x -> x*x) 2 ~?= 256,
        times 0 (**2) 2 ~?= 2
    ])
```

6 “Functors & Foldables”

Betrachten sie die folgende Datenstruktur für 2D-Objekte und die Funktion *changeBy*, mit der die Objekte angepasst werden können. Die Funktion erhält dazu eine Funktion f als Parameter und wendet sie auf die Objekte an.

```
type Point = (Int, Int)
data Shape a = Circle Point a -- Koordinate + Radius
             | Quadrangle Point a a -- Koordinate + Breite + Hhe
             | Seq (Shape a) (Shape a) -- Shape Sequenz
    deriving (Show, Eq)

changeBy :: Num a => (a -> a) -> Shape a -> Shape a
changeBy f shape = fmap f shape

tests :: Test
tests = TestLabel "timesTests" (TestList [
        changeBy (+1) (Circle (1,2) 3) ~?= Circle (1,2) 4,
        changeBy (*2) (Quadrangle (0,0) 4.2 2.2) ~?= Quadrangle (0,0) 8.4 4.4,
        changeBy (**2) (Circle (1,2) 5) ~?= Circle (1,2) 25,
        changeBy (+1) (Seq (Circle (1,2) 3) (Quadrangle (4,4) 2 2))
          ~?= (Seq (Circle (1,2) 4) (Quadrangle (4,4) 3 3))
    ])
```

Implementieren sie die entsprechende Instanz für die Typklasse `Functor` (nur `fmap`).

7 “Funktionale Datenstrukturen”

Betrachten sie die folgende Datenstruktur für Bäume und die gegebene Funktion zur Suche eines Elementes. Diese Funktion macht Gebrauch von der Rekursivität der Datenstruktur.

```
data Tree a = Node a | Fork (Tree a) (Tree a) deriving (Show,Eq)

elem :: Ord a => a -> Tree a -> Bool
elem x (Node y) = x == y
elem x (Fork l r) = (elem x l) || (elem y r)
-- sobald die linke Seite von || zu True ausgewertet wird,
-- wird die rechte Seite nicht mehr evaluiert.
```

Analysieren sie den Aufruf mit *'elem 4 sampleTree'*. Unterstreichen sie in dem Beispielbaum (sampleTree) unten, welche Verzweigungen (Fork) und Knoten (Node) tatsächlich betrachtet werden. Unterstreichen sie entsprechend nur die Ausdrücke, die mittels == verglichen werden.

```
sampleTree = Fork (Node 3)
                  (Fork (Fork (Node 1)
                              (Node 4))
                        (Node 20))
```

8 “Sortieren”

Die gegebene Funktion *sort* sortiert eine Liste aufsteigend mittels Merge-Sort. Der Code lässt sich allerdings nicht ausführen, ohne dass ein Fehler ausgegeben wird. Markieren sie den Fehler im vorhandenen Code und verbessern sie die entsprechende Stelle.

```
sort :: Ord e => [e] -> [e]
sort [] = []
sort [x] = [x]
sort xs = merge (sort $ fst splits, sort $ snd splits)
  where
    splits = (take len xs, drop len xs)
    len = length xs `div` 2

merge :: Ord e => ([e], [e]) -> [e]
merge (xs, []) = xs
merge ([], ys) = ys
merge ((x:xs),(y:ys)) = if x <= y
                        then x : merge xs (y:ys)
                        else y : merge (x:xs) ys
```

9 “Anonyme Funktionen”

Die Funktion numberlists nimmt eine Liste von Zahlen-Paaren und gibt die Liste bestehend aus sortierten Zahlen-Paaren zurück.

```
numberlists ls = map numbers ls
    where
        numbers (x, y) | x < y = (x,y)
                       | x == y = (x,y)
                       | x > y = (y,x)
```

Ersetzen sie die Verwendung der Hilfsfunktion (number) durch einen Lambda-Ausdruck.

10 “Komplexität”

Betrachten sie die folgende Funktion *substring*, die bestimmt, ob der erste gegebene String ein Teil des zweiten Strings ist.

```
substring :: String -> String -> Bool
substring [] _ = True
substring _ [] = False
substring (x:xs) (y:ys) =
    if x==y
     then ((take (len xs) ys) == xs) || substring (x:xs) ys
     else substring (x:xs) ys
```

Geben sie für die folgenden beispielhaften Aufrufe an, welche Rekursionstiefe von *substring* erreicht wird.

Aufruf	Rekursionstiefe
substring ”a” ”Banane”	
substring ”an” ”BananeB”	
substring ”worl” ”Hello world”	
substring ”Hello” ”Hel lo Hello world”	
substring ”Rec” ”Red Banana Recipes”	

Verallgemeinern sie nun für eine Abschätzung des Aufwands. Geben sie eine Formel zur Berechnung der Aufrufe von *substring* an in Abhängigkeit zu der Länge des ersten Strings $LEN1$ und der Länge des zweiten Strings $LEN2$. Geben sie dazu an, wie oft *substring* im besten Fall und im schlechtesten Fall aufgerufen wird.