

04IN1023: Grundlagen der funktionalen Programmierung

Klausur SoSe 2019

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

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	
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. bermäßig 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 *trim*, die einen String einliest und die nicht notwendigen umschließenden Leerzeichen entfernt.

```
trim :: String -> String
```

```
tests :: Test
```

```
tests = TestLabel "trimTests" (TestList [  
    trim "Hello World" ~?= "Hello World",  
    trim "Hello World " ~?= "Hello World",  
    trim "Hello World  " ~?= "Hello World",  
    trim " Hello World" ~?= "Hello World",  
    trim "  Hello World" ~?= "Hello World",  
    trim "   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 | Neg | Zero deriving (Show, Eq)
signum :: Num a => a -> Sign
```

4 “Parameter Polymorphism”

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 *addStrings*, die als Parameter einen String nimmt. In dem String sind Integer Zahlen enthalten, die eingelesen und addiert werden sollen. Es ist nicht erlaubt einen lokalen Scope zu benutzen (keine Hilfsfunktionen und -variablen).

```
addStrings :: String -> Int

-- Hints:
> words "123 345"
["123", "345"]
> words "123.2 345.67 654.65"
["123.2", "345.67", "654.65"]

> read "123" :: Int
123
> read "32.2" :: Float
32.2

tests :: Test
tests = TestLabel "addStringTests" (TestList [
    addStrings "1 2 3" ~?= 6,
    addStrings "42" ~?= 42,
    addStrings "" ~?= 0,
    addStrings "2 3 4 5 5 6" ~?= 25
])
```

6 “Functors & Foldables”

Betrachten sie die folgende Datenstruktur für Bäume.

```
data LLTree a = Leaf a | Fork (LLTree a) (LLTree a)
  deriving (Show, Eq)
```

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

```
sample0 = Leaf 3
sample1 = (Fork (Leaf 0) (Leaf 1))
sample2 = (Fork (Fork (Leaf 'a') (Leaf 'b')) (Leaf 'c'))

-- | A few tests .
tests :: Test
tests = TestLabel "LLTree" (TestList [
  fmap (+1) sample0 ~?= Leaf 4,
  fmap (+1) sample1 ~?= Fork (Leaf 1) (Leaf 2),
  fmap toUpper sample2 ~?= (Fork (Fork (Leaf 'A') (Leaf 'B')) (Leaf 'C')),
  foldr (+) 0 sample0 ~?= 3,
  foldr (+) 6 sample1 ~?= 7,
  foldr (:) "." sample2 ~?= "abc."
])
```


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 = Leaf a | Node a (Tree a) (Tree a)* **deriving** (*Show,Eq*)

```
elem :: Ord a => a -> Tree a -> Bool
elem x (Leaf y) = x == y
elem x (Node l r) | x == y = True
                  | x < y = elem x l
                  | x > y = elem x r
```

Analysieren sie den Aufruf mit `'elem 4 sampleTree'`. Unterstreichen sie in dem Beispielbaum (`sampleTree`) unten, welche der numerischen Werte in Verzweigungen (Node) und Blättern (Leaf) tatsächlich betrachtet werden. Unterstreichen sie entsprechend nur die Werte, die mittels `==`, `<` oder `>` verglichen werden.

sampleTree = Node 6 (Node 3 (Leaf 1) (Leaf 4)) (Node 8 (Leaf 7) (Leaf 9))

8 “Sortieren”

Die gegebene Funktion *sort* sortiert eine Liste aufsteigend mittels Selection-Sort. Die Resultate scheinen jedoch nicht richtig zu sein. Markieren sie den Fehler im vorhandenen Code und verbessern sie die entsprechende Stelle.

```
-- Polymorphic sorting
sort :: Ord a => [a] -> [a]
sort xs = selects xs

-- Repeat selection of smallest element
selects :: Ord a => [a] -> [a]
selects [] = []
selects (x:xs) = selects xs' ++ [x]
  where
    z = largest (head xs) (tail xs)
    xs' = remove z xs

-- Find smallest element
largest :: Ord a => a -> [a] -> a
largest x [] = x
largest x (y:ys) = largest (max x y) ys

-- Remove a given element
remove :: Eq a => a -> [a] -> [a]
remove _ [] = error "Element not found for removal."
remove x (y:ys) =
  if x==y
  then ys
  else y : remove x ys
```

9 “Anonyme Funktionen”

Die Funktion `numberlists` nimmt eine Liste von Zahlen-Paaren und gibt eine Liste zurück, die aus Listen besteht mit allen Zahlen zwischen den Zahlen im jeweiligen Paar.

```
numberlists ls = map numbers ls
  where
    numbers (x, y) | x < y = [x..y]
                  | x == y = [x]
                  | 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 "B" "BananeB"	
substring "Hell" "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.