

04IN1023: Grundlagen der funktionalen Programmierung

Klausur SoSe 2021

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel,
M.Sc., M.Ed. Hakan Aksu
30 July 2021

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
Prüfungsordnung	<input type="checkbox"/> \geq PO2019 <input type="checkbox"/> $<$ 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 *replaceliterals*, die einen *String* und zwei *Char* einliest. Im Text soll überall nach dem zuerst angegebenen Zeichen gesucht und jeweils durch das zweite Zeichen ersetzt werden.

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
    replaceliterals "Programmierung" 'm' 's' ~?= "Prograssierung",
    replaceliterals "Grundlagen der funktionalen Programmierung" 'n' 'x'
    ~?= "Grundlagex der funktioalex Programmierung",
    replaceliterals "Ich bin ein String" 'x' 'a' ~?= "Ich bin ein String",
])

replaceliterals :: String -> Char -> Char -> String
```

2 “Suchen und Sortieren”

Implementieren Sie die Funktion *searchLinear*, die zwei *String*-Arrays einliest und einen *Boolean* zurückliefert. Wenn mindestens ein String aus dem einen Array mit irgendeinem String aus dem zweiten Array übereinstimmt, dann wird *True* geliefert, ansonsten *False*.

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
    searchLinear ["Grundlagen", "Programmierung"] ["Software"] ~?= False,
    searchLinear ["Grundlagen", "Programmierung"] ["Software", "Programmierung"] ~?= True,
    searchLinear ["A", "B", "C"] ["G", "H", "Z"] ~?= False,
    searchLinear ["A", "B", "C"] ["B", "H", "Z"] ~?= True
])

searchLinear :: [String] -> [String] -> Bool
```

3 “Simple Datenmodelle”

Deklarieren Sie einen vereinfachten Datentypen zur Verwaltung von Mietobjekten eines Urlaubsunternehmens. Ein *Mietobjekt* ist entweder ein Hotelzimmer oder ein Ferienhaus. Das Hotelzimmer enthält die Bettenanzahl als Integer-Wert und die Angabe, ob es eine Suit ist, als Boolean-Wert. Das Ferienhaus enthält die Zimmeranzahl als Integer-Wert und eine Liste der Ausstattung. Jede einzelne Ausstattung wird mit einem String-Wert beschrieben. Verwenden Sie bei den Mietobjekten geeignete Begriffe mit *type-Synonymen*, um die Lesbarkeit zu verbessern.

4 “Funktionale Datenstrukturen”

Gegeben ist folgender individueller Stack-ADT (LIFO - Last In First Out) für Integer-Werte:

```
data MyStack = Empty | Push Int MyStack
```

```
push :: [Int] -> MyStack -> MyStack
```

```
pop :: MyStack -> (MyStack, (Int,Int))
```

Implementieren Sie die Funktionen *push* und *pop*. Die Funktion *push* erhält eine Liste von Integer-Werten und fügt von vorne nach hinten jeden einzelnen Wert nach und nach in den Stack hinzu. Die Funktion *pop* nimmt die nächsten beiden Werte heraus und gibt diese als Paar zurück. Wenn nur ein Wert oder kein Wert enthalten ist, wird das Paar mit dem Wert 0 entsprechend aufgefüllt.

Beispiele:

```
> push [1,2,3] (Push 9 $ Push 8 $ Empty)
Push 3 $ Push 2 $ Push 1 $ Push 9 $ Push 8 $ Empty
```

```
> push [] (Push 9 $ Push 8 $ Empty)
Push 9 $ Push 8 $ Empty
```

```
> pop Push 3 $ Push 2 $ Push 1 $ Push 9 $ Push 8 $ Empty
(Push 1 $ Push 9 $ Push 8 $ Empty, (3,2))
```

```
> pop Push 8 $ Empty
(Empty, (8,0))
```

```
> pop Empty
(Empty, (0,0))
```

5 “Funktionen höherer Ordnung”

Implementieren Sie die Funktion *checkMaxLength*, die als Parameter ein Array von 2er-Paaren erhält und ein Boolean-Wert zurückgibt. Das Paar besteht aus einem Array beliebigen Typs und einem Integer-Wert.

Die Funktion überprüft, ob in jedem Paar die Länge des Arrays kleiner gleich als der angegebene Integer-Wert ist. Wenn bei mindestens einem Paar die Bedingung nicht erfüllt sein sollte, so wird insgesamt False zurückgegeben, ansonsten True. 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 [
  checkMaxLength [] ~?= True,
  checkMaxLength [("A",1),("Hallo",5),("B",2)] ~?= True,
  checkMaxLength [(1,23,4),1],([8],1)] ~?= False
])
```

```
checkMaxLength :: [(a, Int)] -> Bool
```

6 “Typ-Klassen Polymorphismus”

Implementieren Sie zwei Instanzen der Klasse *Employee* für die Datentypen *Manager* und *Assistant*.

```
data Manager = Manager Float
data Assistant = Assistant Float
```

```
class Employee a
  where
    bonus :: a -> Float
```

Folgend sind die Formeln für die Bonuszahlungen für den Manager und den Assistenten:

- Managerbonus: $0,3 * grundgehalt$
- Assistentenbonus: $0,15 * grundgehalt$

Anwendungsbeispiel:

```
sample1 = Manager 3600
sample2 = Assistant 2400
```

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
  bonus sample1 ~?= 1080.0,
  bonus sample2 ~?= 360.0 ])
```

7 “Functors & Friends”

Betrachten Sie die folgende Datenstruktur für eine Queue.

```
data DQueue a = Empty | Enqueue a (DQueue a) deriving (Eq, Show, Read)
```

Implementieren Sie die entsprechende Instanz für die Typklasse *Functor* und *Foldable* (*fmap* und *foldr*).

```
tests :: Test
tests = TestLabel "Klausur" (TestList [
  fmap (+1) (Enqueue 1 $ Enqueue 2 $ Enqueue 3 Empty)
    ~?= (Enqueue 2 $ Enqueue 3 $ Enqueue 4 Empty),
  foldr (++) "" (Enqueue "Haskell" $ Enqueue "programming" $ Enqueue "word" Empty)
    ~?= "Haskellprogrammingword",
  foldr (-) 0 (Enqueue 9 $ Enqueue 37 $ Enqueue 23 Empty)
    ~?= -5])
-- Berechnung des letzten Tests: 9 - (37 - (23 - 0)) = -5
```


8 “Unparsing & Parsing”

Implementieren Sie die Funktion `unparseTodoList`, die eine `TodoList` als Parameter nimmt und mithilfe der Bibliothek `HughesPJ` in folgende Form überführt:

- Zu Beginn steht das Schlüsselwort 'Todos:'
- Die einzelnen Einträge werden untereinander aufgelistet und mit einem Semikolon (;) getrennt. (Nach dem letzten Eintrag folgt kein Semikolon!)
- Jeder einzelne Eintrag hat die Form 'Priorität Text'

```
data TodoList = Todos Eintraege
type Eintraege = [(Int,String)]
```

```
sample :: TodoList
sample = Todos [(1,"Fp21 lernen"),(3,"EMail verschicken"),(2,"Staubsaugen")]
```

`show (unparseTodoList sample)` liefert folgende Ausgabe:

```
Todos:
1 Fp21 lernen;
3 EMail verschicken;
2 Staubsaugen
```

```
unparseTodoList :: TodoList -> Doc
```

9 “Monaden”

Folgende Datenstruktur dokumentiert die Flächen einzelner Räume in einer Wohnung;

```
type Wohnung = [Zimmer]
type Zimmer = (String, Flaeche)
type Flaeche = Float
```

```
getBezeichnung :: Zimmer -> String
getBezeichnung (x,-) = x
```

```
getFlaeche :: Zimmer -> Float
getFlaeche (_,x) = x
```

Implementieren Sie die Funktion *evalM*, die eine Wohnung als Parameter nimmt. Die Funktion soll mithilfe des Writer-Monaden zum Einen die Gesamtfläche der Wohnung bestimmen und zum Anderen die einzelnen Zimmer mit ihren Flächen loggen.

```
sample :: Wohnung
sample = [("Wohnzimmer",25.5),("Schlafzimmer", 16.2),("Kueche", 21.4)]
```

```
Prelude> evalM sample
WriterT (Identity (63.1,["Zimmer: Kueche Flaeche: 21.4",
"Zimmer: Schlafzimmer Flaeche: 16.2",
"Zimmer: Wohnzimmer Flaeche: 25.5"]))
```

```
evalM :: Wohnung -> Writer [String] Float
```