04IN1023: Introduction to functional programming

Examination SS 2018

PD Stefan Bosse, Marcel Heinz
Universität Koblenz-Landau, Fakultät Informatik
sbosse@uni-koblenz.de

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. Übermäßig 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 Übungen.
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 Übungen behandelt wurde.

1. “Simple algorithms”

Implementieren Sie die Funktion `substring`, die zwei Strings erwartet und überprüft, ob der erste String Teil des zweiten ist (z.B. “sub” ist in “mysubstring”). Einige Testfälle sind unten aufgeführt.

```
-- hints
> a = "test"
> (y:ys) = a
> y
't'
> ys
"est"

substring :: String -> String -> Bool
tests :: Test
tests = TestLabel "SubstringTests" (TestList [
  substring "" "Hello world!" ~?= True,
  substring "Hello" "" ~?= False,
  substring "42" "Truth" ~?= False,
  substring "42" "Tr42uth" ~?= True,
  substring "42" "4Truth2" ~?= False,
  substring "42" "42" ~?= True
])
```

Reference solution

```
substring :: String -> String -> Bool
substring [] _ = True
substring _ [] = False
substring xs ys = xs == take (length xs) ys
          || substring xs (tail ys)
```

2. “Simple data models”

Deklarieren Sie einen Datentyp für Züge. Ein einzelner Zug hat einen Start- und Zielnamen und besteht aus Waggons. Jeder Wagen hat eine Nummer und nummerierte Sitze, die optional für einen Namen (einer Person) reserviert werden können.

Reference solution

```
type Way = (String, String)
type Seat = (Int, Maybe String)
data Wagon = Wagon Int [Seat]
data Train = Train Way [Wagon]
```

3. "Parametric polymorphism"

Definieren Sie eine polymorphe Funktion *isSorted*, die nach einer Liste benutzerdefinierter Wertepaare sucht (siehe unten), ob sie in aufsteigender Reihenfolge sortiert sind (Rückgabewert ist `True` oder `False`). Ein Paar (x,y) wird mit anderen Paaren verglichen, indem das erste und das zweite Element getrennt betrachtet werden. Das erste Element wird niemals mit dem zweiten eines anderen Paares verglichen. Ein Element eines Paares kann beispielsweise eine Zahl oder eine Zeichenfolge sein.

```
data Pair a = Pair a a deriving Show
issorted :: Ord a => [Pair a] -> Bool

> issorted [Pair 2 1, Pair 4 2]
True
> issorted [Pair "4" "2", Pair "4" "2", Pair "4" "2"]
True
> issorted [Pair 4.2 2.1, Pair 2.1 1.0]
False
> issorted []
True
> issorted [Pair 42 21]
True
> issorted [Pair 1 20, Pair 2 30, Pair 40 20]
False
```

Reference solution

```
issorted :: Ord a => [Pair a] -> Bool
issorted ((Pair x1 y1):(Pair x2 y2):xs) =
  if (x1<=x2) && (y1<=y2)
  then issorted ((Pair x2 y2):xs)
  else False
issorted _ = True
```

4. "Pure functional data structures"

Definieren Sie die konstante Liste *evens* einschließlich ihrer Typ-Signatur, die alle geraden natürlichen Zahlen (> 0) in geeigneter Weise enthält, damit sie noch mit *take* oder *head* verarbeitet werden kann.

Reference solution

```
> :t evens
evens :: [Integer]

evens = evensConstruct 1
  where
   evensConstruct x =
    if even x then x:(evensConstruct (x+1))
    else evensConstruct (x+1)
```

5. "Trees, maps, and Functors"

Teil 1: Implementieren Sie eine Mapping-Funktion *treeMap*, die die Werte der Blätter eines binären Baums (Definition siehe unten) mit einer benutzerdefinierten Funktion f abbildet (transformiert). Der Binärbaum besteht aus Zweigen (Knoten) und Blättern, die Datenwerte tragen.

Teil 2: Implementiere nun diese Mapping-Funktion als Instanz der Typklasse *Functor* mit dem einzigen Element *fmap*.

```
-- Binary tree type
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving (Show)
-- Functor class
class Functor t where
    fmap :: (a -> b) -> t a -> t b

-- hints
instance Functor [] where
    fmap = map
> :t treeMap
treeMap :: (a -> b) -> Tree a -> Tree b
> treeMap (\x -> 2*x) (Branch (Branch (Leaf 1) (Leaf 2)) (Branch (Leaf 3) (Leaf 4)))
Branch (Branch (Leaf 2) (Leaf 4)) (Branch (Leaf 6) (Leaf 8))
> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
> fmap (\x -> 2*x) (Branch (Branch (Leaf 1) (Leaf 2)) (Branch (Leaf 3) (Leaf 4)))
Branch (Branch (Leaf 2) (Leaf 4)) (Branch (Leaf 6) (Leaf 8))
```

Reference solution

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving (Show)

treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf x)            = Leaf (f x)
treeMap f (Branch left right) = Branch (treeMap f left) (treeMap f right)

instance Functor Tree where
    fmap f (Leaf x)            = Leaf   (f x)
    fmap f (Branch left right) = Branch (fmap f left) (fmap f right)
```

6. "Higher-order functions"

Implementieren Sie die Funktion *crosscorrelate*, die die diskrete Approximation der Kreuzkorrelation zweier gegebener (reellwertiger) Funktionen f und g berechnet, gegeben durch:

$$crosscorrelate(f,g)(n) = (f * g)(n) = \sum_{m=a}^{b} f(m) + g(m+n)$$

Das bedeutet *crosscorrelate* konstruiert und gibt eine Funktion zurück, die die diskrete Kreuzkorrelation bei einem gegebenen Punkt n zweier spezifischer Funktionen berechnet. Die konstanten Parameter a und b sind fest und können beispielsweise mit a = -1000, b = 1000 gewählt werden.

Implementieren Sie die Summe $f(m) + g(m+n)$, indem Sie die Funktionen *sum* und *map* verwenden, die auf eine Liste von Zahlen im Bereich [a..b] angewendet werden.

```
-- hints
> a = (-1000)
> b = 1000
> :t sum
sum :: Num a => [a] -> a
> :t map
map :: (a -> b) -> [a] -> [b]
> [1..5]
[1,2,3,4,5]
> :t crosscorrelate
crosscorrelate
  :: (Num a1, Num a, Enum a1) => (a1 -> a) -> (a1 -> a) -> a1 -> a
```

Reference solution

```
crosscorrelate f g = \n -> sum (map (\m -> (f m)*(g (n+m))) [a..b])
```

7. "Sorting"

Im Folgenden nimmt die Funktion *sort* eine Liste vergleichbarer Werte auf und soll den Quick-sort-Algorithmus implementieren. Die Ergebnisse scheinen nicht korrekt zu sein. Markieren Sie die Fehler in der Implementierung und reparieren Sie die Funktion so, dass sie fehlerfrei funktioniert (d.h. schreiben Sie den korrekten Code).

```
-- Polymorphic sorting
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) =
  (sort [l|l<-left, l<x])
  ++ [x]
  ++ (sort [r|r<-right, r>=x])
where
  left = map (\l -> l < x) xs
  right = map (\r -> r >= x) xs
```

Reference solution

```
-- Polymorphic sorting
sort :: Ord a => [a] -> [a]
sort [] = []
sort (pivot:rest) =
  (sort left)
  ++ [pivot]
  ++ (sort right)
where
  left = filter (< pivot) rest
  right = filter (>= pivot) rest
```

8. "Anonymous Functions"

Die Funktion *doublef* erwartet eine Funktion *f* und eine Liste *xs* und wendet die Funktion zweimal auf jedes Element an, indem sie die Hilfsfunktion *twice* verwendet. Konvertieren Sie die benannte Hilfsfunktion in einen Lambda-Ausdruck, sodass keine explizite Hilfsfunktion benötigt wird.

```
doublef f xs = map twice xs
  where
    twice x = f (f x)
```

Reference solution

```
doublef f xs = map (\x -> f (f x)) xs
```

9. "Computational Complexity"

Frage 1: Analysieren Sie den folgenden binären Suchalgorithmus *search* angewandt auf eine sortierte Liste von Zahlen und geben Sie eine Formel $f(n)$ an, die die Anzahl der Einheitsoperationen in Bezug auf die Anzahl n der Elemente der Liste für das **Worst-Case-Szenario** angibt, d.h. der zu suchende Schlüssel ist das erste Element der Liste), und schließlich die Komplexitätsklasse $\Theta(f(n))$.

Frage 2: Geben Sie zusätzlich zwei Funktionen $g(n)$ und $h(n)$ an, die eine untere bzw. obere Schranke von $f(n)$ darstellen, d.h. für jedes $n > n_0$ ist die Funktion g niedriger und h größer als f.

Frage 3: Was ist die beste Laufzeit und die Position des zu durchsuchenden Schlüsselelements?

Hinweis: Die Lösungsfunktion f und $\Theta(f)$ werden durch einfache Ausdrücke beschrieben, und ebenso g und h. Sie können die folgenden Beispiele ausprobieren, um einen Ansatz zu erhalten: `[1,2,3,4,5,6,7,8]` mit $n = 8$ und $key = x = 1$, `[1 , 2,3,4,5,6,7,8, 9,10,11,12,13,14,15,16]` mit $n = 16$, $key = x = 1$.

Vereinfachung: *Jeder Funktionsaufruf ist eine Einheitsoperation.*

```
-- hint
> :t take
take :: Int -> [a] -> [a]
> take 2 [1,2,3,4]
[1,2]
> :t drop
drop :: Int -> [a] -> [a]
> drop 2 [1,2,3,4]
[3,4]
-- Binary Search
search :: Ord a => [a] -> a -> Bool
search [] _ = False
search xs x =
   if x < y then search ys1 x
   else if x > y then search ys2 x
   else True
  where
    ys1 = take l xs
    (y:ys2) = drop l xs
    l = length xs `div` 2
```

Reference solution

$f(n)_{\text{worstcase}} = \log(n)+1$, $\Theta_{\text{worstcase}}(\log\ n)$

$g(n) = 1$, n > 2

$h(n) = n$, n > 0

$\Theta_{\text{bestcase}}(1)$

10. “Unit testing”

Wie würden Sie die binäre Suchfunktion *search* aus der vorherigen Aufgabe testen, die eine Liste von ganzen Zahlen und einen ganzzahligen Schlüssel *key* als Argumente erhält und zurück gibt, ob *key* Teil der Liste ist?

Geben Sie drei grundlegende *eindeutige Testfälle* an (Unterscheidung in Sonderfällen der Eingabeliste und in Bezug auf die Beziehung des Inhalts zum Schlüssel).

Sie können diese informell definieren (mit drei symbolischen Variablen, die die Ergebnisse der Tests darstellen) oder Sie können den Assertion-Operator verwenden.

```
-- hints
:t search
search :: Ord a => [a] -> a -> Bool
```

Reference solution

```
import Test.HUnit -- Assertion

empty = False ~=? search 42 []
exists = True ~=? search 42 [21, 21, 42]
notexists = False ~=? search 42 [21, 63, 105]

-- Or informally
empty = False = search 42 []
exists = True = search 42 [21, 21, 42]
notexists = False ? search 42 [21, 63, 105]
```