04IN1023: Introduction to functional programming

Final—Final SS 2017

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel, M.Sc. Marcel Heinz
21 July 2017

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. If you have any questions regarding the following items, please ask them in the lab or in the lecture. You can ask them during the final or the re-sit as well, but this may be less helpful to you.

2. There are 10 assignments with 0-2 points each. 0 means 'missing' or 'wrong'; 1 means 'arguably appropriate, but significantly incomplete or incorrect'; 2 means 'appropriate and essentially complete and correct'.

3. Grades are computed as follows: 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. The exam lasts 1 hours. Thus, one can spend more than 5 min per assignment. All assignments only require very few lines of code: 1–5 in the reference solution. Overly long code may receive a reduced score. If text is required, a 140 chars limit applies.

5. The overall topics for the exam are defined with the dry-run; see the section headers. These topics are maintained for the actual final and the re-sit of the given course edition. The topics may be somewhat different in the next edition.

6. One should be prepared—systematically—that the text of the assignments relates to the (software) *concepts* that are listed for each lecture. Definitions of the concepts are never inquired, but basic understanding of the concepts is assumed and crucial for passing the exam.

7. One is advised to establish familiarity with the *illustrations* given for all concepts, as available on the wiki. These illustrations are often invoked, perhaps after some modulation, in the exam assignments.

8. Detailed library knowledge (such as combinators of libraries for parsing or pretty printing) is never assumed; relevant hints would be provided, if libraries are to be used. Familiarity with Haskell's *Prelude*, though, is assumed—to the extent it is covered in the lecture.

9. The dry run for the exam also contains some 'metaremarks' to clarify the scope assumed for the exam topics. This helps understanding how much the question in the final or resit may differ from the dry run.

1 "Simple algorithms"

Define a function that given a list of integers counts the number of values for which duplicates exist.

$duplicates[1,2,3,1] \Rightarrow 1$
$duplicates[1,2,3] \Rightarrow 0$
$duplicates[1,2,3,1,2,1] \Rightarrow 2$

Reference solution

```
duplicate xs = length $ duplicate' xs []
  where
    duplicate' (x:xs) ys = if elem x xs
      then if elem x ys
          then duplicate' xs ys
          else duplicate' xs (x:ys)
      else duplicate' xs ys
    duplicate' _ ys = ys
```

2 “Simple data models”

Declare a data type for leaf-labeled rose trees as follows. Each node is either a fork with an arbitrary number of sub-trees or a leaf labelled by a string value.

Reference solution

```
data Tree = Leaf Int | Fork [Tree]
```

3 "Unit testing"

How would you test the *concat* function that takes two strings and concatenates them such that the first one is appended to the second. Provide a number of test cases. Use the HUnit style of describing test cases.

Reference solution

```
import Test.HUnit -- Not required

leftempty = " world" ~=? concat "" " world"
rightempty = "hello" ~=? concat "hello" ""
normal = "hello world" ~=? concat "hello" " world"

-- Not required
main = do
  testresults <- runTestTT $ TestList [leftempty,rightempty,normal]
  print testresults
```

4 "Parametric polymorphism"

Define a polymorphic function including its function signature for 'toDistinct'. The function takes a list of values and turns it into a set such that there are no duplicates. Don't forget to provide the correct type signature.

$toDistinct[1,2,3,1] \Rightarrow [1,2,3]$
$toDistinct[1,2,3] \Rightarrow [1,2,3]$
$toDistinct[1,2,1,2,1] \Rightarrow [1,2]$
$toDistinct[] \Rightarrow []$

Reference solution

```
toDistinct :: Eq a => [a] -> [a]
toDistinct [] = []
toDistinct (x:xs) = x:(toDistinct [z|z<-xs,z/=x])
```

5 "Higher-order functions"

Define a function *isSquare*. The function takes an integer and returns a boolean that states whether the integer is a square number or not. Use higher order functions such as map, foldr, filter or list comprehension. Don't forget to provide the correct type signature.

$isSquare\ 4 \Rightarrow True -- 2 * 2$
$isSquare\ 8 \Rightarrow False$
$isSquare\ 3 \Rightarrow False$
$isSquare\ 225 \Rightarrow True -- 15 * 15$

Reference solution

```
isSquare :: Integral n => n -> Bool
isSquare n = (==1) $ length $ [x| x<-[0..n], x*x==n]
```

6 “Monoids”

Define a monoid for String concatenation (‘concat’), where String concatenation works in the way that is described in the unit testing task.

Reference solution

```
data Concat = Concat String
instance Monoid Concat where
  mempty = Concat ""
  mappend (Concat x) (Concat y) = Concat (x++y)
```

7 “Functors”

Consider lists with an even number of elements as follows.

```
data ListEven a = Empty | TwoMore a a (ListEven a)
```

Implement an instance of the type class `Foldable` with the member `foldr`. Hint: Think about folding to the right...

Reference solution

```
instance Foldable ListEven
  where
    foldr f z Empty = z
    foldr f z (TwoMore x y l) = f x (f y (foldr f z l))
```

8 “Unparsing”

What are the major combinators for unparsing strings in the sense of unparsing with package Text.HughesPJ? You do not need to get the names of the combinators right, but you are asked to list 3+ combinators with a short comment that explains the combinator’s parameters, if any.

Reference solution

text a — print a string a.

<> — concatenate two strings

< + > — concatenate two strings with a space in between

indent n — indentation by n spaces

...

9 "Parsing"

Consider these expression forms:

Compute 3 + 4
Compute 5 − 6
Compute 5 /= 6
Compute 4 *power* 6

Write a parser for the expression forms using combinators such as string, many1, digit, space,char, noneOf. You can transform strings to integers using (*read x* :: *Int*).

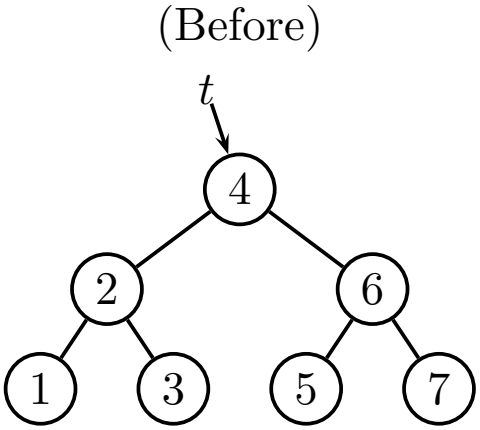
Reference solution

```
import Text.Parsec
data Compute = Compute Int String Int

parseArithm :: Parser Compute
parseArithm = string "Compute "
      >> many1 digit
      >>= \l -> space
      >> many1 (noneOf " ")
      >>= \o -> space
      >> many1 P.digit
      >>= \r -> return (Compute (read l :: Int) o (read r :: Int))
```

10 "Functional data structures"

Consider the following binary search tree t.



Now, assume that 8 is inserted into t, resulting in a tree t'. Draw the tree t' and mark the nodes that are copied from t (in the sense of path copying).

Reference solution