04IN1023: Introduction to functional programming

Final—Final SS 2016

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel, M.Sc. Marcel Heinz
21 July 2016

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 two Strings, returns the longest common prefix.

```
> wordf "Helloworld!" "Trolloworl!"
""
> wordf "Hello" "Hela"
"Hel"
> wordf "21" "2142"
"21"
> wordf "" "Hello"
""
> wordf "Helloworld!" "Hello"
"Hello"
```

Reference solution

```
wordf :: String -> String -> String
wordf [] _ = []
wordf _ [] = []
wordf (x:xs) (y:ys) = if (x==y) then [x]++ (wordf xs ys) else []
```

2 “Simple data models”

Declare a data type for edge-labeled graphs as follows. An edge-labeled graph consists of a list of nodes and a list of edges. The nodes have a name and edges are defined as tuples with an integer value and the pair of the connected nodes.

Reference solution

```
data ELGraph = ELGraph ([Node], [Edge])
type Node = String
type Edge = (Node, Node, Int)
```

3 “Unit testing”

How would you test the *neg* function that takes a number and multiplies it by (-1), when applied to ints? Provide a number of test cases. Use the HUnit style of describing test cases.

Reference solution

```
import Test.HUnit -- Not required

pos = 1 ~=? neg (-1::Int)
neg = (-1) ~=? neg (1::Int)
zero = 0 ~=? neg (0::Int)

-- Not required
main = do
    testresults <- runTestTT $ TestList [ pos, neg, zero ]
    print testresults
```

4 “Parametric polymorphism”

Define a polymorphic function including its function signature for adding a value to the end of the list, provided the value is not yet an element of the list. Otherwise, the original list is returned as is. The membership test should be based on linear search and equality. Consider these illustrations:

```
> endappend [1,2,3] 4
[1,2,3,4]
> endappend ["a", "b"] "a"
["a","b"]
> endappend [1,2,3,4] 3
[1,2,3,4]
> endappend [] 4
[4]
```

Reference solution

```
endappend :: Eq a => [a] -> a -> [a]
endappend [] y = [y]
endappend (x:xs) y = if x==y
                        then x:xs
                        else x:(endappend xs y)
```

5 “Higher-order functions”

Define a function *condgroup* including its function signature that takes the following parameters: (1) a list of values *xs* with a polymorphic element type *a* and (2) a function *f* that maps values of type *a* to Boolean values. *condgroup* returns a pair of two lists. The first list contains the values for which *f* evaluates to *true*; the second list contains the other values. Consider the following examples:

```
> condgroup [1,2,3,4] even
([2,4],[1,3])
> condgroup [-1.1,2.2,-4.5] (<0)
([-1.1,-4.5],[2.2])
> condgroup ["Hello", "world","!"] ((==5) . length)
(["Hello","world"],["!"])
> condgroup [] even
([],[])
```

Reference solution

```
condgroup :: [a] -> (a -> Bool) -> ([a],[a])
condgroup [] _  = ([],[])
condgroup (x:xs) f = if f x then ((x:ts),fs) else (ts,(x:fs))
                    where
                      (ts,fs) = condgroup xs f
```

6 “Monoids”

Define a monoid for Boolean conjunction (‘and’).

Reference solution

```
newtype And = And Bool
instance Monoid And
 where
   mempty = And True
   mappend (And x) (And y) = And (x && y)
```

7 “Functors”

Consider rose trees as follows. Each inner node (instead of a leaf) is labeled by a polymorphic value.

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

Implement an instance of the type class `Functor` with the member `fmap`.

Reference solution

```
instance Functor Tree
  where
    fmap f (Leaf) = Leaf
    fmap f (Fork x ys) = Fork (f x) (map (fmap f) ys)
```

8 “Unparsing & parsing”

What are the major combinators for parsing strings in the sense of parsing with package Text.Parsec? 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

char a — parse a character a.

digit — parse a digit.

many f — parse a possibly empty sequence with the parser f for elements.

many1 f — parse a non-empty sequence with the parser f for elements.

...

9 “Functional data structures”

Consider the following data type of stacks:

```
data Stack a = Empty | Push a (Stack a) deriving (Show)
```

Provide an implementation and type signature of a function *srepeat* that takes a value as a parameter and returns a stack in which the value is repeated infinitely many times.

Reference solution

```
srepeat :: a -> Stack a
srepeat x = Push x (srepeat x)
```

10 “Monads”

Consider these expressions forms:

```
data Expr = Constant Float | Add Expr Expr
```

Consider this interpreter for the expression forms:

```
-- A straightforward interpreter
eval :: Expr -> Float
eval (Constant f) = f
eval (Add e1 e2) = eval e1 + eval e2
```

Convert the interpreter to monadic style.

Reference solution

```
-- A monadic style interpreter in do notation
evalM :: Monad m => Expr -> m Float
evalM (Constant f) = return f
evalM (Add e1 e2) = do
  f1 <- evalM e1
  f2 <- evalM e2
  return (f1 + f2)
```