04IN1023: Introduction to functional programming

Final—Final SS 2015

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel
23 July 2015

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. 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.

4. 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.

5. 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.

6. 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, to provide for the exam assignments or to ask code in the assignments.

7. 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.

8. 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 dry run.

1 “Simple algorithms”

Define a function that given a list of ints, returns the smallest index i, if any, so that all elements with indexes smaller than i are sorted in ascending order while the element at index i is smaller than the element at index $i-1$. (Note: The index of the first element in a list is ‘0’.) Consider this illustration:

```
> findIndex []
Nothing
> findIndex [1]
Nothing
> findIndex [1,2]
Nothing
> findIndex [1,2,3]
Nothing
> findIndex [1,3,2]
Just 2
```

Reference solution

```
findIndex :: [Int] -> Maybe Int
findIndex = f 1
 where
   f _ [] = Nothing
   f _ [_] = Nothing
   f i (x:y:zs) = if y<x then Just i else f (i+1) (y:zs)
```

2 “Simple data models”

Declare a data type for rose trees as follows. Each inner node (as opposed to leaf) is labeled by an *Int*. Each leaf (as opposed to inner node) is labeled by a *String*.

Reference solution

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

3 “Unit testing”

How would you test the *signum* function for signs (1, 0, -1), when applied to ints? Provide a number of test cases.

Reference solution

```
import Test.HUnit -- Not required

pos = 1 ~=? signum (42::Int)
neg = (-1) ~=? signum (-88::Int)
zero = 0 ~=? signum (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 zipping together two lists while returning *Nothing* when the lists are of unequal length. Consider this illustration:

```
> zipMaybe [1,2,3] ['a','b','c']
Just [(1,'a'),(2,'b'),(3,'c')]
> zipMaybe [1,2,3] ['a','b']
Nothing
```

Reference solution

```
zipMaybe :: [a] -> [b] -> Maybe [(a, b)]
zipMaybe [] [] = Just []
zipMaybe (x:xs) (y:ys) =
 maybe Nothing (\zs -> Just ((x, y):zs)) (zipMaybe xs ys)
zipMaybe _ _ = Nothing
```

5 “Higher-order functions”

Define the *map* function in terms of *foldr*. As a reminder, here are the types of the functions:

```
> :t map
map :: (a -> b) -> [a] -> [b]
> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Reference solution

```
map f = foldr ((:) . f) []
```

6 “Monoids”

Define a *newtype* on top of *Int* and instantiate the *Monoid* typeclass for the new type such that ‘1’ is the unit and ‘*’ is the binary operation of the monoid.

Reference solution

```
import Data.Monoid -- Not required

newtype Mult = Mult Int

instance Monoid Mult
  where
    mempty = Mult 1
    (Mult x) ‘mappend‘ (Mult y) = Mult (x*y)
```

7 “Functors”

Here is one algebraic law that is supposed to hold for any *Functor* instance:

```
fmap id = id
```

What other law has to hold?

Reference solution

```
fmap (f . g) = fmap f . fmap g

-- Thanks to Karsten Krmer
-- for pointing out that the first law implies the second one.
-- https://www.fpcomplete.com/user/edwardk/snippets/fmap
```

8 “Unparsing & parsing”

What are the major combinators for composing documents (type *Doc*) in the sense of unparsing with package Text.PrettyPrint? You don’t need to get the names of the combinators right, but suggest 3+ suitable combinators in terms of their type signature and a line comment for explanation.

Reference solution

```
-- The empty docment
empty :: Doc

-- Compose horizontally
(<>) :: Doc -> Doc -> Doc

-- Compose horizontally with extra space for separation
(<+>) :: Doc -> Doc -> Doc

-- Compose vertically
($$) :: Doc -> Doc -> Doc
```

9 “Functional data structures”

Define a function that illustrates the notion of ‘path copying’. Add a line comment to explain what the function does.

Reference solution

```
-- Let’s look at binary search trees
data BST e = Empty | Node (BST e) e (BST e)

-- This is the insert function; it performs path copying
insert e s =
    case s of
      Empty -> Node Empty e Empty
      (Node s1 e’ s2) ->
        if e<e’
          then Node (insert e s1) e’ s2
          else if e>e’
            then Node s1 e’ (insert e s2)
            else Node s1 e’ s2,
```

10 “Monads”

Define the bind function for the *State* monad. As a reminder, here is a possible datatype constructor for the state monad:

```
newtype State s a = State { runState :: s -> (a,s) }
```

Reference solution

```
c >>= f =
  State (\s -> let (x,s’) = runState c s
                in runState (f x) s’)
```