04IN1023: Introduction to functional programming

Final—Dry run SS 2015

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel
15 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”

Metaremark: Simple algorithms for searching, sorting, or other forms of analysis or transformation are considered. List processing and other basic data types may be involved. Pattern matching and recursion may be involved.

Define a function that tests whether a given list of ints is sorted (in any order you favor).

Reference solution

```
-- Signature is not required
sorted :: [Int] -> Bool
sorted [] = True
sorted [x] = True
sorted (x1:x2:xs) = x1 <= x2 && sorted (x2:xs)
```

2 “Simple data models”

Metaremark: Type synonyms, new types, algebraic data types, record types may be considered. The type constructors for lists, tuples, maybes, and eithers may be involved. Typically, a specific problem is considered for which a data model should be authored or commented on.

Declare a data type for shapes as follows. One kind of shape is ‘triangle’ described by three points. Each point consists of two ints for the x/y coordinates. Another kind of shape is ‘circle’ described by one point (the centre) and an Int for the radius. Yet another kind of shape is ‘composite’ described by two shapes that are composed in this way.

Reference solution

```
type Point = (Int,Int)
data Shape = Circle Point Int
           | Triangle Point Point Point
           | Composite Shape Shape
```

3 “Unit testing”

Metaremark: The style of unit testing, as supported by HUnit, is considered. One may be asked to author test cases or comment on test cases.

How would you test the *not* function for Boolean negation? Provide at least one test case.

Reference solution

```
import Test.HUnit -- Not required

-- One case required; several provided
negTrue = False ~=? not True
negFalse = True ~=? not False
doubleNegTrue = doubleNegation True
doubleNegFalse = doubleNegation False
doubleNegation x = x ~=? not (not x)

-- Not required
main = do
    testresults <- runTestTT $
      TestList [negTrue, negFalse, doubleNegTrue, doubleNegFalse]
    print testresults
```

4 "Parametric polymorphism"

Metaremark: Simple scenarios for parametrically polymorphic functions are considered, e.g., functions on tuples, lists, or maybes. Typically, a polymorphic function including its type must be defined or commented on.

Define a polymorphic function including its function signature for swapping the two components of a pair.

Reference solution

```
-- Other definitions welcome.
swap :: (a,b) -> (b,a)
swap x = (snd x, fst x)
```

5 “Higher-order functions”

Metaremark: Standard higher-order functions (such as foldr, foldl, map, filter from the Prelude) or problem-specific higher-order functions are considered. Typically, a higher-order function possibly including its type must be defined or commented on.

Define a polymorphic function *times* which repeats the application of a given argument function, as demonstrated with the following example:

```
> times 22 (+1) 20
42
```

Reference solution

```
-- Signature is not required
times :: Int -> (a -> a) -> a -> a
times 0 f x = x
times n f x = times (n-1) f (f x)
```

6 “Monoids”

Metaremark: The designated type class and typical instances (Sum, Product, []) as well as problem-specific monoids and their applications are considered. The algebraic laws to be met by any monoid instance may also play a role.

Why is Float not readily defined to be an instance of the Monoid type class? *Please, be concise: 140 characters or less.*

Reference solution

Float could be a monoid for summation and multiplication, but there can be only one instance per type.

7 “Functors”

Metaremark: The designated type class and typical instances ([], Maybe, rose trees) as well as problem-specific functors and their applications are considered. The algebraic laws to be met by any functor instance may also play a role.

Consider the following code:

```
-- Sets as lists
data Set a = Set [a]
  deriving Show

-- Construct the empty set
empty :: Set a
empty = Set []

-- Add an element to a set
addTo :: Eq a => a -> Set a -> Set a
addTo x (Set xs) =
  Set $ if elem x xs
          then xs
          else x:xs

-- Functor instance for sets
instance Functor Set
  where
    fmap f (Set xs) = Set (map f xs)
```

The *addTo* function and the functor instance are not perfectly aligned in that an application of *fmap* could lead to a set representation that cannot be possibly reached by a repeated application of *addTo* starting from *empty*. Please explain or give an illustrative sample expression! *Please, be concise: 140 characters or less.*

Reference solution

```
> fmap (const 0) $ addTo 1 $ addTo 2 $ empty
Set [0,0]
```

8 “Unparsing & parsing”

Metaremark: Simple applications of the combinator libraries for unparsing and parsing are considered. Typically, simple combinators need to be exercised or given applications need to be commented on.

Remember the two horizontal composition operators for unparsing:

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

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

Why is the following definition of the latter in terms of the former possibly problematic?

```
x <+> y = x <> text " " <> y
```

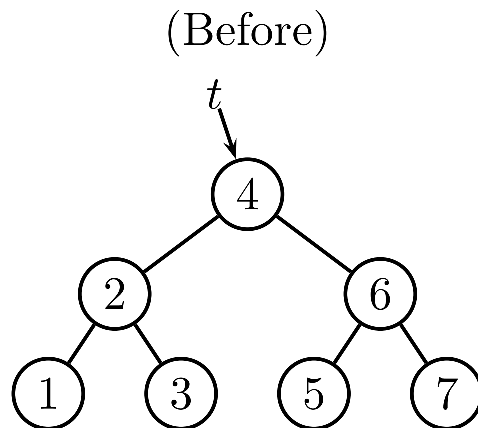
Reference solution

We may prefer that ‘empty’ (say, ””) is a unit of both forms of composition. For instance, if we compose ‘empty’ and, say, ‘*’, even with the second operator, then the result should be ‘*’ rather than ‘ *’.

9 "Functional data structures"

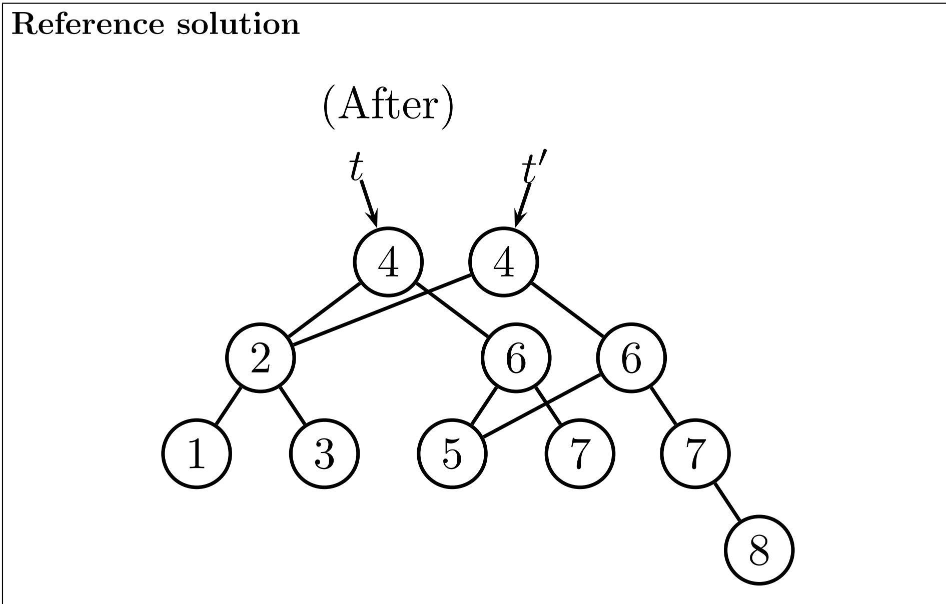
Metaremark: This topic includes simple functional data structures such as stacks, sets, and heaps as well as general aspects of laziness and abstract data types.

Consider the following binary search tree t.



Now, assume that 8 is inserted into t, resulting in a tree t'. Draw the tree t' so that one can see what parts it shares with t.

Reference solution



10 “Monads”

Metaremark: Simple established monads (State, Reader, Writer, Maybe) and possibly trivial variations on these as well as their applications are considered.

Complete the following instance:

```
instance Monad Maybe
  where
    return x = Just x
    Nothing >>= f = ...
    (Just x) >>= f = ...
```

Reference solution

```
instance Monad Maybe
  where
    return x = Just x
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
```