04IN1023: Introduction to functional programming

Final—Dry run SS 2013

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel, M.Sc. Andrei Varanovich
28 June 2013

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 during the dry run. 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'. You might get an extra point for each exam assignment, if you manage to come up with an exceptionally insightful solution. The exam lasts 1 hour. Thus, one can spend more than 5 min per exam assignment.

3. 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 and the per-section footnotes for extra explanations. These topics are maintained for the actual final and the re-sit of the given course edition. The topics may be different in the next edition of this course.

5. One should be prepared—systematically—that the text of the exam assignments relates to the (software) *concepts* that are listed on the wiki pages for the individual lectures. 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. These illustrations may be used, perhaps after some modulation, in the exam assignments or they may be directly related to what's asked for in exam 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.

1 “Simple algorithms”[1]

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

[1]This topic concerns the lectures ‘First steps’ and ‘Searching and sorting’ during which a number of simple algorithms were exercised. For instance, the algorithms for factorial, greatest common divisor, searching, and sorting were exercised. Assignments may involve simple algorithms discussed in the lectures or the labs or variations of those algorithms. All the few relevant algorithms are readily listed on the lecture pages in question.

2 “Simple data models”[2]

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.

[2]This topic concerns the lecture ‘Basic data modeling’, but aspects of data modeling were also encountered in passing in other lectures. One needs to understand declaration forms for data types (algebraic data types, type synonyms, and record types). Further, one needs to use effectively list types, maybe types, product types (perhaps even either types). Of course, one needs to be able to declare recursive data types, possibly involving multiple constructors in some cases with multiple construtor components.

3 “Local scope”[3]

Consider the following code:

```
f x y = g y
  where
    g z = x + z
```

Transform the code such that no local scope is used, i.e., the function g shall be defined at the top-level as opposed to the local scope of f.

[3]This topic was covered in the lectures ‘Basic software engineering for Haskell’, ‘Searching and sorting’, and (to some extent) ‘Higher-order functions’. One should understand that a function defines a local scope in which other functions can be defined for local use indeed. More specifically, one should understand how to exploit or to eliminate local scope so that global declarations are moved into local scope or vice versa.

4 “Parametric polymorphism”[4]

Define a polymorphic function *swap* including its function signature for swapping the two components of a pair. Here is an illustration:

```
> swap (4,2)
(2,4)
> swap ("2",4)
(4,"2")
```

[4]This topic was covered in the lectures ‘Basic software engineering for Haskell’, ‘Searching and sorting’, and ‘Higher-order functions’. We have encountered various polymorphic functions, e.g., the *id* function, the combinator for function composition, and higher-order list-processing combinators.

5 “Higher-order functions”[5]

Define a polymorphic function *times* which repeats the application of a given argument function. Here is an illustration:

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

[5]This topic concerns the lecture with the same name. One needs to understand the notion of higher-order functions and deal with simple examples such as function composition and *twice*. Further, one needs to be fluent in list processing based on appropriate combinators such as *map* or *foldr*. Yet further, one may end up dealing with lambda abstractions as a means of constructing arguments or results of higher-order functions.

6 "Monoids"[6]

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

[6]This topic concerns the lecture 'Type-class polymorphism', but monoids were also encountered in passing in other lectures. Monoids provide an example of an algebraic structure that can be generally established via a type class while allowing for different instances. In this manner, monoids also illustrate the idea of type class-polymorphic functionality which may be parameterized in an arbitrary monoid that may be resolved later. Further, monoids also touch upon the issue of algebraic laws such that a proper instance of the type class *Monoid* must satisfy certain, simple laws. One needs to understand the monoid examples that were encountered in the lecture and the corresponding lab.

7 “Functors”[7]

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.*

[7]This topic concerns the lecture ‘Functors and friends’. Functors provide a powerful generalization of what *map* provided for lists: the idea of applying a certain function to all elements in a container. Functors also come with interesting algebraic laws. One needs to understand the functor examples that were encountered in the lecture and the corresponding lab.

8 "Reasoning"[8]

How would you test the *head* function for retrieving the head of a list? *Please, be concise: 140 characters or less.*

[8]In this (edition of this) course, we have not deeply studied reasoning (such as equational reasoning), but some forms of reasoning poped up occasionally. For instance, we cared about algebraic properties of monoids, functors, and monads. We also reasoned effectively when we used HUnit and QuickCheck for testing, since we would write down and actually test expected properties of functionality for some given or generated data.

9 "Lazy evaluation"[9]

Write down a Haskell expression (based on the Prelude) which terminates fine and prints when entered at the interpreter prompt, which however would diverge (loop)—if Haskell had an eager rather than a lazy evaluation semantics.

[9]In this (edition of this) course, we have not deeply studied laziness (see "Lazy evaluation" on the 101wiki), but some basic awareness of the difference between lazy versus eager evaluation was established in a number of lectures in passing.

10 “Monads”[10]

Add the missing equation for the Maybe’s monad bind operation:

```
data Maybe a = Nothing |  Just a

instance Monad Maybe
  where
    return = Just
    Nothing >>= f = Nothing
```

[10]This is the most difficult subject in this course. Don’t assign much time to it, before you haven’t secured the other subjects. Also, the exam will not drill deep into monads. Instead, the idea is to get you started on the subject: *return*, *bind*, some simple monads such as *State* or *Writer* monad, and perhaps some simple applications of monadic style to *parsing*. One should try to understand the monad examples that appeared in the lecture and the lab.