04IN1023: Introduction to functional programming

Resit—Resit WS 2018/2019

Universität Koblenz-Landau, FB4
PD. Stefan Bosse, Marcel Heinz

11 January 2019

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”

Implement the function *countSmileys* that takes a String and returns the number of smileys found in the String. Only ’:)’ is a valid smiley. A few test cases are provided below.

```
> countSmileys "Hi :)!"
1
> countSmileys "This is not a smiley :("
0
> countSmileys "The cake (::) is a lie :)!"
2
> countSmileys "Hi! :) It’s a trap :-)"
1
```

Reference solution

```
countSmileys :: String -> Int
countSmileys (x:y:xs) = if x == ’:’ && y == ’)’
                            then 1 + countSmileys xs
                            else countSmileys (y:xs)
countSmileys _ = 0
```

2 “Simple data models”

Implement a simplified model for processing online trades as follows. For every trade, there exist two parties. One is buying, the other is selling. A trade party can either be an individual person or a company. For a person, the name and address are provided. For a company, a registration number is additionally stored.

Reference solution

```
data Trade = Trade TradeParty TradeParty
data TradeParty = Person Name Address
                | Company Name Address RegNumber

type Name = String
type Address = String
type RegNumber = String
```

3 “Parametric polymorphism”

Define a polymorphic function *sumprod* that takes a list of any kind of numbers and computes a pair that consists of the sum and the product of the list. Do not forget to provide the correct type signature of *sumprod*.

```
> sumprod [1..6]
(21, 720)
> sumprod [1,3]
(4, 3)
> sumprod [1.2, 5.0]
(6.2, 6.0)
```

Reference solution

```
sumprod :: Num a => [a] -> (a, a)
sumprod xs = (sum xs, foldr (*) 1 xs)
```

4 “Functional data structures”

Define the function *insert* that illustrates the notion of path copying for binary search trees. It takes an integer and inserts it into the binary search tree if it does not already exist. Mark your code snippets, where path copying is illustrated.

```
data BST e = Empty | Node (BST e) e (BST e)
```

Reference solution

```
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,
```

5 “Higher-order functions”

Define the function *zipWith* that takes two lists A and B and a function f. The result is a list R, whether the element R_i is the result of applying f to A_i and B_i. If A and B do not have the same number of elements, an empty list is returned!

```
> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
> zipWith (+) [1..3] [4..6]
[5, 7, 9]
> zipWith (-) [4..6] [2,4]
[]
```

Reference solution

```
zipWith’ f (x:xs) (y:ys) = if length xs == length ys
                            then (f x y) : (zipWith’ f xs ys)
                            else []
zipWith’ f _ _ = []
```

6 “Functors”

Below, you find a data model for graphs in Haskell as algebraic data types with records. Nodes have a value entry that can be of any type. Provide a Functor for the type Graph, so that a function can be applied to the value in nodes.

```
data Graph t = Graph { nodes :: [Node t] }

data Node t = Node {  name :: String,
                      value :: t,
                      edges :: [Edge] }

data Edge = Edge { target :: String }
```

Reference solution

```
instance Functor Graph where
   fmap f g = Graph $ map (fmap f) (nodes g)

instance Functor Node where
   fmap f n = Node (name n) (f $ value n) (edges n)
```

7 “Sorting”

Below, the function *sort* takes a list of comparable values and is supposed to implement the Merge-sort algorithm. The results do not seem to be correct. Mark the mistakes in the implementation and repair the function so that it works correctly.

```
sort :: Ord e => [e] -> [e]
sort [] = []
sort [x] = [x]
sort xs = merge (sort zs) (sort ys)
  where
    (zs,ys) = split xs

split :: [e] -> ([e],[e])
split xs = (\len -> (take len xs, drop len xs)) $ length xs ‘div‘ 2

merge :: Ord e => [e] -> [e] -> [e]
merge _ [] = []
merge [] _ = []
merge (x:xs) (y:ys) = if x<=y
                        then x : merge xs ys
                        else y : merge (x:xs) (y:ys)
```

Reference solution

```
sort :: Ord e => [e] -> [e]
sort [] = []
sort [x] = [x]
sort xs = merge (sort zs) (sort ys)
  where
    (zs,ys) = split xs

split :: [e] -> ([e],[e])
split xs = (\len -> (take len xs, drop len xs)) $ length xs ‘div‘ 2

merge :: Ord e => [e] -> [e] -> [e]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) = if x<=y
                        then x : merge xs (y:ys)
                        else y : merge (x:xs) ys
```

8 “Anonymous Functions”

Below, the function *leapyears* takes a list of years and filters the list based on whether the year is a leap year. Replace the locally defined function by an anonymous function.

```
type Year = Int

leapyears :: [Year] -> [Year]
leapyears ys = filter leapyear ys
    where
       leapyear :: Year -> Bool
       leapyear y = (y ‘mod‘ 4==0) && not (y ‘mod‘ 100==0)
                      || (y ‘mod‘ 400==0)
```

Reference solution

```
type Year = Int

elementsWith :: [Year] -> [Year]
elementsWith = filter (\y -> (y ‘mod‘ 4==0) && not (y ‘mod‘ 100==0)
                                 || (y ‘mod‘ 400==0))
```

9 “Computational Complexity”

Consider the following sorting algorithm for lists (commonly called ”insertion sort”).

Question 1: Analyze the insertion sort algorithm *sort* applied to a list of numbers and give a formula *f(n)* that expresses the number of unit operations in relation to the number n of elements of the list to be sorted for the **worst case scenario**, i.e., a totally unsorted list, finally giving the complexity class $\Theta(f(n))$.

Question 2: Additionally, give two functions $g(n)$ and $h(n)$ that are a lower and upper bound of $f(n)$, respectively, i.e., for any $n > n_0$ the function g is lower and h is higher than f.

Question 3: Does the run-time really depend on the presorting of the list? Are there different results for a good, mean, and worst case sorted list?

```
sort []  = []
sort [x] = [x]
sort (x:xs) = insert (sort xs)
  where
    insert [] = [x]
    insert (y:ys) | x <= y    = x : y : ys
                  | otherwise = y : insert ys
```

Reference solution

1. The complexity class in the worst case is $O(n^2)$.
2. For $n_0 = 1$, $g(n) = n$ and $h(n) = n^3$
3. The run-time depends on the presorting of the list. If the list is sorted, *insert* is never called by itself (see $otherwise = y : insert\ ys$).

10 “Unit testing”

How would you test the *substring* function that takes two Strings and checks whether the first String is part of the second.

Reference solution

```
import Test.HUnit -- Not required

begin = True ~=? substring "Hell" "Hello"
middle = True ~=? substring "2" "424"
end = True ~=? substring "2" "42"
notexists = False ~=? substring "Hello" "No"

-- Not required
main = do
  testresults <- runTestTT $ TestList [empty,exists,notexists]
  print testresults
```