04IN1023: Introduction to functional programming

Final—SS 2014

Universität Koblenz-Landau, FB4
Prof. Dr. Ralf Lämmel
31 July 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)**	**Zusatzpunkt?**
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

1 “Simple algorithms”

Define a function *bst* that tests the property of a *binary search tree* with ints at the nodes. That is, for each node in the tree, it holds that the elements on the left are not greater than the int at the node, whereas the elements on the right are greater.

Here is a declaration of a data type for trees:

```
data Tree = Empty | Fork Int Tree Tree
```

Here is an illustration of the function in question:

```
> bst Empty
True
> bst (Fork 42 Empty Empty)
True
> bst (Fork 42 (Fork 37 Empty Empty) Empty)
True
> bst (Fork 42 (Fork 37 Empty Empty) (Fork 88 Empty Empty))
True
> bst (Fork 42 (Fork 77 Empty Empty) (Fork 88 Empty Empty))
False
```

Reference solution

```
bst Empty = True
bst (Fork i l r) =
  and [
    bst l,
    bst r,
    case l of
      Empty -> True
      (Fork j _ _) -> j <= i,
    case r of
      Empty -> True
      (Fork j _ _) -> j > i ]
```

2 "Simple data models"

Declare a data type for *shapes* as follows. One kind of shape is *circle*; it is described by one point (a pair of floats) for the *centre* and a float for the *radius*. Another kind of shape is *ellipse*; it is described by one point for the *centre* and two floats for the *minor and major radii*. Make use of record notation and introduce a helper record type for points.

Reference solution

```
data Point = Point { getX :: Float, getY :: Float }
data Shape
  = Circle {
      getCentre :: Point,
      getRadius :: Float
    }
  | Ellipse {
      getCentre :: Point,
      getMajorRadius :: Float,
      getMinorRadius :: Float
    }
```

3 "Local scope"

Consider the following function and its illustration:

```
inclist = map (\x -> x + 1)

> inclist [1,2,3]
[2,3,4]
```

Transform the function definition such that no lambda expression is used, but a helper function *f* is defined in the local scope of *inclist* and passed to *map* instead.

Reference solution

```
inclist = map f
  where
    f x = x + 1

-- Alternative solution
inclist = map f
  where
    f = (+1)
```

4 "Parametric polymorphism"

Define a polymorphic function *split* including its function signature for retrieving simultaneously the head and the tail of a list. The result needs to use a *Maybe* type. Here is an illustration:

```
> split []
Nothing
> split [1,2,3]
Just (1,[2,3])
```

Reference solution

```
split :: [a] -> Maybe (a, [a])
split [] = Nothing
split (x:xs) = Just (x, xs)
```

5 "Higher-order functions"

Consider the following function for finding the maximum of a list:

```
findmax :: [Int] -> Maybe Int
findmax [] = Nothing
findmax (x:xs)
  = case findmax xs of
      Nothing -> Just x
      Just y -> Just $ if y > x then y else x
```

Here is an illustration:

```
> findmax [3,1,7]
Just 7
```

Redefine the function *findmax* in terms of *foldr*.

Reference solution

```
findmax :: [Int] -> Maybe Int
findmax = foldr f Nothing
  where
    f x r
      = case r of
          Nothing -> Just x
          Just y -> Just $ if y > x then y else x
```

6 “Monoids”

Define one monoid instance for *Bool*. This instance could be concerned with conjunction (“and”) or disjunction (“or”). Again, you only need to define one instance.

Reference solution

```
-- We reproduce instances of Data.Monoid
-- The import is not required by a solution.

import Data.Monoid hiding (All, Any, getAny, getAll)

-- One option: conjunction

newtype All = All { getAll :: Bool }

instance Monoid All
  where
    mempty = All True
    x `mappend` y = All (getAll x && getAll y)

-- Another option: disjunction

newtype Any = Any { getAny :: Bool }

instance Monoid Any
  where
    mempty = Any False
    x `mappend` y = Any (getAny x || getAny y)
```

7 “Functors”

Consider the following data type of lists with an even number of elements:

```
data ListEven a = Empty | TwoMore a a (ListEven a)
```

Define a functor instance for this data type.

Reference solution

```
instance Functor ListEven
  where
    fmap _ Empty = Empty
    fmap f (TwoMore x y l) = TwoMore (f x) (f y) (fmap f l)
```

8 "Unparsing & parsing"

Consider the following parser:

```
import Text.Parsec

-- Shorthand for the parser type
type Parser = Parsec String ()

-- Parse an int or a string
parseIntOrString :: Parser ()
parseIntOrString =
  ( parseInt >> return () )
  <|>
  ( parseString >> return () )

-- Parse a double-quoted string
parseString :: Parser String
parseString =
  string "\"" >>
  many (noneOf "\"") >>= \xs ->
  string "\"" >>
  return xs

-- Parse an unsigned int
parseInt :: Parser Int
parseInt =
  many1 digit >>= \xs ->
  return ((read xs)::Int)
```

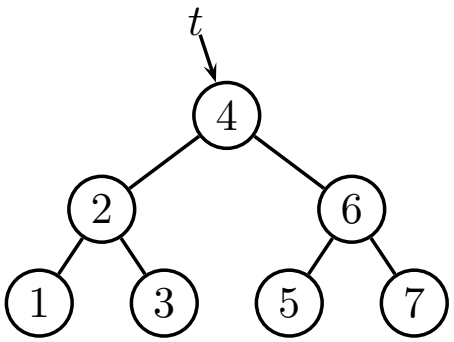
Modify the definition of *parseIntOrString* so that it returns a result of type *Either Int String*. Hint: you need to return intermediate results via *Either*'s constructors *Left* and *Right*.

Reference solution

```
-- The type is not required.
parseIntOrString :: Parser (Either Int String)
parseIntOrString =
  ( parseInt >>= return . Left )
  <|>
  ( parseString >>= return . Right )
```

9 "Functional data structures"

Consider the following binary search tree t.



Consider insertion of an element into t. Please, answer these questions:
i) What is the maximum number of elements that need to be copied
ii) What is the maximum length of a path in the result?
Please, explain. *Please, be concise: 140 characters or less.*

Reference solution

i) 3 (the number of elements on the longest path of the input)
ii) 4 elements or 3 edges (as some path of the input gets extended by a new leaf node)

10 "Reasoning"

Consider the following property that may be worth testing for a company in the sense of the 101system that we implement time and again in the course:

```
-- Some property for testing
prop_what :: Company -> Bool
prop_what c
  = length ns == length (nub ns)
      where ns = map getEmployeeName (getEmployees c)

-- Helper function: Return all employees of a company
getEmployees :: Company -> [Employee]

-- Helper function: Return name of an employee
getEmployeeName :: Employee -> Name

-- Imported from Data.List: Removes all doubles in a list
nub :: Eq a => [a] -> [a]
```

What does the property check?
Why is it worth testing?
Please, be concise: 140 characters or less.

Reference solution

The property checks whether the names of a company's employees are all different. If so, the names could be used to reference the employees unambiguously. (Alternative answer for "why": it is perhaps not even worth testing because persons with the same name can exist.)