Radboud University
Institute for Computing and
Information Sciences

Exam
Functional Programming 2

07.13.2018, 12:30 – 15:30

Note: Please complete in block letters.

Name: ....

Student number: ....

Please read carefully before answering the questions:

- Write your name and student number on *each* sheet.
- Check that the exercise set is complete: the exam consists of 7 questions.
- Read the questions carefully.
- Write your answers *on* the question sheets. Feel free to use the back pages, as well. Two blank pages are included at the rear. Additional paper is available on request.
- Use a pen with a permanent ink.
- Write legibly. Be concise and precise.
- This exam is “closed book”. You are not allowed to use lecture notes, personal notes, books, a notebook, or any other electronic devices.

Question	**1**	**2**	**3**	**4**	**5**	**6**	**7**	$\sum$
max. points:	12	12	12	12	12	12	12	**84**
obtained:								

Result: ....

(Peter Achten)

Question		12 points
1	Name:	(3 × 4)
	Student number:	Σ: .... / S: ....

Propositional calculus

Formulas of the propositional calculus are built up from propositional variables using negation, conjunction, and disjunction. Propositional formulas can be defined by the datatype declaration.

```
data Formula var
  =  Var  var
  |  Not  (Formula var)
  |  And  (Formula var) (Formula var)
  |  Or   (Formula var) (Formula var)
```

To evaluate such a formula, you need to have an *environment* that gives a value to the variables:

```
type Environment var val = [(var,val)]
```

1. Define a function, `eval`, that evaluates a propositional formula, given an environment. Recall the prelude function `lookup` with type `:: Eq a => a -> [(a, b)] -> Maybe b`. If a variable is missing in the environment, then `eval` returns `Nothing`, otherwise it returns the expected boolean result.

   ```
   eval :: (Eq var) => Environment var Bool -> Formula var -> Maybe Bool
   ```

Question		
1	Name: .... Student number:	

Propositional calculus — *continued*

2. A *valuation* of a list of variables of type `var` is an environment of type `Environment var val` that assigns to each of the variables a value of type `val`. Define the function that is given a list of all variables and a list of all possible values, that computes the list of all possible valuations (without duplicates). You are allowed to assume that neither the list of variables nor the list of values have duplicate values.

```
all_valuations :: [var] -> [val] -> [Environment var val]
```

3. Assume to have the function `vars :: (Eq var) => Formula var -> [var]` that retrieves a list of all variables without duplicates from a formula. Use this function, and the ones defined above, to write the function that computes all valuations of a formula that yield `Just True` via the function `eval`.

```
truths :: (Eq var) => Formula var -> [Environment var Bool]
```

Question		**12 points**
	Name:	(3 × 4)
2	Student number:	Σ: .... / S: ....

Evaluation strategies

Suppose the Haskell prelude `length` function has the below given definition. Besides `length`, we introduce two other definitions, `my_fun` and `double`.

```
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs

my_fun   = double (length [length [0]])
double x = x + x
```

1. Evaluate `my_fun` using the *applicative-order* evaluation strategy. Show *all* intermediate steps.

Question 2	Name: .... Student number:	

Evaluation strategies — *continued*

2. Evaluate `my_fun` using the *normal-order* evaluation strategy. Show *all* intermediate steps.

3. Evaluate `my_fun` using the *lazy* evaluation strategy. Show *all* intermediate steps.

Question 3	Name: .... Student number:	**12 points** Σ: .... / S: ....

Imperative programming

Develop a function, `doCommands`, that allows a user to enter commands via the console. This is modeled with the following types:

```
type Commands = [(Command, Action)]
type Command  = String
type Action   = [Argument] -> IO ()
type Argument = String

doCommands :: Commands -> IO ()
```

With each command an action is associated. The function `doCommands` keeps asking for input until the user enters the empty string. In that case, `doCommands` terminates after printing the message `goodbye`. If the string is not empty, but its first word is not a member of one the commands, then an error message should be printed on the console. If the string is not empty, and the first word is a member of the commands, then the remaining words of the input should be passed on to the associated action, and the action should be evaluated. (Recall: `getLine :: IO String` reads a line from the standard input device; `putStr :: String -> IO ()` writes a string to the standard output device; `words :: String -> [String]` breaks a string up into a list of words, which are delimited by white space.)

As an example, suppose we define:

```
theCommands = [ ("echo", \args -> putStrLn (unwords args) )
              , ("fib",  \args -> putStrLn (show (fib (read (head args)))) )
              ]
```

Then `doCommands theCommands` may result in the following session:

```
> echo hello world!
hello world!
> fib 10
89
> exit
exit is not a known command
>
goodbye
```

Question		
3	Name: .... Student number:	

Imperative programming — *continued*

Question		12 points
4	Name: .... Student number:	(6 × 2) Σ: .... / S: ....

Higher-order polymorphism

Define a *total* function for each of the following types, that is, provide a binding for each of the type signatures. (You may assume that functions passed as arguments are total.)

```
g1 :: (Functor f) => f (a, b) -> f (b, a)
```

```
g2 :: (Functor f) => f (f (a, b)) -> f (f (a, b, b, a))
```

```
g3 :: (Functor f, Functor g) => f a -> g b -> f (g (a, b))
```

Question		
4	Name: .... Student number:	

Higher-order polymorphism — *continued*

```
g4 :: (Applicative f, Applicative g) => a -> f (g a)
```

```
g5 :: (Monad m) => a -> m (m a)
```

```
g6 :: (Monad m) => m (m a) -> m a
```

Question			12 points
	Name:	....	(2 × 6)
5	Student number:	....	Σ: .... / S: ....

Functor and Applicative

1. Make `IO` an instance of the type class `Functor`.

2. Make `IO` an instance of the type class `Applicative`.

Question 6	Name: .... Student number:	12 points (2 × 6) Σ: .... / S: ....

Folds and unfolds

Consider the definition of binary trees.

```
data Tree elem = Empty | Node (Tree elem) elem (Tree elem)
```

Its base functor with `Rec (TREE elem) = Tree elem` is defined:

```
data TREE elem tree = EMPTY | NODE tree elem tree

instance Functor (TREE elem) where
    fmap f EMPTY        = EMPTY
    fmap f (NODE l a r) = NODE (f l) a (f r)
```

Use the higher-order operations

```
fold    :: (Base f) => (f a -> a) -> (Rec f -> a)
unfold  :: (Base f) => (a -> f a) -> (a -> Rec f)
```

to implement the following functions. You *must not* use recursion. (You can assume that a suitable instance of `Base` i.e. `Base (TREE elem)` is provided from somewhere; its definition is elided as the details are not relevant for this question.)

1. Define a function, `print`, that prints a binary tree, respecting the branching structure. For instance, if `my_tree = Node Empty 42 (Node (Node Empty 12 Empty) 24 Empty)`, then `print my_tree` should emit `(.) 42 (((.) 12 (.)) 24 (.))`.

   ```
   print :: (Show elem) => Tree elem -> String
   print =
   ```

Question 6	Name: .... Student number:	

Folds and unfolds — *continued*

2. Define a function, `distribute`, that creates a binary tree from a list of values by distributing the list elements over the binary tree such that the *middle* element of the list is at the root of the tree, the elements in the first half get distributed over the left sub tree to be generated, and the elements after the middle element get distributed over the right sub tree to be generated.

   Examples: `print (distribute [1,2])` should emit `((.) 1 (.)) 2 (.)`; `print (distribute [1,2,3])` should emit `((.) 1 (.)) 2 ((.) 3 (.))`.

   ```
   distribute :: [elem] -> Tree elem
   distribute =
   ```

Question		**12 points**
7	Name: .... Student number:	Σ: .... / S: ....

Programming

Write a Haskell program that is given an unsorted list of possibly duplicate numbers (given by `numbers :: [Int]`). Its task is to print the numbers to the console in ascending order, with each number on a new line of output, except that sequences of successive numbers $(n, n+1, \ldots, k-1, k)$ must be printed on a single line of output as $n - k$.

Example: if `numbers = [11,10,5,10,6,5,1,3,8,2,2]`, then the output *must* be:

```
1-3
5-6
8
10-11
```

Question	Name: .... Student number:	

Question	Name: .... Student number:	