University of Bergen

Faculty of Mathematics and Natural Sciences

Examination in:	INF122 – Functional Programming
Day of examination:	9 December 2016
Time of examination:	9:00 – 12:00 (3 hours)
Permitted aids:	None

This problem set consists of 7 pages

Please make sure that your copy of the problem set is complete before you attempt to answer anything

Some general advice and remarks:

- This problem set consists of 6 independent problems.
- If you need a solution to another subproblem, which you did not manage to solve, you can still assume the solution to be available.
- The points from problems one to six sum up to a total of 100 points. The number of points stated on each part indicates the weight of that part.
- Use your time wisely and take into consideration the weight of each question.
- You should read the whole problem set before you start solving the problems.
- Make short and clear explanations!

Good Luck!

Violet Ka I Pun

Problem 1 – Grammar & Parsing (12%)

We consider a language BR consisting of all nonempty sequences of two kinds of brackets,

```
( , ) , [ , ]
```

in which the brackets *match*. For example, the following sequences are included in the language BR:

```
[ ( ) ] ( ) ,
( ( ) ) ( [ ] ) ,
( ( ) ) [ [ ] ] ( ( ) [ ] ) , ...
```

while the following are not included in the language BR:

```
] ( ) [ ( ) ,
( [ ) ] ( [ ] ) , ...
```

(a) Define an unambiguous grammar for the language BR. Remember to identify the start symbol. (5%)

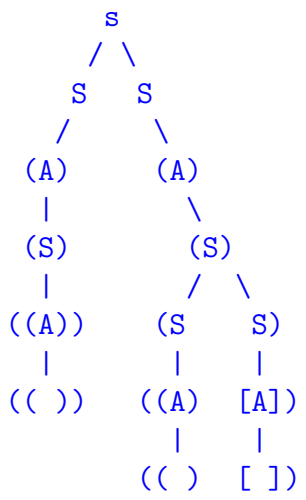
Solution:

```
S = ( A ) | [ A ] | S S
A = S | ϵ
```

□

(b) Draw the parse tree for the sequence `( ( ) ) ( ( ) [ ] )`. (7%)

Solution:



□

Problem 2 – Higher-order functions (12%)

(a) Given the function `foldr` as below:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Define a function

```
append :: [a] -> [a] -> [a]
```

that takes two lists as input and returns their concatenation. For example:

```
> append [4,5,6] [1,2,3]
[4,5,6,1,2,3]
```

(6%)

Solution: (For those who give the answer `[]++[]` can get 2%)

```
append xs ys = foldr (:)  ys xs
```

□

(b) Given the function `foldl` as below:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

What does the following expression return?

```
foldl (\ x -> \ y -> x/2 + y) 4 [2,4,6]
```

(6%)

Solution:

9.0

□

Problem 3 – Phone book (20%)

In this task, you are asked to implement a phone book in Haskell. Use the following type declarations for the implementation.

```
type Name = String
type Pnum = Integer
type Pbook = Name -> Maybe Pnum
```

where `Pbook` is the data type of the phone book, `Name` and `Pnum` are the types of names and phone numbers, respectively.

With this phone book, one can *insert* a new entry by providing a name and an 8-digit number, *lookup* the phone number with a given name, and *delete* an entry with a given name. Each name in the phone book should be *unique*.

(a) Define the function `lookup :: Pbook -> Name -> Maybe Pnum` which returns the corresponding 8-digit number if the input (of type `Name`) exists in the phone book (of type `Pbook`); otherwise, the function should return an appropriate value. (5%)

Solution:
```
lookup b n = b n
```
□

(b) Define the function `insert :: Pbook -> Name -> Pnum -> Pbook` which adds an entry associating the input name (of type `Name`) with the input 8-digit number (of type `Pnum`) in the phone book (of type `Pbook`). If the length of the number is incorrect, the function `insert` should call the function `error :: [Char] -> a` . You do not have to implement the `error` function. (10%)

Solution: (5% for insert, 5% for checking the length of the number)

```
insert b n p = \ x -> if (digitCheck p)
                        then if x == n then Just p else b x
                        else error "Wrong number of digits."

digitCheck :: Pnum -> Bool
digitCheck num = if ((length (show num)) == 8) then True else False
```

□

(c) Define the function `delete :: Pbook -> Name -> Pbook` which removes the entry of the given name (of type `Name`) and the corresponding 8-digit number from the phone book (of type `Pbook`). (6%)

Solution:
```
delete b n =  x -> if x == n then Nothing else b x
```
□

Problem 4 – The game Nim (16%)

We consider in this task the simplified version of Nim, a mathematical game of strategy in which two players *take turns* to remove either 1, 2, or 3 objects from a heap. The player who empties the heap wins. The following example is an illustration that shows how the game, which starts with a heap of 10 objects, is played between two players: Player 1 and Player 2.

Heap size	Moves		
10	Player 1	takes 1 from the heap	
9	Player 2	takes 3 from the heap	
6	Player 1	takes 2 from the heap	$(*)$
4	Player 2	takes 2 from the heap	
2	Player 1	takes the remaining 2 objects from the heap and **wins** the game!	

A winner in Nim: We assume the players in this game are *smart*, and therefore Player 1 in the last round in the illustrating example (∗) will take two objects but *not* one from the heap. That is, the player who is going to remove objects from a heap containing only 1, 2, or 3 objects **wins** the game by default.

4.1 Implementation of Nim

You are going to implement the game Nim with the corresponding interface. The game should start with a heap containing a random number of objects. This random number should be of type `Integer` and be within the range of `10` to `20`. After generating the number, the interface of the game will (i) display the number of objects in the heap, and (ii) ask a player to input the number of objects to be removed from the heap, then (iii) deduct the input number of objects from the current heap. The implementation has to ensure that the player inputs a number between 1 and 3; if the input is out of range, the interface should warn the player about the wrong input and ask for the input again *without aborting*. The game will continue by repeating (i) – (iii), with an alternating player for each round, until a winner is found (see the winner definition above).

The following figure shows how the interface should interact with the players and ultimately find the winner for the illustrating example (∗). Note that those lines with a single number are inputs from the players.

```
Heap size 10.  Player 1:  remove 1 to 3 objects from the heap.
1
Heap size 9.  Player 2:  remove 1 to 3 objects from the heap.
5
Wrong input
Heap size 9.  Player 2:  remove 1 to 3 objects from the heap.
3
Heap size 6.  Player 1:  remove 1 to 3 objects from the heap.
2
Heap size 4.  Player 2:  remove 1 to 3 objects from the heap.
2
Heap size 2.  Player 1 takes all 2 objects and wins!
```

Here is the structure of the implementation of the game Nim in Haskell:

```
import System.IO
import System.Random

-- main starts here

main =
      .
      .                          (a)    -- generate the random number (randNum)
      .
      play randNum 1                    -- call the function play

-- main ends here

-- the play function starts here

play :: Integer -> Integer -> IO ()   -- the type of function play
play =

      .
      .                          (b)    -- implement the play function
      .

-- the play function ends here
```

By following the given structure of the implementation, you are asked to:

(a) complete the `main` function, in which you have to generate the random number (`randNum`) as the initial number of objects for the heap in the game; (6%)

Solution: (Although it is stated in the question, the answer which allows generating a different number everytime the games starts can get extra points, if he/she needs it somewhere.)

```
main = do
     gen <- newStdGen  -- getStdGen is also fine
     let (num, gen') = (randomR (10,20) gen) :: (Integer, StdGen)
     play 10 1
```

□

(b) implement the function `play :: Integer -> Integer -> IO ()` as described in 4.1, where the first argument is the current number of objects in the heap, and the second one is the identity of current player. You can assume the game only allows two players Player 1 and Player 2, and always starts with Player 1. (10%)

Solution:

```
play :: Integer -> Integer -> IO ()
play n p = do
     if (n > 3)
        then do
          putStrLn $ "Heap size " ++ (show n) ++ ". Player "
             ++ (show p) ++ ": remove 1 to 3 objects from the heap."
          r <- getLine
          let o = read r
          if (o > 0 && o < 4)
             then do if (p == 1) then  play (n - o) 2 else play (n - o) 1
             else do putStrLn ("Wrong input")
                     play n p
        else do
          putStrLn $ "Heap size " ++ (show n) ++ ".  Player "
             ++ (show p) ++ " takes all " ++ (show n) ++ " objects and wins!"
```

□

Problem 5 – Type Inference (20%)

Apply the Hindley-Milner algorithm, along with the Martelli-Montanaris unification algorithm (both are provided in the appendix of the problem set), to determine the type of the following Haskell expression by either the *rule-based* or *graph-based* approach, or else to conclude that it has no type in Haskell:

```
\ x -> \ y -> \ z -> (x z) (y z)
```

Solution: (10% for getting the applying the HM-rules/drawing the graph, 10% for unification)

From ghci: (t2 → t → t1) → (t2 → t) → t2 → t1

rule-based:

(t4) $\emptyset \mid \backslash x \to \backslash y \to \backslash z \to (x\ z)\ (y\ z) :: t$ $\{t = t_1 \to t_2\}$

(t4) $x :: t_1 \mid \backslash y \to \backslash z \to (x\ z)\ (y\ z) :: t_2$ $\{t = t_1 \to t_2, t_2 = t_3 \to t_4\}$

(t4) $x :: t_1, y :: t_3 \mid \backslash z \to (x\ z)\ (y\ z) :: t_4$

$S_1 = \{t = t_1 \to t_2, t_2 = t_3 \to t_4, t_4 = t_5 \to t_6\}$

(t3) $x :: t_1, y :: t_3, z :: t_5 \mid (x\ z)\ (y\ z) :: t_6$ S

(t3) $x :: t_1, y :: t_3, z :: t_5 \mid x\ z :: t_7 \to t_6$ $S_2 = S_2' \cup S_2''$

(t2) $x :: t_1, y :: t_3, z :: t_5 \mid x :: t_8 \to (t_7 \to t_6)$ $S_2' = \{t_1 = t_8 \to (t_7 \to t_6)\}$

(t2) $x :: t_1, y :: t_3, z :: t_5 \mid z :: t_8$ $S_2'' = \{t_5 = t_8\}$

(t3) $x :: t_1, y :: t_3, z :: t_5 \mid y\ z :: t_7$ $S_3 = S_3' \cup S_3''$

(t2) $x :: t_1, y :: t_3, z :: t_5 \mid y :: t_9 \to t_7$ $S_3' = \{t_3 = t_9 \to t_7\}$

$x :: t_1, y :: t_3, z :: t_5 \mid z :: t_9$ $S_3'' = \{t_5 = t_9\}$

$S = S_1 \cup S_2 \cup S_3$

$\{t = t_1 \to t_2, t_2 = t_3 \to t_4, \underline{t_4 = t_5 \to t_6}, t_1 = t_8 \to (t_7 \to t_6), t_5 = t_8, t_3 = t_9 \to t_7, t_5 = t_9\}$

$\{t = t_1 \to t_2, \underline{t_2 = t_3 \to (t_5 \to t_6)}, t_1 = t_8 \to (t_7 \to t_6), t_5 = t_8, t_3 = t_9 \to t_7, t_5 = t_9\}$

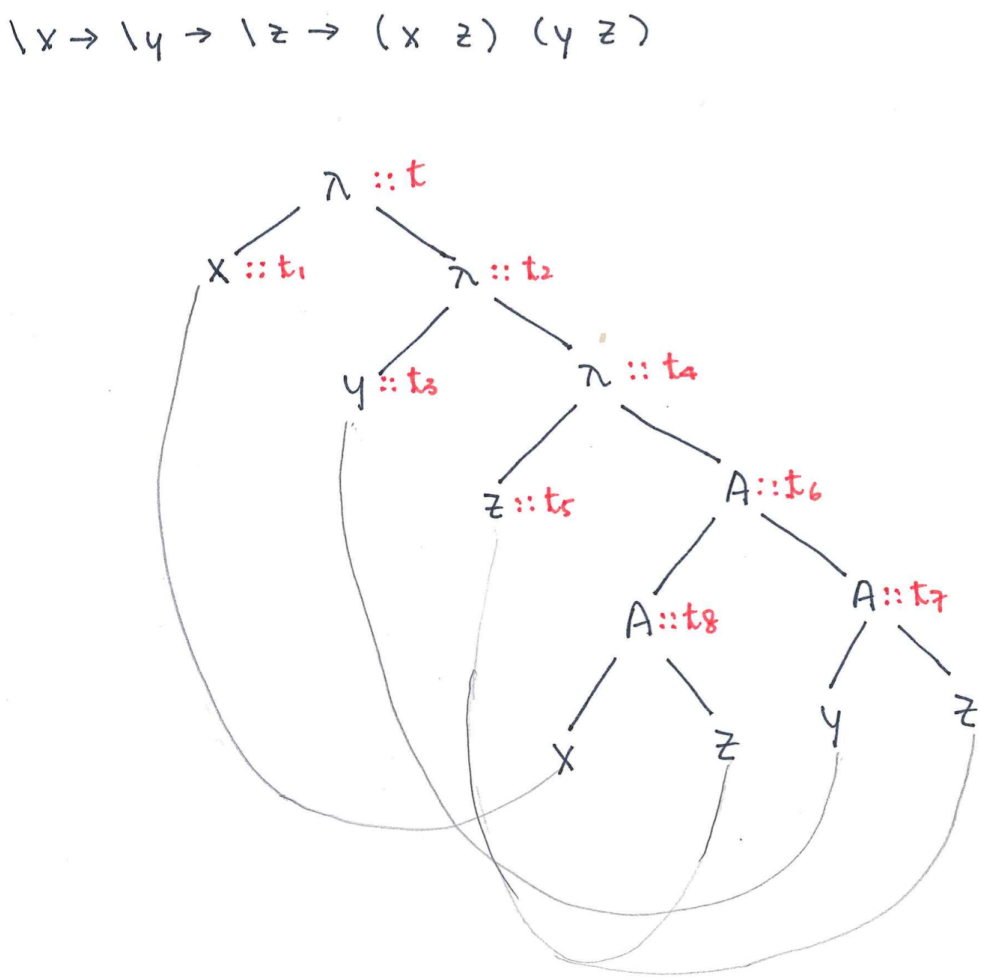
$\{t = t_1 \to (t_3 \to (t_5 \to t_6)), \underline{t_1 = t_8 \to (t_7 \to t_6)}, t_5 = t_8, t_3 = t_9 \to t_7, t_5 = t_9\}$

$\{t = (t_8 \to (t_7 \to t_6)) \to (t_3 \to (t_5 \to t_6)), \underline{t_5 = t_8}, t_3 = t_9 \to t_7, \underline{t_5 = t_9}\}$

$\{t = (t_8 \to (t_7 \to t_6)) \to (t_3 \to (t_5 \to t_6)), \underline{t_5 = t_8 = t_9}, t_3 = t_9 \to t_7\}$

$\{t = (t_5 \to (t_7 \to t_6)) \to (t_3 \to (t_5 \to t_6)), t_5 = t_8 = t_9, \underline{t_3 = t_5 \to t_7}\}$

$\{t = (t_5 \to (t_7 \to t_6)) \to ((t_5 \to t_7) \to (t_5 \to t_6)), t_5 = t_8 = t_9\}$



graph-base:

$\{t = t_1 \to t_2, \underline{t_2 = t_3 \to t_4}, t_4 = t_5 \to t_6, t_8 = t_7 \to t_6, t_1 = t_5 \to t_8, t_3 = t_5 \to t_7\}$
$\{t = t_1 \to (t_3 \to t_4), \underline{t_4 = t_5 \to t_6}, t_8 = t_7 \to t_6, t_1 = t_5 \to t_8, t_3 = t_5 \to t_7\}$
$\{t = t_1 \to (t_3 \to (t_5 \to t_6)), \underline{t_8 = t_7 \to t_6}, t_1 = t_5 \to t_8, t_3 = t_5 \to t_7\}$
$\{t = t_1 \to (t_3 \to (t_5 \to t_6)), \underline{t_1 = t_5 \to (t_7 \to t_6)}, t_3 = t_5 \to t_7\}$
$\{t = (t_5 \to (t_7 \to t_6)) \to (t_3 \to (t_5 \to t_6)), \underline{t_3 = t_5 \to t_7}\}$
$\{t = (t_5 \to (t_7 \to t_6)) \to ((t_5 \to t_7) \to (t_5 \to t_6))\}$

□

Problem 6 – Inductive proof (20%)

Given the type and instance declarations of a binary tree:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

instance Functor Tree where
    fmap g (Leaf x)   = Leaf (g x)
    fmap g (Node l r) = Node (fmap g l) (fmap g r)
```

and the type of the function `fmap :: (a -> b) -> Tree a -> Tree a`. Prove by induction on trees that

(a) `fmap id = id` (7%, 3 for base, 4 for inductive)

(b) `fmap (g . h) = fmap g . fmap h` (13%, 6 for base, 7 for inductive)

Justify *each* step in your equational reasoning with a short comment. You may use the following definitions in the proofs.

(i) `(f . g) x = f (g x)`

(ii) `id x = x`

Solution:

(a) Base case:

```
fmap id (Leaf x) ..............................[by fmap for Leaf]
= Leaf (id x)..................................................[by id]
= Leaf x
```

Inductive case:

```
fmap id (Node l r) ............................[by fmap for Node]
= Node (fmap id l) (fmap id r) .........[by induction hypotheses]
= Node l r
```

(b) In principle, we have to show
`fmap (g . h) (Leaf x) = (fmap g . fmap h) (Leaf x)`
for the base case. However, by the definition of function composition, `(f.g)x = f(g x)`, we just have to show
`fmap (g . h) (Leaf x) = fmap g (fmap h (Leaf x))`.
It is similar in the inductive case.

Base case:

```
fmap (g . h) (Leaf x) .........................[by fmap for Leaf]
= Leaf ((g . h) x) .....................................[by (.)]
= Leaf (g (h x))...............................[by fmap for Leaf]
= fmap g (Leaf (h x))..........................[by fmap for Leaf]
= fmap g (fmap h (Leaf x))
```

```
fmap (g . h) (Node l r).......................[by fmap for Node]
= Node (fmap (g . h) l) (fmap (g . h) r)
  .....................................[by induction hypotheses]
= Node (fmap g (fmap h l)) (fmap g (fmap h r))
  ...................................................[by fmap for Node]
= fmap g (Node (fmap h l) (fmap h r)) .........[by fmap for Node]
= fmap g (fmap h (Node l r))
```

□

Appendix

Martelli-Montanari unification algorithm:

	input	$\Rightarrow$ *result*	*application condition* :
$(u1)$	$E, t = t$	$\Rightarrow E$	
$(u2)$	$E, f(t_1...t_n) = f(s_1...s_n)$	$\Rightarrow E, t_1 = s_1, ..., t_n = s_n$	
$(u3)$	$E, f(t_1...t_n) = g(s_1...s_m)$	$\Rightarrow NO$	$f =/= g \vee n \neq m$
$(u4)$	$E, f(t_1...t_n) = x$	$\Rightarrow E, x = f(t_1...t_n)$	
$(u5)$	$E, x = t$	$\Rightarrow E[x/t], x = t$	$x \notin Var(t)$
$(u6)$	$E, x = t$	$\Rightarrow NO$	$x \in Var(t)$

Hindley-Milner type inference algorithm (a, b are fresh variables):

$(t1)$	$E(\Gamma \mid con :: t)$	$= \{t = \theta(con)\}$ – for a constant con
$(t2)$	$E(\Gamma \mid x :: t)$	$= \{t = \Gamma(x)\}$ – for a variable x
$(t3)$	$E(\Gamma \mid f\ g\ :: t)$	$= E(\Gamma \mid g\ :\ a) \cup E(\Gamma \mid f :: a \rightarrow t)$
$(t4)$	$E(\Gamma \mid \backslash x \rightarrow ex :: t)$	$= \{t = a \rightarrow b\} \cup E(\Gamma, x :: a \mid ex :: b)$