University of Bergen

Faculty of Mathematics and Natural Sciences

Examination in:	INF122 – Functional Programming
Day of examination:	22 February 2017
Time of examination:	9:00 – 12:00 (3 hours)
Permitted aids:	None

This problem set consists of 5 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 7 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.
- You should solve Problems 2 – 5 in Haskell code.
- The points from Problems 1 – 7 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

Grade threshold	
0	F
40	E
50	D
60	C
80	B
90	A

Problem 1 – Grammar & Parsing (10%)

Consider the following grammar:

```
S = S S | S + S | S * | ( S ) | k
```

(a) Draw the parse tree for the string `(k + k) * k` .

(5%)

Solution:

```
            S
           / \
          S   S
         / \   \
        S   *   k
       / | \
      (  S  )
        /|\
       S + S
       |   |
       k   k
```

□

(b) Is the given grammar ambiguous? If yes, justify with examples; otherwise, briefly explain why not.

(5%) 1% for yes, and 4% for giving an example with two different parse trees.

Solution:
yes, it is ambiguous. For example, there are two parse trees for the string `k+k+k`

□

Problem 2 – Higher-order Functions (16%)

In this problem, you are not supposed to use any built-in functions in Haskell except `foldr`, `foldl` and arithmetic/equality operators.

(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)
```

Using the function `foldr`, define a function

```
lengthsum :: (Num a, Num b) => [a] -> (b, a)
```

that takes a list of numbers as input, then returns the length and the sum of the list as a pair. For example:

```
> lengthsum [1,2,3,4,5,6]
(6,21)
```

(6%), 1% for a reasonable try. 3% for a correct answer but used built-in functions like (`fst, snd`, etc.). The order of the sum and the length in the pair does not matter.

Solution: `lengthsum = foldr (\ n (x, y) -> (1+x, n+y)) (0, 0)`
□

(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
```

Using the function `foldl`, define a function

```
inList ::  (Eq a) => a -> [a] -> Bool
```

that takes a value and a list of the same type as inputs, then checks whether the value is an element of the list. For example:

```
> inlist 3 [1,2,3,4,5]
True
```

```
> inlist 3 [6,7,8,9]
False
```

(6%), 1% for a reasonable try. 3% for a correct answer but used built-in functions.

Solution:

```
inList x xs = foldl (\ acc y -> if x == y then True else acc)
False xs
```

□

(c) What do the following expressions return?

(i) `foldr (:)  "hello" "world!"`
(2%) **Solution:** "world!hello" □

(ii) `foldl (\ xs -> \ x -> x:xs) "INF122" "exam"`

(2%) **Solution:** "maxeINF122" □

Problem 3 – An Evaluator (14%)

Consider the following type declaration:

```
data Expr = V Int | M Expr Expr | D Expr Expr
```

You are asked to implement an evaluator

```
eval :: Expr -> Maybe Int
```

which evaluates an expression of type `Expr` defined above. For example:

```
> eval (V 20)
Just 20
> eval (D (M (V 10) (V 2)) (V 4))
Just 5
> eval (M (V 2) (D (V 15) (V 3)))
Just 10
```

```
> eval (V (-20))
Just (-20)
> eval (D (V 10) (V 0))
Nothing
> eval (M (V (-2)) (V 5))
Nothing
```

Note that your implementation should take into account the case of *division by zero*, and the case of multiplication taking only *non-negative* integers.

2% for `V n`, 6% for `M x y`, 6% for `D x y`

Solution:

```
data Expr = V Int | D Expr Expr | M Expr Expr

safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv n m = Just (n `div` m)

eval :: Expr -> Maybe Int
eval (V n) = Just n
eval (M x y) = case eval x of
                    Nothing -> Nothing
                    Just n -> case eval y of
                                   Nothing -> Nothing
                                   Just m -> if (n >= 0 && m >= 0)
                                                then Just (n * m) else Nothing
eval (D x y) = case eval x of
                    Nothing -> Nothing
                    Just n -> case eval y of
                                   Nothing -> Nothing
                                   Just m -> safediv n m
```

□

Problem 4 – Datatypes (10%)

(a) Define a type `Month` to represent the twelve months in a year.

(5%)

Solution:

```
data Month = January | February | March | April | May | June | July | Augus
```

□

(b) Define a function

```
numDays :: Month -> Integer -> Integer
```

which takes a month of type `Month` and a year of type `Integer` as inputs, then calculates the number of days in the given month of the given year. Your solution should take into account that every four year is a *leap year* (e.g., 2016) in which the month February has 29 days. For example:

```
> numDays February 2016          > numDays May 2000
29                               31
```

(5%, only 2% if leap year is not taken care of)

Solution:

```
numDays :: Month -> Integer -> Integer
numDays m y = case m of
                   January -> 31
                   February -> if (leapyear y) then 29 else 28
                   March -> 31
                   April -> 30
                   May -> 31
                   June -> 30
                   July -> 31
                   August -> 31
                   September -> 30
                   October -> 31
                   November -> 30
                   December -> 31

leapyear :: Integer -> Bool
leapyear y = if (y `mod` 4 == 0) then True else False
```

□

Problem 5 – Input and Output (12%)

In this problem, you are not supposed to use any built-in functions in Haskell except `return` and list operators.

(a) Given a list of `IO`-actions, implement a function

```
toDoList :: [IO a] -> IO [a]
```

which executes each of the `IO`-actions in the list and gives back an `IO`-action that returns a list containing the corresponding results in the same order as output.

(6%)

Solution:

```
toDoList :: [IO a] -> IO [a]
toDoList [] = return []
toDoList (a:as) = do
    v <- a
    vs <- toDoList as
    return (v : vs)
```

□

(b) Implement a map function for `IO`-actions

```
mapActions ::  (a -> IO b) -> [a] -> IO [b]
```

which takes a function of type `a -> IO b`, and then applies this function to each item in an input list of type `[a]`. The `mapActions` function finally gives back an `IO`-action that returns a list.

(6%)

Solution:

```
mapActions :: (a -> IO b) -> [a] -> IO [b]
mapActions f [] = return []
mapActions f (x:xs) = do
    y <- f x
    ys <- mapActions f xs
    return (y : ys)
```

□

Problem 6 – Type Inference (20%)

Applying the Hindley-Milner algorithm, along with the Martelli-Montanaris unification algorithm (both are provided in the appendix of the problem set), 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 -> (y x) (x (y x))
```

(10% for applying the HM-rules/drawing the graph to get the *correct* set of equaitons, 10% for unification, full points for the unificatoin if it is done correctly, even though the resulting equations from using HM-rules/the graph is wrong.)

Solution: From ghci:

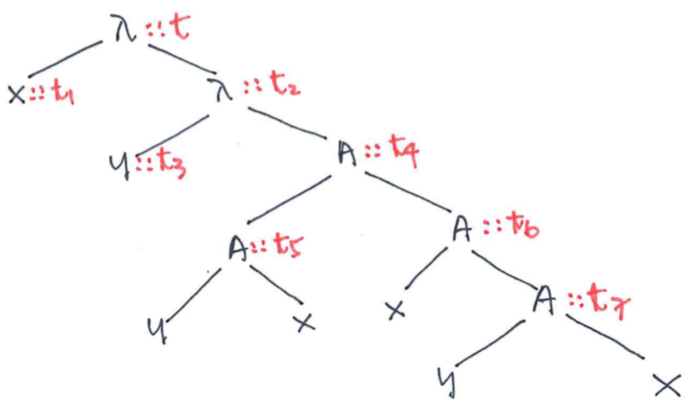
```
((t -> t1) -> t) -> (((t -> t1) -> t) -> t -> t1) -> t1
```

rule-based:

$$
\begin{array}{lll}
(\mathrm{t4}) & \emptyset \mid \backslash x \to \backslash y \to (y\ x)\ (x\ (y\ x)) :: t & \{t = t_1 \to t_2\} \\
(\mathrm{t4}) & x :: t_1 \mid \backslash y \to (y\ x)\ (x\ (y\ x)) :: t_2 & \{t = t_1 \to t_2, t_2 = t_3 \to t_4\} \\
(\mathrm{t3}) & x :: t_1, y :: t_3 \mid (y\ x)\ (x\ (y\ x)) :: t_4 & S \\
\hline
(\mathrm{t3}) & x :: t_1, y :: t_3 \mid y\ x :: t_5 \to t_4 & S_2 = S_2' \cup S_2'' \\
(\mathrm{t2}) & x :: t_1, y :: t_3 \mid y :: t_6 \to (t_5 \to t_4) & S_2' = \{t_3 = t_6 \to (t_5 \to t_4)\} \\
(\mathrm{t2}) & x :: t_1, y :: t_3 \mid x :: t_6 & S_2'' = \{t_1 = t_6\} \\
\hline
(\mathrm{t3}) & x :: t_1, y :: t_3 \mid x\ (y\ x) :: t_5 & S_3 = S_3' \cup S_3'' \\
(\mathrm{t2}) & x :: t_1, y :: t_3 \mid x :: t_7 \to t_5 & S_3' = \{t_1 = t_7 \to t_5\} \\
 & x :: t_1, y :: t_3 \mid y\ x :: t_7 & S_3'' = S_4' \cup S_4'' \\
 & x :: t_1, y :: t_3 \mid y :: t_8 \to t_7 & S_4' = \{t_3 = t_8 \to t_7\} \\
 & x :: t_1, y :: t_3 \mid x :: t_8 & S_4'' = \{t_1 = t_8\} \\
\hline
\end{array}
$$

$$
\begin{array}{r}
S = S_1 \cup S_2 \cup S_3 \\
\{t = t_1 \to t_2,\ \underline{t_2 = t_3 \to t_4},\ t_3 = t_6 \to (t_5 \to t_4),\ t_1 = t_6, t_1 = t_7 \to t_5,\ t_3 = t_8 \to t_7,\ t_1 = t_8\} \\
\{t = t_1 \to (t_3 \to t_4),\ \underline{t_3 = t_6 \to (t_5 \to t_4)},\ t_1 = t_6,\ t_1 = t_7 \to t_5,\ t_3 = t_8 \to t_7,\ t_1 = t_8\} \\
\{t = t_1 \to ((t_6 \to (t_5 \to t_4)) \to t_4),\ \underline{t_1 = t_6},\ t_1 = t_7 \to t_5,\ t_6 \to (t_5 \to t_4) = t_8 \to t_7,\ t_1 = t_8\} \\
\{t = t_6 \to ((t_6 \to (t_5 \to t_4)) \to t_4),\ t_6 = t_7 \to t_5,\ \underline{t_6 \to (t_5 \to t_4) = t_8 \to t_7},\ t_6 = t_8\} \\
\{t = t_6 \to ((t_6 \to (t_5 \to t_4)) \to t_4),\ t_6 = t_7 \to t_5,\ \underline{t_7 = t_5 \to t_4},\ t_6 = t_8\} \\
\{t = t_6 \to ((t_6 \to (t_5 \to t_4)) \to t_4),\ \underline{t_6 = (t_5 \to t_4) \to t_5},\ t_6 = t_8\} \\
\{t = ((t_5 \to t_4) \to t_5) \to ((((t_5 \to t_4) \to t_5) \to (t_5 \to t_4)) \to t_4),\ t_8 = (t_5 \to t_4) \to t_5\}
\end{array}
$$

\x -> \y -> (y x) (x (y x))



graph-base:

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

□

Problem 7 – Inductive Proof (18%)

Given the type of a binary tree:

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

and the following functions:

```
size (Leaf x) = 1
size (Node l r) = size l + size r

balanced (Leaf x) = True
balanced (Node l r) = size l == size r
                      && balanced l && balanced r

mirror (Leaf x) = Leaf x
mirror (Node l r) = Node (mirror r) (mirror l)
```

Prove by induction on trees that

(a) `size (mirror t) = size t` (8%), 2% for base, 6% for inductive

(b) `balanced (mirror t) = balanced t` (10%), 2% for base, 8% for inductive

Justify *each* step in your equational reasoning with a short comment.

Solution:

(a) Base case:

```
size (mirror (Leaf x)) = ...............................[by mirror]
= size (Leaf x)
```

Inductive case:

```
to show: size (mirror (Node l r)) = size (Node l r)
size (mirror (Node l r)) ...............................[by mirror]
= size (Node (mirror r) (mirror l)) ....................[by size]
= size (mirror r) + size (mirror l) ...............[by induction]
= size r + size l ..............................[by commutativity]
= size l + size r .....................................[by size]
= size (Node l r)
```

(b) Base case:

```
balanced (mirror (Leaf x)) ............................[by mirror]
= balanced (Leaf x)
```

Inductive case:

```
to show: balanced (mirror (Node l r)) = balanced (Node l r)
balanced (mirror (Node l r)) ..........................[by mirror]
= balanced (Node (mirror r) (mirror l)) ...........[by balanced]
= size (mirror r) == size (mirror l)
  && balanced (mirror r)
  && balanced (mirror l) ......[by (a) size (mirror t) = size t]
= size r == size l
  && balanced (mirror r)
  && balanced (mirror l) ..........................[by induction]
= size r == size l
  && balanced r && balanced l .................[by commutativity]
= size l == size r
  && balanced l && balanced r ......................[by balanced]
= balanced (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
$(t3)$	$E(\Gamma \mid f\ g\ :: t)$	$= E(\Gamma \mid g\ :\ a) \cup E(\Gamma \mid f :: a \to t)$
$(t4)$	$E(\Gamma \mid \backslash x \to ex :: t)$	$= \{t = a \to b\} \cup E(\Gamma, x :: a \mid ex :: b)$