Lastname: _____

Firstname: _____

Matriculation Number: _____

Exercise	Points	Score
Programming with I/O	16	
Programming with Abstract Datatypes and Functions	21	
Programming with Lists and Trees	23	
Evaluation and Types	20	
$\sum$	80	

- You have 90 minutes to solve the exercises.
- The exam consists of 4 exercises, for a total of 80 points.
- The available points per exercise are written in the margin.
- Don't remove the staple (Heftklammer) from the exam.
- Don't write your solution in red color.

Exercise 1: Programming with I/O 16

Write a program which conducts the following task.

(a) Implement a function `validName :: String -> Bool` which checks whether an input is a valid name, i.e., it is non-empty and only consists of letters 'a', ..., 'z', 'A', ..., 'Z' and the blank symbol. For instance, `"Paul Meyer"` is a name, but `"Daisy76"` is not. (6)

(b) Write a program which does the following. (10)

- It asks the user for a name until a valid name is entered.
- It prints the name in lower case letters. Note that a function `toLower :: Char -> Char` is available in module `Data.Char`.
- Your program has to be compilable as a stand-alone program; it must contain a module declaration and contain all required import statements.
- Of course, you may assume that `validName` is available even if you did not solve part (a).

Example run where system outputs and user inputs alternate in every line.

```
Enter a name:
Daisy76
Invalid, try again:
#fp-exam
Invalid, try again:
Paul Meyer
Lowercase version: paul meyer.
```

Solution:

```
module Main(main) where

import Data.Char

validName :: String -> Bool
validName "" = False
validName xs = all isChar xs

isChar :: Char -> Bool
isChar c
  | c >= 'a' && c <= 'z' = True
  | c `elem` ['A'..'Z'] = True
  | c == ' ' = True
  | otherwise = False

main = dialog "Enter a name:"

dialog m = do
  putStrLn m
  s <- getLine
  if validName s
    then putStrLn $ "Lowercase version: " ++ map toLower s ++ "."
    else dialog "Invalid, try again:"
```

Exercise 2: Programming with Abstract Datatypes and Functions 21

Consider an abstract datatype for *bags*, aka *multisets*, which are almost like sets, except that elements may occur more often than just once. The datatype has the following signature:

```
singleton :: Eq a => a -> Bag a       -- a bag with exactly one element
occurs :: Eq a => a -> Bag a -> Int   -- count how often an element occurs in a bag
union :: Bag a -> Bag a -> Bag a      -- compute union of two bags
```

Some typical equations that should be satisfied are, for example,

```
occurs x bag >= 0
occurs x (singleton x) = 1
occurs x (bag1 `union` bag2) = occurs x bag1 + occurs x bag2
x /= y ==> occurs x (singleton y `union` bag) = occurs x bag
```

(a) Implement a function `bag :: Eq a => [a] -> Bag a` that converts a non-empty list of elements into an equivalent bag of elements by using the functions of the abstract datatype. (4)

Example: `occurs x (bag [1..10])` should be `1` if `x` is between `1` and `10`; it should be `0`, otherwise.

Now, consider an implementation of bags by functions. More concretely, the functions count how often an element occurs in a bag.

```
data Bag a = Occ (a -> Int)

exampleBag :: Bag Integer
exampleBag = Occ (\x -> if x == 5 then 3 else 0)   -- the bag {5,5,5}
```

(b) Implement `singleton` for the given representation. (3)

(c) Implement `occurs` for the given representation. (3)

(d) Implement `union` for the given representation. (5)

(e) Consider the following three further operations on bags and indicate, whether they can be implemented using the above representation (just a yes/no-answer, no implementation required). Each correct answer is worth 2 points, each wrong answer results in -1 points, and giving no answer is worth 0 points. (6)

- create an empty bag ■ **Yes** □ No
- remove all occurrences of an element from a bag ■ **Yes** □ No
- for given `bag`, check whether `occurs (x :: Integer) bag /= 1` for all `x` □ Yes ■ **No**

Solution:

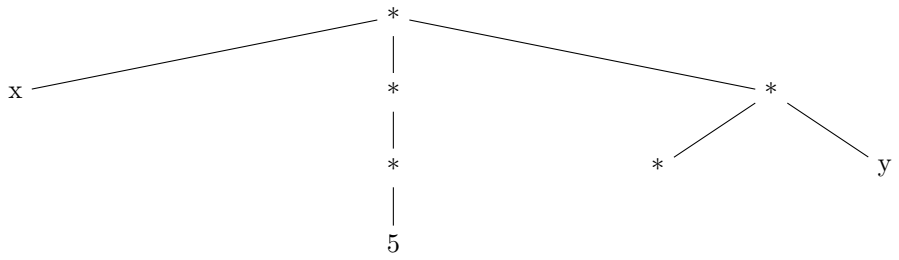
```
bag :: Eq a => [a] -> Bag a
bag xs = foldr1 union (map singleton xs)

data Bag a = Occ (a -> Int)

occurs :: Eq a => a -> Bag a -> Int
occurs x (Occ f) = f x

singleton :: Eq a => a -> Bag a
singleton x = Occ (\y -> if x == y then 1 else 0)

union :: Bag a -> Bag a -> Bag a
union (Occ f) (Occ g) = Occ (\x -> f x + g x)

-- Only the first two operations can be implemented
empty :: Bag a
empty = Occ (\ _ -> 0)

removeAll :: Eq a => a -> Bag a -> Bag a
removeAll x (Occ f) = Occ (\ y -> if y == x then 0 else f y)
```

Exercise 3: Programming with Lists and Trees 23

Consider the following datatype definition to encode expressions involving numbers, variables, and multiplications where each multiplication takes an arbitrary number of subexpressions, including 0.

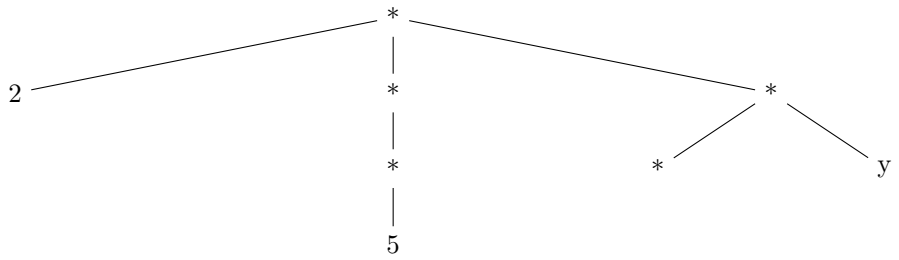
```
data Expr = Num Integer | Var String | Mul [Expr]
```

(a) Define a Haskell constant `example :: Expr` that corresponds to the following expression. (3)



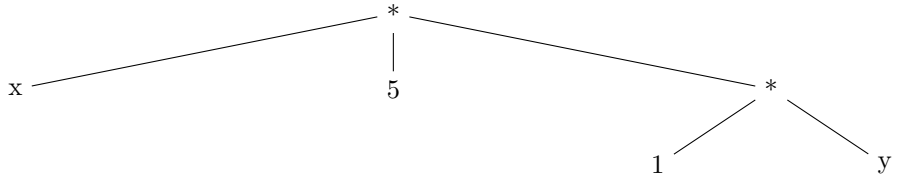
(b) Define a function `eval` to evaluate an expression for a given variable assignment. For instance, `eval (\v -> if v == "x" then 2 else 3) example` evaluates to 30, since 2 * 5 * 3 = 30. Provide both the type and the defining equations of `eval`. (6)

(c) Define a function `subst` where `subst v e1 e2` should deliver a new expression where every occurrence of variable `v` within `e2` is replaced by `e1`. For instance, `subst "x" (Num 2) example` represents the following expression: (6)



Provide both the type and the defining equations of `subst`.

(d) Define a function `simplify :: Expr -> Expr` where `simplify e` should be a simplified version of an expression `e`. In particular, all multiplications with 0 subexpressions should be simplified to the number 1, and each multiplication with exactly one subexpression *se* should be simplified to *se*. As an example, `simplify example` should deliver an expression corresponding to the the following tree: (8)



Solution:

```
data Expr = Num Integer | Var String | Mul [Expr]

example :: Expr
example = Mul [Var "x", Mul [Mul [Num 5]], Mul [Mul [], Var "y"]]

eval :: (String -> Integer) -> Expr -> Integer
eval alpha (Num i) = i
eval alpha (Var x) = alpha x
eval alpha (Mul es) = product (map (eval alpha) es)
-- alternatives:
-- eval alpha (Mul es) = foldr (*) 1 (map (eval alpha) es)
-- eval alpha (Mul es) = foldr (\ x s -> eval alpha x * s) 1 es


subst :: String -> Expr -> Expr -> Expr
subst x e (Num i) = Num i
subst x e (Var y) = if x == y then e else Var y
subst x e (Mul es) = Mul (map (subst x e) es)

simplify :: Expr -> Expr
simplify (Mul []) = Num 1
simplify (Mul [e]) = simplify e
simplify (Mul es) = Mul (map simplify es)
simplify e = e
```

Exercise 4: Evaluation and Types 20

In each multiple choice question, exactly one statement is correct. Marking the correct statement is worth 4 points, giving no answer counts 1 point, and marking multiple or a wrong statement results in 0 points.

Consider the following program.

```
foo x []                 = show x
foo x (y:ys) | x == y    = []
             | otherwise = foo x ys

bar x y z = x z (y z)
```

(a) What is the most general type of `foo`? (4)

- ■ `foo :: (Eq a, Show a) => a -> [a] -> String`
- ☐ `foo :: Ord a => a -> [a] -> [a]`
- ☐ `foo :: a -> [a] -> String`
- ☐ None of the above.

(b) What is the result of invoking `foo 2 [1,2,3,4]`? (4)

- ☐ `"2"`
- ☐ `2`
- ■ `[]`
- ☐ A run-time exception.

(c) What is the most general type of `bar`? (4)

- ☐ `bar :: [a] -> [b] -> [c] -> Int`
- ☐ `bar :: (Eq a, Eq b) => a -> b -> c -> c`
- ■ `bar :: (a -> b -> c) -> (a -> b) -> a -> c`
- ☐ None of the above.

(d) Consider the first two steps in the evaluation of the following expression w.r.t. Haskell's evaluation strategy. (4)

`bar (++) (\x -> x ++ x) [1,2,3]`

Choose the correct statement.

- ☐ First `x ++ x` is evaluated, then the defining equation of `bar` is applied.
- ■ **First the defining equation of `bar` is applied, then `[1,2,3] ++ [1,2,3]` is evaluated.**
- ☐ First the definition of `(++)` is expanded, then the defining equation of `bar` is applied.
- ☐ None of the above statements is correct.

(e) Assume we enter the expression `[y | x <- ["ab", "cde", "fgh"], y <- x]` in `ghci`. What will be the result? (4)

- ■ `"abcdefgh"`
- ☐ `["ab","cde","fgh"]`
- ☐ `["a","b","c","d","e","f","g","h"]`
- ☐ We get a compile error, since the expression is not allowed in Haskell.