

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	
Σ	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 `isName :: 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 upper case letters. Note that a function `toUpper :: 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 `isName` 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, retry:
#fp-exam
Invalid, retry:
Paul Meyer
Uppercase version: PAUL MEYER.
```

Solution:

```
module Main(main) where

import Data.Char

isName :: String -> Bool
isName "" = False
isName 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 isName s
    then putStrLn $ "Uppercase version: " ++ map toUpper s ++ "."
    else dialog "Invalid, retry:"
```

Exercise 2: Programming with Abstract Datatypes and Functions

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
count     :: Eq a => a -> Bag a -> Int -- count how often an element is 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,

```
count x bag >= 0
count x (singleton x) = 1
count x (bag1 `union` bag2) = count x bag1 + count x bag2
x /= y ==> count x (singleton y `union` bag) = count 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: `count 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 is contained in a bag.

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

```
exampleBag :: Bag Integer
```

```
exampleBag = Count (\x -> if x == 10 then 4 else 0)  -- the bag {10,10,10,10}
```

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

(c) Implement `count` 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)

- remove a single occurrence of an element from a bag ■ Yes No
- remove all occurrences of an element from a bag ■ Yes No
- double the number of occurrences of each element in a bag ■ Yes No

Solution:

```
bag :: Eq a => [a] -> Bag a
```

```
bag (x:xs) = foldr1 union (map singleton xs)
```

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

```
count :: Eq a => a -> Bag a -> Int
```

```
count x (Count f) = f x
```

```
singleton :: Eq a => a -> Bag a
```

```
singleton x = Count (\y -> if x == y then 1 else 0)
```

```
union :: Bag a -> Bag a -> Bag a
```

```
union (Count f) (Count g) = Count (\x -> f x + g x)
```

```
-- All three operations can be implemented
```

```
removeOne :: Eq a => a -> Bag a -> Bag a
```

```
removeOne x (Count f) = Count (\y -> if y == x then max 0 (f x - 1) else f y)
```

```
removeAll :: Eq a => a -> Bag a -> Bag a
```

```
removeAll x (Count f) = Count (\y -> if y == x then 0 else f y)
```

```
double :: Bag a -> Bag a
```

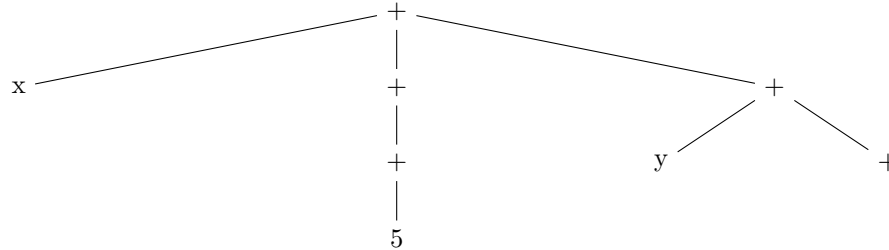
```
double (Count f) = Count ((*2) . f)
```

Exercise 3: Programming with Lists and Trees

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

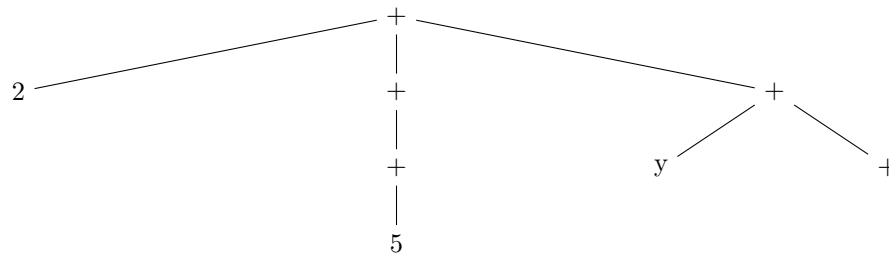
```
data Expr = Num Integer | Var String | Add [Expr]
```

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



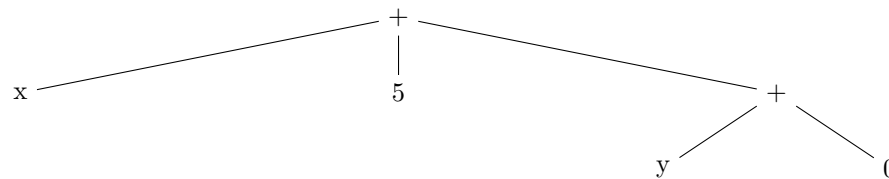
- (b) Define a function `eval` to evaluate an expression for a given variable assignment. For instance, `eval (\v -> if v == "x" then 2 else 3) example` evaluates to 10. Provide both the type and the defining equations of `eval`. (6)

- (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 additions with 0 subexpressions should be simplified to the number 0, and each addition 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 | Add [Expr]

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

eval :: (String -> Integer) -> Expr -> Integer
eval alpha (Num i) = i
eval alpha (Var x) = alpha x
eval alpha (Add es) = sum (map (eval alpha) es)
-- alternatives:
-- eval alpha (Add es) = foldr (+) 0 (map (eval alpha) es)
-- eval alpha (Add es) = foldr (\ x s -> eval alpha x + s) 0 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 (Add es) = Add (map (subst x e) es)

simplify :: Expr -> Expr
simplify (Add []) = Num 0
simplify (Add [e]) = simplify e
simplify (Add es) = Add (map simplify es)
simplify e = e
```

Exercise 4: Evaluation and Types

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 `head (foo 2 [1,2,3,4])`? (4)
- `[1,2,3,4]`
 - `"[1,2,3,4]"`
 - A run-time exception.**
 - `"2"`
- (c) What is the most general type of `bar`? (4)
- `bar :: [a] -> [b] -> [c] -> Int`
 - `bar :: (a -> b -> c) -> (a -> b) -> a -> c`
 - `bar :: (Eq a, Eq b) => a -> b -> c -> 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 definition of `(++)` is expanded, 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.**
  - None of the above statements is correct.
- (e) Assume we enter the expression `[m | n <- ["01", "234", "56"], m <- n]` in `ghci`. What will be the result? (4)
- `"0123456"`
  - `[0,1,2,3,4,5,6]`
  - `["0","1","2","3","4","5","6"]`
  - We get a compile error, since the expression is not allowed in Haskell.