Lastname: _____

Firstname: _____

Matriculation Number: _____

Exercise	Points	Score
Modules and I/O	20	
Programming with Lists and Trees	30	
Programming with Abstract Datatypes and Functions	20	
Evaluation and Types	20	
$\sum$	90	

- You have 90 minutes to solve the exercises.
- The exam consists of 4 exercises, for a total of 90 points (so there is 1 point per minute).
- 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: Modules and I/O 20

Consider the following program. It asks the user for several numbers, which are then sorted and printed.

```
1  module MyProgram(main) where
2
3  import Data.List(sort)
4
5  main :: IO ()
6  main = do
7    putStrLn "Welcome, please enter numbers to be sorted; end with the word \"quit\""
8    run
9
10 run :: [Integer] -> IO ()
11 run xs = do
12    x <- getLine
13    if x == "quit"
14       then finalize xs
15       else run ((read x :: Integer) ++ xs)
16
17 finalize :: [Integer] -> IO ()
18 finalize xs = putStrLn (sort xs)
```

This program contains four mistakes that prevent it from being compilable by `ghc`.

- Identify these mistakes by providing line numbers,
- briefly explain the problem of each mistake, and
- explain how to correct the mistakes.

Note that all four errors are independent of one another.

(a) Mistake #1 (5)

Solution: Line 1, compilation with `ghc` requires module name `Main`, so `MyProgram` must be renamed to `Main`.

(b) Mistake #2 (5)

Solution: Line 8, the function `run` needs a list as argument, i.e., it must be `run []`.

(c) Mistake #3 (5)

Solution: Line 15, `(++)` appends two lists, but `read x` is an `Integer`; so one has to replace `++` by `:` in this line.

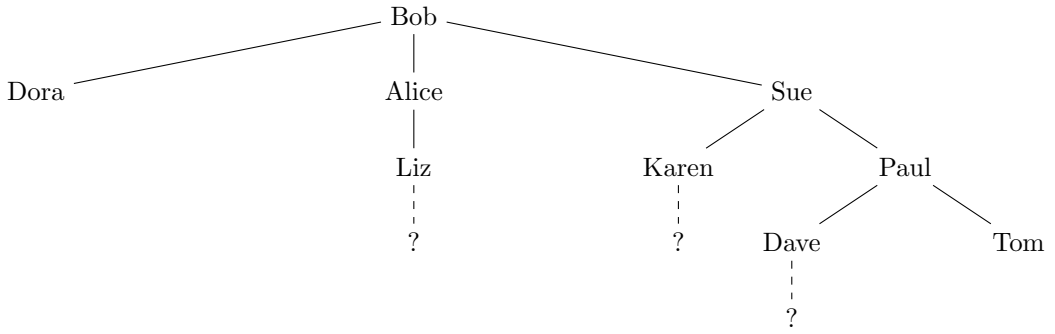
(d) Mistake #4 (5)

Solution: Line 18, the result of `sort` is a list of integers, put `putStrLn` expects a `String`; solution: replace `sort xs` by `show (sort xs)`.

Exercise 2: Programming with Lists and Trees 30

Suppose we have a collection of persons such that for each person we know his or her name and for some persons we also know the list of their children.

For example the following tree represents several persons, where the solid edges represent the is-child-of-relation, and a dashed edge to a question mark indicates that we do not know the children of that person.



For instance, Bob has three children: Dora, Alice and Sue; Dora and Tom have no children; and for Liz, Karen and Dave it is not known how many children they have.

We model the trees by the following datatype definition:

```
data Tree = Node String (Maybe [Tree])
```

For the upcoming tasks you may use arbitrary Prelude functions. In particular `concat :: [[a]] -> [a]`, `map`, and list comprehensions might be useful. You may use the functions defined in task (b) for later tasks, even if you did not solve task (b).

(a) Define constants `tom, karen, paul, sue :: Tree` that correspond to the subtrees of the figure having roots Tom, Karen, Paul and Sue, respectively. (4)

(b) Define two functions `person :: Tree -> String` and `children :: Tree -> [Tree]`. The former returns the name at the root of a tree and the latter returns the direct subtrees. For instance, `person sue = "Sue"`; the expression `children sue` should evaluate to the same list as `[karen, paul]`; and both `children tom` and `children karen` result in the empty list. (4)

(c) Define a function `grandChildren :: Tree -> [String]` which computes a list of all known grandchildren of the person at the root of the tree. (6)

For instance, `grandChildren sue` should result in `["Dave", "Tom"]` since these are the only known grandchildren of Sue.

(d) Define a function `names :: Tree -> [String]` that computes a list of all names that occur within a tree. For instance, `names sue` should be a list which contains the names of Sue, Karen, Paul, Dave and Tom (in any order); and `names` applied on the example tree above (with Bob as root) should result in a list of all names that occur the figure. (7)

(e) Define a function `insertChildren :: String -> [Tree] -> Tree -> Tree` with the following behavior. `insertChildren name ch t` computes a tree which is similar to `t` except that: (9)

- If `name` does not occur in `t`, then `t` is returned.
- If `name` occurs in `t`, such that the children of `name` are unknown in `t`, then in the resulting tree `name` will have children `ch`.
- If `name` already has a known list of children in `t`, an error should be raised.

For instance, `insertChildren "Dave" [] sue` will successfully replace the unknown children of Dave by the empty list of children, `insertChildren "Marc" [] sue` will be same tree as `sue`, and the invocation of `insertChildren "Tom" [] sue` will result in an error.

Solution:

```
data Tree = Node String (Maybe [Tree])

tom, dave, karen, paul, sue :: Tree
tom = Node "Tom" (Just [])
dave = Node "Dave" Nothing
karen = Node "Karen" Nothing
paul = Node "Paul" (Just [dave, tom])
sue = Node "Sue" (Just [karen, paul])

person :: Tree -> String
person (Node name _) = name

children :: Tree -> [Tree]
children (Node _ Nothing) = []
children (Node _ (Just cs)) = cs

grandChildren :: Tree -> [String]
grandChildren t = [ person gc | c <- children t, gc <- children c]
-- alternative:
-- grandChildren = map person . concat . map children . children


names :: Tree -> [String]
names t = person t : concat (map names (children t))

insertChildren :: String -> [Tree] -> Tree -> Tree
insertChildren name cs t@(Node n Nothing)
  | n == name = Node n (Just cs)
  | otherwise = t
insertChildren name cs (Node n (Just chs))
  | n == name = error "cannot replace existing children"
  | otherwise = Node n (Just (map (insertChildren name cs) chs))
```

Exercise 3: Programming with Abstract Datatypes and Functions 20

Consider an abstract datatype for sets, with the following signature:

```
empty :: Set a                              -- the empty set
member :: Eq a => a -> Set a -> Bool        -- check whether element is in a set
insert :: Eq a => a -> Set a -> Set a       -- add an element to a set
complement :: Eq a => Set a -> Set a        -- compute the complement of a set
```

Here, the usual laws of mathematical sets should be satisfied, e.g.,

```
not (member x empty)
member x (insert x set)
x /= y ==> member x (insert y set) = member x set
member x (complement set) <==> not (member x set)
```

(a) Implement a function `setOfList :: Eq a => [a] -> Set a`. It converts a list of elements into an equivalent set of elements by using the functions of the abstract datatype. (4)

Example: `member x (setOfList [1,9,3])` should be satisfied if and only if `x` is one of the numbers 1, 9 or 3.

Next we consider an implementation of sets by functions. To be more precise, the functions encode whether an element is contained in a set or not.

```
data Set a = Fun (a -> Bool)

exampleSet :: Set Integer
exampleSet = Fun (<= 10)   -- the set of all integers below 10
```

(b) Implement `empty`. (3)

(c) Implement `member`. (3)

(d) Implement `complement` and `insert`. (6)

(e) Consider four further set operations: intersection of two sets; union of two sets; testing whether a set is empty; and removal of an element from a set. At least one of these set operations cannot be implemented with our representation. Which one? And why? (4)

Solution:

```
setOfList :: Eq a => [a] -> Set a
setOfList = foldr insert empty

data Set a = Fun (a -> Bool)

empty :: Set a
empty = Fun (\x -> False)

member :: Eq a => a -> Set a -> Bool
member x (Fun f) = f x

insert :: Eq a => a -> Set a -> Set a
insert x s@(Fun f) = if f x then s else Fun (\y -> x == y || f y)

complement :: Eq a => Set a -> Set a
complement (Fun f) = Fun (\x -> not (f x))

-- isEmpty :: Eq a => Set a -> Bool cannot be implemented,
-- since the standard implementation
-- isEmpty (Fun f) = f == (\x -> False)
-- is not permitted, as functions are not comparable
```

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 [] = "{}"
foo xs = "{" ++ foldr (\ x s -> show x ++ "," ++ s) "}" xs

bar (_ : []) [] = 1
bar (x : y) _  = x
```

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

- ☐ `foo :: [String] -> String`
- ☐ `foo :: Show a => [a] -> [a]`
- ■ `foo :: Show a => [a] -> String`
- ☐ `foo :: [a] -> String`

(b) What is the result of invoking `foo $ take 7 [1..]`? (4)

- ☐ `"{1,2,3,"`
- ■ `"{1,2,3,4,5,6,7,}"`
- ☐ `"{1,2,3,4,5,6,7}"`
- ☐ the first character `'{'` is displayed and then the program does not terminate

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

- ☐ `bar :: [a] -> [b] -> Int`
- ■ `bar :: (Num a) => [a] -> [b] -> a`
- ☐ `bar :: (Num c) => [a] -> [b] -> c`
- ☐ `bar :: (Num a, Num c) => [a] -> [b] -> c`

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

`bar (2 + 3 : [4] ++ [5]) ([6] ++ [7])`

Choose the correct statement.

- ☐ First `2 + 3` is evaluated, then `[4] ++ [5]` is evaluated, and finally the second `bar`-equation is applied.
- ☐ First `2 + 3` is evaluated, then `[4] ++ [5]`, then `[6] ++ [7]` and finally the second `bar`-equation is applied.
- ■ **First `[4] ++ [5]` is evaluated, then the second `bar`-equation is applied, and finally `2 + 3` is evaluated.**
- ☐ None of the above statements is correct.

(e) Assume we enter the expression `[(x, y) | x <- ["ab","cde","fgh"], y <- [1..3], y < length x]` in `ghci`. What will be the result? (4)

- ☐ `[("ab",1),("cde",1),("fgh",1),("cde",2),("fgh",2)]`
- ■ `[("ab",1),("cde",1),("cde",2),("fgh",1),("fgh",2)]`
- ☐ `[("a",1),("c",1),("cd",2),("f",1),("fg",2)]`
- ☐ We get a compile error, since the expression is not allowed in Haskell.