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 qualified 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 (x : xs)
16
17 finalize :: [Integer] -> IO ()
18 finalize xs = putStrLn (show (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 7, the double-quotes in the quoted word `"quit"` need to be escaped, i.e., it must be `\"quit\"`.

(c) Mistake #3 (5)

Solution: Line 15, the result of getLine is a `String`, but `xs :: [Integer]`, so `x : xs` is not type correct; the solution is to replace `x : xs` by `read x : xs`.

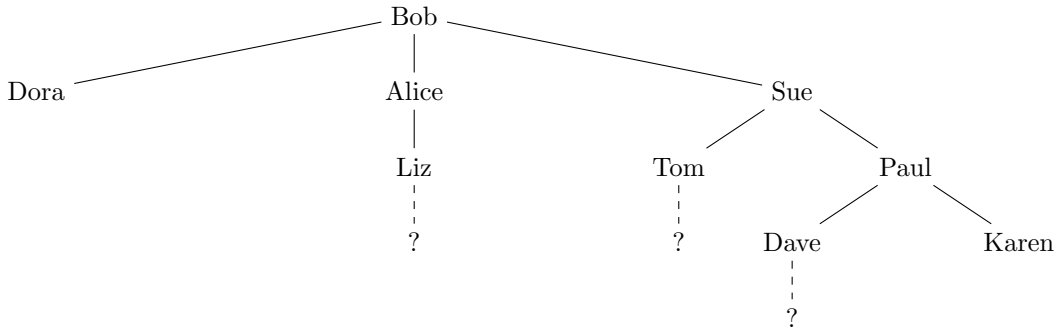
(d) Mistake #4 (5)

Solution: Line 3 or 18, the qualified import demands that `sort` is used with qualifier; either drop `qualified` in line 3, or use `Data.List.sort` in line 18.

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 Karen have no children; and for Liz, Tom 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 `[tom, 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", "Karen"]` since these are the only known grandchildren of Sue.

(d) Define a function `descendants :: Tree -> [String]` which computes a list of all known descendant of a specific person. For instance, `descendants sue` should result in a list which contains the names of Tom, Paul, Dave and Karen (in any order); and the descendants of the example tree above (with Bob as root) should be a list of all names of the figure except for `"Bob"` himself. (8)

(e) Define a function `unknownGC :: Tree -> Bool` which should return True if and only if the person at the root of the input tree might have unknown grandchildren. For instance, `unknownGC sue` evaluates to `True`, since we do not know the children of Tom. By contrast, `unknownGC` invoked on Bob should deliver `False`, since his grandchildren are exactly Liz, Tom and Paul. (8)

Solution:

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

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

descendants :: Tree -> [String]
descendants t = map person cs ++ [ d | c <- cs, d <- descendants c]
  where cs = children t

unknownChild :: Tree -> Bool
unknownChild (Node _ Nothing) = True
unknownChild _ = False

unknownGC :: Tree -> Bool
unknownGC t = unknownChild t || any unknownChild (children t)
```

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 `listToSet :: 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 (listToSet [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; computing the cardinality (i.e., size) of a set; 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:

```
listToSet :: Eq a => [a] -> Set a
listToSet = 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))

-- the cardinality cannot be computed, since there is no way to
-- iterate over the elements of a set; one cannot even check
-- whether a set is empty (since comparisons of functions are
-- not computable in general), so in particular we cannot
-- compute the cardinality and then compare this number to 0
```

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

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

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

(b) What is the result of invoking `take 7 $ foo [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 :: (Eq a, Num a) => [a] -> [b] -> a`
- ☐ `bar :: (Eq a, Num c) => [a] -> [b] -> c`
- ☐ `bar :: (Eq a, 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 `[6] ++ [7]` is evaluated, then the second `bar`-equation is applied, and finally `2 + 3` is evaluated.
- ■ **First `2 + 3` is evaluated, and then 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.
- ☐ None of the above statements is correct.

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

- ☐ `[("ab",1),("cde",1),("cde",2),("fgh",1),("fgh",2)]`
- ☐ `[("ab",1),("cde",1),("fgh",1),("cde",2),("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.**