Lastname: _____

Firstname: _____

Matriculation Number: _____

Exercise	Points	Score
Program Analysis Including Modules and I/O	20	
Programming with Lists	32	
Datatypes and Higher-Order Functions	26	
Evaluation and Types	12	
$\sum$	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.
- Textual answers can be formulated in either English or German.

Exercise 1: Program Analysis Including Modules and I/O 20

Consider the following program.

```
1  import qualified Text.Read
2
3  data Expr = Div Expr Expr | Num Double deriving Read
4
5  eval :: Expr -> Double
6  eval (Num x) = x
7  eval e@(Div e1 e2) = let
8      x1 = eval e1
9      x2 = eval e2
10   in if x2 /= 0
11        then x1 / x2
12        else error $ "div-by-0 error in expression " ++ show e
13
14 main :: IO ()
15 main = do
16   putStrLn "enter expression:"
17   s <- getLine
18   case readEither s of
19     Left errorMessage -> do
20       putStrLn $ "error in input: " ++ errorMessage
21       main
22     Just expr -> do
23       result <- eval expr
24       putStrLn $ "the result is " ++ show result
```

This program contains four mistakes that cause compilation errors.

- 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 mistakes are independent of one another.
Further note that `readEither :: Read a => String -> Either String a` is exported by module `Text.Read` where `data Either a b = Left a | Right b`.

(a) Mistake #1 (5)

(b) Mistake #2 (5)

(c) Mistake #3 (5)

(d) Mistake #4 (5)

Exercise 2: Programming with Lists 32

A word w is a palindrome, if reading w from right-to-left is the same as reading w from left-to-right. For instance, the words "hannah", "refer", and "a" are palindromes, whereas "paul" and "valid" are not.

A palindrome can be generalized to arbitrary lists, e.g., also $[1, 2, 7, 2, 1]$ is a palindrome, whereas $[1, 8, 9, 1]$ is not.

For the upcoming programming tasks except task (b) you may use arbitrary Prelude functions, e.g., functions such as `map`, `length`, `take`, `drop`, `words`, `unwords`, `[i .. j]`, and so on.

(a) Define a Haskell-function `palindrome` that determines whether a given list is a palindrome. Also specify a type for `palindrome` that should be as general as possible. (4)

Examples:

- `palindrome "kayak" && palindrome "" && palindrome [1,2,7,2,1]` should evaluate to `True`
- `palindrome "paul" || palindrome [1,2]` should evaluate to `False`

(b) Define a function `partition :: (a -> Bool) -> [a] -> ([a], [a])` with the following behavior. Whenever `partition p xs = (ys, zs)`, then `ys` contains those elements of `xs` that satisfy predicate `p`, and `zs` contains the other elements of `xs`. (8)
For example, `partition (> 5) [4,10,7,3,2] == ([10,7], [4,3,2])`.

For task (b) it is not allowed to use any predefined functions on lists, except for the list constructors!

(c) Define a Haskell-function `specialSentence :: String -> Bool` that determines whether a sentence is special, i.e., whether at least half of the words in the sentence are palindromes. (8)

- the input is a sentence that is represented as a Haskell `String`, and the words within the sentence are separated by blanks
- each occurrence of a word is counted separately, i.e., `"a bob is a fast vehicle"` is a sentence that consists of 6 words, and it is special as it contains (at least) 3 palindromes `"a"`, `"bob"` and `"a"`
- `"malayalam is a nice language"` is not a special sentence, as it only contains 2 palindromes but consists of 5 words

Remark: You may of course use `palindrome` and `partition`, even if you did not solve those parts.

(d) Define a Haskell-function `subPalindromes` such that `subPalindromes xs` is a list of all non-trivial palindromes that occur as sublists of `xs`. (12)

- a non-trivial palindrome has a length of at least 4
- a sublist of `xs` is obtained by dropping arbitrary many elements at the front and at the rear of `xs`

Example: `subPalindromes "hello to otto and hannah"` should evaluate to a list that contains exactly the strings `"to ot"`, `" otto "`, `"otto"`, `"hannah"` and `"anna"` (in any order).

Hint: list-comprehensions might be useful.

Exercise 3: Datatypes and Higher-Order Functions 26

Consider the following program.

```
import Data.List(nub, sort)
-- nub :: Eq a => [a] -> [a]
-- "nub" removes all duplicates from the given list
-- sort :: Ord a => [a] -> [a]
-- sum :: Num a => [a] -> a
-- "sum" computes the sum of all elements in a list of numbers
-- map :: (a -> b) -> [a] -> [b]
data Tree a = Tree a [Tree a]

node (Tree x _) = x
subtrees (Tree _ ts) = ts
mapTree f (Tree x ts) = Tree (f x) (map (mapTree f) ts)
foldTree f (Tree x ts) = f x (map (foldTree f) ts)
```

(a) Write down the most general types of `node`, `subtrees` and `mapTree`. (4)

(b) Assume we want to define a function `sumTrees :: [Tree Int] -> Int` that computes the sum of all nodes of a given list of integer-trees. (4)

Example: `sumTrees [Tree 1 [], Tree 3 [Tree 4 [], Tree 3 []]] = 1 + 3 + 4 + 3 = 11`

Choose a suitable implementation (4 points for the correct solution, 1 point for making no choice, 0 points for marking a wrong solution)

- ☐ `sumTrees = sum . map node`
- ☐ `sumTrees = sum . map (mapTree id)`
- ☐ `sumTrees = sum . subtrees`
- ☐ `sumTrees = sum . map (foldTree (\ x xs -> x + sum xs))`

(c) Assume we want to write a function `totalSum :: Tree Int -> Tree Int`, replacing each node in the given integer-tree by the sum of all integers in the subtree starting at the node. (6)

Example: `totalSum (Tree 1 [Tree 2 [], Tree 3 [Tree 4 [], Tree 5 []]]) =`
`Tree 15 [Tree 2 [],Tree 12 [Tree 4 [],Tree 5 []]]`

Further assume our implementation uses the following structure:

```
totalSum = foldTree undefined
```

Replace `undefined` by a suitable λ-expression or argue why `totalSum` cannot be implemented via `foldTree`.

(d) Assume we want to define a function `set :: Ord a => Tree a -> [a]` that computes, given a tree, the sorted list of all nodes in the tree without duplicates (you might also say, a set-representation of the tree content). Below are three different attempts to implement `set`: (12)

```
set1 = sort . nub . mapTree id
set2 = nub . sort . foldTree (\x ts -> x : concat ts)
set3 = sort . nub . foldTree (\x -> concat)
```

For each of the functions `set1`, `set2` and `set3`, indicate whether it is a correct implementation of `set` or not; and for the incorrect ones, give a brief description of the problem.

Exercise 4: Evaluation and Types 12

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

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

- ☐ `[Int] -> Int`
- ☐ `[Int] -> [Int]`
- ☐ `Num a => [a] -> [a]`
- ☐ `[a] -> [a]`

(b) What is the result of invoking `foo [1,2,3,4,5]`? (4)

- ☐ `[1,3,6,10,15]`
- ☐ `[0,1,3,6,10]`
- ☐ `15`
- ☐ none of the above

(c) Assume that we evaluate `foo xs` for some finite list `xs :: [Int]`. (4)
Which of the following statements is correct?

- ☐ The memory consumption is constant for both innermost and lazy evaluation.
- ☐ The memory consumption is constant when using lazy evaluation, but grows linearly in the length of `xs` when using innermost evaluation.
- ☐ The memory consumption is unbounded when using innermost evaluation, since the function call leads to an infinite computation.
- ☐ none of the above