SOLUTION

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 Text.Read(readEither)
2
3  data Expr = Div Expr Expr | Num Double deriving Read
4
5  eval :: Expr -> IO Double
6  eval (Num x) = return x
7  eval e@(Div e1 e2) = do
8    x1 <- eval e1
9    x2 <- eval e2
10   if x2 /= 0
11     then return (x1 / x2)
12     else let message = "div-by-0 error in expression " ++ show e
13          in putStrLn message
14
15 main :: IO ()
16 main = do
17   putStrLn "enter expression:"
18   s <- getLine
19   case readMaybe s of
20     Nothing -> main
21     Just e -> do
22       let result = eval e
23       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 `readMaybe :: Read a => String -> Maybe a` is exported by module `Text.Read`.

(a) Mistake #1 (5)

Solution: Line 12: Mistake: there is no `Show`-instance for `Expr`, but this would be required for expression `show e`. Solution: add `Show` to `derive` in Line 3.

(b) Mistake #2 (5)

Solution: The return type in Line 13 is wrong, since `putStrLn message :: IO ()`. Solution: replace `putStrLn` by `error`

(c) Mistake #3 (5)

Solution: Line 1: `readMaybe` is used in Line 19, but only `readEither` is imported via Line 1. So the import has to be changed to `readMaybe`.

(d) Mistake #4 (5)

Solution: Line 22: since `eval e :: IO Double`, we need to write `result <- eval e` instead of using `let`

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 `magicSentence :: String -> Bool` that determines whether a sentence is magic, 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 magic as it contains (at least) 3 palindromes `"a"`, `"bob"` and `"a"`
- `"malayalam is a nice language"` is not a magic 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 3
- 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"`, `"o o"`, `" otto "`, `"otto"`, `"hannah"` and `"anna"` (in any order).

Hint: list-comprehensions might be useful.

Solution:

```
palindrome :: Eq a => [a] -> Bool
palindrome xs = xs == reverse xs

partition :: (a -> Bool) -> [a] -> ([a], [a])
partition p [] = ([], [])
partition p (x : xs)
  | p x = (x : ys, zs)
  | otherwise = (ys, x : zs)
  where (ys, zs) = partition p xs

magicSentence :: String -> Bool
magicSentence s = let
  ws = words s
  (ps, nps) = partition palindrome ws
  in length ps >= length nps

subPalindromes :: Eq a => [a] -> [[a]]
subPalindromes xs = let n = length xs in
   [ ys | i <- [0..n-3],
      let suff = drop i xs,
      j <- [3 .. n - i],
      let ys = take j suff,
      palindrome ys]
```

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)

Solution:

```
node :: Tree a -> a
subtrees :: Tree a -> [Tree a]
mapTree :: (a -> b) -> Tree a -> Tree b
```

(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 3 [], Tree 2 [Tree 1 [], Tree 4 []]] = 3 + 2 + 1 + 4 = 10`

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

(c) Assume we want to write a function `cumulativeSum :: 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:
```
cumulativeSum (Tree 1 [Tree 1 [], Tree 1 [Tree 1 [], Tree 1 []]]) =
                                        Tree 5 [Tree 1 [],Tree 3 [Tree 1 [],Tree 1 []]]
```

Further assume our implementation uses the following structure:

```
cumulativeSum = foldTree undefined
```

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

Solution:
```
cumulativeSum = foldTree (\x ts -> Tree (x + sum (map node ts)) ts)
```

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

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.

Solution:

- `set1` compiles, but does not compute the correct result. The reason is that the expression `foldTree (\x -> concat) t` always results in the empty list `[]`, independent of the specific tree `t` (as long as it is defined).
- `set2` is correct.
- `set3` does not compile, since `nub` expects a list, but `mapTree id t` results in a tree.

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 + y) : bar (x + y) ys
```

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

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

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