Lastname: ______________________________

Firstname: ______________________________

Matriculation Number: ______________________________

Exercise	Points	Score
Program Analysis	12	
Programming	14	
Type-Classes and Modules	10	
Recursion and Efficiency	9	
$\sum$	45	

- You have 90 minutes time to solve the exercises.
- The exam consists of 4 exercises, for a total of 45 points (so there is 1 point per 2 minutes).
- 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: Program Analysis 12

Consider the following Haskell code:

```
sub_lists [] = []
sub_lists (ys @ (x : xs)) = ys : sub_lists xs

f = foldr (:)
```

In each multiple choice question, exactly one statement is correct. Marking the correct statement is worth 3 points, giving no answer counts 1 point, and marking multiple or a wrong statement results in 0 points.

(a) The evaluation of `sub_lists "abc"` results in: (3)

- ☐ `["abc","bc","c",""]`
- ■ `["abc","bc","c"]`
- ☐ `["","c","bc","abc"]`
- ☐ `["c","bc","abc"]`
- ☐ none of the above

(b) The evaluation of `map take 4 . take 2 . sub_lists $ ['e'..]` results in: (3)

- ☐ `["ef", "fg", "gh", "hi"]`
- ☐ `["efgh", "fghi"]`
- ☐ non-termination
- ■ **a type-error**
- ☐ none of the above

(c) Write down the most general type of `f` (3)

Solution:
`f :: [a] -> [a] -> [a]`

(d) An equivalent definition of `f` is: (3)

- ☐ `(++)`
- ☐ `\ xs ys -> reverse xs ++ ys`
- ☐ `\ xs ys -> xs ++ reverse ys`
- ■ `\ xs ys -> ys ++ xs`
- ☐ `reverse`

Exercise 2: Programming 14

Consider a list where persons are stored in combination with their favorite function on numbers.

```
favorites :: [(String, Double -> Double)]
favorites = [("INA", id), ("elton", error "notle"), ("Mini", min), ("norbert", negate), ...]
```

(a) Does the definition of `favorites` compile? If not, which pair(s) must be removed so that it compiles. (2)

Solution:
It does not compile, since `min` is not of type `Double -> Double`. So the pair of Mini must be removed.

(b) Write a function `evaluate :: String -> Double -> Double` which takes a name of a person and a number and evaluates the stored function of that person from `favorites`. You can assume that for the provided name, there will be exactly one pair in the list. (3)

For example, `evaluate "norbert" 7.0 = -7.0`.

Solution:

```
evaluate name = head [ f | (n,f) <- favorites, n == name]
```

(c) Implement a function `to_lower :: Char -> Char` which takes a character c and returns either the lower-case version of c if $c \in \{$'A', ..., 'Z'$\}$, or c itself otherwise. Of course, here it is not allowed to use the predefined Haskell function `toLower`. (4)

Solution:

```
to_lower c
  | c >= 'A' && c <= 'Z' = toEnum (fromEnum c + fromEnum 'a' - fromEnum 'A')
  | otherwise = c

-- alternative solution
to_lower 'A' = 'a'
...
to_lower 'Z' = 'z'
to_lower c   = c
```

(d) Write a function `sort_ignore_case :: [(String,a)] -> [(String,a)]` which sorts a list of pairs by their first component, but where the upper-case and lower-case letters in the strings are identified. For instance, `sort_ignore_case favorites = [("elton", ...), ("INA", ...), ...]` although `'I' < 'e'`. (5)

- You can use any function you want, in particular `sortBy`, `compare` and `to_lower` might be helpful.
- The sorted list must contain the same pairs as the input list.

Solution:

```
sort_ignore_case =
  sortBy (\ (s1, _) (s2, _) -> compare (map to_lower s1) (map to_lower s2))
```

Exercise 3: Type-Classes and Modules 10

Consider the following Haskell module for complex numbers which are represented by pairs consisting of the angle and the radius of the complex number.

```
type Angle = Double
type Radius = Double
data Complex = Polar Angle Radius deriving Eq

standardize_angle :: Angle -> Angle
standardize_angle phi
  | phi > 2 * pi = standardize_angle (phi - 2 * pi)
  | phi < 0      = standardize_angle (phi + 2 * pi)
  | otherwise    = phi

create_polar :: Angle -> Radius -> Complex
create_polar phi r = Polar (standardize_angle phi) r
```

(a) Two complex numbers (φ_1, r_1) and (φ_2, r_2) are equal if and only if $r_1 = r_2 = 0$ or both $r_1 = r_2$ and φ_1 and φ_2 represent the same angle (modulo 2π). (4)

Does the **deriving** Eq implementation of equality on type Complex correctly implement equality on complex numbers if one assumes that all complex numbers have been constructed via create_polar?

Provide a yes/no answer and if the answer is "no", also provide a corrected definition of create_polar such that **deriving** Eq is a correct implementation for equality.

> **Solution:** The answer is no, since `create_polar 0 0` and `create_polar (pi / 2) 0` are the same complex number, but `create_polar 0 0 == create_polar (pi / 2) 0` evaluates to `False`. A corrected implementation looks as follows:

```
create_polar phi r
 | r == 0 =    Polar 0 0
 | otherwise = Polar (standardize_angle phi) r
```

(b) Extend the program so that Complex becomes an instance of Show where a complex number (φ, r) should be represented by the string e^φi * r. (Here, e^ and i * are concrete strings!) (3)

> **Solution:**

```
instance Show Complex where
  show (Polar phi r) = "e^" ++ show phi ++ "i * " ++ show r
```

(c) Add a module definition for the complex numbers. (3)

- The name of the module should be Complex_Polar.
- Access should be given to the type Complex, to create_polar, and to the equality and show functions for complex numbers.
- Access should be forbidden to all other functions, and in particular to the constructor Polar.
- Provide all required Haskell keywords, i.e., if one copies your definition in front of the existing implementation, then the resulting code should compile.

> **Solution:**

```
module Complex_Polar(Complex, create_polar) where
```

Exercise 4: Recursion and Efficiency 9

Consider the following Haskell code:

```
fun x
  | x >= 3 = fun (x - 3) + 2 * fun (x - 2)
  | otherwise = 4
```

(a) Specify which of the following properties fun are satisfied. (3)

Each correct answer is worth one point, each wrong answer reduces one point. If the overall score of this part would be negative, then it is set to 0.

fun uses linear recursion.	☐ yes	■ no
fun uses guarded recursion.	☐ yes	■ no
fun uses nested recursion.	☐ yes	■ no

(b) The current definition of fun has exponential complexity. Provide an equivalent definition of fun that requires only linearly many recursive calls. (6)

- Of course, you may specify auxiliary functions.
- You only need to consider non-negative inputs.

Solution:

```
fun x
  | x <= 2 = 4
  | otherwise = fun_aux 2 4 4 4
  where
    fun_aux i fi2 fi1 fi
      | i == x = fi
      | otherwise = fun_aux (i+1) fi1 fi (fi2 + 2 * fi1)
```