

Lastname: _____

Firstname: _____

Matriculation Number: _____

Exercise	Points	Score
Program Analysis	12	
Programming	14	
Type-Classes and Modules	10	
Recursion and Efficiency	9	
Σ	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

Consider the following Haskell code:

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

```
function = 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 `part_lists [1,2,3]` results in: (3)
- `[[[], [3], [2,3], [1,2,3]]`
 - `[[3], [2,3], [1,2,3]]`
 - `[[1,2,3], [2,3], [3], []]`
 - `[[1,2,3], [2,3], [3]]`
 - none of the above
- (b) The evaluation of `map take 3 . take 2 . part_lists $ [4..]` results in: (3)
- `[[4,5,6], [5,6,7]]`
 - `[[4,5], [5,6], [6,7]]`
 - a type-error
 - non-termination
 - none of the above
- (c) Write down the most general type of `function` (3)

- (d) An equivalent definition of `function` is: (3)
- `reverse`
 - `(++)`
 - `\ xs ys -> ys ++ xs`
 - `\ xs ys -> reverse xs ++ ys`
 - `\ xs ys -> xs ++ reverse ys`

Exercise 2: Programming

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

```
fun_list :: [(String, Integer -> Integer)]
```

```
fun_list = [("nena", negate), ("Ida", id), ("MAX", max), ("egon", error "noge"), ...]
```

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

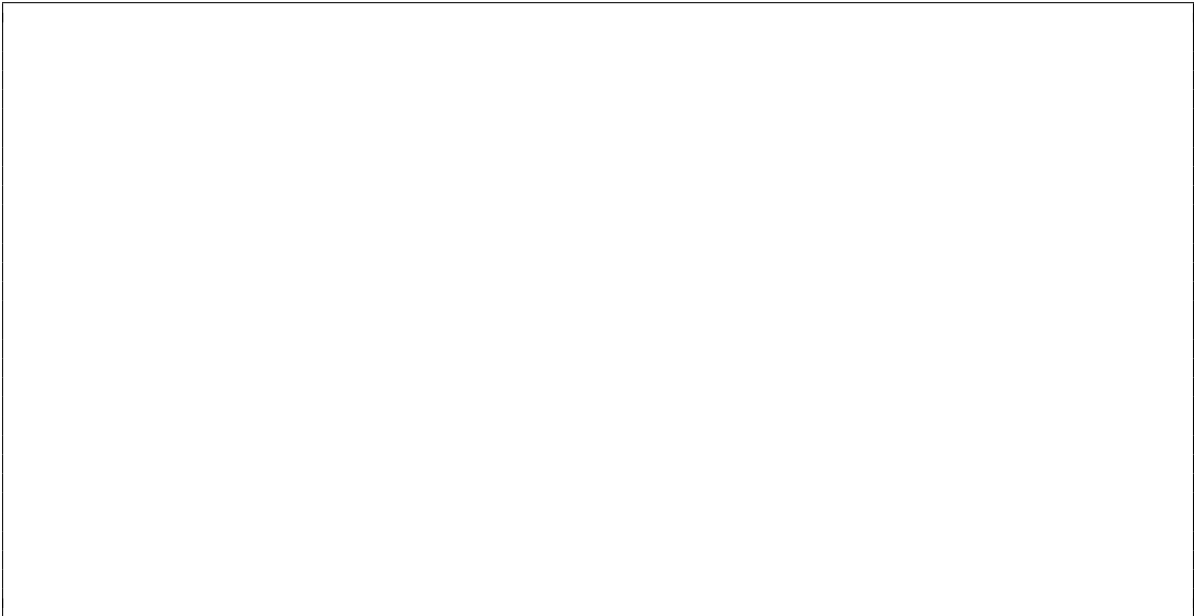
(2)

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

(3)

For example, `eval_fun "nena" 5 = -5`.

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



- (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 fun_list = [("egon", ...), ("Ida", ...), ...]` 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.



Exercise 3: Type-Classes and Modules

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

```

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

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

```

```

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

```

- (a) Two complex numbers (r_1, φ_1) and (r_2, φ_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.

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

- (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.

Exercise 4: Recursion and Efficiency

Consider the following Haskell code:

```
f x
| x >= 3 = f (x - 2) + 5 * f (x - 3)
| otherwise = 7
```

- (a) Specify which of the following properties **f** 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.

f uses nested recursion. yes no

f uses linear recursion. yes no

f uses guarded recursion. yes no

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

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