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. It chooses a random non-negative number and then asks the user to guess it.

```
1  module Main(main, finalize) where
2
3  import System.Random(randomIO)  -- randomIO :: IO Integer
4
5  main = do
6    num <- randomIO
7    putStrLn "Try to guess my number"
8    guessingGame (abs num) 1
9
10 guessingGame :: Integer -> Integer -> IO String
11 guessingGame num n = do
12   str <- getLine
13   x <- (read str :: Integer)
14   let reason = if x < num then "small" else "large"
15   if x == num then finish n
16   else do
17     putStrLn $ "Your guess was too " ++ reason ++ ". Try again"
18     guessingGame num n + 1
19
20 finish n = putStrLn $ "You guessed my number using " ++ show n ++ " tries"
```

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.

(a) Mistake #1 (5)

> **Solution:** Line 1, `finalize` cannot be exported as it is not defined in this program. Either remove `finalize` or rename it to `finish`.

(b) Mistake #2 (5)

Solution: Line 10, the type of `guessingGame` is wrong, it must be `Integer -> Integer -> IO ()`.

(c) Mistake #3 (5)

Solution: Line 13, `read` is not a monadic operation, hence `x <- read str` must be replaced by `let x = read str`

(d) Mistake #4 (5)

Solution: Line 18, the expression is parsed as `(guessingGame num n) + 1`, but it should have been `guessingGame num (n + 1)`, so parentheses or a `$` are required.

Exercise 2: Programming with Lists 32

Periodic functions can be represented by a list of initial values `vs = [v_1,...,v_k]` and a non-empty list of values `ws = [w_1,...,w_m]` that is repeated over and over again.

The function `periodic :: [a] -> [a] -> Int -> a` is then defined as follows: `periodic vs ws n` is the `n`-th element of the infinite list `[v_1,...,v_k,w_1,...,w_m,w_1,...,w_m,w_1,...,w_m, ...]` for every non-negative integer `n`. For example, if `f = periodic [4,2] [5,1,3]` then the results of evaluating `f` for arguments `0`, ..., `8` are shown in the following table:

f 0	f 1	f 2	f 3	f 4	f 5	f 6	f 7	f 8	...
4	2	5	1	3	5	1	3	5	...

Your task is to develop different (equivalent) implementations of `periodic`. You may freely use all Prelude functions. In particular, `take`, `drop`, `splitAt`, `(!!)`, `(++)`, `filter`, `map` and `lookup` might be useful.

(a) Define a function `infList` for the infinite list described above, i.e., `infList vs ws` should evaluate to `[v_1,...,v_k,w_1,...,w_m,w_1,...,w_m, ...]`, and provide the most general type of `infList`. Further define a function `periodicInf` as an implementation of `periodic` that is based on `infList`. (6)

(b) Define a function `periodicN` that implements `periodic` without constructing an infinite list and *without using any predefined functions on lists* (except for the list constructors). Evaluating `periodicN vs ws n` should require approximately `n` steps. (6)

(c) Define a function `periodicFast :: [a] -> [a] -> Int -> a` as an implementation of `periodic`. You should make use of the periodicity so that evaluating `periodicFast vs ws n` does not need many more than `length vs + length ws` steps, even if `n` is very large. (8)

Hint: Haskell contains functions `div` and `mod` to compute the quotient and the remainder of an integer division respectively.

(d) Assume that a function `g :: Int -> a` is periodic, i.e. there exists a finite list `vs` and a non-empty finite list `ws` such that `g = periodic vs ws`. Assume also that the list `vs ++ ws` is *distinct*, i.e., it contains no duplicates. (12)

Define a function `getLists :: Eq a => (Int -> a) -> ([a], [a])` such that `getLists g` reconstructs `vs` and `ws` from `g`. In particular `getLists (periodic vs ws) == (vs, ws)` should be satisfied whenever `vs` and `ws` are two finite lists such that `vs ++ ws` is distinct and `ws` is non-empty.

Remark: you can get half of the points for this part if you instead implement an easier function `getListSimple :: Eq a => (Int -> a) -> [a]`. Here we assume that the input is a periodic function `g = periodic [] ws` and only `ws` is computed via `getListSimple`. In particular, the property `getListSimple (periodic [] ws) == ws` should be satisfied whenever `ws` is a finite, non-empty and distinct list.

Solution:

```
infList :: [a] -> [a] -> [a]
infList vs ws = vs ++ infList ws ws
periodicInf vs ws n = infList vs ws !! n

periodicN (x : _) _ 0 = x
periodicN (_ : vs) ws n = periodicN vs ws (n - 1)
periodicN [] ws n = periodicN ws ws n

periodicFast vs ws n
  | n < lvs = vs !! n
  | otherwise = ws !! ((n - lvs) `mod` length ws)
  where lvs = length vs

getListSimple :: Eq a => (Int -> a) -> [a]
getListSimple g = g 0 : takeWhile (/= g 0) (map g [1..])

-- solution based on lookup / take / splitAt
getLists :: Eq a => (Int -> a) -> ([a], [a])
getLists g = search 1 where
  gis = map (\ i -> (f i, i)) [0..]
  search n = case lookup (g n) (take (n - 1) gis) of
     Just i -> let (vs, long) = splitAt i (map fst gis)
       in (vs, take (n - i) long)
     Nothing -> search (n + 1)

-- solution based on elem / take / span / filter
getLists :: Eq a => (Int -> a) -> ([a], [a])
getLists g = let
  xs = map g [0..]
  duplIndex = head (filter (\ i -> g i `elem` take i xs) [0..])
  vsWs = take duplIndex xs
 in span (/= g duplIndex) vsWs
```

Exercise 3: Datatypes and Higher-Order Functions 26

Consider the following program.

```
import Data.List(sort, sortOn)
-- sort   :: Ord a => [a] -> [a]
-- sortOn :: Ord b => (a -> b) -> [a] -> [a]
-- sortOn f xs provides a sorted list ys of xs,
--    such that f (ys !! (i - 1)) <= f (ys !! i) for all 1 <= i < length ys;
-- sort and sortOn are closely related: sort = sortOn id

import Data.Char(toUpper)
-- toUpper :: Char -> Char

type Name = [String] -- a name might be composed, e.g., John Paul van de Boes

data Employee = Empl
  Name
  Int      -- age
  Float    -- salary

nameOf (Empl name _ _) = name
mapEmp f g h (Empl name age salary) = Empl (f name) (g age) (h salary)
```

(a) Write down the most general types of `nameOf` and of `mapEmp`. (4)

> **Solution:**
> ```
> nameOf :: Employee -> Name
> mapEmp ::
>   (Name -> Name) ->
>   (Int -> Int) ->
>   (Float -> Float) ->
>   (Employee -> Employee)
> ```

(b) Assume we want to write a function `raiseSalary :: Employee -> Employee` where the new salary is computed by the formula (4)

$$\text{new-salary} = \text{old-salary} + \text{age} \times 10$$

Further assume our implementation uses the following structure.

```
raiseSalary = mapEmp undefined undefined undefined
```

Either replace each `undefined` by a suitable λ-expression, or argue why `raiseSalary` cannot be implemented via `mapEmp`.

> **Solution:** It cannot be implemented via `mapEmp`, since the new salary depends on the age and the old salary. However, in `mapEmp` the new salary is computed by a function that only gets the old salary as input.

(c) Assume we want to define a function `toUpperEmployees :: [Employee] -> [Employee]` that changes all names of all employees in a list so that they are written with uppercase letters. Choose a suitable implementation (4 points for the correct solution, 1 point for making no choice, 0 points for marking a wrong solution) (4)

- ☐ `toUpperEmployees = toUpper`
- ☐ `toUpperEmployees = map (mapEmp (map toUpper))`
- ■ `toUpperEmployees = map (mapEmp (map (map toUpper)) id id)`
- ☐ `toUpperEmployees = map (mapEmp (map toUpper) id id)`

(d) Assume we want to define a function `sortedUppercaseNames :: [Employee] -> [Name]` that returns a sorted list of the names of all employees in a list converted to uppercase. The sorting should also be done using the uppercase names. There are four different attempts to implement `sortedUppercaseNames` (`sun` for brevity). (14)

```
sun1 = map nameOf . sort . toUpperEmployees
sun2 = sort . toUpperEmployees . map nameOf
sun3 = map nameOf . sortOn nameOf . toUpperEmployees
sun4 = map nameOf . sortOn (nameOf . toUpperEmployees)
```

For each of the functions `sun1`, `sun2`, `sun3` and `sun4`, indicate whether they are correct implementations of `sortedUppercaseNames` or not; and for the incorrect ones, give a brief description of the problem.

Solution:

- `sun1` does not compile, since `Employee` does not instantiate `Ord`.
- `sun2` does not compile, since `map nameOf` produces a list of names, but `toUpperEmployees` expects a list of employees as input.
- `sun3` is correct.
- `sun4` does not compile for several reasons. For instance, `toUpperEmployees` requires a *list* of employees as input, but `sortOn` will invoke this function on singleton employees; or the result of `toUpperEmployees`, a list of employees, is passed to `nameOf`, which expects a singleton employee. Moreover, even if it compiled, the final result would not be in uppercase, as `sortOn` produces a permutation of the input list.

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 n = bar 0 1 n
bar x y n
  | n == 0 = x
  | otherwise = bar y (x + y) (n - 1)
```

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

- ☐ `foo :: Int -> Int`
- ☐ `foo :: (Eq a, Num a) => a -> a`
- ■ `foo :: (Eq a, Num a, Num b) => a -> b`
- ☐ `foo :: (Eq a, Num a, Eq b, Num b) => a -> b`

(b) What is the result of invoking `foo n` for some positive natural number `n`? (4)

- ☐ `0 + 1 + 2 + ... + n`
- ☐ `n * n`
- ■ **The `n`-th element of the list of Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, . . .**
- ☐ None of the above

(c) Assume that we evaluate `foo n :: Int` for some positive `n :: Int`. (4)
Choose the correct statement.

- ☐ The memory consumption is constant for both innermost evaluation and lazy evaluation.
- ☐ The memory consumption grows linearly in `n` for both innermost evaluation and lazy evaluation.
- ☐ The memory consumption grows linearly in `n` when using innermost evaluation, but is constant when using lazy evaluation.
- ■ **The memory consumption is constant when using innermost evaluation, but grows linearly in `n` when using lazy evaluation.**