

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	
$\Sigma$	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**

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

```
1 module Main(main, game) where
2
3 import System.Random(randomIO) -- randomIO :: IO Integer
4
5 main :: IO Integer
6 main = do
7   num <- randomIO
8   putStrLn "Try to guess my number"
9   guessingGame (abs num) 1
10
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)

(b) Mistake #2 (5)

(c) Mistake #3 (5)

(d) Mistake #4 (5)

**Exercise 2: Programming with Lists**

Periodic functions can be represented by a list of initial values  $vs = [v_1, \dots, v_k]$  and a non-empty list of values  $ws = [w_1, \dots, w_m]$  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  $[v_1, \dots, v_k, w_1, \dots, w_m, w_1, \dots, w_m, w_1, \dots, w_m, \dots]$  for every non-negative integer  $n$ . For example, if `f = periodic [4,2] [5,1,3]` then the results of evaluating `f` for arguments  $0, \dots, 8$  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 `infiList` for the infinite list described above, i.e., `infiList vs ws` should evaluate to  $[v_1, \dots, v_k, w_1, \dots, w_m, w_1, \dots, w_m, \dots]$ , and provide the most general type of `infiList`. Further define a function `periodicInfi` as an implementation of `periodic` that is based on `infiList`. (6)
- (b) Define a function `periodicLinear` that implements `periodic` without constructing an infinite list and *without using any predefined functions on lists* (except for the list constructors). Evaluating `periodicLinear vs ws n` should require approximately  $n$  steps. (6)
- (c) Define a function `periodicFaster :: [a] -> [a] -> Int -> a` as an implementation of `periodic`. You should make use of the periodicity so that evaluating `periodicFaster 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 `reconLists :: Eq a => (Int -> a) -> ([a], [a])` such that `reconLists g` reconstructs `vs` and `ws` from `g`. In particular `reconLists (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 `reconListSimple :: Eq a => (Int -> a) -> [a]`. Here we assume that the input is a periodic function `g = periodic [] ws` and only `ws` is computed via `reconListSimple`. In particular, the property `reconListSimple (periodic [] ws) == ws` should be satisfied whenever `ws` is a finite, non-empty and distinct list.



**Exercise 3: Datatypes and Higher-Order Functions**

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
  Integer      -- skill-rating
  Double       -- salary

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

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

(4)

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

(4)

$$\text{new-salary} = \text{old-salary} + 20 \times \text{skill-rating}$$

Further assume our implementation uses the following structure.

```
increaseSalary = mapEmp undefined undefined undefined
```

Either replace each `undefined` by a suitable  $\lambda$ -expression, or argue why `increaseSalary` cannot be implemented via `mapEmp`.

- (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 toUpper) id id)`
- `toUpperEmployees = map (mapEmp (map (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 = sort . toUpperEmployees . map nameOf
sun2 = map nameOf . sort . toUpperEmployees
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.

**Exercise 4: Evaluation and Types**

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 (x + y) (y + 1) (n - 1)
```

- (a) What is the most general type of `foo`? (4)
- `foo :: (Ord a, Num a, Eq b, Num b) => a -> b`
  - `foo :: (Eq a, Num a, Num b) => a -> b`
  - `foo :: (Eq a, Num a) => a -> a`
  - `foo :: Int -> Int`
- (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 grows linearly in `n` for both innermost evaluation and lazy evaluation.
  - The memory consumption is constant for both innermost evaluation and lazy evaluation.
  - The memory consumption is constant when using innermost evaluation, but grows linearly in `n` when using lazy evaluation.
  - The memory consumption grows linearly in `n` when using innermost evaluation, but is constant when using lazy evaluation.