Name: _____

Matriculation Number: _____

Exercise	Points	Score
Types	12	
Evaluation	11	
Programming	15	
I/O and Modules	7	
$\sum$	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: Types 12

Consider the following Haskell code:

```
data Foo a = Bar | Com a Int (Foo a) deriving Eq

f x y z = if x == Bar then y else z
g x = if x < Bar then replicate 9 '.' else "World"
c = Com
d = \ x -> Com x x Bar
```

In each multiply 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 the wrong statement results in 0 points.

(a) The most general type of `f` is (3)

- ☐ `(Eq a, Eq b) => Foo a -> b -> b -> b`
- ■ `Eq a => Foo a -> b -> b -> b`
- ☐ `Eq a => Foo a -> a -> a -> a`
- ☐ `Foo a -> b -> b -> b`
- ☐ `f` is not type-correct.

(b) The most general type of `g` is (3)

- ☐ `Eq a => Foo a -> String`
- ☐ `Ord a => Foo a -> String`
- ☐ `Foo a -> String`
- ☐ `Foo String -> String`
- ■ `g` **is not type-correct.**

(c) The most general type of `c` is: (3)

- ☐ `Eq a => a -> Int -> Foo a`
- ☐ `Eq a => a -> Int -> Foo a -> Foo a`
- ■ `a -> Int -> Foo a -> Foo a`
- ☐ `Foo a -> a -> Int -> Foo a -> Foo a`
- ☐ `c` is not type-correct.

(d) The most general type of `d` is: (3)

- ☐ `a -> Foo a`
- ☐ `a -> Foo (a,a)`
- ■ `Int -> Foo Int`
- ☐ `Eq a => a -> Foo a`
- ☐ `d` is not type-correct.

Exercise 2: Evaluation 11

Consider the following Haskell code:

```
front_A, front_B, front_C, front_D, front_E :: [a] -> [a]
front_A xs = take (length xs - 1) xs
front_B xs = map fst (zip xs (tail xs))
front_C = reverse . tail . reverse
front_D = drop 1 . reverse
front_E xs = [ xs !! j | i <- [1 .. length xs], let j = i - 1]
```

(a) Assume the input is a non-empty finite list $[x_1, \ldots, x_n]$. Then most of the `front_X`-functions return the list $[x_1, \ldots, x_{n-1}]$. Write down all `front_X`-functions that return a *different list* and also give the results of these functions. (3)

Solution:
`front_D` results in $[x_{n-1}, \ldots, x_1]$ and `front_E` results in $[x_1, \ldots, x_n]$.

(b) Next we consider the empty list as input. Write down the result of `front_X []` for `X = C,D,E` and provide a step by step evaluation of `front_B []`. (5)

As a reminder, here are the definitions of `zip` and `tail`.

```
tail (_ : xs)          = xs
tail []                = error "empty list"
zip []       _         = []
zip _        []        = []
zip (x : xs) (y : ys) = (x,y) : zip xs ys
```

Solution:
```
front_B [] = map fst (zip [] (tail [])) = map fst [] = []
front_C [] = error "empty list"
front_D [] = []
front_E [] = []
```

(c) Now assume the input is an infinite list. Write down all `front_X`-functions which satisfy that `front_X [0..]` evaluates to `[0..]`. (3)

Solution:
Only `front_B` satisfies the property. All other versions do not terminate while computing the reverse or the length of the infinite list.

Exercise 3: Programming 15

Consider a function `search` which given a key k and a list of key-value pairs, returns v if (k, v) is the *first entry* in the list with key k, or nothing if no such pair exists.

Examples:

- `search 'c' [('a',1), ('z',26)] = Nothing`
- `search 5 [(3, "a"), (5, "b"), (5, "c"), (2, "g")] = Just "b"`

(a) Give a suitable type-definition of `search`. In particular, the examples above should be type-correct, and one should be able to implement `search` with your type. (2)

Solution:

```
search :: Eq a => a -> [(a,b)] -> Maybe b
```

(b) Provide a *recursive definition* of `search` that does not use any library functions on lists, except for the list constructors. (3)

Solution:

```
search k [] = Nothing
search k ((key, val) : xs)
  | k == key = Just val
  | otherwise = search k xs
```

(c) Provide a *non-recursive definition* of `search` that is based on *list-comprehensions*. (3)

Solution:

```
search k xs = case [ val | (key,val) <- xs, key == k] of
  [] -> Nothing
  (v : _) -> Just v
```

(d) Provide a *non-recursive definition* of `search` that is based on `foldr`. (3)

Solution:

```
search k = foldr ( \ (key,val) res -> if key == k then Just val else res) Nothing
```

(e) Write a function `good_object :: [(String,String)] -> Maybe String` which returns an object that is rated as good, if such an object exists. (4)

- The input list of rated objects is always given in pairs of the form (object, rating), e.g., as in `[("toast", "medium"), ("bread", "good"), ("oat-slime", "bad"), ...]`.
- If there are many objects that are rated as good, return the one which is *last in alphabetical order*. You may assume that all object names are provided in lower-case letters.
- In the definition you may use `search` from above and standard list functions like `sort`, `map`, `reverse`, ..., but neither list-comprehensions nor `filter`.

Solution:

```
good_object = search "good" . map ( \ (i,r) -> (r,i) ) . reverse . sort
```

Exercise 4: I/O and Modules 7

Consider the following Haskell module.

```
module Volume where

volume :: Double -> Double
volume r = 4 / 3 * pi * r ^ 3
```

Write a Haskell program (outside of the module `Volume`) which asks the user for a radius and then prints the volume of a ball with that radius, *precisely* as formatted in the two lines between the `prompt>...`-lines.

```
prompt> ./my_program    # start program
Enter radius: 1.5
Volume of ball with radius 1.5 is 14.137166941154067.
prompt>                 # program has ended
```

- The program should be compilable via `ghc --make`.
- The user made exactly one input, namely the first occurrence of the number 1.5.
- For the calculation, the method `volume` has to be invoked.
- You can ignore buffering problems.

Solution:

```
import Volume

main = do
  putStr "Enter radius: "
  str <- getLine
  let r = (read str :: Double)
  let res = volume r
  putStrLn $ "Volume of ball with radius " ++ str ++ " is " ++ show res ++ "."
```