Lastname: ______________________________

Firstname: ______________________________

Matriculation Number: ______________________________

Exercise	Points	Score
Program Analysis	17	
Datatypes	20	
Programming with Lists	31	
Programming with I/O	22	
$\sum$	90	

- You have 90 minutes time 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.

Exercise 1: Program Analysis 17

Consider the Haskell following program.

```
module A where
foo True = "hello"

module B where
foo :: Int -> Int
foo x = x + 1

module C where

import A
import B

bar :: Bool -> String
bar b = foo b ++ " world"
```

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.

(a) (4)

- ☐ Module `A` does not compile since there is no defining equation for `foo False`.
- ☐ Module `A` does not compile since there is no type-definition for `foo`.
- ☐ Module `A` does not compile because of some other error.
- ☐ Module `A` compiles without errors.

(b) (4)

- ☐ Module `C` does not compile even when dropping the definition of `bar`: the clash of the imported name `foo` coming from both `A` and `B` is not permitted.
- ☐ Module `C` does not compile because of an unresolved name clash of `foo` in the definition of `bar`.
- ☐ Module `C` compiles successfully, since `foo` in `C` must refer to `A.foo`, as `B.foo` would lead to a type-error.
- ☐ Module `C` compiles successfully: `foo` in `C` refers to `A.foo`, as `A` was imported before `B`.

(c) Consider the Haskell following program: (9)

```
isEmpty1 xs = case xs of {[] -> True; _ -> False}
isEmpty2 xs = length xs == 0
isEmpty3 xs = xs == []
```

As usual, we say that two functions are equivalent if they have the same type and deliver the same output for each input.

For each of the three combinations of `isEmptyX`-functions, indicate whether they are equivalent or not, and if not provide a brief justification why they are not equivalent.

Exercise 2: Datatypes 20

Consider following Haskell code:

```
data TypeA a b = A a b | B Int (TypeA a b) | C a a | EmptyA

func1 = B
func2 x y = A x y == B 2 (A x y)
func3 = \ x y -> B x (A x y)

g :: Eq a => TypeA a b -> TypeA a b -> Bool
h :: TypeA Int a -> Int -> Bool
```

For each question, exactly one answer is correct. Marking the correct statement is worth 4 points, giving no answer counts 1 point, and marking multiple or the wrong statement results in 0 points.

(a) What is the most general type of `func1`? (4)

- ☐ `func1:: a -> TypeA a b -> TypeA a b`
- ☐ `func1:: Int -> TypeA a b -> TypeA a b`
- ☐ `func1:: a -> b -> TypeA a b`
- ☐ `func1 is not type-correct`

(b) What is the most general type of `func2`? (4)

- ☐ `func2 :: a -> a -> Bool`
- ☐ `func2 :: Int -> a -> TypeA a b`
- ☐ `func2 :: Int -> a -> Bool`
- ☐ `func2 is not type-correct`

(c) What is the most general type of `func3`? (4)

- ☐ `func3 :: Int -> a -> TypeA Int a`
- ☐ `func3 :: a -> b -> TypeA a b`
- ☐ `func3 :: Int -> a -> TypeA a b`
- ☐ `func3 is not type-correct`

(d) Which equation is allowed in the function definition of `g`? (4)

- ☐ `g (A x y) (C z u) = x > z`
- ☐ `g (A x y) (C z u) = z == y`
- ☐ `g (A x y) (B z (C u t)) = u == x && t == x`
- ☐ `g (A x y) (B z (A u t)) = y /= t || u == x`

(e) Which equation is allowed in the function definition of `h`? (4)

- ☐ `h (A x y) z = y < z`
- ☐ `h (C x y) z = x == y && z < x`
- ☐ `h (B x y) z = if x > z then 1 else 0`
- ☐ `h (B x y) z = y == z`

Exercise 3: Programming with Lists 31

(a) Write a function `lefts :: [Either a b] -> [a]` that takes a list of values of type `Either a b` and returns the list consisting of all the `Left`-values in the same order. (5)

Example:

```
lefts [Left 10, Left 42, Right 23, Left 37, Right 19] == [10, 42, 37]
```

(b) We say that a list is ascending if each value in it is strictly smaller than the one following it. That is, if an ascending list is of the form $[x_1, \ldots, x_n]$, then $x_i < x_{i+1}$ for any $1 \leq i < n$. (8)

Define a function `check :: Ord a => [a] -> Maybe Int` that returns `Nothing` if the list is ascending and `Just i` otherwise, where `i` is the least index that violates the above condition. (that is, $x_i \geq x_{i+1}$)

Example:

```
check []           == Nothing
check [1, 5, 7]    == Nothing
check [1, 5, 5, 9] == Just 2
```

(c) Define a function `run :: (a -> Maybe a) -> a -> [a]`. It takes as arguments a function `f` and a starting value `x`. The idea is that `f` either returns some result (`Just x'`) or indicates that you are done (`Nothing`). The task of `run` is to repeatedly apply `f` to obtain new values until it gets `Nothing`, and returns a list of all the intermediate results you encountered in order. (8)

Note that it is possible that `f` will never return `Nothing`, in which case the result list will be infinite.

Example:

```
run (\x -> if x >= 5 then Nothing else Just (x + 1)) 1 == [1, 2, 3, 4, 5]
run (\xs -> Just ('a' : xs)) "" == ["", "a", "aa", "aaa", "aaaa", ...] -- infinite
```

(d) Write a function `sumUp` that takes a (possibly infinite) list $xs = [x_1, x_2, x_3, \ldots]$ of numbers and returns the list of numbers obtained by summing up the first 0, 1, 2, etc. numbers of xs, i.e. (10)

$$0,\ x_1,\ x_1{+}x_2,\ x_1{+}x_2{+}x_3,\ \ldots$$

Also specify the type of `sumUp` which should be as general as possible.

Examples:

```
sumUp []              == [0]
sumUp [1, 2, 3]       == [0, 1, 3, 6]
sumUp [2, 5, 3, 4]    == [0, 2, 7, 10, 14]
sumUp [1, 1, 1, ...] == [0, 1, 2, 3, ...] -- infinite lists
```

Exercise 4: Programming with I/O 22

(a) Define a function `getNegNumber :: IO Int`, that asks the user for a number, until a negative number is entered. In the following example dialog, the contents until the ": " is printed via `getNegNumber`, and the text afterwards was entered by the user. (14)

```
-- getNumber is invoked
enter negative number: minus-five
retry: -0
retry: -5
-- getNumber now returns the integer -5
```

For your implementation the function `readMaybe :: Read a => String -> Maybe a` might be useful in addition to the various I/O functions. (You don't have to be provide an import statement!)

(b) Define a Haskell program that can be compiled by `ghc` and then be executed without `ghci`. Here, the program should ask the user for a negative number, and the difference to the absolute value should printed, as it is illustrated in the following dialog. Here, only the number -48 in the first line was entered by the user. You may assume that `getNegNumber` was implemented correctly. (8)

```
enter negative number: -48
-48 + 96 = |-48|
```