VO Functional Programming December 12, 2022
LVA 703024 Test Exam

Last Name: ______________________________

First 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.

Remarks:

- This is an old exam that was designed as a closed book exam, i.e., no notes, slides, books, computers, . . . were allowed.
- Blank paper for making notes were made available to all participants.
- 50 % of the points were required to pass the exam.

**Exercise 1: Types** 12

Consider the following Haskell code:

```
data Type a = Empty | Node a Int (Type a) deriving Eq

c = Node
d = \ x -> Node x x Empty
f x y z = if x == Empty then y else z
g x = if x > Empty then "Hello" else replicate 10 '!'
```

In each multiple choice question, exactly one statement is correct. Marking the correct statement is worth 3 points, giving no answer counts as 1 point, and marking multiple or the wrong statement results in 0 points.

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

□ `Type a -> a -> Int -> Type a -> Type a`
■ `a -> Int -> Type a -> Type a`
□ `Eq a => a -> Int -> Type a -> Type a`
□ `Eq a => a -> Int -> Type a`
□ `c` is not type-correct.

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

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

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

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

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

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

**Exercise 2: Evaluation** 11

Consider the following Haskell code:

```
drop_last_A, drop_last_B, drop_last_C, drop_last_D, drop_last_E :: [a] -> [a]
drop_last_A xs = take (length xs - 1) xs
drop_last_B = drop 1 . reverse
drop_last_C = reverse . tail . reverse
drop_last_D xs = map fst (zip xs (tail xs))
drop_last_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 `drop_last_X`-functions return the list $[x_1, \ldots, x_{n-1}]$. Write down all `drop_last_X`-functions that return a *different list* and also give the result of these functions. (3)

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

(b) Next we consider the empty list as input. Write down the result of `drop_last_X []` for `X = B,C,E` and provide a step by step evaluation of `drop_last_D []`. (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:**

```
drop_last_B [] = []
drop_last_C [] = error "empty list"
drop_last_D [] = map fst (zip [] (tail [])) = map fst [] = []
drop_last_E [] = []
```

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

**Solution:**
Only `drop_last_D` 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 `find` 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:

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

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

**Solution:**

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

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

**Solution:**

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

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

**Solution:**

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

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

**Solution:**

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

(e) Write a function `bad_item :: [(String,String)] -> Maybe String` which returns an item that is rated poorly, if such an item exists. (4)

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

**Solution:**

```
bad_item = find "poor" . map ( \ (i,r) -> (r,i) ) . reverse . sort
```

**Exercise 4: I/O and Modules** 7

Consider the following Haskell module.

```
module Area where

area :: Double -> Double
area r = pi * r * r
```

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

```
prompt> ./my_program    # start program
Enter radius: 6.72
Area of circle with radius 6.72 is 141.8692976878693.
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 6.72.
- For the calculation, the method `area` has to be invoked.

**Solution:**

The following program is a correct solution according to the course, where the two commented lines are not present.

However, the program will not behave as intended when you compile it, since the first `putStr` is not immediately displayed because of buffered I/O, a topic that was not discussed in the lecture.

To make the program behave as intended, one would have to uncomment the two comments.

```
import Area
-- import Sytem.IO

main = do
  putStr "Enter radius: "
--  hFlush stdout
  str <- getLine
  let r = (read str :: Double)
  let res = area r
  putStrLn $ "Area of circle with radius " ++ str ++ " is " ++ show res ++ "."
```