December 11, 2023

Test Exam

Lastname: ______________________________

Firstname: ______________________________

Matriculation Number: ______________________________

Exercise	Points	Score
Programming with Strings and I/O	25	
Programming with Abstract Datatypes and Lists	37	
Programming with Trees and Functions	16	
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.
- 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.*
- *Solutions could be formulated in German or English.*

Exercise 1: Programming with Strings and I/O

25

(a) Implement a function `getMonth :: String -> Int` which extracts the number of the month for a given date. You may assume that dates are provided as strings using the format `"DD.MM.YYYY"`. For instance, `getMonth "19.09.2023"` results in `9`. (5)

(b) Implement a function `monthName :: Int -> String` which converts a number between 1 and 12 into the corresponding name of the month (abbreviated by its first three letters). For instance, `monthName 1` should return `"Jan"` for January. (You may use German or English names of months). (5)

(c) Write a program which does the following. (15)

- It asks the user for its birthday, e.g., `"Hello, when were you born?"` and receives an answer, i.e., a date, e.g., `"19.09.2000"`.
- It prints the name of the month with some surrounding text, e.g., `"Oh, in Sep."`.
- It further prints the quarter of the year in which this month is located, e.g., `"This is in the 3rd quarter of the year."`.

Your program has to be compilable as a stand-alone program; it must contain a module declaration and contain all required import statements.

You do *not* have to integrate error handling, e.g., for handling user answers that do not specify a date in the described format.

Of course, you may assume that `monthName` and `getMonth` are available even if you did not solve parts (a) and (b).

Solution:

```
module Main(main) where

months :: [String]
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

monthName :: Int -> String
monthName x = months !! (x - 1)

getMonth :: String -> Int
getMonth (d1 : d2 : dot : m1 : m2 : _) = read [m1,m2]

quarterName :: Int -> String
quarterName m
  | 1 <= m && m <= 3 = "1st"
  | 4 <= m && m <= 6 = "2nd"
  | 7 <= m && m <= 9 = "3rd"
  | otherwise = "4th"

main = do
  putStrLn "Hello, when were you born?"
  date <- getLine
  let m = getMonth date
  putStrLn $ "Oh, in " ++ monthName m ++ "."
  putStrLn $ "This is in the " ++ quarterName m ++ " quarter of the year."
```

Exercise 2: Programming with Abstract Datatypes and Lists 37

Consider an abstract datatype for *sets*.

Abstract datatypes will be introduced in Lecture 12 on January 15. Just skip exercises (a) and (b) for the moment. You can however already solve tasks (c)–(g). These just ask you to implement certain operations on sets for the concrete datatype `Set a` *that is specified on the next page.*

The datatype has the following signature:

```
empty :: Set a                          -- the empty set
insert :: Eq a => a -> Set a -> Set a   -- insert a single element into a set
complement :: Set a -> Set a            -- complement a given set
member :: Eq a => a -> Set a -> Bool    -- testing membership
```

Some typical equations that should be satisfied are, for example,

```
not (member x empty)
member x (insert x set)
x /= y ==> member y (insert x set) = member y set
member x (complement set) = not (member x set)
```

(a) Implement a function `remove :: Eq a => a -> Set a -> Set a` that removes an element of a set. Here, you may only use the functions of the abstract datatype. (4)

(b) Specify two useful equations that are satisfied by `remove`. (4)
(You do *not* have to prove that these equations are indeed satisfied!)

Now, consider a list-based implementation of sets by the following datatype. Here, one can either specify a list of elements that should be included in the set, or specify a list of elements that should be excluded from the set.

```
data Set a = Incl [a] | Excl [a]

exampleSet1, exampleSet2 :: Set Integer
exampleSet1 = Incl [4,9,2]  -- the set {2,4,9}
exampleSet2 = Excl [2,5]    -- the set of all integers excluding 2 and 5
```

For all of the following implementation tasks, you may freely use arbitrary Prelude functions.
All tasks can be solved independently.

(c) Implement `empty` for the given representation. (2)

(d) Implement `complement` for the given representation. (3)

(e) Implement `member` for the given representation. (5)

(f) Implement `insert` for the given representation. Here, your implementation should preserve the following invariant: no duplicates appear in the lists that represent sets. (7)

(g) Implement a function `subsetInteger :: Set Integer -> Set Integer -> Bool` that checks whether the first set is a subset ($\subseteq$) of the second set. You may assume that `member` is available, even if you did not solve that task. (8)

(h) Provide a module-declaration such that all required functionality is available to an external user, and such that the internally used invariant cannot be violated. (4)

Solution:

```
module Set(Set, remove, empty, complement, insert, member, subsetInteger) where

remove :: Eq a => a -> Set a -> Set a
remove x s = complement (insert x (complement s))

{-
equations:
not (member x (remove x set))
x /= y ==> member y (remove x set) = member y set
-}

data Set a = Incl [a] | Excl [a]

empty :: Set a
empty = Incl []

complement :: Set a -> Set a
complement (Incl xs) = (Excl xs)
complement (Excl xs) = (Incl xs)

member :: Eq a => a -> Set a -> Bool
member x (Incl xs) = x `elem` xs
member x (Excl xs) = not (x `elem` xs)

insert :: Eq a => a -> Set a -> Set a
insert x s@(Incl xs) = (if x `elem` xs then s else Incl (x : xs))
insert x s@(Excl xs) = (if x `elem` xs then Excl (filter (/= x) xs) else s)

subsetInteger :: Set Integer -> Set Integer -> Bool
subsetInteger (Incl xs) ys = all (\ x -> member x ys) xs
subsetInteger xs (Excl ys) = all (\ y -> not (member y xs)) ys
subsetInteger (Excl _) (Incl _) = False
```

Exercise 3: Programming with Trees and Functions $\boxed{16}$

Consider the following datatype definition for binary trees.

```
data Tree a = Node (Tree a) (Tree a) | Leaf a
```

(a) Assume you are given the following function (9)

```
foldTree :: (b -> b -> b) -> (a -> b) -> Tree a -> b
```

that is the canonical fold-function on trees, i.e., `foldTree f g t` replaces in `t` every `Node` by `f` and every `Leaf` by `g`.
Implement all of the following three functions on trees. They are one-liners if you use `foldTree`, but it is also permitted to write time-consuming recursive definitions.

- `sumTree` computes the sum of all numbers in a tree. Provide a type-definition for `sumTree`, too.
- `prodTree` computes the product of all numbers in a tree. Provide a type-definition for `prodTree`, too.
- `mapTree :: (a -> b) -> Tree a -> Tree b` applies a function to all elements in a tree.

(b) Implement `foldTree`. (7)

Solution:

```
data Tree a = Node (Tree a) (Tree a) | Leaf a

foldTree :: (b -> b -> b) -> (a -> b) -> Tree a -> b
foldTree f g (Node l r) = f (foldTree f g l) (foldTree f g r)
foldTree f g (Leaf x) = g x

sumTree, prodTree :: Num a => Tree a -> a
sumTree = foldTree (+) id
prodTree = foldTree (*) id

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f = foldTree Node (Leaf . f)
```

Exercise 4: Evaluation and Types $\boxed{12}$

In each of the three multiple choice questions, 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 x []     = []
foo x (y:ys) = ((0 :) . (x :)) ys : foo y ys
```

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

- ☐ `foo :: a -> [a] -> [[a]]`
- ☐ `foo :: Eq a => a -> [a] -> [[a]]`
- ■ `foo :: Num a => a -> [a] -> [[a]]`
- ☐ None of the above.

(b) What is the result of invoking `foo 0 [1,2,3]`? (4)

- ☐ `[[0,2,3],[1,0,3],[1,2,0]]`
- ■ `[[0,0,2,3],[0,1,3],[0,2]]`
- ☐ `[[0,0,2,3],[0,1,3],[0,2],[]]`
- ☐ `[[0,1,0,2,0,3],[0,2,0,3],[0,3],[]]`

(c) Assume we enter the expression `[(n,m) | n <- [0 .. 2], 3 < m && m <= 5]` in `ghci`. What will be the result? (4)

- ☐ `[(0,4),(1,4),(2,4),(0,5),(1,5),(2,5)]`
- ☐ `[(0,4),(0,5),(1,4),(1,5),(2,4),(2,5)]`
- ☐ we get a compile error, since type-annotations are missing in the expression, such as `n :: Integer` or `m :: Double`
- ■ **we get a compile error, since the expression is not allowed in Haskell, even if one would add type-annotations**