Lab Report

23CSE212 – Principles of Programming Languages

Criteria	Excellent	Good	Poor
Timely Submission			
Correctness of lab assignment			
		Total	
Faculty Signature			

Lab Session No: 8.2
Date:

CO3: Apply advanced Haskell concepts, including abstract data types, evaluation, streams, input/output operations (IO), applicative factors, monads

Code	**Testcases (Input & Output)**

Question 1:

```
myLookup134 :: Eq a => a -> [(a,
b)] -> Maybe b
myLookup134 _ [] = Nothing
myLookup134 key ((k,v):xs)
  | key == k  = Just v
  | otherwise = myLookup134 key xs
```

```
ghci> myLookup "b" [("a",1),("b",2)]
Just 2
ghci> myLookup "z" [("a",1),("b",2)]
Nothing
```

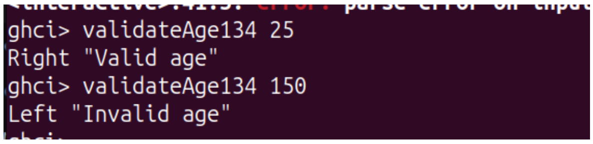
Question 2:

```
lookupE134 :: Eq a => a -> [(a,
b)] -> Either String b

lookupE134 _ [] = Left "Key not
found"

lookupE134 key ((k,v):xs)

  | key == k  = Right v

  | otherwise = lookupE134 key xs
```

```
ghci> lookupE134 "b" [("a",1),("b",2)]
Right 2
ghci> lookupE134 "z" [("a",1),("b",2)]
Left "Key not found"
```

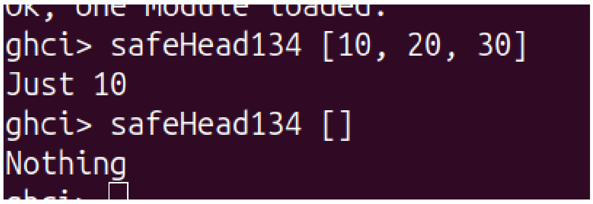
Question 3:

```
validateAge134 :: Int -> Either
String String
validateAge134 age
  | age >= 0 && age <= 120 = Right
"Valid age"
  | otherwise              = Left
"Invalid age"
```



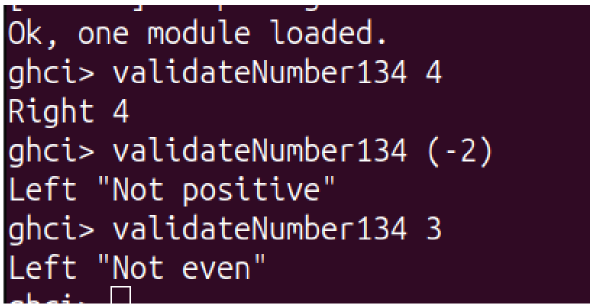
Question 4:

```
safeHead134 :: [a] -> Maybe a
safeHead134 []   = Nothing
safeHead134 (x:_) = Just x
```



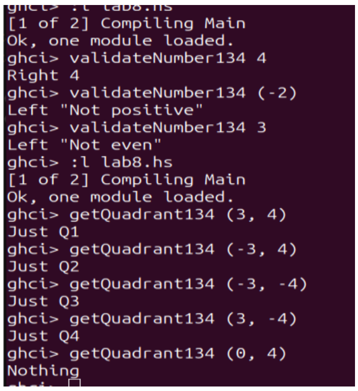
Question 5:

```
checkPositive134 :: Int -> Either
String Int
checkPositive134 n
  | n > 0     = Right n
  | otherwise = Left "Not
positive"

checkEven134 :: Int -> Either
String Int
checkEven134 n
  | even n    = Right n
  | otherwise = Left "Not even"

validateNumber134 :: Int -> Either
String Int
validateNumber134 n = do
  x <- checkPositive134 n
  checkEven134 x
```



Question 6:

```
type Point = (Float, Float)

data Move134 = LeftM | RightM | Up
| Down deriving Show
data Quadrant134 = Q1 | Q2 | Q3 |
Q4 deriving Show

getQuadrant134 :: Point -> Maybe
Quadrant134


getQuadrant134 (x, y)
  | x == 0 || y == 0 = Nothing
  | x > 0 && y > 0   = Just Q1
  | x < 0 && y > 0   = Just Q2
  | x < 0 && y < 0   = Just Q3
  | x > 0 && y < 0   = Just Q4
```



b)

b)

```
data Point = Point Float Float
deriving (Show, Eq)

mover :: Float -> Point -> Point
mover angle (Point x y) =
  let step = 134
      dx = step * cos angle
      dy = step * sin angle
  in Point (x + dx) (y + dy)
```

```
Ok, one module loaded.
ghci> let p = Point 0 0
ghci> mover (pi/4) p
Point 94.752304 94.752304
ghci>
```

c)

```
distance :: Point -> Point ->
Float
distance (Point x1 y1) (Point x2
y2) =
  sqrt ((x2 - x1)^2 + (y2 - y1)^2)
```

```
ghci> let p1 = Point 0 0
ghci> let p2 = Point 3 4
ghci> distance p1 p2
5.0
```

d)

```
-- Define the Point type data
Point = Point Float Float deriving
(Show, Eq)

-- Define offsetPoint to add (134,
134) to a Point offsetPoint ::
Point -> Point offsetPoint (Point
x y) = Point (x + 134) (y + 134)
```

```
[1 of 2] Compiling Main             ( lab8.hs, interp
Ok, one module loaded.
ghci> let p = Point 1 2
ghci> offsetPoint p
Point 135.0 136.0
```

e)

```
-- Define Point type data Point =
Point Float Float deriving (Show,
Eq)

-- Define midpoint function
midpoint :: Point -> Point ->
Maybe Point midpoint (Point x1 y1)
(Point x2 y2) | x1 == x2 = Nothing
| otherwise = Just (Point ((x1 +
x2) / 2) ((y1 + y2) / 2))
```

```
ghci> :l lab8.hs
[1 of 2] Compiling Main             ( lab8.hs, interpr
Ok, one module loaded.
ghci> let p1 = Point 2 4
ghci> let p2 = Point 4 6
ghci> midpoint p1 p2
Just (Point 3.0 5.0)
```

Question 7:

a)

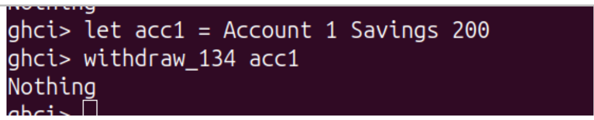
```
-- Define AccountType before using
it in Account data AccountType =
Savings | Current deriving (Show,
Eq)

 AccountType data Account =
Account { accNumber :: Int ,
```

A)

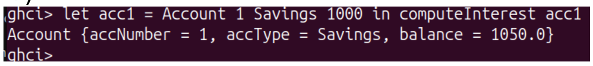
```
ghci> :l lab8.hs
[1 of 2] Compiling Main             ( lab8.hs, interpreted )
Ok, one module loaded.
ghci> let acc1 = Account 1 Savings 1000
ghci> deposit_134 acc1
Account {accNumber = 1, accType = Savings, balance = 1134.0}
ghci>
```

B)

```
accType :: AccountType , balance
:: Float } deriving (Show, Eq)

withdraw_134 :: Account -> Maybe
Account

withdraw_134 acc

  | newBalance < minBalance =
Nothing

  | otherwise = Just acc { balance
= newBalance }

where  newBalance = balance acc -
134  minBalance = case accType acc
of Savings -> 100 Current -> 0

c) computeInterest :: Account ->
Account

computeInterest acc@(Account _
Savings bal) =

  acc { balance = bal + bal * 0.05
}

computeInterest acc = acc  -- No
interest for Current accounts
```



C)

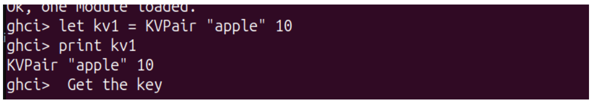


Question 8:

```
-- Define a polymorphic queue data
Queue a = Queue [a] [a] deriving
(Show)

-- Create an empty queue
emptyQueue :: Queue a emptyQueue =
Queue [] []

-- a. Enqueue: Add element to the
end of the queue enqueue :: a ->
Queue a -> Queue a enqueue x
(Queue front back) = Queue front
(x : back)

-- b. Dequeue: Remove and return
the front element of the queue
dequeue :: Queue a -> Maybe (a,
Queue a) dequeue (Queue [] []) =
Nothing dequeue (Queue [] back) =
dequeue (Queue (reverse back) [])
dequeue (Queue (x:xs) back) = Just
(x, Queue xs back)
```

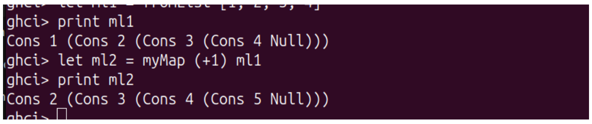
Question 9:

```
-- Define polymorphic Key-Value
Pair data KVPair k v = KVPair k v
deriving (Show)

-- a. getKey: Retrieve the key
from a key-value pair getKey ::
KVPair k v -> k getKey (KVPair k
_) = k

-- b. getValue: Retrieve the value
from a key-value pair getValue ::
KVPair k v -> v getValue (KVPair _
v) = v

-- c. mapValue: Apply a function
to transform the value, keeping
the key unchanged mapValue :: (v -
> v') -> KVPair k v -> KVPair k v'
mapValue f (KVPair k v) = KVPair k
(f v)
```



Question 10:

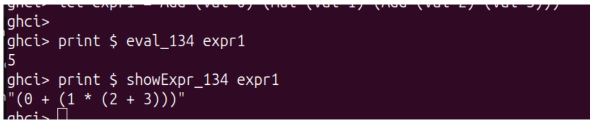
```
-- Define recursive list data type
data MyList a = Null | Cons a
(MyList a) deriving (Show)

-- a. myLength: Return the number
of elements in the list myLength
:: MyList a -> Int myLength Null =
0 myLength (Cons _ xs) = 1 +
myLength xs

-- b. myMap: Apply a function to
each element myMap :: (a -> b) ->
MyList a -> MyList b myMap _ Null
= Null myMap f (Cons x xs) = Cons
(f x) (myMap f xs)

-- c. myAppend: Concatenate two
MyLists myAppend :: MyList a ->
MyList a -> MyList a myAppend Null
ys = ys myAppend (Cons x xs) ys =
Cons x (myAppend xs ys)

-- d. myToList: Convert MyList to
regular list myToList :: MyList a
-> [a] myToList Null = [] myToList
(Cons x xs) = x : myToList xs

-- e. fromList: Convert regular
list to MyList fromList :: [a] ->
MyList a fromList [] = Null
```



```
fromList (x:xs) = Cons x (fromList
xs)
```

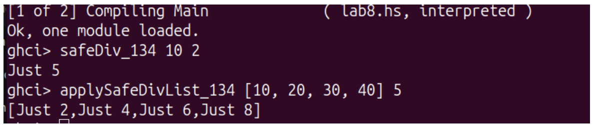
Question 11:

```
-- Define Expr data type data Expr
= Val Int | Add Expr Expr | Mul
Expr Expr deriving (Show)

-- Evaluate the expression
(eval_134) eval_134 :: Expr -> Int
eval_134 (Val n) = n eval_134 (Add
e1 e2) = eval_134 e1 + eval_134 e2
eval_134 (Mul e1 e2) = eval_134 e1
* eval_134 e2

-- Convert expression to readable
string (showExpr_134) showExpr_134
:: Expr -> String showExpr_134
(Val n) = show n showExpr_134 (Add
e1 e2) = "(" ++ showExpr_134 e1 ++
" + " ++ showExpr_134 e2 ++ ")"
showExpr_134 (Mul e1 e2) = "(" ++
showExpr_134 e1 ++ " * " ++
showExpr_134 e2 ++ ")"
```



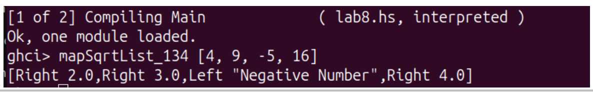
Question 12:

```
-- safeDiv_134: Safe division
returning Maybe Int safeDiv_134 ::
Int -> Int -> Maybe Int
safeDiv_134 _ 0 = Nothing
safeDiv_134 x y = Just (x div y)

-- applySafeDivList_134: Divide
list by divisor safely
applySafeDivList_134 :: [Int] ->
Int -> [Maybe Int]
applySafeDivList_134 xs divisor =
map (safeDiv_134 divisor) xs

-- extractJusts_134: Extract
values from Just, filter Nothing
extractJusts_134 :: [Maybe Int] ->
[Int] extractJusts_134 = foldr (\x
acc -> case x of Just v -> v :
acc; Nothing -> acc) []
```



Question 13:

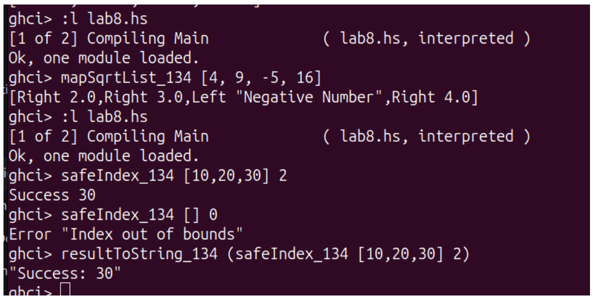
```
import Data.Either (Either(..))
```



```
-- safeSqrt_134: returns Right sqrt or Left error message
safeSqrt_134 :: Float -> Either String Float safeSqrt_134 x | x < 0 = Left "Negative Number" | otherwise = Right (sqrt x)

-- mapSqrtList_134: Apply safeSqrt_134 to list of floats
mapSqrtList_134 :: [Float] -> [Either String Float]
mapSqrtList_134 = map safeSqrt_134
```

Question 14:

```
-- Define Result type data Result a = Success a | Error String deriving (Show)

-- a. safeHead_134 safeHead_134 :: [a] -> Result a safeHead_134 [] = Error "Empty list" safeHead_134 (x:_) = Success x

-- b. safeIndex_134 safeIndex_134 :: [a] -> Int -> Result a safeIndex_134 xs i | i < 0 || i >= length xs = Error "Index out of bounds" | otherwise = Success (xs !! i)

-- c. resultToString_134
resultToString_134 :: Show a => Result a -> String
resultToString_134 (Success x) = "Success: " ++ show x
resultToString_134 (Error msg) = "Error: " ++ msg
```



Question 15:

```
-- Define binary tree type data BTree a = Empty | Node (BTree a) a (BTree a) deriving (Show)

-- a. insert_134: Insert element in BST insert_134 :: (Ord a) => a -> BTree a -> BTree a insert_134 x Empty = Node Empty x Empty
insert_134 x (Node left val right) | x < val = Node (insert_134 x left) val right | x > val = Node left val (insert_134 x right) | otherwise = Node left val right -- no duplicates
```

```
-- b. search_134: Search for a
value search_134 :: (Ord a) => a -
> BTree a -> Bool search_134 _
Empty = False search_134 x (Node
left val right) | x == val = True
| x < val = search_134 x left |
otherwise = search_134 x right

-- c. inOrder_134: In-order
traversal to list inOrder_134 ::
BTree a -> [a] inOrder_134 Empty =
[] inOrder_134 (Node left val
right) = inOrder_134 left ++ [val]
++ inOrder_134 right

-- d. treeHeight_134: Calculate
height of tree treeHeight_134 ::
BTree a -> Int treeHeight_134
Empty = 0 treeHeight_134 (Node
left _ right) = 1 + max
(treeHeight_134 left)
(treeHeight_134 right)

-- e. countNodes_134: Count nodes
countNodes_134 :: BTree a -> Int
countNodes_134 Empty = 0
countNodes_134 (Node left _ right)
= 1 + countNodes_134 left +
countNodes_134 right
```

```
ghci> -- 3
ghci> :l lab8.hs
[1 of 2] Compiling Main             ( lab8.hs, interpreted )
Ok, one module loaded.
ghci> let tree0 = Empty
ghci> let tree1 = insert_134 10 tree0
ghci> let tree2 = insert_134 5 tree1
ghci> let tree3 = insert_134 15 tree2
ghci>
ghci> search_134 10 tree3
True
ghci>
ghci> search_134 7 tree3
False
ghci>
ghci> inOrder_134 tree3
[5,10,15]
ghci>
ghci> treeHeight_134 tree3
2
ghci>
ghci> countNodes_134 tree3
3
ghci>
```