Experiment No: 05

Aim: To design and develop Haskell code for given programming problems Part 02.

Lab Objective: Design and implement declarative programs in functional and logic programming languages.

Lab Outcomes: Design and Develop solution based on declarative programming paradigm using functional and logic programming (LO2)

Requirements: Any Text Editor and Glasgow Haskell Compiler 8.0+ Version

Performance: [Note: While writing the write up student need to change the wording such that it coveys that students have done all following steps. Also where ever output is generated the output must be written by the student Sample Code need to be executed and execution steps along with output must be recorded by the students.]

Problem Statement 1: Nim Game

Nim is a mathematical game of strategy in which two players take turns removing (or "nimming") objects from distinct heaps or piles. On each turn, a player must remove at least one object, and may remove any number of objects provided they all come from the same heap or pile. Depending on the version being played, the goal of the game is either to avoid taking the last object or to take the last object.

Nim is the most famous two-player algorithm game. The basic rules for this game are as follows:

- The game starts with a number of piles of stones. The number of stones in each pile may not be equal.
- The players alternately pick up one or more stones from pile
- The player to remove the last stone wins.

For example, there are n=3 piles of stones having piles = [3,2,4] stones in them. Play may proceed as follows:

```
Initial pile=[3,2,4]
Player Takes            Leaving
1      2 from pile[1]   pile=[3,4]
2      2 from pile[1]   pile=[3,2]
1      1 from pile[0]   pile=[2,2]
2      1 from pile[0]   pile=[1,2]
1      1 from pile[1]   pile=[1,1]
2      1 from pile[0]   pile=[0,1]
1      1 from pile[1]   Player 1 WINS
```

```
Consider Following Haskell Code :
```

```
import Data.Char  -- Required for digitToInt and isDigit
```

```
{-
For simplicity the player number is
represented as an integer (1 or 2).
-}
next :: Int -> Int
next 1 = 2
next 2 = 1


{-
In turn, we represent the board as a list comprising the number of stars that
remain on each row, with the initial board given by the list [5,4,3,2,1] and the
game being finished when all rows have no stars left.
-}
type Board = [Int]
initial :: Board
initial = [5,4,3,2,1]

finished :: Board -> Bool
finished = all (== 0)


{-
A move in the game is specified by a row number and the number of stars to be
removed, and is valid if the row contains at least this many stars.
Example:
-- The first row on the initial board contains at least 3 stars
> valid initial 1 3
True
-- The 4th row contains fewer than 3 stars
> valid inital 4 3
False
-}
valid :: Board -> Int -> Int -> Bool
valid board row num = board !! (row - 1) >= num
{-
A valid move can then be applied to a board to give a new board by using a list
comprehension
```

to update the number of stars taht remain in each row.
Example:
-- 3 stars have been removed in the 1 row
> move inital 1 3
[2,4,3,2,1]
-}

```
move :: Board -> Int -> Int -> Board
move board row num = [update r n | (r, n) <- zip [1..] board]
 where update r n = if r == row then (n - num) else n


-- IO Utils
putRow :: Int -> Int -> IO ()
putRow row num = do putStr (show row)
 putStr ": "
 putStrLn (concat (replicate num "* "))


putBoard :: Board -> IO ()
putBoard [a,b,c,d,e] = do putRow 1 a
 putRow 2 b
 putRow 3 c
 putRow 4 d
 putRow 5 e


getDigit :: String -> IO Int
getDigit prompt = do putStr prompt
 x <- getChar
 newline
 if isDigit x then
 return (digitToInt x)
 else
 do putStrLn "ERROR: Invalid digit"
 getDigit prompt


newline :: IO ()
```

```
newline = putChar '\n'
-- Game of nim
play :: Board -> Int -> IO ()
play board player = do newline
 putBoard board
 if finished board then
 do newline
 putStr "Player "
 putStr (show (next player))
 putStrLn " wins!"
 else
 do newline
  putStr "Player "
  putStrLn (show player)
  row <- getDigit "Enter a row number: "
  num <- getDigit "Stars to remove: "
  if valid board row num then
  play (move board row num) (next player)
  else
  do newline
   putStrLn "ERROR: Invalid move"
   play board player
nim :: IO ()
nim = play initial 1
```

Problem Statement 02 :

Part 02: Write Haskell code to create a simple calculator that performs binary operations of add, sub, multiply, exponentiation, div as per user choice.

```
calc :: (Integral a, Num a) => a -> a -> Char -> Maybe a
calc x y op
 | op == '+' = Just (x+y)
 | op == '-' = Just (x-y)
 | op == '*' = Just (x*y)
 | op == '^' = Just (x^y)
 | otherwise = Nothing
```

[**Note:** Students must create appropriate main function for calculator above]

Conclusion: Thus we have understood how to create functional solution to programming

problems using Haskell.

Reference:

1. Glasgow Haskell Project Home Page. https://www.haskell.org/
2. Learn You a Haskell for Great Good ! A Beginner's Guide http://learnyouahaskell.com/
3. Michael L Scott, 'Programming Language Pragmatics', 3rd Edition, Elsevier Publication.