universität innsbruck

Functional Programming	WS 2023/2024	LVA 703025
Exercise Sheet 10, 10 points		Deadline: Tuesday, January 9, 2024, 8pm

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_10.tgz` provided on the proseminar page.
- Your `*.hs` files must be compilable with `ghc`.
- Upload your `*.hs` files in OLAT. Please upload each file separately and do *not* use zip or upload any build files (`*.exe`, `*.o`, `*.hi`, etc.)

Exercise 1 *I/O* **4 p.**

1. Implement a stand-alone Haskell program that prints the name of all .txt files in the current directory prefixed by a number, starting from 1. For instance, if the current directory contains files

   ```
   Main.hs
   foo.txt
   whatever.jpg
   bar.txt
   ```

   then starting the program from the command line should print the following two lines:

   ```
   1: foo.txt
   2: bar.txt
   ```

   Try to separate the I/O task and formatting the list of file names into different functions. You might find it helpful to define auxiliary functions `filterTxtfiles :: [FilePath] -> [FilePath]` and `formatFiles :: [FilePath] -> String`. (1.5 points)

   Hint: A Haskell function that returns a list of files in a given directory can be found in the `System.Directory` documentation.[1] The function `isSuffixOf`[2] might be useful to check whether a file ends in `".txt"`.

2. Add interaction to your program. First, print the .txt files in the current directory, as in the previous task, then read a number input by the user. If the user inputs 0, the program should quit, otherwise the contents of the .txt file with the given index should be printed.

   You may assume that the user always gives a valid input and that the functionality from exercise 1.1 is available. An example program run could look like this:

   ```
   ... program starts ...
   1: foo.txt
   2: bar.txt
   Input 0 to quit or a file number to view the file contents:
   1
   ... print the contents of foo.txt ...
   ```

   (2 points)

3. Extend your program so that after a file has been printed, the user is asked to proceed by pressing the enter key. After pressing enter, the program should start again. You may assume that the functionality from question 1.2 is available even if you did not solve this exercise.

[1] https://hackage.haskell.org/package/directory-1.3.8.1/docs/System-Directory.html
[2] https://hackage.haskell.org/package/base-4.19.0.0/docs/Data-List.html#v:isSuffixOf

```
... program starts ...
1: foo.txt
2: bar.txt
Input 0 to quit or a file number to view the file contents:
1
... print the contents of foo.txt ...
Press enter to restart
... user presses enter ...
1: foo.txt
2: bar.txt
Input 0 to quit or a file number to view the file contents:
0
... program quits ...
```

(0.5 points)

Solution 1

```haskell
module Main where

import System.Directory (listDirectory)

{-
-- Solution for task 1.1
main :: IO ()
main = do
  files <- listDirectory "." -- Get files in current directory
  putStr $ formatFiles (filterTxtfiles files)

-}

-- Solution for task 1.2 and 1.3
main :: IO ()
main = do
  files <- listDirectory "." -- Get files in current directory
  let txtFiles = filterTxtfiles files
  putStr $ formatFiles txtFiles
  putStrLn "Input 0 to quit or a file number to view the file contents:"
  instructionStr <- getLine
  let n = (read instructionStr :: Int)
  if n == 0
    then return ()
    else do
      putStrLn (txtFiles !! (n - 1))
      fileContents <- readFile (txtFiles !! (n - 1))
      putStrLn fileContents
      putStrLn "Press enter to restart"
      getLine
      main

filterTxtfiles :: [FilePath] -> [FilePath]
filterTxtfiles = filter ((== "txt.") . take 4 . reverse)

formatFiles :: [FilePath] -> String
formatFiles fs = unlines $ [show i ++ ": " ++ f | (i, f) <- zip [1 ..] fs]
```

Exercise 2 *Connect Four* **6 p.**

In this exercise, you will extend the implementation of Connect Four from the lecture in various ways. Note that all subtasks can be solved independently of one another.

1. The user interface does not check whether input moves are valid: it is not checked whether the input from the user really is a number or whether an input number is a valid move. Both cases may lead to unintended behavior or crashes of the program. Therefore, you should modify the user interface so that it repeatedly asks for input until a valid move has been entered, e.g. as follows:

   ```
   Choose one of [0,1,2,3,4,5,6]: five
   five is not a valid move, try again: 8
   8 is not a valid move, try again: 3
   ... accept and continue ...
   ```

   (2 points)

2. Extend the Connect Four implementation so that it can save and load games, e.g. in a file `connect4.txt`. The user interface might look like this:

   ```
   Welcome to Connect Four
   (n)ew game or (l)oad game: l
   ... game starts by loading state from connect4.txt ...
   Choose one of [0,2,3,5,6] or (s)ave: s
   ... game is saved in file connect4.txt and program quits ...
   ```

   For the implementation, note that `read . show = id` and that one can automatically derive `Read` instances in datatype definitions via `deriving Read`. (2 points)

3. Modify the function `winningPlayer` in the game logic so that diagonals are also taken into account. (2 points)

Solution 2

```haskell
{- user interface with I/O -}
module Main(main) where

import Text.Read
import System.IO
import Logic

file :: FilePath
file = "connect4.txt"

main :: IO()
main = do
  hSetBuffering stdout NoBuffering
  putStrLn "Welcome to Connect Four"
  newOrLoad "(n)ew game or (l)oad game: "

newOrLoad :: [Char] -> IO ()
newOrLoad str = do
  putStr str
  answer <- getLine
  if elem answer ["n","new"] then newGame
  else if elem answer ["l","load"] then loadGame
  else newOrLoad "unknown answer, please type \"n\" or \"l\": "

loadGame :: IO()
loadGame = do
  stateStr <- readFile file
  let state = read stateStr
```

```
  game state

newGame :: IO()
newGame = game initState

game :: State -> IO ()
game state = do
  putStrLn $ showState state
  case winningPlayer state of
    Just player -> putStrLn $ showPlayer player ++ " wins!"
    Nothing -> let moves = validMoves state in
      if null moves then putStrLn "Game ends in draw."
      else do
        codeMove <- getMove state moves
        case codeMove of
          Right move -> game (dropTile state move)
          Left code ->
            if code == 's'
            then saveGame state
            else error $ "unknown internal code: " ++ [code]

saveGame :: State -> IO ()
saveGame state = do
  writeFile file $ show state
  putStrLn "Game saved. Goodbye!"

getMove :: State -> [Move] -> IO (Either Char Move)
getMove state moves = do
  putStr $ "Choose one of " ++ show moves ++ " or (s)ave game: "
  moveStr <- getLine
  if moveStr `elem` ["s","save"] then return $ Left 's'
  else case extractMove moveStr of
    Nothing -> do
      putStrLn $ moveStr ++ " is not a valid move"
      getMove state moves
    Just move -> return (Right move)
  where
    extractMove moveStr = do
      move <- readMaybe moveStr
      if move `elem` moves then return move else Nothing


{- logic of Connect Four -}

module Logic(State, Move, Player,
  initState, showPlayer, showState,
  winningPlayer, validMoves, dropTile) where

type Tile   = Int   -- 0, 1, or 2
type Player = Int   -- 1 and 2
type Move   = Int   -- column number
data State = State Player [[Tile]]  deriving (Show,Read)-- list of rows

empty :: Tile
empty = 0

numRows, numCols :: Int
numRows = 6
```

```haskell
numCols = 7

startPlayer :: Player
startPlayer = 1

initState :: State
initState = State startPlayer
  (replicate numRows (replicate numCols empty))

otherPlayer :: Player -> Player
otherPlayer = (3 -)

dropTile :: State -> Move -> State
dropTile (State player rows) col = State
  (otherPlayer player)
  (reverse $ dropAux $ reverse rows)
    where
      dropAux (row : rows) =
        case splitAt col row of
         (first, i : last) ->
           if i == empty
             then (first ++ player : last) : rows
             else row : dropAux rows

validMoves :: State -> [Move]
validMoves (State _ rows) =
  map fst . filter ((== empty) . snd) . zip [0..] $ head rows

showPlayer :: Player -> String
showPlayer 1 = "X"
showPlayer 2 = "O"

showTile :: Tile -> Char
showTile t = if t == empty then '.' else head $ showPlayer t

showState :: State -> String
showState (State player rows) =
  unlines $ map (head . show) [0 .. numCols - 1] :
    map (map showTile) rows
     ++ ["\nPlayer " ++ showPlayer player ++ " to go"]

transposeRows ([] : _) = []
transposeRows xs = map head xs : transposeRows (map tail xs)

zipr :: (a -> b -> b) -> [a] -> [b] -> [b]
zipr f (x:xs) (y:ys) = f x y : zipr f xs ys
zipr _ _ ys = ys

diags :: [[a]] -> [[a]]
diags [xs] = map (\x -> [x]) xs
diags ((x:xs):xss) = [x] : zipr (:) xs (diags xss)
diags _ = []

diagonals :: [[a]] -> [[a]]
diagonals xs = diags xs ++ diags (reverse xs)

winningLine :: Player -> [Tile] -> Bool
winningLine player [] = False
```

```haskell
winningLine player row = take 4 row == replicate 4 player
  || winningLine player (tail row)

winningPlayer :: State -> Maybe Player
winningPlayer (State player rows) =
  let oplayer = otherPlayer player
      longRows = rows ++ transposeRows rows ++ diagonals rows
    in if any (winningLine oplayer) longRows
      then Just oplayer
      else Nothing
```