universität innsbruck

Functional Programming	WS 2021	LVA 703025
Exercise Sheet 10, 10 points		Deadline: Wednesday, January 12, 2022, 6am

- 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 should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Connect Four* **10 p.**

In this exercise we want to extend the implementation of Connect Four from the lecture in various ways. Note that all sub-tasks can be solved independently.

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, and whether this number is a valid move. Both cases may lead to unintended behavior or crashes of the programs. Therefore, you should modify the user-interface in a way 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. Modify the user interface so that after a match has been completed, it asks whether another round should be played. If so, the starting player should be switched. Clearly, this also requires a change in the type of `initialState`. (2 points)

3. Extend the implementation so that it can save and load games, e.g., via 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. (2 points)

4. Modify the function `winningPlayer` in the game logic, so that also diagonals are taken into account.

   (2 points)

5. Extend the implementation so that it can give hints. To be more precise, the user interface should inform the current player, whenever she can win within 1 or 2 moves by providing a hint. Winning in 2 moves means that after following the move from the hint, you will win the game no matter how the opponent moves in between your moves. In that case the player can type "`h`" to see a first move that leads to success.

   ```
   Choose one of [0,2,3,5,6] or see (h)int to win within 2 moves: h
   Hint: Drop a piece in column 2
   Choose one of [0,2,3,5,6]: 3
   ... the game continues since the user is not forced to follow hints
   ```

   (2 points)

Solution 1

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

import Text.Read
import System.IO
import Logic

file :: FilePath
file = "connect4.txt"

startPlayer :: Player
startPlayer = 1

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 startPlayer
  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 (startPlayer,state) = read stateStr
  game startPlayer state

newGame :: Player -> IO()
newGame startPlayer = game startPlayer (initState startPlayer)

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

```
saveGame :: Player -> State -> IO ()
saveGame startPlayer state = do
  writeFile file $ show (startPlayer, state)
  putStrLn "Game saved. Good Bye!"

getMove :: State -> [Move] -> IO (Either Char Move)
getMove state moves = let
  hint = winMoves state
  (hintStr, hintStr2) = case hint of
    Nothing -> ("", "I don't have any hints for you")
    Just col -> (", (h)int", "Hint: Drop a piece in column " ++ show col)
 in do
  putStr $ "Choose one of " ++ show moves ++ hintStr ++ " or (s)ave game: "
  moveStr <- getLine
  if moveStr `elem` ["s","save"] then return $ Left 's'
  else if moveStr `elem` ["h", "hint"] then do
    putStrLn hintStr2
    getMove state moves
  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

anotherRound :: Player -> IO ()
anotherRound startPlayer = anotherRoundMain
  "Another round, (y)es or (n)o: "
 where
  anotherRoundMain str = do
    putStr str
    answer <- getLine
    if elem answer ["y", "yes"]
      then newGame (otherPlayer startPlayer)
      else if elem answer ["n", "no"]
        then return ()
        else
          anotherRoundMain
          $  "I don't understand \""
          ++ answer
          ++ "\", please type \"y\" or \"n\": "

{- logic of Connect Four -}

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

import Data.List
import Data.Maybe


type Tile   = Int   -- 0, 1, or 2
type Player = Int   -- 1 and 2
type Move   = Int   -- column number
```

```haskell
data State = State Player [[Tile]]  deriving (Show,Read)-- list of rows

empty :: Tile
empty = 0

numRows, numCols :: Int
numRows = 6
numCols = 7

initState :: Player -> State
initState startPlayer = 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)

diagonals :: [[Tile]] -> [[Tile]]
diagonals []   = []
diagonals rows@(row : remRows) =
  if length row < winNum || length rows < winNum then
    []
  else
    checkableCs ++ diagonals remRows
  where
    winNum          = 4
    numCheckableCs = numCols - winNum
```

```
    extract          = zipWith (!!) rows
    checkableCs      =
      concatMap (\c -> let ps = map (+ c) [0 .. winNum-1]
                       in [extract ps, extract $ reverse ps])
        [0 .. numCheckableCs]

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

winMove :: State -> Move -> Bool
winMove state = isJust . winningPlayer . dropTile state

winMoves :: State -> Maybe Move
winMoves state = find (checkOnlyWins . dropTile state) moves
 where
  moves = validMoves state
  checkOnlyWins state =
    let opMoves = validMoves state
    in  (isJust $ winningPlayer state)
          || all
               (\m ->
                 let afterOp         = dropTile state m
                     pMovesAfterOp = validMoves afterOp
                 in  (isNothing $ winningPlayer afterOp)
                       && (any (winMove afterOp) pMovesAfterOp)
               )
               opMoves && not (null opMoves)
```