Tic Tac Toe

COMP40009 – Computing Practical 1

8th – 12th November 2021

Aims

- To provide experience of working with Haskell I/O and monads.
- To build a simple interactive system for playing Tic Tac Toe on an $N \times N$ board.

Introduction

This is the final Haskell exercise and is designed to give you the opportunity to experiment with monads, and monadic I/O in particular. The exercise involves implementing a generalisation of the classic "Tic Tac Toe" game on a board of size $N \times N$ – traditionally it's played on a 3×3 board.

Because I/O is hard to test the autotester will only be applied to the various utility functions in Part I. If you're up to speed on Haskell thus far, you should find these quite easy to define. The main technical challenge is to get the I/O right (Part II). You can make the user interaction as simple or as elaborate as you wish. The specification that follows assumes simple text-based I/O; you don't need to follow it exactly, but it will give you some ideas.

Important: You will not need to implement I/O for any of the Haskell tests, so if you get stuck on the I/O, or run out of time, don't worry. With that said, monads are an important concept in Computing, so you should to use the exercise to try to understand the basics, at least.

Tic Tac Toe

You've doubtless played this game before: two players take it in turns to place a mark on one of the empty squares in a 3×3 grid. Traditionally the two players use the marker symbols 'O' and 'X' respectively, hence the alternative name "noughts and crosses". The game is won if one of the players forms a line of three of their own symbol, either horizontally, vertically or along one of the two diagonals. In this version of the game we'll apply the same principles to an $N \times N$ board, where $N \geq 1$[1].

Your program should allow the two players to move in turn and should display the board state after each move. You'll need to read in and check each move to ensure that it defines a valid, and empty, location on the board. You'll also need to check whether a player has won after each move, terminating the program if they have.

Representing the board

The board will be represented by a pair comprising a list of N^2 `Cells` and the board's size, N, an `Int`. The cells are stored in "row major" order, e.g. the first N elements correspond to the first row on the board and are indexed $(0, 0)$, $(0, 1)$, ..., $(0, N-1)$; each board `Position` is thus a pairs of integers. Initially all cells are `Empty`. The two players are identified by the constructors `O` and `X` and player `X` is assumed to go first.

The following types are declared in the skeleton file:

```
data Player = O | X
            deriving (Eq, Show)

data Cell = Empty | Taken Player
          deriving (Eq, Show)

type Board = ([Cell], Int)

type Position = (Int, Int)
```

Getting started

As per the previous exercise, you will use the `git` version control system to get the repository with the skeleton files for this exercise and its (incomplete) test suite. You can get your repository with the following (remember to replace the *`username`* with your own username).

[1]You might note that for $N = 1$ and $N = 2$ the first player to go is guaranteed to win, so they are not very interesting! But you should support them anyway.

```
git clone https://gitlab.doc.ic.ac.uk/lab2122_autumn/haskelltictactoe_username.git
```

You will notice that the skeleton files have been zipped and encrypted with a password. In order to extract them, you will need to use an application that supports 7z files[2]. On the lab machines, this has already been installed for you, so you can extract the archive by typing:

```
7z x skel.7z -p9E010CED
```

(*n.b., 9E010CED is the password*). Make sure that you extract the contents of the archive directly inside your repo, i.e. do not create an extra subfolder named "skel", because that would break the LabTS autotesting process. Once this is done, you can then safely delete the 7z archive. Remember to commit and push all the extracted files back into the repo.

A number of functions from, or based on, the unassessed problem sheets turn out to be quite useful here, so their definitions are included in the skeleton file. In summary:

```
-- Preserves Just x iff x satisfies the given predicate. In all other cases
-- (including Nothing) it returns Nothing.
filterMaybe :: (a -> Bool) -> Maybe a -> Maybe a

-- Replace nth element of a list with a given item.
replace :: Int -> a -> [a] -> [a]

-- Returns the rows of a given board.
rows :: Board -> [[Cell]]

-- Returns the columns of a given board.
cols :: Board -> [[Cell]]

-- Returns the diagonals of a given board.
diags :: Board -> [[Cell]]
```

Various sample boards have also been defined for testing purposes.

[2] https://www.7-zip.org/download.html

What to do

Part I: Utilities

The first part of the exercise involves writing some utility functions that will be useful when implementing the game (Part II).

- Define a function `gameOver :: Board -> Bool` that returns `True` if a given board represents a winning situation; `False` otherwise. Use the `rows`, `cols` and `diags` functions above and note that to win, one or more of the rows, columns and diagonals must contain either all `X`s or all `O`s. For example,

  ```
  *Main> gameOver testBoard1
  True
  *Main> gameOver testBoard2
  False
  *Main> gameOver testBoard3
  True
  ```

  Hint: Try using `nub` from `Data.List`: if every element of a list is the same then `nub` will return a singleton. **Do not use `length` to check for singletons!**

- Define a function `parsePosition :: String -> Maybe Position` that, given an input string determines whether it can be parsed as two integers. The result is equivalent to a `Maybe (Int, Int)` so if the string cannot be parsed as two integers it should return `Nothing`. Note that it is not necessary to check whether the integers represent a valid board position – that will be done by `tryMove` below. For example,

  ```
  *Main> parsePosition "0 2"
  Just (0,2)
  *Main> parsePosition "-2 15"
  Just (-2,15)
  *Main> parsePosition "garbage 3"
  Nothing
  ```

  Hint: A function you might find useful is `readMaybe` from module `Text.Read`, which is imported in the skeleton. This will attempt to parse a string as an object of some type `a` that's a member type of class `Read`. It returns `Nothing` if the parse fails. For example:

```
*Main> :t readMaybe
readMaybe :: Read a => String -> Maybe a
*Main> readMaybe "True" :: Maybe Bool
Just True
*Main> readMaybe "123" :: Maybe Int
Just 123
*Main> readMaybe "True" :: Maybe Int
Nothing
```

- Define a function `tryMove :: Player -> Position -> Board -> Maybe Board` that will try to write the given player's marker (`X` or `O`) at the given board position. Recall that the board is represented in row-major order, so position (i, j) on a board of size $N \times N$ corresponds to index $i \times N + j$ in the list. If the player enters a position that lies outside the bounds of the board, or if the position is already occupied then the function should return `Nothing`. Otherwise it should `replace` the given board position with the current player's marker. If successful the value returned should be (`Just b`) where `b` is the updated board. For example,

```
*Main> testBoard2
([Taken X,Empty,Empty,Empty],2)
*Main> tryMove X (0,0) testBoard2
Nothing
*Main> tryMove O (-1,2) testBoard2
Nothing
*Main> tryMove O (1,1) testBoard2
Just ([Taken X,Empty,Empty,Taken O],2)
```

Part II: Game management and I/O

You are now in a position to implement the game, so all the remaining functions will need to do I/O. To start this off, define a function `prettyPrint :: Board -> IO ()` that will print a given board. You can do this any way you like, but something like this will do fine:

```
*Main> testBoard1
([Taken O,Taken X,Empty,Taken O,Taken O,Empty,Taken X,Taken X,
Taken O,Empty,Empty,Taken X,Taken O,Taken X,Empty,Empty],4)
*Main> prettyPrint testBoard1
O X - O
```

```
O - X X
O - - X
O X - -
```

If you're feeling ambitious you could use fancy unicode characters to print the board in the form of a grid.
Hint: Use `rows` above and the prelude function `putStrLn`. You may also find the function `intersperse` (from `Data.List`) useful, e.g.

```
*Main> intersperse ' ' "Hello"
"H e l l o"
*Main> intersperse '.' (intersperse ' ' "Hello")
"H. .e. .l. .l. .o"
```

Managing the game

You can implement the game management in any way you see fit, but the suggested approach is to implement the following functions:

```
-- Repeatedly read a target board position and invoke tryMove until
-- the move is successful (Just ...).
takeTurn :: Board -> Player -> IO Board

-- Manage a game by repeatedly: 1. printing the current board, 2. using
-- takeTurn to return a modified board, 3. checking if the game is over,
-- printing the board and a suitable congratulatory message to the winner
-- if so.
playGame :: Board -> Player -> IO ()

-- Print a welcome message, read the board size, invoke playGame and
-- exit with a suitable message.
main :: IO ()
```

Hints:

- Reading the board size and each individual move may involve repetition, e.g. if the user enters unparsable input or an invalid board position. It's a good idea to abstract this repeated 'try action until successful' pattern in the form of a higher-order function. One way to do this is to define something like:

  ```
  doParseAction :: (String -> Maybe a) -> IO a
  ```

The idea here is to repeatedly read input from the keyboard using `getLine` and apply the given function to it until the function 'succeeds', by returning something of the form `Just ...`. The function will (presumably) try to parse, and possibly also check the validity of, the given input. If it gives `Nothing` you print an error message and try again (recursion); otherwise you `return` the value read. Note that the returned value (wrapped in a `Just`) will be an `Int` if you're reading a board size, a pair of `Int`s, i.e. a `Position` if you're reading a board position, or even a new `Board` if your action combines parsing (`parsePosition`) and moving (`tryMove`); the latter works just fine, as the monadic composition of the two is also a `Maybe`. For example:

```
*Main> n <- doParseAction readMaybe :: IO Int
120
*Main> n
120
*Main> n <- doParseAction readMaybe :: IO Int
xyz
Invalid input, try again: 0
*Main> n
0
```

If you want you can tidy up the error messages by passing an additional `String` parameter that contains all, or part of, the "try again" message.

- Remember that you're now using monads, and that `Maybe`s are themselves monads, so you can use `do` notation, or `>>=`, to chain together functions that return `Maybe`s, should you need to.

- Beware of 'mixing your monads'! If a function has return type `IO a` then every expression in its `do` block must itself return something of the form `IO b` (the `a` and `b` don't have to match), otherwise the expressions cannot be composed (type error). If you need to invoke functions that manipulate other monads, e.g. `Maybe`s, you have to nest them, e.g. by doing something like:

```
do
  <IO actions>
  let m = do <Maybe actions>
  <more IO actions>
```

Alternatively you can use `>>=` instead of the nested `do`. After the `let` you can test `m` if you need to and perform different I/O actions depending on the result, e.g.

```
maybe (do <IO actions 1>) (do <IO actions 2>) m
```

Yes, you can inline the `m`, of course, if you don't want/need the `let`. Remember that `do` blocks are just syntactic sugar for expressions built using `>>=` and `>>`.

Here's an example of a simple interaction for a 2×2 board:

```
*Main> main
Welcome to tic tac toe on an N x N board
Enter the board size (N): oops
Invalid input, try again: 0
Invalid input, try again: 2
- -
- -
Player X, make your move (row col): 0 2
Invalid move, try again: 0 0
X -
- -
Player O, make your move (row col): A 1
Invalid move, try again: 0 1
X O
- -
Player X, make your move (row col): 1 1
X O
- X
Player X has won!
Thank you for playing
```

Submission

As with all previous exercises, you will need to use the commands `git add`, `git commit` and `git push` to send your work to the GitLab server. Then, as always, log into the LabTS server, `https://teaching.doc.ic.ac.uk/labts`, click through to your `HaskellTicTacToe` exercise `https://gitlab.doc.ic.ac.uk/lab2122_autumn/haskelltictactoe` and request an autotest of your submission.

IMPORTANT: Make sure that you submit the correct commit to CATe – you can do this by checking that the key submitted to CATe matches the `Commit Hash` of your commit on LabTS/GitLab.

Assessment

In general, the assessment for laboratory exercises uses the following scheme:

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```