Tic-Tac-Toe Analysis

Haskell Chapter 7

Brennan W. Fieck

Department of Engineering Physics
Department of Electrical Engineering and Computer Science
Colorado School of Mines

CSCI400 - Programming Languages November 2016

Tic-Tac-Toe

Rather than go over the entire program line for line in the order that functionality appears, I'll start by explain some data structures used and then step chronologically through the logic used to run the game.

1.1 Preliminary Logic

Firstly, we have to have a way to uniquely identify the two players playing the game. This tic-tac-toe game does so using an enumerable datatype called `Player` which can have only the values `One` and `Two`.

```
data Player = One | Two
    deriving(Eq, Show, Read, Bounded, Enum)
```

Listing 1.1 Player Datatype Definition

Simple enough, a player is always either Player One or Player Two, and the datatype derives from other typeclasses that allow the value of a `Player` to be converted to a string, compared to other `Players`, and to be incremented and decremented.
To identify the different places a player may place either an 'X' or an 'O', a type exists to enumerate all of the different available spaces, named `Positions`.

```
data Position = TL | TM | TR | ML | MM | MR | BL | BM | BR
    deriving (Eq, Show, Read, Bounded, Enum)
```

Listing 1.2 Position Datatype

Each of these represents a single box; the first letter denotes the row and the second the column ('T'=top, 'M'=middle, 'R'=right etc.). At this point, we have enough types to fully define a move, which after all consists only of a player selecting a space. A move, therefore is aliased by a tuple that packs this information.

```
type Move = (Player, Position)
```

Listing 1.3 Move Type Macro

This next bit is kind of long, and can get a little confusing, but it actually is trying to say something very simple. It deals with defining a `Game` and it's state as either `Finished` or `Unfinished`.

```
data Unfinished = Unfinished [Move]
data Finished = Finished [Move]

-- this Either is so that 'move' can
-- return a potentially finished game
type Game = Either Unfinished Finished

class TicTacToe a where
    playerAt :: a -> Position -> Maybe Player
instance TicTacToe Unfinished where
    playerAt (Unfinished moves) = findPlayer moves
instance TicTacToe Finished where
    playerAt (Finished moves) = findPlayer moves

instance Show Finished where
    show (Finished moves) = showBoard moves
instance Show Unfinished where
    show (Unfinished moves) = showBoard moves
```

Listing 1.4 Gamestate Class and Datatypes

Simply put, whether the game is `Finished` or `Unfinished`, you can fully describe it as the list of all the moves made by either player. The `TicTacToe` class can be any kind of `Game` (`Either Unfinished` or `Finished`), and asks only that types that instantiate it be able to find what player has placed their 'X' or 'O' at some given `Position` by implementing its member function `playerAt` (if any; `Maybe` no one has put an 'X' or an 'O' at that position yet). The last two `instances` only serve to describe how Haskell should `show` a `Finished` or `Unfinished Game`, which is accomplished by showing the board (more on that later).

1.2 Main Play

The game begins by entering at the `main` function, which does nothing more than call the `playGame` function that returns a `Finished game`, then prints the winner and exits.

```
main :: IO ()
main = do
    game <- playGame newGame
    putStrLn $ "Winner is: " ++ show (whoWon game)
```

```
    print game
```

Listing 1.5 Main Function

`newGame` serves as basically a "constructor" for a `Game`, that just returns an `Unfinished Game` with no moves having been made by any player.

```
newGame :: Unfinished
newGame = Unfinished []
```

Listing 1.6 newGame Function

The real backbone of the game is the `playGame` function, but before that can be described, it behooves us to discuss the last line of `main`, because it's the same as the first line of `playGame`: `print game`.

1.2.1 Showing the Board

Recall from Section 1.1 that `showing` a `Game` (which `print` implicitly does) entails a call to the `showBoard` function.

```
showBoard :: [Move] -> String
showBoard moves =
        "+---+---+---+\n" ++
        "| " ++ showCell TL moves ++ " | " ++ showCell TM moves ++ "
           | " ++ showCell TR moves ++ " |\n" ++
        "+---+---+---+\n" ++
        "| " ++ showCell ML moves ++ " | " ++ showCell MM moves ++ "
           | " ++ showCell MR moves ++ " |\n" ++
        "+---+---+---+\n" ++
        "| " ++ showCell BL moves ++ " | " ++ showCell BM moves ++ "
           | " ++ showCell BR moves ++ " |\n" ++
        "+---+---+---+\n"

showCell :: Position -> [Move] -> String
showCell p ms = case (findPlayer ms p) of
        Nothing -> " "
        Just One -> "X"
        Just Two -> "O"
```

Listing 1.7 showBoard Function

This pretty clearly just prints out a tic-tac-toe board with a cute little border, and each cell of the board is filled with the output from `showCell`, passing all of the moves in the game and the position of the cell as arguments. `showCell` then passes these arguments along to

`findPlayer`, which will return the player that has marked that cell, or `Nothing` if the cell is unclaimed. Then the cell can be blank, or filled with the player's respective mark.

```
findPlayer :: [Move] -> Position -> Maybe Player
findPlayer moves position
    | null move = Nothing
    | otherwise = Just $ (fst . head) move
    where
        move = filter ((== position) . snd) moves
```

Listing 1.8 findPlayer Function

The first thing `findPlayer` does is `filter` the list of moves made in the game for those that were made concerning the same cell as the one passed as the `position` parameter (recall from Listing 1.3 in Section 1.1 that the form of a `Move` is a tuple containing (`Player`,`Position`), in that order). If the list is empty, then no player has made a move to mark this cell, so the function returns `Nothing`. Otherwise, it's assumed that only one move has been regarding this cell (handled in the `move` function), and so the `Player` part of that value is returned.

1.2.2 Anatomy of a Turn

Now that the display has been explained, it'll be easier to describe what is actually happening during a single turn of this game. Once again: the backbone of the game is the `playGame` function.

```
playGame :: Unfinished -> IO Finished
playGame game = do
    print game
    position <- getPosition
    let nextGame = move game position
    case nextGame of
      Just goodGame ->
          case goodGame of
              Left unfinishedGame ->
                  playGame unfinishedGame
              Right finishedGame ->
                  return finishedGame
        Nothing ->
        do
          invalidMove
        playGame game
```

Listing 1.9 playGame Function

So as just a broad overview, this function apparently gets a position from a player, then gets a new `Game` via the `move` function, passing in the retrieved position, the current `Game` containing the list of all moves that have been made. Next, it seems that if the position entered by the player represents an invalid move, the `move` function returns `Nothing`, in which case the game simply prints an error, then continues to play the game.

```
invalidMove :: IO ()
invalidMove = putStrLn "Invalid move."
```

Listing 1.10 invalidMove Function

However, if the function returns something that is not `Nothing`, it will be a `Game` (which can be inferred from the fact that this return value goes on to be both passed as an argument to and returned from `playGame`, which has type `Unfinished -> IO Finished`), and furthermore if this `Game` uses its `Left` constructor it is an `Unfinished Game` so the game must continue. On the other hand, if the `Game` uses its `Right` constructor, it is a `Finished Game`, and so it's ready to be returned so that a winner can be declared. It may help to recall the form of the constructor for a `Game` from Listing 1.4 in Section 1.1. All that remains to be seen is how a position is retrieved from a player, and how this position is used to make a move.
Before a move can be made, the position of the move must be decided by the player and validated by the game.

```
getPosition :: IO Position
getPosition = do
    putStrLn "Please enter move:"
    rawPos <- try getLine
    case rawPos of
        Left (SomeException e) ->
            do
            putStrLn "Error. "
            getPosition
        Right rawPosStr ->
            case (reads rawPosStr) of
                [] ->
                    do
                    putStrLn "Invalid position."
                    printValidMoves
                    getPosition
                [(goodPos,_)] ->
                    return goodPos
```

Listing 1.11 getPosition Function

This function is simple enough, first it prompts the user for input, then it attempts to retrieve said input and interpret it as a `Position`, then it tries to validate it. If an error occurs when reading input, the function simply asks again for input. If the input is read successfully, but fails to resolve to a valid `Position`, then it outputs a list of valid moves, which is just the list of all positions.

```
validMoves :: [Position]
validMoves = [minBound .. maxBound]
```

Listing 1.12 validMoves Function

In the final case that the input is read successfully, AND it corresponds to a valid `Position`, then and only then does the function return.
Now that a position has been read in, the game attempts to make a move with it using the `move` function.

```
-- applies a move to a game
-- must only accept unfinished games
move :: Unfinished -> Position -> Maybe Game
move (Unfinished []) position = Just $ Left (Unfinished [(One,
   position)])
move game@(Unfinished moves) position
    | not isValid = Nothing
    | isFinished (Unfinished moveList) = Just $ Right (Finished
       moveList)
    | otherwise = Just $ Left (Unfinished moveList)
    where
        isValid = validMove game position
        player = whoseTurn game
        moveList = (player, position):moves
```

Listing 1.13 move Function

The first definition for `move` applies to games that have just started, so it saves some time by optimizing this function call to not check if the move has already been made (because that would be impossible). If some moves have already been made in the game, then the program first checks if the proposed move is valid, in that the cell in question has not already been claimed.

```
validMove :: Unfinished -> Position -> Bool
validMove (Unfinished moves) position = all ((/= position) . snd)
   moves
```

Listing 1.14 validMove Function

This is easily accomplished by checking that none of the moves already made affected the cell in the proposed move. If such a move *has* already been made, then the function returns `Nothing`, which, as discussed earlier in this section, signals to the `playGame` function that the move is invalid.
If the move *is* valid, then it's possible that the game has been won, so victory conditions must be checked via the `isFinished` function.

```
isFinished :: Unfinished -> Bool
isFinished (Unfinished []) = False
isFinished (Unfinished moves@((player,_):_)) =
    any null (foldl removeMove winningPatterns lastPlayersMoves)
    where
        lastPlayersMoves = map snd $ filter ((== player) . fst) moves
        removeMove winningMoves move = map (filter (/= move))
           winningMoves
```

Listing 1.15 isFinished Function

At first glance, this looks somewhat complex, but it's not too difficult to puzzle out what's happening with a little effort. The functions output is a left-fold over `lastPlayersMoves`. Since every move is prepended to the list of moves in a game with each successive move (see Listing 1.13), the most recent move's respective player is easily pattern-matched, and then a filter is run that extracts all the `Moves` that were played by that player from the moves in the game and a further mapping extracts only the `Position` data from each of those moves. Now, this list is folded with the accumulator `winningPatterns`, which is simply a list of all the possible lists of `Positions` that result in a tic-tac-toe victory.

```
winningPatterns :: [[Position]]
winningPatterns = [[TL,TM, TR], [ML, MM, MR], [BL, BM, BR],
    [TL, ML, BL], [TM, MM, BM], [TR, MR, BR], [TL, MM, BR],
    [TR, MM, BL]]
```

Listing 1.16 winningPatterns

The function used to fold `lastPlayersMoves` with these winning combinations is defined by first filtering each of the currently accumulated winning move sets such that a move picked from the player's moves is not in any of them. Then, this filtered list of sets replaces the old one so that by the end, if any of those sets are empty, it means that the player has completed all of the moves necessary to win with that pattern.
If it is determined that the game is over, a `Finished` list of `Moves` is handed back to `playGame`, otherwise an `Unfinished` list is handed back, but either way the current move is prepended. The player to whom that move belongs is determined by a function that uses the most recent

move to determine who went last, and therefore can ascertain that the other player must be taking their turn presently.

```
-- this should just work on unfinished games
whoseTurn :: Unfinished -> Player
whoseTurn (Unfinished []) = One
whoseTurn (Unfinished ((player,_):_)) = otherPlayer player

otherPlayer :: Player -> Player
otherPlayer One = Two
otherPlayer Two = One
```

Listing 1.17 whosTurn and otherPlayer Functions

Incidentally, this function is also the one that decides which player takes his or her turn first (clearly seen to be `Player One`).
The last line executed in `main` utilizes a function somewhat similar to find the winner, given the full list of moves in a `Finished Game`.

```
-- this should only work on finished games
whoWon :: Finished -> Player
whoWon (Finished ((winner,_):_)) = winner
```

Listing 1.18 whoWon Function

1.3 Tie

In order to facilitate a tie, it would be best for the `isFinished` function to check not only for victory conditions, but for a full board as well. On the other hand, with a little restructuring, it could also happen in `playGame`, which could return `Nothing` if the game ties, or a game with a special move that indicates a tie, or it could simply return the game as it is, and then logic could be added in `main` that checks for a winner or a tie. If done in `isFinished`, it could be implemented by introducing a third type of `Player` that has the value `Tie` to indicate such an event, in which case a simple `if/else` or `unless` in `main` would allow the program to function almost exactly the way it does now.