Assignment 2 - Unbeatable Tic-Tac-Toe using Minimax Algorithm

Rory Stephenson 300160212 stepherory

14 April 2014

Data and Instance Declarations

`SQ` represents the three possible squares in tic-tac-toe: X, O and E which is an empty square. `Player` is just a synonym for the `SQ` type for readability in the code. A `Board` is made up of a list of `SQ`, which in this implementation will always be nine squares. Each group of three (in the order they are given) make up a row. `Mtree` is the tree structure which will contain the possible moves from each state of the board, to be used by the minimax algorithm to choose what move to make. Each node/leaf in the `Mtree` contains a `Board` and the `Player` who just had a turn. `Node`s also contain a list of possible moves from the current `Board` state. Creating a tree of every move does not carry a large performance cost becuase Haskell uses lazy evaluation so each successive list of moves is only actually generated when it is needed. This means only a very small subtree of the whole tree will be evaluated.

`Eq` and `Ord` instances of MTree allow ordering of children in the tree. A `Win` is always considered greater than a `Node` and equal to another win. When comparing two `Node`s they are evaluated based on the value they receive when `minimax` is evaluated on them. `MTree`'s instance of `Show` pretty prints the `Node` or `Win`'s board.

```
data SQ = X | O | E
type Player = SQ
type Board = [SQ]
data MTree = Node Board Player [MTree] | Win Board Player

instance Eq MTree where
  (Win _ _) == (Win _ _) = True
  (Win _ _) == (Node _ _ _) = False
```

```
  (Node _ _ _) == (Win _ _) = False
  l@(Node _ _ _) == r@(Node _ _ _) = minimax 3 l == minimax 3 r


instance Ord MTree where
  (Win _ _) `compare` (Win _ _) = EQ
  (Win _ _) `compare` (Node _ _ _)        = GT
  (Node _ _ _) `compare` (Win _ _)        = LT
  l@(Node _ _ _) `compare` r@(Node _ _ _) = minimax 3 l `compare` minimax 3 r

instance Show MTree where
  show (Node b _ _) = showBoard b
  show (Win b _) = showBoard b

instance Show SQ where
  show X = "X"
  show O = "O"
  show E = " "

instance Eq SQ where
  X == X = True
  O == O = True
  E == E = True
  _ == _ = False
```

The Game

main issues some instructions before finding out if the player wants to go first or second. It then calls gameSubTree to get a tree of all possible moves before passing this tree to play to start the game.

```
main :: IO()
main =
  do
    putStrLn "Welcome to Tic Tac Toe.\n"
    putStrLn "Make your selection by entering 1-9 where those numbers "
    putStrLn "match an empty slot on the board. The board is numbered "
    putStrLn "left to right, top to bottom.\n\n"
    putStrLn "Enter 1 to go first or 2 to let the computer go first.\n"
    firstTurn <- pickTurn
    play (gameSubTree (firstTurn) emptyBoard)
```

`pickTurn` chooses the computer as the first player if the user enters anything but 1.

```
pickTurn :: IO Player
pickTurn = getLine >>= (\ e -> return $ turn (read e :: Int))
             where turn 1 = (X :: Player)
                   turn _ = (O :: Player)
```

`play` takes an initial game state and alternates between giving the player a turn and giving the computer a turn. It does so by descending down the provided tree of possible moves based on the computer's or the player's chosen move. When a win is encountered the winning player in announced, otherwise an empty list of possible moves prompts declaration of a draw. The current `Board` state is shown at the start of the player's turn.

```
play :: MTree -> IO ()
play (Node brd _ []) = putStrLn (showBoard brd ++ "\n\n" ++ "It's a draw!")
play (Node brd X subtree) =
  putStrLn (showBoard brd) >> chooseMove brd subtree >>= play
play (Node _ O subtree) = play (last (mySort subtree))
play (Win brd a) =
  putStrLn (showBoard brd ++ "\n\nPlayer " ++ show a ++ " wins!\n")
play (Node _ E _) = error "Cannot play tic tac toe as an empty tile"
```

`chooseMove` waints for a valid numerical input from the human player and passes back the subtree of moves based on the player's chosen move.

```
chooseMove :: Board -> [MTree] -> IO MTree
chooseMove brd options =
  do
    n <- getLine
    if (fst (setSq (read n - 1 :: Int) X brd) == E) then
      return $ findMove (snd (setSq (read n - 1 :: Int) X brd)) options
    else
      putStrLn (showBoard brd) >> chooseMove brd options
```

`findMove` returns the MTree node/leaf containing the specified board state.

```
findMove :: Board -> [MTree] -> MTree
findMove brd tree =
  head $ foldr (\ n@(Node b _ _) r -> select (eqBrd b brd) n r) [] tree
                where select True e r = e:r
                      select False _ r = r
```

eqBrd evaluates whether two boards contain the same squares in the same order.

```haskell
eqBrd :: Board -> Board -> Bool
eqBrd (b:rd) (b2:rd2) = b == b2 && eqBrd rd rd2
eqBrd [] [] = True
eqBrd _ _ = False
```

comp declares the computer's player token

```haskell
comp :: Player
comp = O
```

emptyBoard is a board containing all empty tiles

```haskell
emptyBoard :: Board
emptyBoard = replicate 9 E
```

threes decomposes a board into its winnable rows/columns/diagonals

```haskell
threes :: Board -> [(SQ, SQ, SQ)]
threes (a:b:c:d:e:f:g:h:i:_) = [(a, b, c), (d, e, f), (g, h, i), -- rows
                                (a, d, g), (b, e, h), (c, f, i), -- columns
                                (a, e, i), (g, e, c)]            -- diagonals
threes _ = error "Board must be 3x3"
```

showBoard pretty prints a Board

```haskell
showBoard :: Board -> String
showBoard (a:b:c:d:e:f:g:h:i:_) =
  show a ++ sep ++ show b ++ sep ++ show c ++ rowSep ++
  show d ++ sep ++ show e ++ sep ++ show f ++ rowSep ++
  show g ++ sep ++ show h ++ sep ++ show i ++ "\n"
    where rowSep = "\n---------\n"
          sep = " | "
showBoard _ = "Invalid Board dimensions, 3x3 required"
```

gameSubTree recursively generates a tree containing all valid board states made by player's taking consecutive turns starting with the specified Player and Board. Every branch terminates at a win or a full board.

```haskell
gameSubTree :: Player -> Board -> MTree
gameSubTree player board | gameOver board =
  Win board (next player) -- The game was won by the player who just went
gameSubTree player board =
  Node board player $
    foldr (\ e r -> (gameSubTree (next player) e) : r)
      [] (boardNextMoves player board)
```

`minimax` returns the score for the `Board` in the given `Mtree` node/leaf by searching to the specified depth and applying the evaluation function on the successive moves.

```haskell
minimax :: Int -> MTree -> Int
minimax n (Node board _ children) | n == 0 || length children == 0
  = evaluate  board
minimax _ (Win board _) = evaluate board
minimax n (Node _ p children) =
  foldr ((maxOrMin p) . (minimax (n-1))) (startVal p) children -- sub moves
    where maxOrMin p1 | p1 == comp = max
                      | otherwise = min
          startVal p1 | p1 == comp = (-999999999 :: Int)
                      | otherwise = (999999999 :: Int)
```

`gameOver` returns true if a player has won

```haskell
gameOver :: Board -> Bool
gameOver board =
  foldr (\ (a, b, c) r -> r || (a == b && a == c && a /= E))
    False $ threes board
```

`evaluate` in combination with `evaluateLine` return the sum of the heuristic value of each winnable triple of squares in the specified `Board`. The score of three winnable `SQ` is 100, 10, 1 for 3, 2, 1 in a line respectively and the negative value equivalent for the human player.

```haskell
evaluate :: Board -> Int
evaluate board = foldr ((+) . evaluateLine) 0 (threes board)


evaluateLine :: (SQ, SQ, SQ) -> Int
evaluateLine (s1, s2, s3) =
  tileThr s3 $ tileTwo s2 $ tileOne s1
```

```haskell
        where tileOne s     | s == comp = 1
                            | s == next comp = -1
                            | otherwise = 0
              tileTwo s n   | s == comp && n == 1 = 10
                            | s == next comp && n == 1 = 0
              tileTwo s n   | s == comp && n == -1 = 0
                            | s == next comp && n == -1 = -10
              tileTwo s n   | s == comp && n == 0 = 1
                            | s == next comp && n == 0 = -1
              tileTwo _ n = n
              tileThr s n   | s == comp && n > 0 = n * 10
                            | s == comp && n < 0 = 0
                            | s == comp && n == 0 = 1
                            | s == next comp && n < 0 = n * 10
                            | s == next comp && n > 1 = 0
                            | s == next comp = -1
                            | otherwise = n
```

boardNextMoves generates the possible moves by the specified player on the given Board. add and setSq are used to duplicate a Board with a new move placed on it. setSq returns a tuple containing the replaced tile as well as the new Board so that the replacement can be checked for validity i.e. making sure the move was made on an E square.

```haskell
boardNextMoves :: Player -> Board -> [Board]
boardNextMoves p brd =
  foldr (\(n, b) r -> add n p b r) [] $ [0..8] `zip` (replicate 9 brd)

add :: Int -> Player -> Board -> [Board] -> [Board]
add n p brd brds | sq == E = ((newBoard):brds)
                 where (sq, newBoard) = setSq n p brd
add _ _ _ brds = brds

{- Sets the specified square and returns a tuple (old sq, new board) -}
setSq :: Int -> Player -> Board -> (SQ, Board)
setSq 0 p (b:bs) = (b, p:bs)
setSq n p (b:bs) = (square, b:board)
                 where (square, board) = (setSq (n-1) p bs)
setSq _ _ [] = (E, [])


```

```haskell
next :: Player -> Player
next X = O
next O = X
next E = E
```

`mySort` accepts a list of elements which are a subtype of `Ord` and returns that list in ascending order. It uses recursive mergesort to achieve the sorting. It is used on `MTree`s to order their children based on how good the outcomes of the current `Node`'s `Board` are for the computer.

```haskell
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort [a] = [a]
mySort xs = merge (mySort xs1) (mySort xs2)
            where
              (xs1, xs2) = splitAt (quot (length xs) 2) xs

merge :: Ord a => [a] -> [a] -> [a]
merge [] y = y
merge x [] = x
merge (x:xrest) (y:yrest) | x <= y = x: merge xrest (y:yrest)
merge (x:xrest) (y:yrest) | otherwise = y: merge (x:xrest) yrest
```