Jordan Lynn
CS 470, Project 2

For Project 2 I implemented a connect 4 board game both with a computer player and human player implementations. The human player uses whatever strategy or algorithm they'd like to play with on their own, while the computer will (probably) use a superior minimax algorithm with alpha-beta pruning. This is my second Scala project and so I'm striving to stay 'idiomatic' while designing my program. For project 1 I was delayed in handing in due to just learning this new language, so setting out for this project I had the goal of 'make everything an object' due to my last Scala project not having enough objects and I ended up cramming the objects with methods and fields that didn't belong there. Overall the project was a success, I have an AI that plays connect 4 and makes the 'correct' choice every time to a certain depth, currently with our ten second limit imposed comes to a depth of about eight or nine. Once the program was in it's completed state I have yet to beat it, partially because if I as a player make a mistake the program has no problem directing the game towards a winning state for it, but also because it can be hard to read the game board sometimes.

If I make the computer play against itself I initially didn't expect to have the same game run over and over again, but looking back of course that would be the result because there isn't any 'randomness' to this game, so if I choose to have the computer play itself and both of these players have the same depth the game will either draw every time or one will win every time depending on which player went first and what depth they're searching to.

Minimax is a way to make decisions based on which player's turn it is, for ever other level in the game state tree we either try and '-max' our potential game state score or try to 'Mini-' the score when it is our opponents score. This evaluation is simple and takes place in the 'BetterEvaluation.scala' trait. To help speed up our algorithm and get it to better depth in the state tree I also implemented alpha-beta pruning. This process just reduces the number of nodes we search in the state tree. This preemptive pruning of the state tree allows us to search deeper in the tree, hopefully giving this algorithm an edge over others in class when we do the in class competition.

While the game is in progress we use a evaluation function that will assign $Integer.MIN_VALUE$ to a state where the opponent wins, or the $Integer.MAX_VALUE$ value if there's a state where AI can win. Otherwise we just add +1 for every disc in a row we have.

Playing the game against the AI I do not believe it's possible to beat it without going to a further depth in the state tree or 'tricking' it if such a strategy exists. The AI will always push the game towards a win for itself, and always try and block you from winning.

Any improvements would entail maybe hard-coding some beginning states into the AI that it should push for, currently it prioritizes playing to the middle of the board, but maybe there is additional strategy that could improve it. Also the AI doesn't attempt any 'tricks' to deceive or beat its opponent which when playing against itself results in the same game over and over again.

Now the scala code:

```
package com

import com.Discs.Disc
import com.GridStates.{GameInProgress, Full, FourInALine}

/**
 * Better than DefaultEvaluation.
 * It gives a score based on how many times we can still make 4 in
     a line for given player.
 * It has shortcuts in case player has 4 in line, other player has
     4 in line or grid is full.
 * It also favours early win situation by taking into account the
    number of discs present in the grid.
 *
 */
trait BetterEvaluation extends Evaluation {

  case class CurrentAndMax(current:Int, max:Int)

  override def evaluate(grid: Grid, player: Player): Int = {
    grid.state match {
      case GameInProgress => calcInProgress(grid, player)
      case FourInALine => calc4InALine(grid, player)
      case Full => 0 // Draw condition?
    }
  }

  private def calc4InALine(grid: Grid, player: Player) =
    if (grid.winningDisc == Some(player.disc))
      Integer.MAX_VALUE - grid.NumOfCol
    else
      Integer.MIN_VALUE

  private def calcInProgress(grid: Grid, player: Player): Int = {
    val value = grid.cells.foldLeft(0)(
      (curVal, cell) =>
        if (grid.value(cell) == Some(player.disc)) {
          curVal + (calc(Grid.horizontal(cell.col, cell.row), grid
              , player.disc) +
          calc(Grid.vertical(cell.col, cell.row), grid, player.
              disc) +
          calc(Grid.diagonalTopLeftToBottomRight(cell.col, cell.
              row), grid, player.disc) +
```

```
          calc(Grid.diagonalBottomLeftToTopRight(cell.col, cell.
             row), grid, player.disc))
        }
        else curVal
    )
    value - grid.NumOfCol
  }

  private def calc(seq: Array[Cell], grid: Grid, disc: Disc): Int
     = {
    val value = seq.foldLeft(CurrentAndMax(0, 0))(
      (currentAndMax, cell) =>
        if (validCellForGrid(cell, grid) && (grid.value(cell) ==
           Some(disc) || grid.value(cell) == None)) {
          val current = currentAndMax.current + 1
          CurrentAndMax(current, math.max(current, currentAndMax.
             max))
        }
        else
          CurrentAndMax(0, currentAndMax.max)
    )
    if (value.max >= 4)
      value.max
    else
      0
  }

  private def validCellForGrid(cell:Cell, grid:Grid): Boolean =
    (cell.col >= 0) &&
      (cell.col < grid.NumOfCol) &&
      (cell.row >= 0) &&
      (cell.row < grid.NumOfRows)


}
```

```
package com

case class Cell(col:Int, row:Int)
```

```
package com

import com.Discs.{Player2, Player1, Disc}
import com.GridStates.GameInProgress

```

```scala
/**
 * Computer player that uses minimax.
 *
 * @constructor
 * @param name Player name
 * @param disc Disc player will use.
 * @param algoDepth How many steps will the computer player calculate upfront to determine best next move.
 */
class ComputerPlayer(val name: String, val disc: Disc, val algoDepth: Int) extends Player with Evaluation {

  private val otherPlayerDisc = {
    if (disc == Player1) Player2 else Player1
  }

  /**
   * Select next move. Determine column in which to drop disc.
   */
  def next(grid: Grid): Int = {
    if (grid.state != GameInProgress) {
      throw new IllegalStateException("Expected game to be still in progress.")
    }
    var alpha = Integer.MIN_VALUE
    var beta = Integer.MAX_VALUE
    var choice = 0
    for (col <- 0 until grid.NumOfCol) {
      if (grid.dropPossible(col)) {
        val newGrid = grid.drop(col, disc)
        val value = minimax(newGrid, algoDepth - 1, false, alpha, beta)
        if (value > alpha) {
          alpha = value
          choice = col
        }
      }
    }
    choice
  }


  private def minimax(grid: Grid, depth: Int, max: Boolean, alpha: Int, beta: Int): Int = {
    if (depth == 0 || grid.state != GameInProgress) {
```

```
      evaluate(grid, this)
    }
    else if (max) {
      var prune = false
      var alphaCopy = alpha
      for (col <- 0 until grid.NumOfCol if prune == false) {
        if (grid.dropPossible(col)) {
          val newGrid = grid.drop(col, disc)
          alphaCopy = math.max(alphaCopy, minimax(newGrid, depth -
              1, false, alphaCopy, beta))
          if (beta <= alphaCopy) prune = true
        }
      }
      alphaCopy
    }
    else {
      var prune = false
      var betaCopy = beta
      for (col <- 0 until grid.NumOfCol if prune == false) {
        if (grid.dropPossible(col)) {
          val newGrid = grid.drop(col, otherPlayerDisc)
          betaCopy = math.min(betaCopy, minimax(newGrid, depth -
             1, true, alpha, betaCopy))
          if (betaCopy <= alpha) prune = true
        }
      }
      betaCopy
    }
  }

}
```

```
package com

import com.Discs.Disc

/**
 * Represents a Player.
 */
trait Player {

  val name:String
  val disc:Disc

  /**
```

```
    * Determines next move based on given Grid state.
    *
    * @return Column in which to drop a disc.
    */
  def next(grid: Grid): Int

}
```

```
package com

import com.Discs.{Player2, Player1}
import com.GridStates.{Full, GameInProgress, FourInALine}

object Main extends App {
    val NumOfCol = 7
    val NumOfRow = 6
    val treeDepth = 8

    val grid = Grid (NumOfCol, NumOfRow)
    /*
    val playerTwo = new HumanPlayer ("Player 1", Player1)
    val playerOne = new ComputerPlayer("Computer", Player2,
       treeDepth) with BetterEvaluation
    */
    val playerTwo = new ComputerPlayer("Computer", Player2, 8)
       with BetterEvaluation
    val playerOne = new ComputerPlayer("Computer2", Player1, 8)
       with BetterEvaluation


    draw (grid, playerOne, playerTwo)
    val finalGrid = playGame (playerOne, playerTwo, grid)

    finalGrid.state match {
        case FourInALine =>
            println ("Four in a row!")
            val winningPlayer = if (finalGrid.winningDisc == Some
               (playerOne.disc)) {
                playerOne.name
            } else {
                playerTwo.name
            }
            println (s"Player $winningPlayer won.")
        case Full =>
            println ("Draw, the board is full")
```

```
        case _ =>
            throw new IllegalStateException ("Oh no the board
                entered an illegal state!")
    }


    private def playGame (currentPlayer: Player, otherPlayer:
        Player, grid: Grid): Grid = {
        val next = currentPlayer.next (grid)
        val updatedGrid = grid.drop (next, currentPlayer.disc)

        draw (updatedGrid, playerOne, playerTwo)

        if (updatedGrid.state == GameInProgress) {
            playGame (otherPlayer, currentPlayer, updatedGrid)
        } else updatedGrid
    }

    private def draw(grid: Grid, playerOne: Player, playerTwo:
        Player): Unit = {
        println(s"""${playerOne.name}: ${playerOne.disc.
            asciiRepresentation}, ${playerTwo.name}: ${playerTwo.
            disc.asciiRepresentation}""")

        // Print board
        for (row <- 0 until NumOfRow) {
            for (col <- 0 until NumOfCol) {
                grid.value(col, row) match {
                    case Some(player) => print(player.
                        asciiRepresentation)
                    case None => print("_")
                }
            }
            println()
        }
    }
}
```

```
package com

import com.Discs.Disc
import com.GridStates.GameInProgress

import scala.io.StdIn
```

```
/**
 * Player instance that request changes from console (= from a
   human).
 *
 * @constructor Creates a new Human Player instance.
 * @param name Name of player.
 * @param disc Disc type player plays with.
 */
class HumanPlayer(val name:String, val disc:Disc) extends Player {

  def next(grid: Grid): Int = {
     if (grid.state != GameInProgress) {
         throw new IllegalStateException("Game needs to be in 
            progress!")
        }
     var choice:Int = 0
     do {
        printf("Column choice (0-"+(grid.NumOfCol-1)+") : ")
        try{
                choice = StdIn.readInt()
        } catch {
            case e: Exception => println("Error not a valid number
               ")
            next(grid)
        }

     } while(grid.dropPossible(choice) == false)
     choice
  }

}
```

```
package com

import com.Discs.Disc
import com.GridStates._

case class Grid private (   NumOfCol: Int,
                            NumOfRows: Int,
                            private val content:List[Option[Disc
                               ]],
                            state:GridState,
                            winningDisc:Option[Disc]  ) {

    def value(col: Int, row: Int): Option[Disc] = {
```

```
      if (validCol(col) && validRow(row)) {
        content(indexFor(col, row))
      }
      else throw new IllegalArgumentException(s"Invalid col and/
        or row: col: $col, row: $row.")
    }


  def value(cell:Cell): Option[Disc] =
    value(cell.col, cell.row)

  def numOfDiscs =
    content.filter(_ != None).length

  def dropPossible(col:Int):Boolean =
    validCol(col) && value(col, 0) == None && state != FourInALine

   def drop(col:Int, disc:Disc): Grid =
    if (dropPossible(col)) {
      drop(col, NumOfRows-1, disc) match {
        case (newContent, row) =>
          if (winner(newContent, col, row, disc))
            new Grid(NumOfCol, NumOfRows, newContent, FourInALine,
                Some(disc))
          else if (row == 0 && allColumnsFull(newContent))
            new Grid(NumOfCol, NumOfRows, newContent, Full, None)
          else
            new Grid(NumOfCol, NumOfRows, newContent,
              GameInProgress, None)
      }
    } else
      throw new IllegalStateException(s"Dropping disc at col $col
        not possible.")

    def cells : Seq[Cell] =
        for(
          col <- 0 until NumOfCol;
          row <- 0 until NumOfRows
        ) yield(Cell(col, row))

    private def winner(content:List[Option[Disc]], col:Int, row:
      Int, disc:Disc) : Boolean =
    winner(content, Grid.horizontal(col, row), disc) ||
    winner(content, Grid.vertical(col, row), disc) ||
    winner(content, Grid.diagonalTopLeftToBottomRight(col, row),
```

```
        disc) ||
      winner(content, Grid.diagonalBottomLeftToTopRight(col, row),
        disc)

    private def winner(content:List[Option[Disc]], cellArray:Array
        [Cell], disc:Disc):Boolean = {
         val cells = cellArray.foldLeft(List[Cell]())(
           (list, cell) =>
             if (list.length == 4)
               list
             else if (validCol(cell.col) && validRow(cell.row) &&
                     (content(indexFor(cell.col, cell.row)) ==
                       Some(disc))
                     )
               cell :: list
             else
               List()
         )
         cells.size == 4
  }

  private def drop(col:Int, row:Int, disc:Disc): (List[Option[Disc
      ]],Int) = {
    value(col, row) match {
      case None =>
        val newContent = content.updated(indexFor(col, row), Some(
          disc))
        newContent -> row
      case Some(value) => drop(col, row-1, disc)
    }
  }

  private def allColumnsFull(content:List[Option[Disc]]) : Boolean
      =
    (0 until NumOfCol).filter(col => content(indexFor(col, 0)) !=
        None).length == NumOfCol

  private def validCol(col:Int) = col >= 0 && col < NumOfCol

  private def validRow(row:Int) = row >= 0 && row < NumOfRows

  private def indexFor(col:Int, row:Int) =
    col + (NumOfCol * row)
}
```

```
object Grid {

  def apply(NumOfCol:Int, NumOfRows:Int) = {
    val content = List.fill[Option[Disc]](NumOfCol * NumOfRows)(
       None)
    new Grid(NumOfCol, NumOfRows, content, GameInProgress, None)
  }

  /**
   * Return the cells for given col+row which need to be checked
      to verify if we
   * can have 4 in a row horizontally.
   *
   * @param col Column.
   * @param row Row.
   * @return Cells.
   */
  def horizontal(col: Int, row: Int): Array[Cell] =
    Array(
      Cell(col - 3, row), Cell(col - 2, row),
      Cell(col - 1, row), Cell(col, row),
      Cell(col + 1, row), Cell(col + 2, row), Cell(col + 3, row)
    )

  /**
   * Return the cells for given col+row which need to be checked
      to verify if we
   * can have 4 in a row vertically.
   *
   * @param col Column.
   * @param row Row.
   * @return Cells.
   */
  def vertical(col: Int, row: Int): Array[Cell] =
    Array(
      Cell(col, row - 3), Cell(col, row - 2),
      Cell(col, row - 1), Cell(col, row),
      Cell(col, row + 1), Cell(col, row + 2), Cell(col, row + 3)
    )

  /**
   * Return the cells for given col+row which need to be checked
      to verify if we
   * can have 4 in a row diagonally from top left to bottom right.
```

```
   *
   * @param col Column.
   * @param row Row.
   * @return Cells.
   */
  def diagonalTopLeftToBottomRight(col: Int, row: Int): Array[Cell
     ] =
    Array(
      Cell(col - 3, row - 3), Cell(col - 2, row - 2),
      Cell(col - 1, row - 1), Cell(col, row),
      Cell(col + 1, row + 1), Cell(col + 2, row + 2), Cell(col +
         3, row + 3)
    )

  /**
   * Return the cells for given col+row which need to be checked
     to verify if we
   * can have 4 in a row diagonally from bottom left to top right.
   *
   * @param col Column.
   * @param row Row.
   * @return Cells.
   */
  def diagonalBottomLeftToTopRight(col: Int, row: Int): Array[Cell
     ] =
    Array(
      Cell(col - 3, row + 3), Cell(col - 2, row + 2),
      Cell(col - 1, row + 1), Cell(col, row),
      Cell(col + 1, row - 1), Cell(col + 2, row - 2), Cell(col +
         3, row - 3)
    )
}
```

```
package com


object GridStates {

  sealed abstract class GridState

  case object GameInProgress extends GridState
  case object Full extends GridState
  case object FourInALine extends GridState

}
```

```
package com


/**
 * Used to evaluate game board in given state for given player.
 */
trait Evaluation {

  /**
   * Evaluate given grid for given player.
   * Returns a positive integer value (0 inclusive). The higher
      the value the better the situation
   * for the given player.
   *
   * @param grid Game grid.
   * @param player Player for who to evaluate board.
   */
  def evaluate(grid: Grid, player: Player): Int = 0
}
```

```
package com

/**
 * Represents the different 'discs' that can be dropped in the
    board.
 */
object Discs {

  sealed abstract class Disc(val asciiRepresentation:Char)

  case object Player1 extends Disc('o')
  case object Player2 extends Disc('x')

}
```