

# The Checkers

v 1.0

Gorka Suárez García & Enrique López Mañas

## 1.- A brief introduction.

Checkers game is a checkers version, developed under the Haskell platform for Windows **WinHugs**. It uses a minimax logic for the machine movements, with alpha-beta pruning. Distinguishing scope between difficulty levels has been enhanced, and a minimal text interface has been implemented to use within.

## 2.- What is Checkers? (from Wikipedia)

English draughts, also called American checkers or "straight checkers", commonly called checkers in the U.S., but commonly called draughts in some other countries, is a form of the draughts board game played on an 8×8 board with 12 pieces on each side that may only move and capture forward.

### The rules

As in all draughts variants, English draughts is played by two people, on opposite sides of a playing board, alternating moves. One player has dark pieces, and the other has light pieces. Pieces move diagonally and pieces of the opponent are captured by jumping over them.

The rules of this variant of draughts are:

- \* **Board** The board is an 8×8 grid, with alternating dark and light squares, called a checkerboard (in the US, in reference to its checkered pattern, also the source of the name checkers). The playable surface consists of the 32 dark squares only. A consequence of this is that, from each player's perspective, the left and right corners encourage different strategies.

- \* **Pieces** The pieces are usually made of wood and are flat and cylindrical. They are invariably split into one darker and one lighter color. Traditionally, these colors are red and white. There are two kinds of pieces: "men" and "kings". Kings are differentiated as consisting of two normal pieces of the same color, stacked one on top of the other. Often indentations are added to the pieces to aid stacking.

- \* **Starting Position** Each player starts with 12 pieces on the three rows closest to their own side, as shown in the diagram. The row closest to each player is called the "crownhead" or "kings row". The black (darker color) side moves first.

- \* **How to Move** There are two ways to move a piece: simply sliding a piece diagonally forwards (also diagonally backwards in the case of kings) to an adjacent and unoccupied dark square, or "jumping" one of the opponent's pieces. In this case, one piece "jumps over" the other, provided there is a vacant square on the opposite side for it to land on. Again, a man (uncrowned piece) can only jump diagonally forwards, and a king can also move diagonally backwards. A piece that is jumped is captured and removed from the board. Multiple-jump moves are possible if, when the jumping piece lands, there is another piece that can be jumped. Jumping is mandatory and cannot be passed up to make a non-jumping move, nor can fewer than the maximum jumps possible be taken in a multiple-jump move. When there is more than one way for a player to jump, one may choose which sequence to make, not necessarily the sequence

that will result in the most amount of captures. However, one must make all the captures in that sequence. (Under traditional draughts rules jumping is not mandatory. If it is not done, the opponent may either force the move to be reversed, huff the piece or carry on regardless.)

- \* **Kings** If a player's piece moves into the kings row on the opposing player's side of the board, that piece is said to be "crowned" (or often "kinged" in the US), becoming a "king" and gaining the ability to move both forwards and backwards. If a player's piece jumps into the kings row, the move terminates (it cannot jump out (as in a multiple-jump move) until that move has ended and the piece has been crowned).

- \* **How the Game Ends** A player wins by capturing all of the opposing player's pieces, or by leaving the opposing player with no legal moves.

In tournament English draughts, a variation called three-move restriction is preferred. The first three moves are drawn at random from a set of accepted openings. Two games are played with the chosen opening, each player having a turn at either side. This tends to reduce the number of draws and can make for more exciting matches. Three-move restriction has been played in the United States championship since 1934. A two-move restriction was used from 1900 until 1934 in the United States and in the British Isles until the 1950s. Before 1900, championships were played without restriction: this style is called go-as-you-please (GAYP).

One rule of long standing that has fallen out of favor is the "huffing" rule. In this variation, jumping is not mandatory, but a piece that could have jumped, but failed to do so, may be taken — or "huffed" — by the opposing player at the beginning of his or her next turn. After huffing the offending piece, the opponent then takes his or her turn as normal. Huffing has been abolished by both the American Checker Federation and the English Draughts Association.

Three common misinterpretations of the rules are,

- \* that the game ends in a draw when a player has no legal move but still pieces remaining (true in Chess but not in draughts, see stalemate)
- \* that capturing with a king precedes capturing with a regular piece
- \* a piece which in the current move has become a king can then in the same move go on to capture other pieces

### **3.- An approach over the game**

Checkers 1.0 uses the *kernel* concept to represents the current state of the application. This means the last command being executed, the board situation and so on. Kernel changes whenever a new command is introduced into the application (typically, a new movement is made by the player)

Game has been divided in two main sections:

### 3.1.- Console

Composed by other two files, *Console.hs* and *Utils.hs*. This group of files manages whatever is related with the Console displaying stuff, as is expectable by its own name. We will make a further analysis over this files:

#### 3.1.1.- Console.hs

Composed by the following functions and types:

##### Types:

- **Command:** This type defines the set of command we use in the game. Possible command's values can be *CmdHelp*, *CmdLookBoard*, *CmdNewGame PColor Level*, *CmdRestart*, *CmdExit*, *CmdMove Position Position*, *CmdNone*
- **Kernel:** Represents the current state of the full application. The status is represented by a combination of the *Game Data* and the last *movement*.

##### Functions:

- **initKernel:** This function simply gives us an initialized kernel.
- **checkCommand:** Checks if the line is a certain command, and if it's true the function will return the command inside value. The header of this function is  

```
checkCommand::String -> String -> Command -> Command
```
- **checkCommandEx:** Very similar function to the last one. We only extend the functionality to accept one more value. The header is:

```
checkCommand::String -> String -> Command -> Command
```

- **parcePieceColor:** Gets a PieceColor from a string. By default, value will be White. The header of the function is:

```
parsePieceColor::String -> PColor
```

- **parseLevel:** Gets a Level from a string. By default, value will be Easy. The header of the function is:

```
parseLevel::String -> Level
```

- **getStrPieceColor:** This time, we will transform a string into a PieceColor. The header of the function is:

```
getStrPieceColor::PColor -> String
```

- **getStrLevel:** Gets a string from a Level. The header of this function is:

```
getStrLevel::Level -> String
```

- **parsePosition:** Gets a BoardPosition from a string. This function uses the help of the auxiliary function *parsePositionEx*. The header of this function is:

```
parsePosition::String -> Position
```

- **parsePositionEx:** Auxiliary function of parsePosition, used to get a BoardPosition from a string.. The header of this function is:

```
parsePositionEx::String -> String -> Position -> Position
```

- **getStrCmdNew3:** Get a string like "new <level> game" from a Command.. The header of this function is:

```
getStrCmdNew3::Command -> String
```

- **getStrCmdNew4:** Get a string like "new game with <color>" from a Command. The header of this function is:

```
getStrCmdNew4::Command -> String
```

- **getStrCmdNew5:** Get a string like "new <level> game with <color>" from a Command. The header of this function is:

```
getStrCmdNew5::Command -> String
```

- **checkCmdNewEx3:** Checks if the line is a new game command, and if it's true the function will return the command inside value. The header of this function is:

```
checkCmdNewEx3::String -> Command
```

- **checkCmdNewEx4:** Checks if the line is a new game command, and if it's true the function will return the command inside value. The header of this function is:

```
checkCmdNewEx4::String -> Command
```

- **checkCmdNewEx5:** Checks if the line is a new game command, and if it's true the function will return the command inside value. The header of this function is:

```
checkCmdNewEx5::String -> Command
```

- **checkCmdNew:** Checks if the line is a new game command, and if it's true the function will return the command inside value. The header of this function is:

```
checkCmdNew::String -> Command
```

- **getStrLevel:** Gets a string from a Level. The header of this function is:

```
getStrLevel::Level -> String
```

- **checkCmdMovEx:** Checks if the line is a move command, and if it's true the function will return the command inside value. The header of this function is:

```
checkCmdMovEx::String -> Int -> Command
```

- **checkCmdMov:** Checks if the line is a move command, and if it's true the function will return the command inside value. The header of this function is:

```
checkCmdMov::String -> Command
```

- **parseCmd:** Gets a Command from a string.. The header of this function is:

```
parseCmd::String -> Command
```

- **getCmd:** Gets a Command from a string.. The header of this function is:

```
getCmd::Kernel -> IO Kernel
```

- **doCmdHelp:** Shows the help of the game (all the information related to the game's commands, and some examples of use. The header of this function is:

```
doCmdHelp::Kernel -> IO Kernel
```

- **printSquare:** Prints the value of a square on screen.. The header of this function is:

```
printSquare::Square -> IO ()
```

- **printHBorderLn:** Prints the upper and bottom horizontal border. The header of this function is:

```
printHBorderLn::IO ()
```

- **printHNumbersLn:** Prints the horizontal numbers of the columns. The header of this function is:

```
printHNumbersLn::IO ()
```

- **printHSeparator:** Prints the horizontal separator between squares.. The header of this function is:

```
printHSeparator::IO ()
```

- **printHSeparatorLn:** Prints the horizontal separator between squares.. The header of this function is:

```
printHSeparatorLn::IO ()
```

- **printBoardCols:** Prints the columns of a row.. The header of this function is:

```
printBoardCols::Board -> Int -> Int -> IO ()
```

- **printBoardRow:** Prints a row of squares.. The header of this function is:

```
printBoardRow::Board -> Int -> IO ()
```

- **printBoardRowLn:** Prints the upper and bottom horizontal border. The header of this function is:

```
printBoardRowLn::Board -> Int -> IO ()
```

- **printBoardRows:** Prints some rows of the board.. The header of this function is:

```
printBoardRows::Board -> Int -> IO ()
```

- **doCmdLookBoard:** Shows on screen the current board of the game. The header of this function is:

```
doCmdLookBoard::Kernel -> IO Kernel
```

- **doCmdNewGame:** Starts a new game with a new configuration.. The header of this function is:

```
doCmdNewGame::Kernel -> IO Kernel
```

- **doCmdRestart:** Starts a new game with the current configuration.. The header of this function is:

```
doCmdRestart::Kernel -> IO Kernel
```

- **getMoveError:** Gets the string that explains the last error in a move. The header of this function is:

```
getMoveError::Int -> String
```

- **doCmdMove:** Executes a player's move inside the board.. The header of this function is:

```
doCmdMove::Kernel -> IO Kernel
```

- **doCmdError:** Shows an error if the user puts an invalid command.. The header of this function is:

```
doCmdError::Kernel -> IO Kernel
```

- **doCmdEx:** Executes the last command introduced.. The header of this function is:

```
doCmdEx::Kernel -> Command -> IO Kernel
```

- **doCmd:** Executes the last command introduced. The header of this function is:

```
doCmd::Kernel -> IO Kernel
```

- **mainLoop:** This is the main loop of the game. The header of this function is:

```
mainLoop::Kernel -> IO ()
```

### 3.1.2.- Utils.sh

Composed by the following functions:

#### Functions:

- **dropTokenEx:** Drops the first token inside the string.. The header of this function is

```
dropTokenEx::String -> Int -> String
```

- **dropToken:** Drops the first token inside the string.. The header is:

```
dropToken::String -> String
```

- **getToken:** Gets a token inside the string.. The header is:

```
getToken::String -> Int -> String
```

- **getTokens:** Gets a some tokens inside the string. The header is:

```
getTokens::String -> Int -> String
```

- **countTokensEx:** Gets the number of tokens inside the string. The header is:

```
countTokensEx::String -> Int -> Int
```

- **countTokens:** Gets the number of tokens inside the string.The header is:

```
countTokens::String -> Int
```

- **getTokenPositionEx:** Gets the position of a token inside the text.The header is:

```
getTokenPositionEx::String -> String -> Int -> Int
```

- **getTokenPosition:** Gets the position of a token inside the text. The header is:

```
getTokenPosition::String -> String -> Int
```

- **checkText:** Checks if some tokens inside "text" are equal to "cmd". The header is:

```
checkText::String -> String -> Int -> Bool
```

- **isDigit:** Checks if the character is a digit. The header is:

```
isDigit:: Char -> Bool
```



- **quitSpacesEx:** Erases the redundant blank spaces in a string. The header is:

```
quitSpacesEx::String -> Bool -> String
```

- **quitSpaces:** Erases the redundant blank spaces in a string. The header is:

```
quitSpaces::String -> String
```

- **isDigit:** Checks if the character is a digit. The header is:

```
isDigit:: Char -> Bool
```

## 3.2.- Game

Composed by other two files, *Data.hs* and *Logic.hs*.

### 3.2.1.- Data.hs

Composed by the following functions and types:

#### Types:

- **PColor:** This new type is to define the color of a piece.. Possible PColor's values can be *White* and *Black*.
- **PType:** This new type is to define the type of a piece: normal or king. The main difference between the types is the movement options.
- **Piece:** This represents the data of a piece: the color and the type.
- **Square:** This new type is to define the state of a square in the board of the game. There are only three options: an empty square, a square with a piece or a square that is not usable in the game.
- **Position:** This represents a position inside the board.
- **BoardRow:** This represents a file in the board of the game.
- **Board:** This represents the board of the game.
- **Level:** This new type is to define the difficulty in the game. Possible values can be *Easy*, *Medium* or *Hard*.
- **Config:** This represents the current configuration of the game.
- **LastMove:** This represents the last move of the IA.
- **GameData:** This represents the current state of the whole game.

#### Functions:

- **initBoardRow:** This gives an initialized BoardRow. The header of this function is:

```
initBoardRow::Square -> Int -> BoardRow
```

- **initBoardSquares:** This gives an initialized fragment of a Board. The header of this function is

```
initBoardSquares::Square -> Int -> Board
```

- **initBoard:** This gives an initialized Board. The header is:

```
initBoard::Board
```

- **checkLimits:** Checks if the coordinates are inside the board. The header of the function is:

```
checkLimits::Int -> Int -> Bool
```

- **checkEvenLimits:** Checks if the coordinates are an even square. The header of the function is:

```
checkEvenLimits::Int -> Int -> Bool
```

- **checkOddLimits:** Checks if the coordinates are an odd square. The header of the function is:

```
checkOddLimits::Int -> Int -> Bool
```

- **getBoardCol:** Gets a square inside a BoardRow value.. The header of the function is:

```
getBoardCol::BoardRow -> Int -> Square
```

- **getBoardRow:** Gets a row inside a Board value. The header of the function is:

```
getBoardRow::Board -> Int -> BoardRow
```

- **getBoardSquare:** Gets a square inside a Board value. The header of the function is:

```
getBoardSquare::Board -> Int -> Int -> Square
```

- **setBoardCol:** Sets a square inside a BoardRow value. The header of the function is:

```
setBoardCol::BoardRow -> Int -> Square -> BoardRow
```

- **setBoardRow:** Sets a square inside a Board value. The header of the function is:

```
setBoardRow::Board -> Int -> Int -> Square -> Board
```

- **setBoardSquare:** Sets a square inside a Board value. The header of the function is:

```
setBoardSquare::Board -> Int -> Int -> Square -> Board
```

- **initLastMove:** This gives an initialized LastMove. The header of the function is:

```
initLastMove::LastMove
```

- **initConfig:** This gives an initialized Config. The header of the function is:

```
initConfig::Config
```

- **initGameData:** This gives an initialized GameData. The header of the function is:

```
initGameData::GameData
```

- **initConfig:** This gives an initialized Config. The header of the function is:

```
initConfig::Config
```

### 3.2.2.- Logic.hs

Composed by the following functions and types:

#### Types:

- **Vector:** This represents a vector inside the board. It is composed by a couple of integer values.
- **RelCoords:** This represents a relative coordinates inside the board. It is composed by a couple of integer values.
- **NodePiece:** This represents a piece and all its possible moves. It is composed by the position and the tree with all the possible values.
- **TreeMoves:** This new type is to create a tree with all the moves in a turn.

#### Functions:

- **checkSquareColor:** Checks if a square have a piece with the same color. The header of this function is:

```
checkSquareColor::Square -> PColor -> Bool
```

- **validateOrigin:** Validates that the origin position is a piece of a player. The header of this function is

```
validateOrigin::Board -> PColor -> Position -> Bool
```

- **getVector:** Gets a directional vector from a relative coordinates. The header of this function is:

```
initBoard::Board
```

- **getPrevSquare:** Gets the previous square to the destination one.. The header of the function is:

```
getPrevSquare::Board -> Position -> (Int, Int) -> Square
```

- **validateEnemyPrevSquare:** Checks if the next square have an enemy we want to kill. The header of the function is:

```
validateEnemyPrevSquare::PColor -> Square -> Bool
```

- **validateNormalMoveEx:** Validates the move that a normal piece wants to do. The header of the function is:

```
validateNormalMoveEx::Board -> PColor -> Position -> Position -> (Int, Int) -> Bool
```

- **validateNormalMove:** Validates the move that a normal piece wants to do. The header of the function is:

```
validateNormalMove::Board -> PColor -> Position -> Position -> Bool
```

- **countPieces:** Counts the enemy pieces between the origin and the destination.. The header of the function is:

```
countPieces::Board -> PColor -> Vector -> Position -> Position -> Int -> Int
```

- **validateKingMove:** Validates the move that a king piece wants to do. The header of the function is:

```
validateKingMove::Board -> PColor -> Position -> Position -> Bool
```

- **validateMoveEx:** Validates the move the player wants to do. The header of the function is:

```
validateMoveEx::PType -> Board -> PColor -> Position -> Position -> wBool
```

- **validateMove:** Validates the move the player wants to do. The header of the function is:

```
validateMove::Board -> Position -> Position -> Bool
```

- **killPiece:** We kill a piece. The header of the function is:

```
killPiece::Board -> Vector -> Position -> Position -> Board
```

- **makeMove:** This gives an initialized LastMove. The header of the function is:

```
makeMove::Board -> Position -> Position -> Board
```

- **getAllNormalMoves:** Gets all the moves of a normal piece. The header of the function is:

```
getAllNormalMoves::PColor -> [RelCoords]
```

- **getAllKingMoves:** Gets all the moves of a king piece.. The header of the function is:

```
getAllKingMoves::[RelCoords]
```

- **getAllMoves:** Gets all the moves of a piece.. The header of the function is:  
`getAllMoves::PType -> PColor -> [RelCoords]`
- **getRelCoords:** Gets a relative coordinates from a move. The header of the function is:  
`getRelCoords::Position -> Position -> RelCoords`
- **calcPrevSquare:** Gets the position of the previous square to the destine. The header of the function is:  
`calcPrevSquare::Position -> Vector -> Position`
- **checkSquareColor:** Checks if a square have a piece with the same color. The header of the function is:  
`checkSquareColor::Square -> PColor -> Bool`
- **checkEnemySquare:** Checks if the square have an enemy piece. The header of the function is:  
`checkEnemySquare::PColor -> Square -> Bool`
- **transformSquare:** This transforms a normal piece into a king. The header of the function is:  
`transformSquare::Square -> Position -> Square`
- **makeMove:** Moves a piece from an origin to a destination. The header of the function is:  
`makeMove::Board -> Position -> Position -> Board`
- **getAllBoardPositions:** Gets all the board positions. The header of the function is:  
`getAllBoardPositions::[Position]`
- **countTotalPiecesEx:** Counts the total number of pieces of a color. The header of the function is:  
`countTotalPiecesEx::Board -> PColor -> PType -> Int`
- **countTotalPieces:** Counts the total number of pieces of a color. The header of the function is:  
`countTotalPieces::Board -> PColor -> Int`
- **getAllPieceMovesEx:** Gets all the posible moves of a piece. The header of the function is:  
`getAllPieceMovesEx::Board -> Position -> Square -> [(Position, Position)]`

- **getAllPieceMoves:** Gets all the possible moves of a piece. The header of the function is:

```
getAllPieceMoves::Board -> Position -> [(Position, Position)]
```

- **oppColor:** Gets the opposite color. The header of the function is:

```
oppColor::PColor -> PColor
```

- **killNormalCondition:** Gets if a normal piece can check the stillKillMove condition. The header of the function is:

```
killNormalCondition::Position -> Position -> Square -> Bool
```

- **stillKillMove:** Gets if a piece can still move to kill an enemy. The header of the function is:

```
stillKillMove::Board -> PColor -> Position -> Position -> Bool
```

- **evalColorSide:** Calculates the value of a color side. The header of the function is:

```
evalColorSide::Board -> PColor -> Int
```

- **evalBoardBlack:** Evaluator for the computer when the player is the black side. The header of the function is:

```
evalBoardBlack::Board -> Int
```

- **evalBoardWhite:** Evaluator for the computer when the player is the white side. The header of the function is:

```
evalBoardWhite::Board -> Int
```

- **getPiecesPositions:** Gets all the positions of the pieces of a color. The header of the function is:

```
getPiecesPositions::Board -> PColor -> [Position]
```

- **getNodeNextKill:** Gets all the kill moves inside a list of nodes. The header of the function is:

```
getNodeNextKill::Board -> PColor -> Position -> [TreeMoves]
```

- **getNodeNextKillEx:** Gets all the kill moves inside a list of nodes. The header of the function is:

```
getNodeNextKillEx::Board -> PColor -> Position -> Position -> [TreeMoves]
```

- **getNodeNextAll:** Gets all the moves inside a list of nodes. The header of the function is:

```
getNodeNextAll::Board -> PColor -> Position -> [TreeMoves]
```

- **getNodePiece:** Gets all the node pieces in the root of the tree. The header of the function is:

```
getNodePiece::Board -> PColor -> [Position] -> [NodePiece]
```

- **getTreeMoves:** Gets a tree with all the moves in this turn. The header of the function is:

```
getTreeMoves::Board -> PColor -> TreeMoves
```

- **getNodeTotalMoves:** Gets the total number of moves from a NodePiece. The header of the function is:

```
getNodeTotalMoves::NodePiece -> Int
```

- **getTreeTotalMoves:** Gets the total number of moves from a TreeMoves.. The header of the function is:

```
getTreeTotalMoves::TreeMoves -> Int
```

- **getMoveFromNodes:** Gets a move from a node of a TreeMoves. The header of the function is:

```
getMoveFromNodes::[TreeMoves] -> Int -> [Position] -> [Int] -> [Position]
```

- **getMoveFromNodePieces:** Gets a move from a NodePiece of a TreeMoves. The header of the function is:

```
getMoveFromNodePieces::[NodePiece] -> Int -> [Int] -> [Position]
```

- **getMoveFromRoot:** Gets a move from the root of a TreeMoves. The header of the function is:

```
getMoveFromRoot::TreeMoves -> Int -> [Position]
```

- **getMoveFromTree:** Gets a move from a TreeMoves. The header of the function is:

```
getMoveFromTree::TreeMoves -> Int -> [Position]
```

- **initialAlpha:** A constant for the initial alpha in the minimax algorithm. By default, the value is -1000000. The header of the function is:

```
initialAlpha::Int
```

- **initialBeta:** A constant for the initial beta in the minimax algorithm. By default, the value is -1000000. The header of the function is:

```
initialBeta::Int
```

- **maximizeMoves:** Gets the maximum value for the computer's choice. The header of the function is:

```
maximizeMoves::Board -> PColor -> Int -> Int -> Int -> (Board -> Int) -> TreeMoves -> Int -> Int -> Int -> Int
```

- **maximize:** Gets the maximum value for the computer's choice. The header of the function is:

```
maximize::Board -> PColor -> Int -> Int -> Int -> (Board -> Int) -> Int
```

- **minimizeMoves:** Gets the minimum value for the player's choice.. The header of the function is:

```
minimizeMoves::Board -> PColor -> Int -> Int -> Int -> (Board -> Int) -> TreeMoves -> Int -> Int -> Int -> Int
```

- **minimize:** Gets the minimum value for the player's choice. The header of the function is:

```
minimize::Board -> PColor -> Int -> Int -> Int -> (Board -> Int) -> Int
```

- **minimaxMoves:** Gets the best move for the computer. The header of the function is:

```
minimaxMoves::Board -> PColor -> Int -> Int -> Int -> (Board -> Int) -> TreeMoves -> Int -> Int -> Int -> [Position] -> [Position]
```

- **minimax:** Gets the best move for the computer. The header of the function is:

```
minimax::Board -> PColor -> Int -> Int -> Int -> [Position]
```

- **makeAIMove:** Executes a list of moves for the computer. The header of the function is:

```
makeAIMove::Board -> [Position] -> Board
```

- **getLastMoveFromListLast:** Gets the last position inside the list. The header of the function is:

```
getLastMoveFromListLast::[Position] -> Position -> Position
```

- **getLastMoveFromList:** Transforms a list of moves into a LastMove structure. The header of the function is:

```
getLastMoveFromList::[Position] -> LastMove
```

- **aiMove:** Calculates and executes the best move for the computer. The header of the function is:

```
aiMove::Board -> Config -> (Board, LastMove)
```



- **execMove:** Executes a player's move inside the board. The header of the function is:

```
execMove::Board -> PColor -> Position -> Position -> (Board, Int)
```

- **moveEx:** Executes a player's move inside the board. The header of the function is:

```
moveEx::GameData -> Position -> Position -> GameData
```

- **move:** Executes a player's move inside the board. The header of the function is:

```
move::GameData -> Position -> Position -> GameData
```

### 3.3.- Other files

Composed by other two files, *Data.hs* and *Logic.hs*.

#### 3.3.1.- Checkers.hs

This file is the main file of the game. When loaded, and when called the function main, we initialize a new kernel and enter into the game main loop

#### 3.3.2.- Checkers.hs

This file is the main file of the game. When loaded, and when called the function main, we initialize a new kernel and enter into the game main loop

#### Functions:

- **debugPiece:** This function shows the content of a Piece value.. The header of this function is:

```
debugPiece::Piece -> IO ()
```

- **debugSquare:** This function shows the content of a Square value. The header of this function is:

```
debugSquare::Square -> IO ()
```

- **debugPosition:** This function shows the content of a Position value. The header of this function is:

```
debugPosition::Position -> IO ()
```

- **debugBoard:** This function shows the content of a Board value. The header of this function is:

```
debugBoard::Board -> IO ()
```

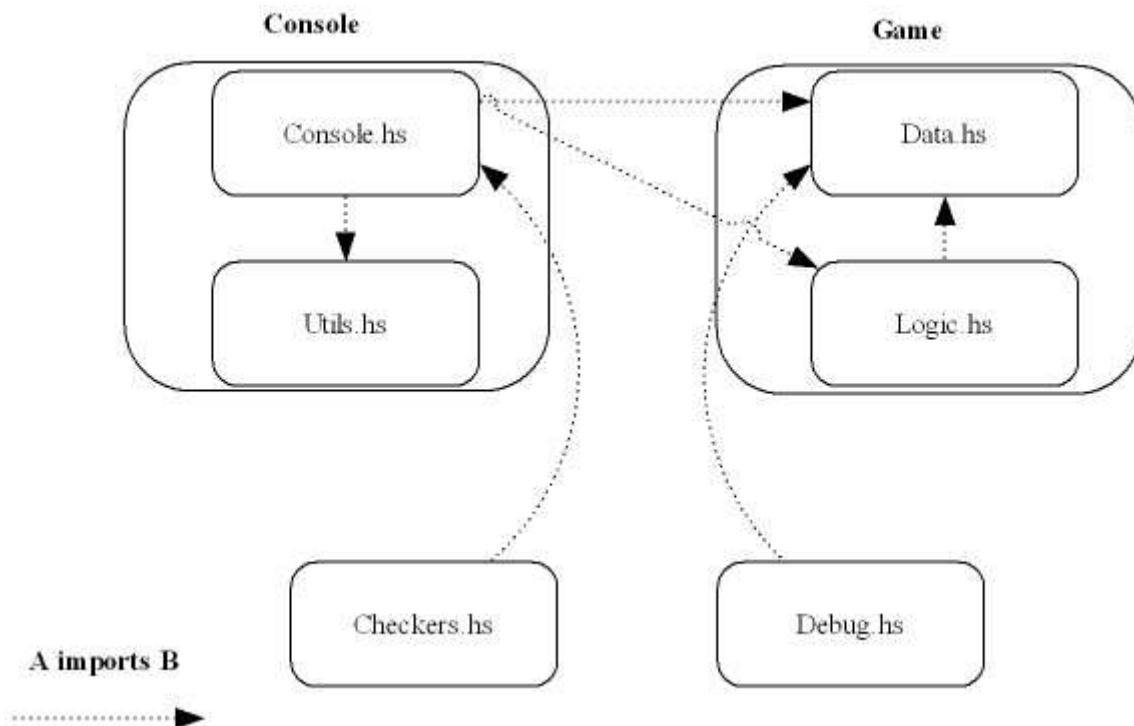
- **debugConfig:** This function shows the content of a Config value. The header of this function is:

```
debugConfig::Config -> IO ()
```

- **debugGameData:** This function shows the content of a Kernel value. The header of this function is:

```
debugGameData::GameData -> IO ()
```

### 3.4.- Dependency Map



## 4.- Game's proof of concept and behaviour

### 4.1.- Game initialization

Initially, **Checkers.hs** is the .hs file which is expected to be loaded into WinHugs. Checkers will paint on the screen a small help tip, and will load the main loop.

The main loop is a function located within Console.hs. This function has two options: exit from the application, or it calls itself with a new kernel state.

When recalled, we will send a message to doCmd. doCmd is a function that will execute the last command introduced, calling the auxiliary function doCmdEx.

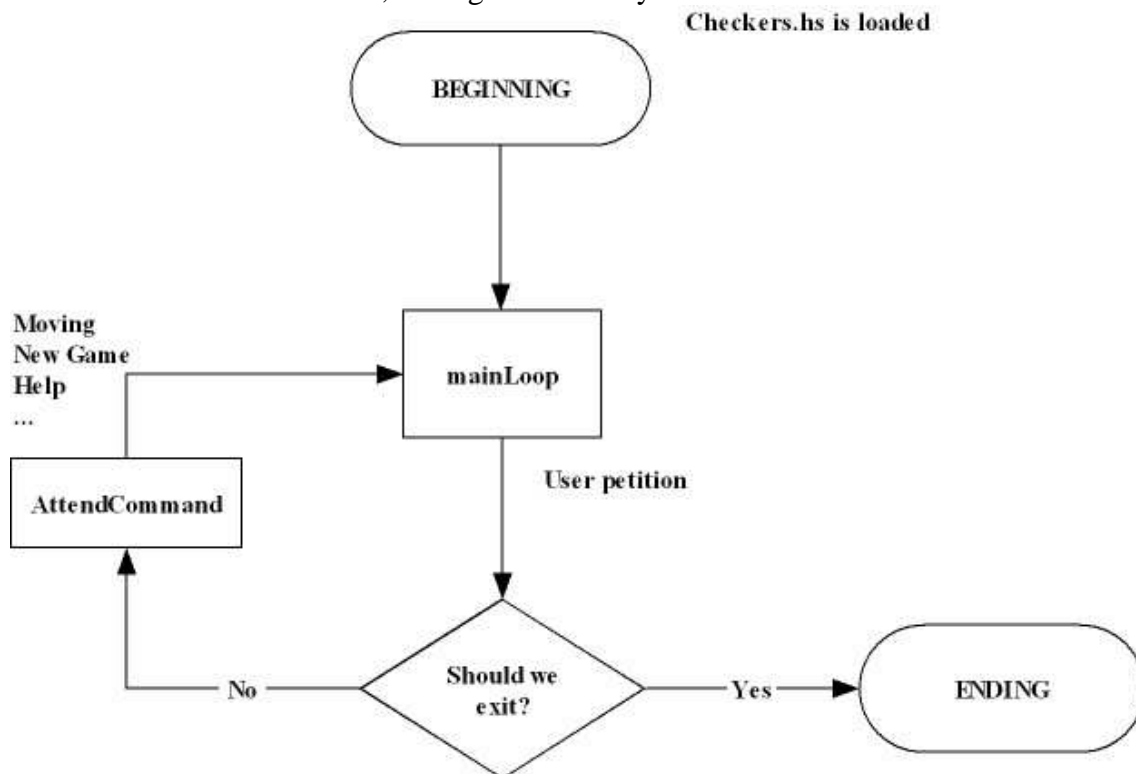


Figure 4.1.: Game engine's behaviour diagram

#### 4.2.- The kernel concept

The kernel concept has been introduced to represent the internal state of the entire application.

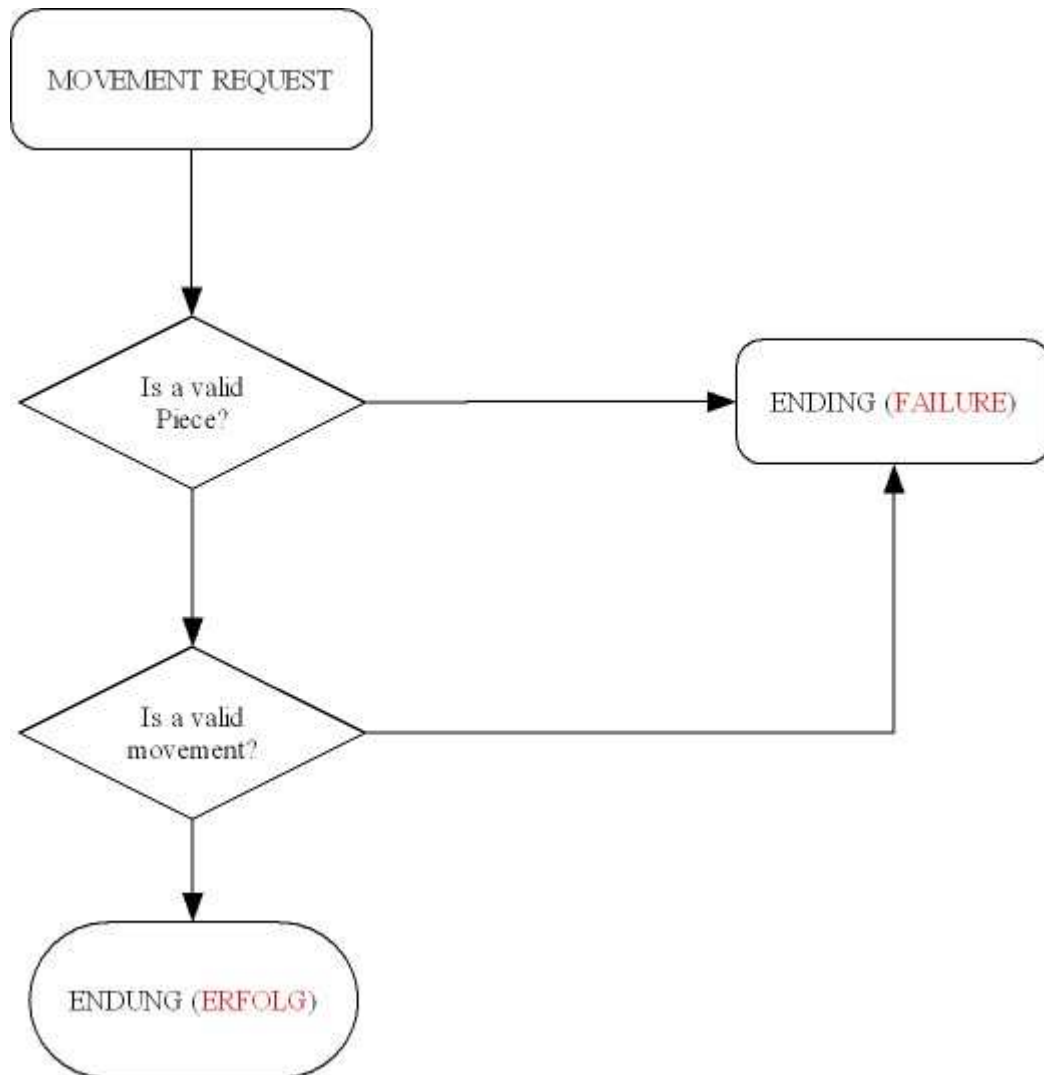
## Kernel Composition

Game Data				Command
Board	(int, int)	Config		<b>CmdHelp</b> <b>CmdLook Board</b> <b>CmdRestart</b> <b>CmdExit</b> <b>CmdNone</b> <b>CmdNewGame</b> Pcolor Level White Easy Black Medium Hard  <b>CmdMove</b> Position Position (int, int) (int, int)
BoardRow		Level	PColor	
Square SquareEmpty SquareInvalid SquarePiece		Easy Medium Hard	White Black	
PType Normal King			PColor White Black	

Of course, kernel status is changed whenever we executed any action involving one of the kernel's attribute. More specifically, every time we execute a new command, kernel status changes.

### 4.3.- Movement logic

Here is an explanation about the movement logic. Player's movement control is quite simple: we just take a look on some conditions to evaluate the validity of the movement



The IA Logic for the movement follows a more complex process. We use a minimax methodology to calculate the best option of the AI. What is exactly a minimax strategy? Let's take a look:

#### 4.3.1.- Minimax strategy (from Wikipedia)

A minimax algorithm is a recursive algorithm for choosing the next move in an n-player game, usually a two-player game. A value is associated with each position or state of the game. This value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves. If it is A's turn to move, A gives a value to each of his legal moves.

A possible allocation method consists in assigning a certain win for A as +1 and for B as -1. This leads to combinatorial game theory as developed by John Horton Conway. An alternative is using a rule that if the result of a move is an immediate win for A it is assigned positive infinity and, if it is an immediate win for B, negative infinity. The value to A of any other move is the minimum of the values resulting from each of B's possible replies. For this reason, A is called the *maximizing player* and B is called the *minimizing player*, hence the name *minimax algorithm*. The above algorithm will assign a value of positive or negative infinity to any position since the value of every position will be the value of some final winning or losing position. Often this is generally only possible at the very end of complicated games such as chess or go, since it is not computationally feasible to look ahead as far as the completion of the game, except towards the end, and instead positions are given finite values as estimates of the degree of belief that they will lead to a win for one player or another.

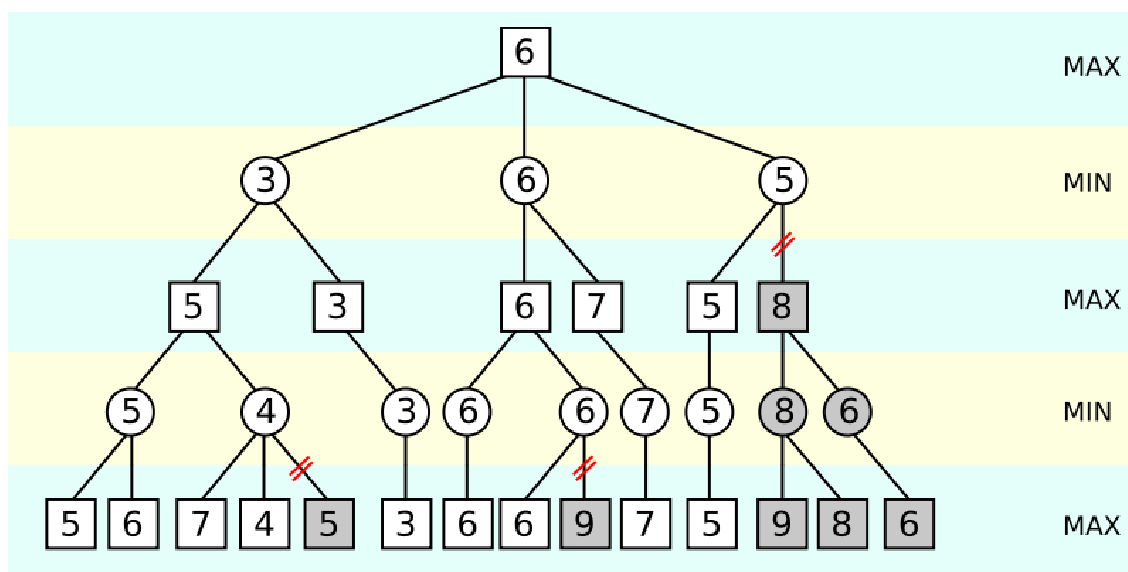
This can be extended if we can supply a heuristic evaluation function which gives values to non-final game states without considering all possible following complete sequences. We can then limit the minimax algorithm to look only at a certain number of moves ahead. This number is called the "look-ahead", measured in "plies". For example, the chess computer Deep Blue (that beat Garry Kasparov) looked ahead 12 plies, then applied a heuristic evaluation function.

The algorithm can be thought of as exploring the nodes of a *game tree*. The *effective branching factor* of the tree is the average number of children of each node (i.e., the average number of legal moves in a position). The number of nodes to be explored usually increases exponentially with the number of plies (it is less than exponential if evaluating forced moves or repeated positions). The number of nodes to be explored for the analysis of a game is therefore approximately the branching factor raised to the power of the number of plies. It is therefore impractical to completely analyze games such as chess using the minimax algorithm.

The performance of the native minimax algorithm may be improved dramatically, without affecting the result, by the use of alpha-beta pruning. Other heuristic pruning methods can also be used, but not all of them are guaranteed to give the same result as the un-pruned search.

## What is alpha-beta pruning?

**Alpha-beta pruning** is a search algorithm that reduces the number of nodes that need to be evaluated in the search tree by the minimax algorithm. It is a search with adversary algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go... Checkers, etc.). It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. Alpha-beta pruning is a sound optimization in that it does not change the result of the algorithm it optimizes. Here we can see a schema of a minimax tree using alpha-beta pruning to improve the performance of the algorithm:



When evaluating from the left to the right side, greyed out sub trees doesn't need to be explored, since we know the group of subtrees yields the value of an equivalent subtree or worse, and as such cannot influence the final result.

Focusing on our particular application, when the human movement has been executed, there is a call in *execMove* function to *aiMove*. And here begins the entire AI engine

