Faculdade de Engenharia
da Universidade do Porto

Relatório Final

Programação em Lógica

Mestrado Integrado em Engenharia Informática e Computação

Corrida de Reis

Autores:
Afonso Jorge Ramos up201506239@fe.up.pt
João Dias Conde Azevedo up201503256@fe.up.pt

Novembro 2017

Resumo

Ao longo das últimas semanas de aulas, o grupo desenvolveu um jogo em PROLOG, denominado de "Racing Kings", em português "Corrida de Reis", uma das muitas variantes do jogo tradicional de Xadrez. O trabalho teve como objetivo aplicarmos os conhecimentos teóricos de programação em lógica na implementação do jogo, revelando-se este o maior desafio, pois PROLOG é uma linguagem de programação com um paradigma diferente do habitual. Com alguma pesquisa e consulta aos materiais fornecidos pelos docentes foi possível uma melhor compreensão e aplicação das novas abordagens aqui requeridas. Com recorrência a diversos predicados já existentes e novos criados pelo grupo a solução à grande maioria dos problemas foi encontrada, e a totalidade dos mesmos não foi resolvida por falta de tempo e/ou de conhecimento. A cooperação entre os membros do grupo e curiosidade foram essenciais para o que foi conseguido. Como resultado final do projeto temos um jogo simples e apelativo, sendo uma variante do xadrez, como também mais ou menos robusto e eficiente visto que foi desenvolvido para execução na linha de comandos. Como conclusões finais a este trabalho podemos dizer que o nosso conhecimento acerca da linguagem PROLOG aumentou consideravelmente, sendo possível a consolidação dos conceitos aprendidos nas aulas teóricas.

Conteúdo

1 Introdução

No âmbito da unidade curricular de Programação em Lógica, do curso Mestrado Integrado em Engenharia Informática e Computação, foi-nos sugerido o desenvolvimento de um jogo utilizando a linguagem de programação PROLOG. De entre as várias opções que nos foram disponibilizados pelos docentes foi escolhido o jogo Corrida de Reis, pois ambos os membros são amantes do xadrez, então seria um jogo no qual ficaríamos bastante mais motivados a fazer pelas suas parecenças com o mesmo. Assim como no xadrez, é necessário haver um bom raciocínio mental e estratégico com o desenrolar de uma partida entre dois elementos.

O objetivo deste trabalho foi a aplicação dos primeiros conceitos interiorizados nas aulas teóricas e desenvolvidos nas aulas práticas da cadeira. Este método de avaliação torna-se importante pois permite-nos avaliar os conhecimentos que adquirimos até então e saber se somos ou não capazes de, com uma linguagem de programação nova e um paradigma completamente diferente do que estamos habituados, produzir algo de útil para o quotidiano e futuro.

2 Corrida de Reis

Esta variante do xadrez tradicional, Corrida de Reis, inventada por Vernon R. Parton em 1961, tem por objectivo levar o próprio rei até à última linha antes do adversário.

Vernon Rylands Parton foi um entusiasta de xadrez e um inventor prolífico de variantes para o mesmo, sendo o Xadrez de Alice a variante por ele criada mais conhecida. Muitas das variantes por ele inventadas possuíam inspiração de personagens fictícias e histórias dos trabalhos de Lewis Caroll. Parton, tal como Lewis Caroll, dedicou grande parte da sua vida académica à matemática, mas possuia interesses vários na ciência e era um forte apoiante de Esperanto.

Já face ao jogo em causa, Corrida de Reis, cada jogador começa o jogo com todas as peças normalmente usadas no xadrez exceto os peões, ou seja, começa com 1 rei, 1 rainha, 2 torres, 2 bispos e 2 cavalos. Todas as peças (brancas e pretas) são colocadas nas primeiras duas linhas do tabuleiro e ambos os jogadores veem o jogo da mesma perspetiva. Assim, o tabuleiro inicial tem o aspeto especificado na figura1.

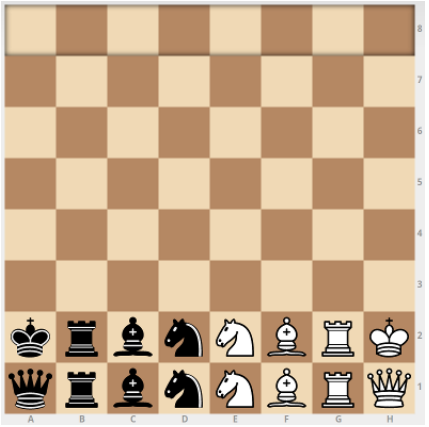

Figura 1: Tabuleiro base.

Como já referido, o objetivo é ser o primeiro a levar o próprio rei até à última linha (linha 8), usando as regras do Xadrez tradicional para mover e capturar as peças.

Contudo, impõem-se restrições adicionais, listadas abaixo:

- Não é permitido atacar o rei adversário, isto é, não se podem efetuar jogadas que coloquem o rei adversário em cheque;
- Um rei não pode mover-se para uma casa coberta por uma peça adversária.

Visto que para alcançar a vitória, um jogador deve mover o seu rei para a última linha, o jogador correspondente às peças brancas possuiria uma vantagem clara, por começar primeiro. Assim, quando é o rei branco que chega primeiro à última linha, o jogador preto tem uma ronda extra para que, caso consiga colocar o seu rei na última linha nessa jogada, declara-se um empate. Desta forma compensa-se a vantagem que as brancas têm por jogarem primeiro.

3 Lógica do Jogo

3.1 Representação do Estado do Jogo

O estado de jogo é guardado no tabuleiro, representado por uma lista de listas. O tabuleiro é de 8x8 e, por isso, a primeira lista conterá outras 8, cada uma dessas com 8 elementos (peças).

Para exemplificação, o código e comentários abaixo representam em linguagem PROLOG as posições iniciais das peças ao longo do tabuleiro. Cada número de representação interna ao programa é traduzido em um ou mais caracteres na consola do SICStus.

A chave de tradução é também apresentada em baixo.

```
/* Starting game board */
initialBoard([
        [0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0],
        [6,8,9,10,5,4,3,1],
        [7,8,9,10,5,4,3,2]]).

/* Associates each number with a chess piece */
translate(0,' .. ').
translate(1,' WK '). %White King
translate(2,' WQ '). %White Queen
translate(3,' WR '). %White Rook
translate(4,' WB '). %White Bishop
translate(5,' Wk '). %White Knight
translate(6,' BK '). %Black King
translate(7,' BQ '). %Black Queen
translate(8,' BR '). %Black Rook
translate(9,' BB '). %Black Bishop
translate(10,' Bk '). %Black Knight
```

Listing 1: Board inicial e Tradução.

3.2 Visualização do Tabuleiro

A representação interna do tabuleiro inicial será como a anteriormente apresentada, traduzindo-se no seguinte output de consola no SICStus Prolog:

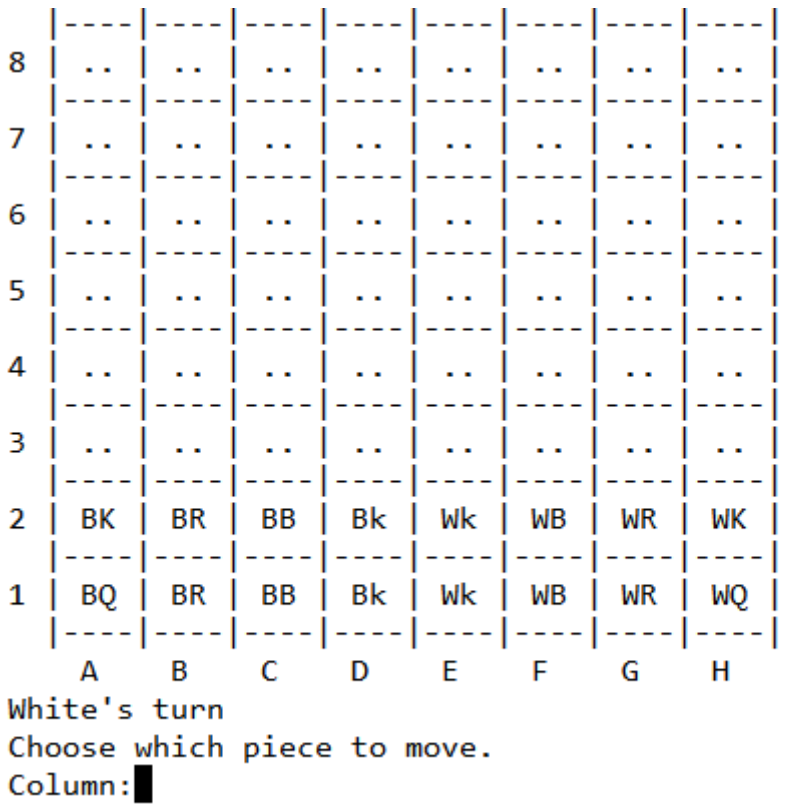

Figura 2: Layout de jogo inicial.

Face a esta representação, necessária, devido ao facto de um tabuleiro de xadrez ser disposto desta forma, com letras (crescentes da esquerda para a direita) para identificar as colunas e números (crescentes de baixo para cima) para identificar as linhas, conseguimos que para o utilizador, não haja dificuldades de adaptação a um tabuleiro diferente do utilizado no xadrez. No entanto, dificultou a implementação de inúmeras funções e do nosso raciocínio lógico, pois tínhamos de considerar qualquer input de linhas como o módulo da subtração do número máximo de linhas pela linha da posição escolhida pelo utilizador.

O tabuleiro inicial de jogo é criado usando o predicado *initialBoard(-Board)* em que X contem o tabuleiro inicial.

Para efeitos de apresentação foi construído o predicado *printBoard(+Board, +Y)* que recebe uma matriz Board (lista de listas) e a imprime na consola, sendo que o tabuleiro

atual é sempre representado no final de cada jogada, depois de a jogada ter sido aprovada pelos vários predicados de verificação. É, claramente, um predicado recursivo, que se auxilia noutro predicado, também ele recursivo, *printLine(+X)*. que recebe uma matriz X de elementos a imprimir na consola. O argumento Y é a lista de números a apresentar como sendo o nome das linhas do tabuleiro.

Assim, é passado ao predicado *printBoard(+Board, +Y)*. o tabuleiro de jogo a imprimir. O mesmo separa a matriz na notação [H—T] em que 'H' representa a cabeça da lista (Head) e 'T' a cauda da lista (Tail). A cabeça apresenta-se como uma lista com os elementos da linha a imprimir, sendo passada ao *printLine(+X)*. A cauda assume-se como uma lista de listas, sendo passada novamente (chamada recursiva) ao predicado *printBoard(+X, +Y)*. Cada linha é processada recursivamente, dividida em [H—T], sendo que agora a cabeça da lista representa um elemento que é traduzido usando a chave referida e impresso na consola. A restante linha assume-se como uma lista de elementos a imprimir, sendo feita uma chamada recursiva a *printLine(+X)*. Ambos os predicados apresentam como caso base o processamento de uma lista vazia.

3.3 Execução de Jogadas

Em cada jogada, é pedido ao jogador, que possui a vez de jogar, as coordenadas da peça no tabuleiro que deseja movimentar. Logo depois de as coordenadas serem validadas como coordenadas válidas dentro do tabuleiro de jogo, o programa verifica se as coordenadas correspondem a uma peça do jogador que tem a vez de jogar e não a uma das peças do oponente para além de se não são nulas. O pedido das coordenadas é feito com o predicado *getPiecePos(-Column, -Row)*, já o predicado *getPiece(+Column, +Row, +Board, -Piece)* vê que peça é para depois passar na verificação do dono da peça.

Uma vez validadas as coordenadas da peça a mover, é pedido ao jogador que insira as coordenadas do destino da peça que selecionou usando o predicado *getDestPos(-DestColumn, -DestRow)*. Após essas coordenadas serem validadas, verifica-se se as coordenadas de origem e destino do movimento são passíveis *validateMove(+Piece, +Row, +Column, +DestRow, +DestColumn, +Board)*. Caso as coordenadas de origem e destino não sejam válidas, o programa retrocede até ao ponto inicial.

Imediatamente a seguir, o predicado *setPiece(+DestColumn, +DestRow, +Piece, +Board2, -UpdatedBoard)* é chamado duas vezes para, efetivamente, mover a peça. Sendo que na primeira chamada coloca a posição anterior como vazia, e na seguinte coloca a peça na posição de destino.

Finalmente, depois da jogada completa ter sido efetuada, é novamente chamado o

predicado printBoard(+Board, +Y) que se responsabiliza por imprimir o tabuleiro atual de jogo.

Cada jogada de qualquer modo de jogo ora ocorre dentro do predicado playerMove(+Board, +Player, -UpdatedBoard), que pode ter aridade 2 (sem Player) se for jogador contra o computador, ora ocorre no botMove(+Board, -UpdatedBoard, +Color), na qual o argumento color decide de que cor é a "inteligência artificial", para os jogos de Computador versus Computador.

As chamadas para os predicados das jogadas, ocorrem nos predicados *main* de cada modo de jogo, por exemplo, *mainPlayerVPlayer*, sendo que neste último a cada iteração alteramos o jogador atual, já nos outros predicados *main* são chamados os predicados de jogada duas vezes por iteração.

3.4 Avaliação do Tabuleiro e Jogadas Válidas

A avaliação do tabuleiro ao longo do jogo é feita através do predicado indicado a seguir.

```
%Generic validate
validateMove(Piece, R, C, DesR, DesC, Board):-

 getPiece(DesC, DesR, Board, DesPiece),

  %Check if destiny has friendly piece.
  ite(Piece =< 5, (DesPiece >= 6 ; DesPiece == 0), true),
  ite(Piece >= 6, DesPiece =< 5, true),

  %Check if movement is allowed.
  ite((Piece==1 ; Piece==6),validateKingMove(R, C, DesR, DesC),true),
  ite((Piece==2 ; Piece==7),validateQueenMove(R, C, DesR, DesC),true),
  ite((Piece==3 ; Piece==8),validateRookMove(R, C, DesR, DesC),true),
  ite((Piece==4 ; Piece==9),validateBishopMove(R, C, DesR, DesC),true),
  ite((Piece==5 ; Piece==10),validateKnightMove(R, C, DesR, DesC),true).
```

Listing 2: Validação.

Cada peça tem o seu próprio predicado de validação devido às limitações individuais de cada peça, e, por isso, é chamado um predicado diferente para cada. Nós conseguimos utilizar todo este predicado de validação através da utilização do predicado ite, que equivale a um simples If, Then, Else.

```
ite(If,Then,_Else):-
  If,!,Then.

ite(_If,_Then,Else):-
  Else.
```

Listing 3: If, then, else.

3.5 Final do Jogo

A verificação do término do jogo, ao contrário dos predicados de validação é bastante simples de implementar, visto que, no nosso jogo, a única condição de fim de jogo é quando um dos reis chega à linha final.

```

gameOver(Board):-
  nth0(0, Board, LastRow),
  ite((member(6,LastRow) ; member(1,LastRow)),
    ite((member(6,LastRow) , member(1,LastRow)),
    drawMessage, gameOverMessage), true).
```

Listing 4: If, then, else.

No entanto, não nos podemos esquecer que na possibilidade de o rei branco chegar primeiro, temos depois de dar a oportunidade de empate para o jogador preto, para tal verificamos a condição de fim de jogo apenas no fim da jogada do jogador preto.

3.6 Jogada do Computador

Implementámos portanto dois predicados de movimento de computador, um completamente aleatório, e outro ganancioso. botMove(+Board, -UpdatedBoard, +Color) O bot normal escolhe aleatoriamente uma peça aleatória do tabuleiro até uma dessas peças ser uma das suas e tenta movimenta-la para uma posição qualquer do mapa até um desses movimentos ser possível. botHardMove(+Board, -UpdatedBoard, +Color) O bot ganancioso, em cada jogada, avança em primeira instância o seu rei, e, apenas se não o puder fazer faz qualquer outro movimento. Em segunda instância, o bot avança com um movimento aleatório do bot anterior.

4 Interface com o Utilizador

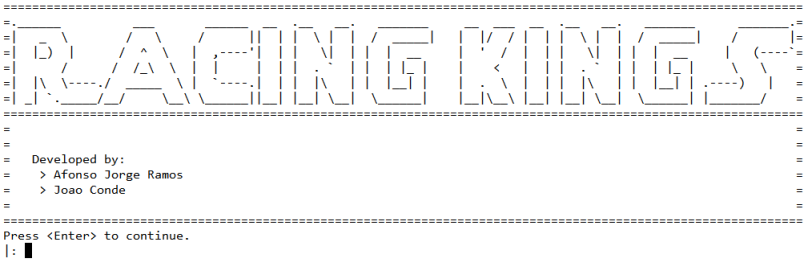

Figura 3: Splash screen inicial.

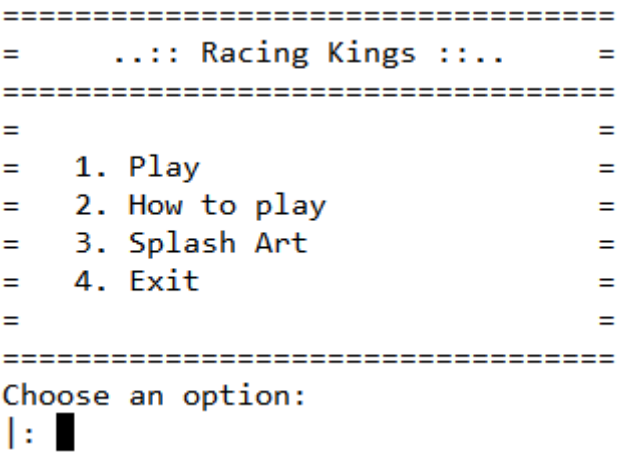

Figura 4: Menu.

```
================================================================================
=                        ..:: How to play ::..                                 =
================================================================================
=                                                                              =
= White pieces start.                                                          =
=                                                                              =
= The objective is to move your king in order to reach the last row first.     =
=                                                                              =
= No move can place your or the enemy's king in check.                         =
=                                                                              =
= If the White king reaches the last row first, the Black king has one         =
= extra move to attempt to reach this last row in order to tie the match.      =
=                                                                              =
= All other rules related to each piece are the same as normal chess.          =
=                                                                              =
================================================================================
Press <Enter> to continue.
|:
```

Figura 5: Regras.

```
==================================
=      ..:: Game Mode ::..       =
==================================
=                                =
=   1. Player vs. Player         =
=   2. Player vs. Computer       =
=   3. Computer vs. Computer     =
=   4. Back                      =
=                                =
==================================
Choose an option:
|:
```

Figura 6: Modo de Jogo.

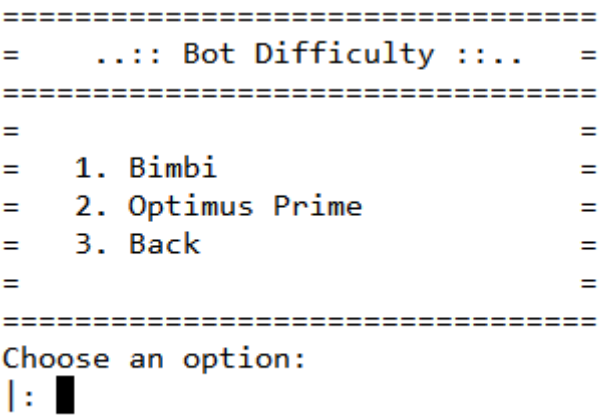

Figura 7: Dificuldades dos bots.

5 Conclusões

O jogo Racing Kings exigiu imenso tempo a ambos os elementos do grupo no seu desenvolvilmento. Por fim, apresentamos as nossas conclusões finais relativamente à implementação do mesmo. O grupo apesar de apreciar o resultado final do seu trabalho, considera que mais e melhor poderia ter sido feito com, possivelmente, mais tempo. Contudo, os conhecimentos adquiridos durante o desenvolvimento do projeto são significáveis e relevantes para o futuro caso necessitemos de desenvolver uma aplicação que recorra ao PROLOG. Apesar do escasso tempo para a entrega final do projeto e as consecutivas sobreposições de trabalhos de outras unidades curriculares, conseguimos concluir a grande maioria do que havíamos planeado. Racing Kings mostrou-se um desafio interessante que, com esforço, dedicação e empenho se tornou numa variante do xadrez muito apelativo e simples que proporciona ao jogador uma boa prática mental e é, como se esperava, um bom passatempo. As dificuldades encontradas foram superadas, porém poderiam haver melhorias, nomeadamente na forma como o algoritmo ganacioso da escolha do movimento randomizado dos bots foi implementado. Em suma o grupo gostou da experiência de desenvolvimento de um jogo na linguagem PROLOG. Ao contrário do que estamos habituados, este tipo de linguagem requer um pensamento lógico em cada predicado desenvolvido.

6 Anexos

6.1 menus.pl

```
%=================%
%= @@ game menus =%
%=================%
mainMenu:-
  printMainMenu,
  getChar(Input),
  (
    Input = '1' -> gameModeMenu, mainMenu;
    Input = '2' -> helpMenu, mainMenu;
    Input = '3' -> splashScreen;
    Input = '4';

    nl, write('Error: invalid input.'), nl,
    pressEnterToContinue, nl,
    mainMenu
  ).

```

```
18 splashScreen:-
19   clearScreen,
20   write('
       ==========================================================================================
       '), nl,
21   write('=.------.            ---        ------   --  .--    --.    -------
       --   ---   --  .--    --.    -------       --------.='), nl,
22   write('=|   _  \\         /   \\      /       ||  | |  \\ |  |  /  -----|
       |  |/  / |  | |  \\ |  |  /  -----|    /        |='), nl,
23   write('=|  |_)  |       /  ^  \\    |  ,----\'|  | |    \\|  | |  |  --
       |  \'  /  |  | |   \\|  | |  |  --      |   (----\'='), nl,
24   write('=|       /       /  /_\\  \\  |  |      |  | |  . \'  | |  | |_ |
       |    <   |  | |  . \'  | |  | |_ |      \\    \\     ='), nl,
25   write('=|  |\\  \\----./  -----  \\ |  \'----.|  | |  |\\    | |  |__| |
       |  .  \\  |  | |  |\\   | |  |__| .----)   |    ='), nl,
26   write('=| _| \'.-----/__/     \\__\\ \\------||__| |__| \\__|  \\------
       |    |__|\\__\\ |__| |__| \\__|  \\------| |-------/    ='), nl,
27   write('
       ==========================================================================================
       '), nl,
28   write('=
                                                          ='), nl,
29   write('=
                                                          ='), nl,
30   write('=   Developed by:
                                                          ='), nl,
31   write('=    > Afonso Jorge Ramos
                                                          ='), nl,
32   write('=    > Joao Conde
                                                          ='), nl,
33   write('=
                                                          ='), nl,
34   write('
       ==========================================================================================
       '), nl,
35   pressEnterToContinue, mainMenu.
36
37 printMainMenu:-
38   clearScreen,
39   write('=========================================='), nl,
40   write('=     ..:: Racing Kings ::..       ='), nl,
41   write('=========================================='), nl,
42   write('=                                      ='), nl,
43   write('=   1. Play                            ='), nl,
44   write('=   2. How to play                     ='), nl,
45   write('=   3. Splash Art                      ='), nl,
46   write('=   4. Exit                            ='), nl,
47   write('=                                      ='), nl,
48   write('=========================================='), nl,
```

```
49    write('Choose an option:'), nl.
50
51  gameModeMenu:-
52    printgameModeMenu,
53    getChar(Input),
54    (
55      Input = '1' -> clearScreen, playerVPlayer;
56      Input = '2' -> clearScreen, difficultyMenu;
57      Input = '3' -> clearScreen, botVBot;
58      Input = '4';
59
60      nl,
61      write('Error: invalid input.'), nl,
62      pressEnterToContinue, nl,
63      gameModeMenu
64    ).
65
66  difficultyMenu:-
67    printDifficultyMenu,
68    getChar(Input),
69    (
70      Input = '1' -> clearScreen, playerVBot;
71      Input = '2' -> clearScreen, playerVBotHard;
72      Input = '3';
73
74      nl,
75      write('Error: invalid input.'), nl,
76      pressEnterToContinue, nl,
77      difficultyMenu
78    ).
79
80  printDifficultyMenu:-
81    clearScreen,
82    write('======================================'), nl,
83    write('=    ..:: Bot Difficulty ::..      ='), nl,
84    write('======================================'), nl,
85    write('=                                    ='), nl,
86    write('=    1. Bimbi                        ='), nl,
87    write('=    2. Optimus Prime                ='), nl,
88    write('=    3. Back                         ='), nl,
89    write('=                                    ='), nl,
90    write('======================================'), nl,
91    write('Choose an option:'), nl.
92
93  printgameModeMenu:-
94    clearScreen,
95    write('======================================'), nl,
96    write('=       ..:: Game Mode ::..          ='), nl,
97    write('======================================'), nl,
```

```
98    write('=                                    ='), nl,
99    write('=    1. Player vs. Player            ='), nl,
100   write('=    2. Player vs. Computer          ='), nl,
101   write('=    3. Computer vs. Computer        ='), nl,
102   write('=    4. Back                         ='), nl,
103   write('=                                    ='), nl,
104   write('======================================'), nl,
105   write('Choose an option:'), nl.
106
107 helpMenu:-
108   clearScreen,
109   write('
      ================================================================================
      '), nl,
110   write('=                          ..:: How to play ::..
                ='), nl,
111   write('
      ================================================================================
      '), nl,
112   write('=
                ='), nl,
113   write('= White pieces start.
                ='), nl,
114   write('=
                ='), nl,
115   write('= The objective is to move your king in order to reach the last
       row first. ='), nl,
116   write('=
                ='), nl,
117   write('= No move can place your or the enemy\'s king in check.
                ='), nl,
118   write('=
                ='), nl,
119   write('= If the White king reaches the last row first, the Black king
       has one     ='), nl,
120   write('= extra move to attempt to reach this last row in order to tie
       the match.  ='), nl,
121   write('=
                ='), nl,
122   write('= All other rules related to each piece are the same as normal
       chess.      ='), nl,
123   write('=
                ='), nl,
124   write('
      ================================================================================
      '), nl,
125   pressEnterToContinue, nl.
```

6.2 utils.pl

```
/*
  0 - empty space
  1 - white king
  2 - white queen
  3 - white tower
  4 - white bishop
  5 - white knight
  6 - black king
  7 - black queen
  8 - black tower
  9 - black bishop
 10 - black knight
*/

/* Associates each number with a chess piece */
translate(0,' .. ').
translate(1,' WK '). %King
translate(2,' WQ '). %Queen
translate(3,' WR '). %Rook
translate(4,' WB '). %Bishop
translate(5,' Wk '). %Knight
translate(6,' BK '). %King
translate(7,' BQ '). %Queen
translate(8,' BR '). %Rook
translate(9,' BB '). %Bishop
translate(10,' Bk '). %Knight


columnToInt('A', 0).
columnToInt('B', 1).
columnToInt('C', 2).
columnToInt('D', 3).
columnToInt('E', 4).
columnToInt('F', 5).
columnToInt('G', 6).
columnToInt('H', 7).
columnToInt('a', 0).
columnToInt('b', 1).
columnToInt('c', 2).
columnToInt('d', 3).
columnToInt('e', 4).
columnToInt('f', 5).
columnToInt('g', 6).
columnToInt('h', 7).

/* Starting game board */
initialBoard([[0,0,0,0,0,0,0,0],
```

```
48                  [0,0,0,0,0,0,0,0],
49                  [0,0,0,0,0,0,0,0],
50                  [0,0,0,0,0,0,0,0],
51                  [0,0,0,0,0,0,0,0],
52                  [0,0,0,0,0,0,0,0],
53                  [6,8,9,10,5,4,3,1],
54                  [7,8,9,10,5,4,3,2]]).
55
56
57 lineNumbers([8,7,6,5,4,3,2,1]).
58
59 clearScreen:-write('\e[2J').
60
61 pressEnterToContinue:-
62   write('Press <Enter> to continue.'), nl,
63   waitForEnter, !.
64
65 waitForEnter:-
66   get_char(_).
67
68 getChar(Input):-
69   get_char(Input),
70   write(Input),nl,
71   get_char(_).
72
73 getCode(Input):-
74   get_code(TempInput),
75   get_code(_),
76   Input is TempInput - 48.
77
78 getInt(Input):-
79   get_code(TempInput),
80   get_code(_),
81   Input is TempInput - 48.
82
83 discardInputChar:-
84   get_code(_).
85
86 /* Recursive function to print a ASCII CODE a specific number of times */
87 %Ended up not being used since there was a need to use a specific font
       that I didn't like.
88 putCode(Times, Code) :-
89       Times > 0, !,
90       put_code(Code),
91       TimesN is Times - 1,
92       putCode(TimesN, Code).
93
94 putCode(0,[]).
95
```

```
96  /*IF-THEN-ELSE*/
97  ite(If,Then,_Else):-
98    If,!,Then.
99
100 ite(_If,_Then,Else):-
101   Else.
```

6.3 board.pl

```
printInitialBoard :-
	initialBoard(X),
	lineNumbers(Y),
	printBoard(X,Y).

printMidBoard :-
	midgameBoard(X),
	lineNumbers(Y),
	printBoard(X,Y).

printEndBoard :-
	endgameBoard(X),
	lineNumbers(Y),
	printBoard(X,Y).

player1(white).
player2(black).

playerTurn(white, 'White').
playerTurn(black, 'Black').

/* Recursive function to print current board state */
printBoard([],[]) :-
	write('  |----|----|----|----|----|----|----|----|'), nl,
	write('    A    B    C    D    E    F    G    H   ').

/*printBoard([],[]) :-
	write('   '), putCode(25,205), nl,
	write('   A  B  C  D  E  F  G  H ').*/

printBoard([Line|Board],[LineNumb|Remainder]) :-
	write('  |----|----|----|----|----|----|----|----|'), nl,
	%write('   '), putCode(25,205),  nl,
	write(LineNumb), write(' '),
	printLine(Line),
	write('|'), nl,
	printBoard(Board,Remainder).

/* Recursive function to print each board's line */
printLine([]).
printLine([Head|Tail]) :-
	translate(Head,T),
	write('|'),
	%put_code(186),
	write(T),
	printLine(Tail).

```

```
48 /* Get element at (row,col) */
49 getPiecePos(Column, Row):-
50   repeat,
51       write('Choose which piece to move.'), nl,
52   write('Column:'),
53   getChar(Char),
54   columnToInt(Char, Column),
55   write('Row:'),
56   getInt(R),
57   R =< 8,
58   R >= 1,
59   Row is R - 1.
60
61
62 getDestPos(Column, Row):-
63       repeat,
64       write('Choose where to move.'), nl,
65       write('Column:'),
66       getChar(Char),
67       columnToInt(Char, Column),
68       write('Row:'),
69       getInt(R),
70       R =< 8,
71       R >= 1,
72       Row is R - 1.
73
74 /* MOVES VALIDATION*/
75
76 %Generic validate
77 validateMove(Piece, R, C, DesR, DesC, Board):-
78       getPiece(DesC, DesR, Board, DesPiece),
79
80       %Check if destiny has friendly piece.
81       ite(Piece =< 5, (DesPiece >= 6 ; DesPiece == 0), true),
82       ite(Piece >= 6, DesPiece =< 5, true),
83
84       %lmao
85       DesPiece \= 1,
86       DesPiece \= 6,
87
88       %Check if movement is allowed.
89       ite((Piece==1 ; Piece==6),validateKingMove(R, C, DesR, DesC),true),
90       ite((Piece==2 ; Piece==7),validateQueenMove(R, C, DesR, DesC),true)
    ,
91       ite((Piece==3 ; Piece==8),validateRookMove(R, C, DesR, DesC),true),
92       ite((Piece==4 ; Piece==9),validateBishopMove(R, C, DesR, DesC),true
    ),
93       ite((Piece==5 ; Piece==10),validateKnightMove(R, C, DesR, DesC),
    true).
```

```
94
95
96
97
98  checkForCheck(0,_,_,_,_,_,_,_,_,_,_).
99  checkForCheck(Rows, [Head|Tail], WKingR, WKingC, BKingR, BKingC, Board)
        :-
100     Rows > 0,
101     Rows1 is Rows-1,
102     ite((checkForCheckList(8, Head, WKingR, WKingC, BKingR, BKingC, Board
        ),write('PORQUE')),(write('CARALHO'),true),(!,fail)),
103     checkForCheck(Rows1,Tail, WKingR, WKingC, BKingR, BKingC, Board).
104
105
106 checkForCheckList(0,[],_,_,_,_,_,_,_,_).
107 checkForCheckList(Cols, [Head|Tail], WKingR, WKingC, BKingR, BKingC,
        Board) :-
108       Cols > 0,
109       Cols1 is (Cols - 1),
110       write('CHECKLIST: '),write(Cols1),nl,
111       Head \= 1, Head \= 6,
112       getPiece(PieceC, PieceR, Board, Head),
113       write(Head),nl,write(PieceR),nl,write(PieceC),nl,write(BKingR),nl,
        write(BKingC),nl,write(WKingR),nl,write(WKingC),nl,
114       pressEnterToContinue,
115       ite(Head =< 5,ite(validateMove(Head,PieceR, PieceC,BKingR,BKingC,
        Board),(write('FAIL'),fail),true),true),
116       ite(Head >= 6,ite(validateMove(Head,PieceR, PieceC,WKingR,WKingC,
        Board),(write('FAIL'),fail),true),true),
117
118       checkForCheckList(Cols1, Tail, WKingR, WKingC, BKingR, BKingC,
        Board).
119
120
121 validateKingMove(KingR,KingC,Row,Col):-
122       Row =< (KingR + 1),
123       Row >= (KingR - 1),
124       Col >= (KingC - 1),
125       Col =< (KingC + 1).
126
127 validateRookMove(RookR,RookC,Row,Col):-
128       RookC =:= Col ; RookR =:= Row.
129
130
131 validateQueenMove(QueenR,QueenC,Row,Col):-
132   validateRookMove(QueenR,QueenC,Row,Col) ;
133   validateBishopMove(QueenR,QueenC,Row,Col).
134
135
```

```
136 validateBishopMove(BishopR,BishopC,Row,Col):-
137       %Distance is 0,
138       %maxDistanceBishop(Distance,BishopR,BishopC,Row,Col,Board), %doenst
      allow to skip pieces (should)
139       DifR is abs(BishopR - Row),
140       DifC is abs(BishopC - Col),
141       %DifC < Distance, DifR < Distance,
142       BishopC \= Col , BishopR \= Row,
143       DifC == DifR.
144
145 validateKnightMove(KnightR,KnightC,Row,Col):-
146       (Row =:= (KnightR + 2), Col =:= (KnightC - 1));
147       (Row =:= (KnightR + 1), Col =:= (KnightC - 2));
148       (Row =:= (KnightR + 2), Col =:= (KnightC + 1));
149       (Row =:= (KnightR + 1), Col =:= (KnightC + 2));
150       (Row =:= (KnightR - 2), Col =:= (KnightC - 1));
151       (Row =:= (KnightR - 1), Col =:= (KnightC - 2));
152       (Row =:= (KnightR - 2), Col =:= (KnightC + 1));
153       (Row =:= (KnightR - 1), Col =:= (KnightC + 2)).
154
155
156 %max distance allowed to travel calculator for bishop
157 maxDistanceBishop(MaxDistance,BishopR,BishopC,Row,Col,Board):-
158     DeltaR is (Row - BishopR),
159     DeltaC is (Col - BishopC),
160     ite(DeltaC > 0,
161             ite(DeltaR > 0, maxDistanceBishopDir1(MaxDistance,BishopR,
    BishopC,Row,Col,Board), maxDistanceBishopDir2(MaxDistance,BishopR,
    BishopC,Row,Col,Board)),
162             ite(DeltaR > 0, maxDistanceBishopDir4(MaxDistance,BishopR,
    BishopC,Row,Col,Board), maxDistanceBishopDir3(MaxDistance,BishopR,
    BishopC,Row,Col,Board))).
163
164
165 maxDistanceBishopDir1(MaxDistance,BishopR,BishopC,Row,Col,Board):-
166   BishopR < Row,
167   BishopC < Col,
168   NBishopC is (BishopC + 1),
169   NBishopR is (BishopR + 1),
170   getPiece(NBishopC,NBishopR,Board,Piece),
171   Piece == 0,
172   MaxDistance1 is (MaxDistance + 1),
173   maxDistanceBishopDir1(MaxDistance1,NBishopC,NBishopR,Row,Col,Board).
174
175 maxDistanceBishopDir2(MaxDistance,BishopR,BishopC,Row,Col,Board):-
176   BishopR > Row,
177   BishopC < Col,
178   NBishopC is (BishopC + 1),
179   NBishopR is (BishopR - 1),
```

```
180   getPiece(NBishopC,NBishopR,Board,Piece),
181   Piece == 0,
182   MaxDistance1 is (MaxDistance + 1),
183   maxDistanceBishopDir2(MaxDistance1,NBishopC,NBishopR,Row,Col,Board).
184
185
186 maxDistanceBishopDir3(MaxDistance,BishopR,BishopC,Row,Col,Board):-
187   BishopR > Row,
188   BishopC > Col,
189   NBishopC is (BishopC - 1),
190   NBishopR is (BishopR - 1),
191   getPiece(NBishopC,NBishopR,Board,Piece),
192   Piece == 0,
193   MaxDistance1 is (MaxDistance + 1),
194   maxDistanceBishopDir3(MaxDistance1,NBishopC,NBishopR,Row,Col,Board).
195
196
197 maxDistanceBishopDir4(MaxDistance,BishopR,BishopC,Row,Col,Board):-
198   BishopR < Row,
199   BishopC > Col,
200   NBishopC is (BishopC - 1),
201   NBishopR is (BishopR + 1),
202   getPiece(NBishopC,NBishopR,Board,Piece),
203   Piece == 0,
204   MaxDistance1 is (MaxDistance + 1),
205   maxDistanceBishopDir4(MaxDistance1,NBishopC,NBishopR,Row,Col,Board).
```

6.4 botting.pl

```
botMove(Board, UpdatedBoard, Color) :-
  lineNumbers(Y1),
  %trace,
  randomMove(C, R, DesC, DesR, Board, Piece, Color),
  write(R),
  %AbsR is abs(7 - R),
  setPiece(C, R, 0, Board, Board2),
  %notrace,
  %NewDesR is abs(7 - DesR),
  setPiece(DesC, DesR, Piece, Board2, UpdatedBoard),
  clearScreen,
  printBoard(UpdatedBoard, Y1),
  nl, ite(Color == 0,(write('Black Bot\'s turn')), (write('White Bot\'s
    turn'))),
  nl, pressEnterToContinue.


botHardMove(Board, UpdatedBoard, Color) :-
  lineNumbers(Y1),
  ite(randomAdvancedMove(C, R, DesC, DesR, Board),Piece = 6,
      randomMove(C, R, DesC, DesR, Board, Piece, Color)),
  setPiece(C, R, 0, Board, Board2),
  setPiece(DesC, DesR, Piece, Board2, UpdatedBoard),
  write(DesC),nl, write(DesR), nl,
  %clearScreen,
  printBoard(UpdatedBoard, Y1),
  nl, write('Black Bot\'s turn'),
  nl, pressEnterToContinue.


randomAdvancedMove(Col, Row, DesC, DesR, Board):-
  getPiece(Col, Row, Board, 6),
  DesC is Col,
  DesR is (Row-1),
  validateMove(6, Row, Col, DesR, DesC, Board).


randomMove(Col, Row, DesC, DesR, Board, Piece, Color) :-
  randomPiece(Col,Row,Board,Piece, Color),
  randomDestination(Col, Row, DesC, DesR, Board, Piece).


randomDestination(Col, Row, DesC, DesR, Board, Piece):-
  repeat,
  random(0, 8, DesC),
  random(0, 8, WrongDesR),
  DesR is abs(7 - WrongDesR),
```

```
47    validateMove(Piece, Row, Col, DesR, DesC, Board).
48
49  randomPiece(Col, Row, Board, Piece, Color):-
50    repeat,
51    random(0, 8, Col),
52    random(0, 8, WrongRow),
53    Row is abs(7 - WrongRow),
54    getPiece(Col, Row, Board, Piece),
55    Piece \= 0,
56    ite(Color == 1,Piece >= 6, Piece =< 5).
57
58  %randomPiece(Col, Row, Board, Piece).
```

6.5 racingkings.pl

```
:- use_module(library(lists)).
:- use_module(library(random)).

:- include('menus.pl').
:- include('utils.pl').
:- include('board.pl').
:- include('botting.pl').

racingkings:-
  splashScreen.

playerVPlayer:-
  initialBoard(T),
  lineNumbers(Y),
  printBoard(T,Y),
  nl,
  player1(PlayerInit),
  mainPlayerVPlayer(T, PlayerInit).

playerVBot:-
  initialBoard(T),
  lineNumbers(Y),
  printBoard(T,Y),
  nl,
  mainPlayerVBot(T).

playerVBotHard:-
  initialBoard(T),
  lineNumbers(Y),
  printBoard(T,Y),
  nl,
  mainPlayerVBotHard(T).

botVBot:-
  initialBoard(T),
  lineNumbers(Y),
  printBoard(T,Y),
  nl, write('White Bot\'s turn'), nl ,pressEnterToContinue,
  mainBotVBot(T).

mainPlayerVPlayer(Board, CurrentPlayer):-
  playerMove(Board, CurrentPlayer, UpdatedBoard),
  ite((CurrentPlayer == white), player2(NextPlayer), player1(NextPlayer))
    ,
  ite((CurrentPlayer == black), gameOver(UpdatedBoard), true),
  mainPlayerVPlayer(UpdatedBoard, NextPlayer).

```

```
47 mainPlayerVBot(Board):-
48   playerMove(Board, UpdatedBoard),
49   botMove(UpdatedBoard, UpdatedBotBoard, 1),
50   gameOver(UpdatedBoard),
51   mainPlayerVBot(UpdatedBotBoard).
52
53
54 mainPlayerVBotHard(Board):-
55   playerMove(Board, UpdatedBoard),
56   botHardMove(UpdatedBoard, UpdatedBotBoard, 1),
57   gameOver(UpdatedBoard),
58   mainPlayerVBotHard(UpdatedBotBoard).
59
60
61 mainBotVBot(Board):-
62   botMove(Board, UpdatedBoard, 0),
63   botMove(UpdatedBoard, UpdatedBotBoard, 1),
64   gameOver(UpdatedBoard),
65   mainBotVBot(UpdatedBotBoard).
66
67 playerMove(Board1, P, UpdatedBoard):-
68   lineNumbers(Y1),
69   playerTurn(P, PlayerName),
70   write(PlayerName), write('\'s turn'),nl,
71   getPiecePos(C,R),
72   AbsR is abs(7 - R),
73   getPiece(C, AbsR, Board1, Piece),
74   Piece \= 0,
75   ite((P==white),Piece =< 5, Piece >= 6),
76   once(getDestPos(DesC,DesR)),
77   AbsDesR is abs(7 - DesR),
78   validateMove(Piece,AbsR,C,AbsDesR,DesC, Board1),
79
80   setPiece(C, AbsR, 0, Board1, Board2),
81   setPiece(DesC, AbsDesR, Piece, Board2, UpdatedBoard),
82 /*
83   %check for kings check
84   getPiece(BlackKingC, BlackKingR, UpdatedBoard, 6),
85   getPiece(WhiteKingC, WhiteKingR, UpdatedBoard, 1),
86
87   NewBoard = UpdatedBoard,
88   ite(checkForCheck(8, UpdatedBoard, WhiteKingR, WhiteKingC, BlackKingR,
       BlackKingC, NewBoard),true,(!,fail)),
89 */
90   clearScreen,
91   printBoard(UpdatedBoard, Y1),
92   nl, pressEnterToContinue.
93
94 playerMove(Board, UpdatedBoard):-
```

```
95    lineNumbers(Y1),
96    write('Your turn'),nl,
97    getPiecePos(C,R),
98    AbsR is abs(7 - R),
99    getPiece(C, AbsR, Board, Piece),
100   Piece \= 0, Piece =< 5,
101   once(getDestPos(DesC, DesR)),
102   AbsDesR is abs(7 - DesR),
103   validateMove(Piece, AbsR, C, AbsDesR, DesC, Board),
104   setPiece(C, AbsR, 0, Board, Board2),
105   setPiece(DesC, AbsDesR, Piece, Board2, UpdatedBoard),
106   clearScreen,
107   printBoard(UpdatedBoard, Y1),
108   nl, pressEnterToContinue.
109
110 getPiece(Col, Row, Board, Piece):-
111   nth0(Row, Board, RowList),
112   nth0(Col, RowList, Piece).
113
114 setPiece(ElemCol, 0, NewElem, [RowAtTheHead|RemainingRows], [
      NewRowAtTheHead|RemainingRows]):-
115   setPieceList(ElemCol, NewElem, RowAtTheHead, NewRowAtTheHead).
116
117 setPiece(ElemCol, ElemRow, NewElem, [RowAtTheHead|RemainingRows], [
      RowAtTheHead|ResultRemainingRows]):-
118   ElemRow > 0,
119   ElemRow1 is ElemRow-1,
120   setPiece(ElemCol, ElemRow1, NewElem, RemainingRows, ResultRemainingRows
      ).
121
122 setPieceList(0, Elem, [_|L], [Elem|L]).
123 setPieceList(I, Elem, [H|L], [H|ResL]):-
124   I > 0,
125   I1 is I-1,
126   setPieceList(I1, Elem, L, ResL), !.
127
128
129 gameOverMessage:-
130   write('GAME OVER'),nl,
131   pressEnterToContinue,
132   mainMenu.
133
134 drawMessage:-
135   write('It\'s a tie.'),nl,
136   pressEnterToContinue,
137   mainMenu.
138
139 gameOver(Board):-
140   nth0(0, Board, LastRow),
```

```
141   ite((member(6,LastRow) ; member(1,LastRow)),ite((member(6,LastRow) ,
        member(1,LastRow)), drawMessage, gameOverMessage), true).
```