Crab Stack

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo 2:

Sara Santos - up201402814

Nuno Castro - up201406990

Faculdade de Engenharia da Universidade do Porto

Rua Roberto Frias, sn, 4200-465, Porto, Portugal

12 de Novembro de 2016

Resumo

O projeto consiste num jogo de tabuleiro para duas pessoas em que cada jogador possuir 3 caranguejos grandes, 3 médios e 3 pequenos. O jogador ganha quando o outro jogador não conseguir efetuar jogadas válidas. O objetivo era fazer uma representação o mais fiel possível ao jogo original, adaptando-o para apenas dois jogadores. Tendo em conta que os caranguejos podem formar torres, contruímos um tabuleiro baseado em listas em que cada posição, também uma lista, representa a Stack de caranguejos, sendo possível apenas movimentar o que está no topo.

No final obtivemos um jogo bastante idêntico ao original e que nos ajudou a consolidar conhecimentos obtidos desde o inicio do semestre. Deparámo-nos com algumas dificuldades, nomeadamente na documentação de certos predicados, que foram ultrapassados com alguma pesquisa da nossa parte.

Conteúdo

1. Introdução

Este projeto surge no âmbito da unidade curricular Programação em Lógica do 3º ano do Mestrado Integrado em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto, com o objetivo de implementar um jogo de tabuleiro em PROLOG com interface textual. Foi utilizado o software recomendado, SicStus-Prolog.

O principal objetivo era conseguirmos obter uma implementação fiel ao jogo original. Como tal, desenvolvemos uma interface simples e adaptámos o jogo apenas a 2 jogadores (inicialmente seria até 4 jogadores).

2. O Jogo Crab Stack

Crab Stack é um jogo de tabuleiro criado pela companhia Blue Orange Games, fundada em 2000 por Julien Mayot e Thierry Denoual. O objetivo do jogo é ser o último jogador com pelo menos um caranguejo (peça) que se consiga movimentar seguindo as instruções originais, as regras e o tabuleiro foram adaptados para a versão de apenas dois jogadores.

Existem diferentes tipos de peças de diferentes cores (verde e vermelho) e tamanhos (pequeno, médio e grande).

Cada jogador pode mover apenas um dos seus caranguejos no seu respetivo turno, respeitando o tipo de movimento correspondente ao seu tamanho. O jogador só pode mover os seus caranguejos através de outros caranguejos, não podendo movimentá-los através de rochas vazias ou através da água e não podem voltar a uma posição em que já tenham estado nesse turno.

Quando os caranguejos formam uma coluna (Stack), apenas o caranguejo do topo se pode mexer. Formar colunas com os caranguejos segue algumas regras:

- Os caranguejos grandes podem formar colunas com qualquer outro caranguejo, incluindo outros caranguejos grandes.
- Os caranguejos médios podem formar colunas apenas com outros caranguejos médios ou caranguejos pequenos.
- Os caranguejos pequenos apenas podem formar colunas com outros caranguejos pequenos.

Se um jogador não conseguir mover nenhum dos seus caranguejos no inicio do seu turno é eliminado do jogo. O último jogador a ser eliminado é o vencedor. Se o jogo chegar a um ponto em que os jogadores continuam a repetir as mesmas jogadas por não haver outras possíveis é considerado um empate e será jogado mais um jogo para decidir o vencedor.

3. Lógica do Jogo

3.1. Representação do Estado do Jogo

O tabuleiro do jogo é representado através de listas. São utilizados diferentes átomos que representam o tabuleiro e os diferentes tipos de peças. Dependendo das suas características os átomos são representados da seguinte maneira:

Tipo	*Átomo*	*Representação no tabuleiro*
Casa Vazia	.	.
Caranguejos pequenos Jogador 1	cp1	a
Caranguejos médios Jogador 1	cm1	A
Caranguejos grandes Jogador 1	cg1	*
Caranguejos pequenos Jogador 2	cp2	b
Caranguejos médios Jogador 2	cm2	B
Caranguejos grandes Jogador 2	cg2	+

```
empty_board(Board):-
    Board=  [[[.],[.],[.]],
             [[.],[.],[.],[.]],
             [[.],[.],[.],[.],[.]],
             [[.],[.],[.],[.]],
             [[.],[.],[.]]].
```

```
initial_board(Board):-
    Board = [[[cp1,.],[cg1,.],[cm1,.]],
             [[cp2,.],[cg1,.],[cg2,.],[cm2,.]],
             [[cg2,.],[cm1,.],[cp1,.],[.],[cp2,.]],
             [[cm2,.],[cg1,.],[cg2,.],[cm2,.]],
             [[cp1,.],[cp2,.],[cm1,.]]].
```

FIGURA 1 TABULEIRO VAZIO E INICIAL

3.2. Visualização do Tabuleiro

Foi implementado o predicado para a visualização, recebendo como argumento um tabuleiro:

```
print_gameboard(Board):-
    write('   '),
    write_spaces(1),
    top_borders(1),nl,
    printBoard(Board,0,5).

printBoard(_,F,F).
printBoard([B|Bs],I,F):-
    I1 is I + 1,
    write(I1),
    write('|'),
    write_spaces(I1),
    display_line(B),
    write('|'),
    nl,
    write('   '),
    I2 is I1 + 1,
    write_spaces(I2),
    top_borders(I2),nl,
    printBoard(Bs,I1,F).
```

```
display_line([]).
display_line([L|Ls]):-
    display_position(L),
    display_line(Ls).

display_position([P|Ps]):-
    write('|'),
    translate(P,V),
    write(V).

write_spaces(1):- write('      ').
write_spaces(2):- write('    ').
write_spaces(3):- write('  ').
write_spaces(4):- write('    ').
write_spaces(5):- write('      ').
write_spaces(6):- write('       ').

top_borders(0).
top_borders(1):- write(' _ _ _ _ _ _').
top_borders(2):- write(' _ _ _ _ _ _ _ _').
top_borders(3):- write(' _ _ _ _ _ _ _ _ _ _').
top_borders(4):- write(' _ _ _ _ _ _ _ _').
top_borders(5):- write(' _ _ _ _ _ _').
top_borders(6):- write('- - - - - -').

translate(r,' O ').
translate(cp1,' a ').
translate(cm1,' A ').
translate(cg1,' * ').
translate(cp2,' b ').
translate(cm2,' B ').
translate(cg2,' + ').
translate(.,' . ').
```

O predicado implementado tem como resultado o seguinte output:

```
1|      |_._|_._|_._|
2|    |_._|_._|_._|_._|
3|  |_._|_._|_._|_._|_._|
4|    |_._|_._|_._|_._|
5|      |_._|_._|_._|
       _ _ _ _ _ _
```

FIGURA 2 TABULEIRO VAZIO

```
1|      |_a_|_*_|_A_|
2|    |_b_|_*_|_+_|_B_|
3|  |_+_|_A_|_a_|_._|_b_|
4|    |_B_|_*_|_+_|_B_|
5|      |_a_|_b_|_A_|
       _ _ _ _ _ _
```

FIGURA 3 TABULEIRO PREENCHIDO

3.3. Lista de Jogadas Válidas

No inicio de cada jogada, é pedido ao jogador a peça que ele deseja mover. É aí que o utilizador insere o número da linha e da coluna correspondente à peça que deseja mover. Caso as coordenadas sejam inválidas ou a peça não seja do jogador em questão, é apresentada uma mensagem de erro. Por outro lado, caso a peça seja válida assim como as coordenadas, é apresentado ao jogador uma lista de todas as jogadas possíveis que ele pode efetuar com aquela peça.

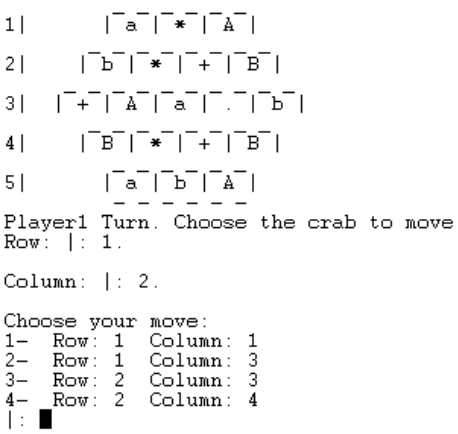


FIGURA 4 EXEMPLO DE LISTA DE JOGADAS POSSÍVEIS

3.4. Execução de Jogadas

A execução da jogada para cada jogador é efetuada da mesma maneira:

- O jogador insere um par de coordenadas
- O jogo valida as coordenadas, repetindo o 1º passo caso sejam inválidas
- Verifica se o caranguejo na posição inserida é do jogador, caso não seja volta ao inicio da jogada
- Apresenta a lista de jogadas possíveis daquela peça
- O jogador escolhe a jogada
- O jogo atualiza o tabuleiro

No final de cada jogada repete-se todos os passos para o outro jogador, caso esse mesmo jogador não tenha perdido.

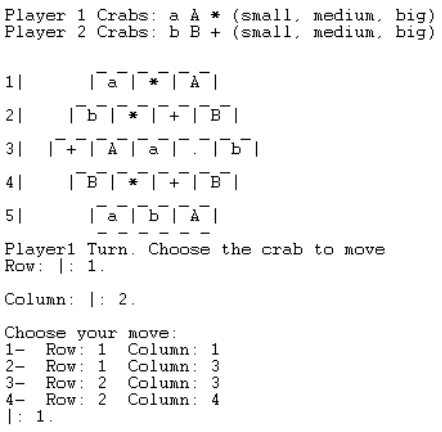


FIGURA 5 EXEMPLO DE SEQUÊNCIA DE JOGADA

3.5. Avaliação do Tabuleiro

Conforme foi mencionado acima, os caranguejos têm regras especificas para o seu movimento. Como tal, tivemos a necessidade em implementar alguns predicados para nos auxiliar a validar cada movimentação de cada caranguejo, nomeadamente verificar se na nova posição o caranguejo que se apresenta no topo da Stack permite que o caranguejo do jogador fique no seu topo ou se existe algum caranguejo. Foram por isso implementados os predicados:

3.6. Jogada do Computador

Apesar de ter sido implementado um modo de Jogador VS Computador e Computador VS Computador, não foram implementados níveis de dificuldade. No entanto, o comportamento do Computador em termos das jogadas é igual em ambos os modos de jogo.

Primeiramente é escolhida aleatoriamente uma posição do campo. De seguida verifica-se se esse caranguejo pertence ou não ao Computador: se não pertencer volta a repetir-se este passo. Caso pertença, escolhe-se também aleatoriamente uma jogada da lista de jogadas possíveis da peça escolhida. Caso a peça não tenha jogadas possíveis, é escolhida novamente outra peça aleatória.

4. Interface com o Utilizador

O módulo de interface com o utilizador em modo de texto é bastante simples:

```
=====================================================
= #### ####    #  ###    #### #####  #  #### #  # =
= #    #  #   # # #  #   #      #   # # #    #  # =
= #    ####  #### ####   ####   #  #### #    ###  =
= #    #  #  #  # # #      #   #  #  # #    #  # =
= #### #   # #  # ###    ####   #  #  # #### #  # =
=     SARA SANTOS NUNO CASTRO PLOG 2016/2017      =
=====================================================

                1. Play Game
                2. Intructions
                3. Exit
   _
```

No inicio do jogo, o utilizador é apresentado com o menu principal, tendo apenas como opções jogar, ler as instruções do jogo e sair do jogo.

Caso o utilizador escolha jogar (opção 1):

```
=====================================================
= #### ####    #  ###    #### #####  #  #### #  # =
= #    #  #   # # #  #   #      #   # # #    #  # =
= #    ####  #### ####   ####   #  #### #    ###  =
= #    #  #  #  # # #      #   #  #  # #    #  # =
= #### #   # #  # ###    ####   #  #  # #### #  # =
=     SARA SANTOS NUNO CASTRO PLOG 2016/2017      =
=====================================================

                1. Player vs Player
                2. Player vs Computer
                3. Computer vs Computer
```

É neste menu que o jogador escolhe o tipo de jogo que quer: se quer jogar contra outra pessoa, jogar contra o computador ou então computador contra computador. Em qualquer uma das escolhas o utilizador será reencaminhado para o jogo, excetuando a terceira opção em que o jogo começa a correr sem qualquer interferência:

```
*********************************************************************************

Player 1 Crabs: a A * (small, medium, big)
Player 2 Crabs: b B + (small, medium, big)

    _ _ _ _ _ _ _
1|  |_a_|_*_|_A_|
    _ _ _ _ _ _ _
2|  |_b_|_*_|_+_|_B_|
    _ _ _ _ _ _ _
3|  |_+_|_A_|_a_|_._|_b_|
    _ _ _ _ _ _ _
4|  |_B_|_*_|_+_|_B_|
    _ _ _ _ _ _ _
5|  |_a_|_b_|_A_|
       _ _ _ _ _ _
Player1 Turn. Choose the crab to move
Row: |: 
```

Conforme mencionado em cima, o jogo termina quando um dos jogadores não consegue efetuar nenhum movimento com nenhum dos seus caranguejos. Isto é verificado a partir de um predicado que percorre todo o tabuleiro verificando se pelo menos uma peça de cada jogador se consegue mover, caso contrário o jogo termina.

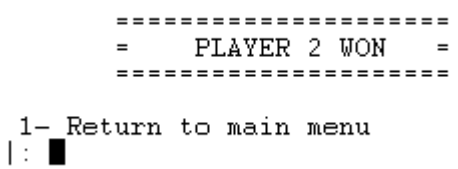
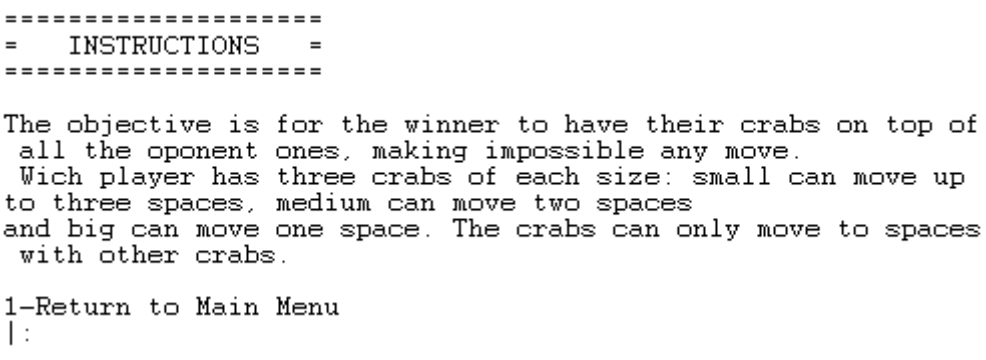


FIGURA 6 EXEMPLO DE FINAL DE JOGO

Caso o utilizador escolha ler as instruções (opção 2):



Aqui é apresentado uma breve explicação das regras de jogo.

Caso o utilizador escolha sair do jogo (opção 3), o jogo é terminado e o utilizador volta à consola.

5. Conclusões

Realizado no âmbito da Unidade Curricular de Programação em Lógica, este trabalho permitiu-nos consolidar todos os conhecimentos obtidos até agora ao nível da linguagem de PROLOG. No entanto, achamos que teria sido mais simples se existisse mais documentação da própria linguagem a nível de predicados que pudéssemos utilizar.

No entanto surpreende-nos o facto da simplicidade das regras do jogo físico terem sido um pouco mais complicadas de implementar nesta linguagem, apesar de as termos implementado na sua maioria.

Achamos também que teríamos conseguido consolidar ainda mais este projeto caso tivéssemos mais tempo, pois houveram algumas regras do jogo que tivemos que descartar por não termos tempo para terminar a sua implementação.

A. Anexos

BOARD.PL

```
:-include('utilities.pl').

% ************************** %
% REPRESENTATION OF THE BOARD %
% ************************** %

empty_board(Board):-
        Board=  [[[.],[.],[.]],
                        [[.],[.],[.],[.]],
                        [[.],[.],[.],[.],[.]],
                        [[.],[.],[.],[.]],
                        [[.],[.],[.]]].

initial_board(Board):-
        Board = [[[cp1,.],[cg1,.],[cm1,.]],
                        [[cp2,.],[cg1,.],[cg2,.],[cm2,.]],
                        [[cg2,.],[cm1,.],[cp1,.],[.],[cp2,.]],
                        [[cm2,.],[cg1,.],[cg2,.],[cm2,.]],
                        [[cp1,.],[cp2,.],[cm1,.]]].


% ************************************** %
%      UPDATES THE BOARD IN EACH PLAY       %
%  Moves a piece from one position to another %
% ************************************** %

updateBoard(BoardO,BoardU,OldR,OldC,NewR, NewC,[OldP|OldPs]):-
        Row1 is OldR - 1,
        Col1 is OldC -1,
        append(OldPs,[],NEWPOS),
        replace(BoardO,Row1,Col1,NEWPOS,Temp1),
        nth1(NewR, Temp1, NewRowList ),
        nth1(NewC, NewRowList, NP),
        append([OldP],NP, NewPosition),
        Row2 is NewR - 1,
        Col2 is NewC - 1,
        replace(Temp1, Row2, Col2, NewPosition, BoardU).

replace( L , X , Y , Z, R) :-
 append(RowPfx,[Row|RowSfx],L),
```

```
  length(RowPfx,X) ,
  append(ColPfx,[_|ColSfx],Row) ,
  length(ColPfx,Y) ,
  append(ColPfx,[Z|ColSfx],RowNew) ,
  append(RowPfx,[RowNew|RowSfx],R).


% ************** %
%   PRINT BOARD  %
% ************** %

print_gameboard(Board):-
        write('  '),
        write_spaces(1),
        top_borders(1),nl,
        printBoard(Board,0,5).

printBoard(_,F,F).
printBoard([B|Bs],I,F):-
        I1 is I + 1,
        write(I1),
        write('|'),
        write_spaces(I1),
        display_line(B),
        write('|'),
        nl,
        write('  '),
        I2 is I1 + 1,
        write_spaces(I2),
        top_borders(I2),nl,
        printBoard(Bs,I1,F).


display_line([]).
display_line([L|Ls]):-
        display_position(L),
        display_line(Ls).

display_position([P|Ps]):-
        write('|'),
        translate(P,V),
        write(V).

write_spaces(1):- write('      ').
write_spaces(2):- write('    ').
```

```
write_spaces(3):- write('  ').
write_spaces(4):- write('    ').
write_spaces(5):- write('      ').
write_spaces(6):- write('       ').

top_borders(0).
top_borders(1):- write(' _ _ _ _ _ _').
top_borders(2):- write(' _ _ _ _ _ _ _ _').
top_borders(3):- write(' _ _ _ _ _ _ _ _ _ _').
top_borders(4):- write(' _ _ _ _ _ _ _ _').
top_borders(5):- write(' _ _ _ _ _ _').
top_borders(6):- write('- - - - - -').

translate(r,' O ').
translate(cp1,' a ').
translate(cm1,' A ').
translate(cg1,' * ').
translate(cp2,' b ').
translate(cm2,' B ').
translate(cg2,' + ').
translate(., ' . ').
```

CRABSTACK.PL

```
:- include('menu.pl').
:- include('utilities.pl').
:- include('board.pl').
:- use_module(library(lists)).
:- use_module(library(random)).

crabstack:- main_menu.

% ***************************** %
%    PLAY GAME FUNCTIONS      %
%    For each game mode       %
% ***************************** %

play_game(B):-   %Player Vs Player
   FirstTurn = 1,
   Computer = 0,
          clear_screen,
          playerTurn(B, FirstTurn,Computer).

play_game_computer(B):- %Player Vs Computer
```

```
        FirstTurn = 1,
        Computer = 1,
        clear_screen,
        playerTurn(B,FirstTurn,Computer).

play_computer_computer(B):-  %Computer Vs Computer
        FirstTurn = 1,
        Computer = 2,
        clear_screen,
        playerTurn(B,FirstTurn, Computer).



% ******************************** %
%                 GAME TURN FUNCTIONS      %
% ******************************** %

playerTurn(PlayerBoard,Turn, Mode):-  %Calls the right player to play a turn
        print_separador,
        (Turn == 1 -> player1_turn(PlayerBoard,Mode);
        Turn == 2 -> player2_turn(PlayerBoard,Mode);
        write('Wrong player Turn')).

player1_turn(B1,PlayerMode1):-      %Player 1 Turn
        moves_available_player1(B1,B1,0) ->
                %If there is any moves available, player picks a crab to move
                                (display_players_info,nl,nl,
                                print_gameboard(B1),
                                write('Player1 Turn. Choose the crab to move'),nl,
                                (PlayerMode1 == 2 -> (random(1,5,RowR),
random(1,5,ColR));
                                                                (write('Row: '),
read(RowO), nl,write('Column: '),read(ColO),nl)),
                                (PlayerMode1 == 2 -> (Row1 = RowR, Col1 = ColR ) ;
(Row1 = RowO, Col1 = ColO )),
                                (isInsideBoard(Row1,Col1)-> true  ; (write('coord
inv'),nl,NewBoard1 = B1, player1_turn(NewBoard1,PlayerMode1))),
                                nth1(Row1,B1,BoardRow1),
                                nth1(Col1,BoardRow1,Pos1),
                                (isplayer1Crab(Pos1) -> true; (false, write('Not your
crab!'),nl, NewBoard1 = B1, player1_turn(NewBoard1,PlayerMode1))),
                                write('Choose your move: '),nl,
                                build_moves(B1,RL1,CL1,Row1,Col1,Pos1),
                                list_empty(CL1,ColEmpty), list_empty(RL1, RowEmpty),
```

```
                                ( (ColEmpty;RowEmpty)  -> (write('No available moves'),
nl , NewBoard = B1, player1_turn(NewBoard,PlayerMode1));
                                 (PlayerMode1 == 2 ->
(select_random_play(RL1,CL1,Row1,Col1,B1,Pos1,0,PlayerMode1));

(display_available_moves(RL1,CL1, 0), read(O1), length(RL1,L), ((O1 > L; O1<1) ->
player1_turn(B1,PlayerMode1);
true),select_mov(O1,RL1,CL1,Row1,Col1,B1,Pos1,1,PlayerMode1))))));
        %Otherwise, player 2 is the winner
                                player2_won_menu.

player2_turn(B2,PlayerMode2):-     %Player 2  Turn
        moves_available_player1(B2,B2,0) ->
                %If there is any moves available, player picks a crab to move
                                (display_players_info,nl,nl,
                                print_gameboard(B2),
                                write('Player2 Turn. Choose the crab to move'),nl,
                                (PlayerMode2 == 0 -> (write('Row: '), read(RowO),
nl,write('Column: '),read(ColO),nl);

(random(1,5,RowR), random(1,5,ColR))),
                                (PlayerMode2 == 0 -> (Row2 = RowO, Col2 = ColO );
(Row2 = RowR, Col2 = ColR )),
                                write(Row2),nl,write(Col2),nl,
                                (isInsideBoard(Row2,Col2)-> true ; (false,write('coord
inv'),nl,NewBoard = B2, player2_turn(NewBoard,PlayerMode2))),
                                nth1(Row2,B2,BoardRow2),
                                nth1(Col2,BoardRow2,Pos2),
                                (isplayer2Crab(Pos2) -> true; (false,write('Not your
crab!'),nl,NewBoard = B2, player2_turn(NewBoard,PlayerMode2))),
                                write('Choose your move: '),nl,
                                build_moves(B2,RL2,CL2,Row2,Col2,Pos2),
                                list_empty(CL2,ColEmpty), list_empty(RL2,RowEmpty),
                                ( (ColEmpty;RowEmpty)  -> (write('No available moves'),
nl , NewBoard = B2, player2_turn(NewBoard,PlayerMode2));
                                 (PlayerMode2 == 0 -> (display_available_moves(RL2,CL2,
0), read(O2), length(RL2,L), ((O2 > L; O2<1) -> player2_turn(B2, PlayerMode2); true),
select_mov(O2,RL2,CL2,Row2,Col2,B2,Pos2,2,PlayerMode2));

(select_random_play(RL2,CL2,Row2,Col2,B2,Pos2,0,PlayerMode2))))));
                %Otherwise, player 2 is the winner
                                player1_won_menu.

%AUX FUNCTION - Selects a random crab move
select_random_play(RL,CL,Rw,Cl,Bd,Ps,Tn,PMd):-
```

```
        length(RL,Len),
        random(1, Len, Choice),
        write(Choice),
        select_mov(Choice,RL,CL,Rw,Cl,Bd,Ps,Tn,PMd).



% ********************************** %
%    FUNCTION TO VALIDATE MOVEMENT    %
% ********************************** %

validSmallMove(BoardSmall, RowCoordSmall, ColCoordSmall):-
        nth1(RowCoordSmall, BoardSmall, NewRowB),
        nth1(ColCoordSmall, NewRowB, NewPos),
        (newPosSmall(NewPos) -> true; false).
newPosSmall([NS|NSs]):-
        (NS == .) -> false;
        (NS == cp1) -> true;
        (NS == cp2) -> true; false.

validMediumMove(BoardMedium, RowCoordMedium, ColCoordMedium):-
        nth1(RowCoordMedium, BoardMedium, NewRowB),
        nth1(ColCoordMedium, NewRowB, NewPos),
        (newPosMedium(NewPos) -> true; false).
newPosMedium([NS|NSs]):-
        (NS == .) -> false;
        (NS == cp1 ; NS == cp2) -> true;
        (NS == cm1 ; NS == cm2) -> true; false.

validBigMove(BoardBig, RowCoordBig, ColCoordBig):-
        nth1(RowCoordBig, BoardBig, NewRowB),
        nth1(ColCoordBig, NewRowB, NewPos),
        (newPosBig(NewPos) -> true; false).
newPosBig([NS|NSs]):-
        (NS == .) -> false;
        true.

isInsideBoard(Row,Col):-   % Checks if coordenates are inside the game board
        (Row > 5) -> false;
        (Col > 5) -> false;
        (Row < 1) -> false;
        (Col < 1) -> false;
        ((Row == 1; Row == 5), Col > 3) ->  false;
        ((Row == 2; Row == 4), Col > 4) -> false;
        ((Row == 3), Col > 5) -> false; true.
```

```
isplayer1Crab([P1|P1s]):- % Checks if a crab belongs to player 1
        P1 == cp1 -> true;
        P1 == cm1 -> true;
        P1 == cg1 -> true; false.

isplayer2Crab([P2|P2s]):- % Checks if crab belongs to player 2
        P2 == cp2 -> true;
        P2 == cm2 -> true;
        P2 == cg2 -> true; false.



% ************************ %
%  BUILD AVAILABLE MOVES    %
%   Given a player crab     %
% ************************ %

build_moves(BoardGame,R,C,PRow, PCol,[Peca|PecaT]):-
        (Peca == cp1 ; Peca == cp2) ->(build_small_moves(BoardGame,R,C,PRow, PCol));
        (Peca == cm1 ; Peca == cm2) -
>(build_medium_moves(BoardGame,R,C,PRow,PCol));
        (Peca == cg1 ; Peca == cg2) ->(build_big_moves(BoardGame,R,C,PRow,PCol));
        (append([],[],R), append([],[],C)).


build_small_moves(BS,SmallRowList,SmallColList,RowS, ColS):-
        X1 is ColS - 3,
        X2 is ColS + 3,
        Y1 is RowS +3,
        X3 is (ColS - (Y1 - 3)),
        X4 is (ColS + (3 - RowS)),
        Y2 is RowS - 3,
        X5 is (ColS - (3 - Y2)),
        X6 is (ColS + (RowS - 3)),
        ((isInsideBoard(RowS,X1), validSmallMove(BS, RowS, X1) )->
(append(EmptyList1,[X1],Temp1), append(EmptyList2,[RowS],Temp2)) ;
(append(EmptyList1,[],Temp1), append(EmptyList2,[],Temp2))),
        ((isInsideBoard(RowS,X2), validSmallMove(BS, RowS, X2) )->
(append(Temp1,[X2],Temp3), append(Temp2,[RowS],Temp4)) ; (append(Temp1,[],Temp3),
append(Temp2,[],Temp4))),
        ((isInsideBoard(Y1,X3), validSmallMove(BS, Y1, X3) )->
(append(Temp3,[X3],Temp5), append(Temp4,[Y1],Temp6)); (append(Temp3,[],Temp5),
append(Temp4,[],Temp6))),
```

```
        ((isInsideBoard(Y1,X4), validSmallMove(BS, Y1, X4) )->
(append(Temp5,[X4],Temp7), append(Temp6,[Y1],Temp8)) ; (append(Temp5,[],Temp7),
append(Temp6,[],Temp8))),
        ((isInsideBoard(Y2,X5), validSmallMove(BS, Y2, X5) )->
(append(Temp7,[X5],Temp9), append(Temp8,[Y2],Temp10)); (append(Temp7,[],Temp9),
append(Temp8,[],Temp10))),
        ((isInsideBoard(Y2,X6), validSmallMove(BS, Y2, X6) )->
(append(Temp9,[X6],Temp11), append(Temp10,[Y2],Temp12)) ;
(append(Temp9,[],Temp11), append(Temp10,[],Temp12))),
        SmallColList = Temp11, SmallRowList = Temp12.

build_medium_moves(BM,MediumRowList, MediumColList, RowM, ColM):-
        X1 is ColM - 2,
        X2 is ColM + 2,
        Y1 is RowM + 2,
        X3 is (ColM - (Y1 - 3)),
        X4 is (ColM + (3 - RowM)),
        Y2 is RowM - 2,
        X5 is (ColM - (3 - Y2)),
        X6 is (ColM + (RowM - 3)),
        ((isInsideBoard(RowM,X1), validMediumMove(BM, RowM, X1) )->
(append(EmptyList1,[X1],Temp1), append(EmptyList2,[RowM],Temp2)) ;
(append(EmptyList1,[],Temp1), append(EmptyList2,[],Temp2))),
        ((isInsideBoard(RowM,X2),validMediumMove(BM, RowM, X2)) ->
(append(Temp1,[X2],Temp3), append(Temp2,[RowM],Temp4)) ;
(append(Temp1,[],Temp3), append(Temp2,[],Temp4))),
        ((isInsideBoard(Y1,X3), validMediumMove(BM, Y1, X3) )->
(append(Temp3,[X3],Temp5), append(Temp4,[Y1],Temp6)); (append(Temp3,[],Temp5),
append(Temp4,[],Temp6))),
        ((isInsideBoard(Y1,X4), validMediumMove(BM, Y1, X4) )->
(append(Temp5,[X4],Temp7), append(Temp6,[Y1],Temp8)) ; (append(Temp5,[],Temp7),
append(Temp6,[],Temp8))),
        ((isInsideBoard(Y2,X5), validMediumMove(BM, Y2, X5) )->
(append(Temp7,[X5],Temp9), append(Temp8,[Y2],Temp10)); (append(Temp7,[],Temp9),
append(Temp8,[],Temp10))),
        ((isInsideBoard(Y2,X6), validMediumMove(BM, Y2, X6) )->
(append(Temp9,[X6],Temp11), append(Temp10,[Y2],Temp12)) ;
(append(Temp9,[],Temp11), append(Temp10,[],Temp12))),
        MediumColList = Temp11, MediumRowList = Temp12.

build_big_moves(BB,BigRowList, BigColList, RowB, ColB):-
        X1 is ColB - 1,
        X2 is ColB + 1,
        Y1 is RowB + 1,
        X3 is (ColB - (Y1 - 3)),
```

```
        X4 is (ColB + (3 - RowB)),
        Y2 is RowB - 1,
        X5 is (ColB - (3 - Y2)),
        X6 is (ColB + (RowB - 3)),
        ((isInsideBoard(RowB,X1), validBigMove(BB, RowB, X1) )->
(append(EmptyList1,[X1],Temp1), append(EmptyList2,[RowB],Temp2)) ;
(append(EmptyList1,[],Temp1), append(EmptyList2,[],Temp2))),
        ((isInsideBoard(RowB,X2), validBigMove(BB, RowB, X2) )->
(append(Temp1,[X2],Temp3), append(Temp2,[RowB],Temp4)) ; (append(Temp1,[],Temp3),
append(Temp2,[],Temp4))),
        ((isInsideBoard(Y1,X3), validBigMove(BB, Y1, X3) )-> (append(Temp3,[X3],Temp5),
append(Temp4,[Y1],Temp6)); (append(Temp3,[],Temp5), append(Temp4,[],Temp6))),
        ((isInsideBoard(Y1,X4), validBigMove(BB, Y1, X4) )-> (append(Temp5,[X4],Temp7),
append(Temp6,[Y1],Temp8)) ; (append(Temp5,[],Temp7), append(Temp6,[],Temp8))),
        ((isInsideBoard(Y2,X5), validBigMove(BB, Y2, X5) )-> (append(Temp7,[X5],Temp9),
append(Temp8,[Y2],Temp10)); (append(Temp7,[],Temp9), append(Temp8,[],Temp10))),
        ((isInsideBoard(Y2,X6), validBigMove(BB, Y2, X6) )-> (append(Temp9,[X6],Temp11),
append(Temp10,[Y2],Temp12)) ; (append(Temp9,[],Temp11),
append(Temp10,[],Temp12))),
        BigColList = Temp11, BigRowList = Temp12.



% ********************************** %
%    SELECTS THE PLAYER MOVEMENT     %
% Updates the board and calls next turn %
% ********************************** %

select_mov(MovOption, MovRow, MovCol,OldRow,OldCol,GameB,OldPos, CurrentTurn,
GameMode):-
        nth1(MovOption, MovRow, NewRow),
        nth1(MovOption, MovCol, NewCol),
        (CurrentTurn == 1 -> NextTurn = 2; NextTurn = 1),
        updateBoard(GameB, NewGameB, OldRow, OldCol, NewRow, NewCol, OldPos),
        playerTurn(NewGameB,NextTurn,GameMode).


% ************************************** %
%          EVALUATES THE GAME STATE/BOARD       %
% Checks if a player has any available moves %
% ************************************** %

moves_available_player1([],Board,RowI):- false.
moves_available_player1([B|Bs],Board,RowI):-
         (moves_available_row_player1(B,Board,RowI,0) -> true;
```

```
                (NewI is RowI +1, moves_available_player1(Bs,Board, NewI))).

moves_available_row_player1([],Board,IndexR,IndexC):- false.
moves_available_row_player1([R|Rs],Board,IndexR, IndexC):-
        (is_available_player1_crab(Board,IndexR, IndexC) -> true;

                        (NewCI is IndexC +1, moves_available_row_player1(Rs,Board,
IndexR, NewCI))).

is_available_player1_crab(Board, IR , IC):-
        nth1(IR,Board, TempRowList),
        nth1(IC,TempRowList, Crab),
        build_moves(Board,RLis,CLis,IR, IC,Crab),
        list_empty(RLis, T),
        ( T -> !, false ;(isplayer1Crab(Crab) -> true; !,false)).

moves_available_player2([],Board,RowI):- false.
moves_available_player2([B|Bs],Board,RowI):-
         (moves_available_row_player2(B,Board,RowI,0) -> true;

                (NewI is RowI +1, moves_available_player2(Bs,Board, NewI))).

moves_available_row_player2([],Board,IndexR,IndexC):- false.
moves_available_row_player2([R|Rs],Board,IndexR, IndexC):-
        (is_available_player2_crab(Board,IndexR, IndexC) -> true;

                        (NewCI is IndexC +1, moves_available_row_player2(Rs,Board,
IndexR, NewCI))).

is_available_player2_crab(Board, IR , IC):-
        nth1(IR,Board, TempRowList),
        nth1(IC,TempRowList, Crab),
        build_moves(Board,RLis,CLis,IR, IC,Crab),
        list_empty(RLis, T),
        ( T -> !, false ;(isplayer2Crab(Crab) -> true; !,false)).
```

MENU.PL

```
:- include('board.pl').
:- include('utilities.pl').

% ****************** %
% MAIN MENU FUNCTIONS %
```

```
% ******************* %
display_title:-    write('================================================='), nl,
                                write('= #### ####   #  ###   #### #####  #  #### #  # ='),
nl,
                                write('= #   # #  # # # #  #    #  # # #  #  # ='), nl,
                                write('= #   #### #### ####  ####  # #### #   ### ='),
nl,
                                write('= #   # # # # # #    #  # # # # #   # # ='),nl,
                                write('= #### #  # # # ###   ####  # #  # #### #  # ='),nl,
                                write('=    SARA SANTOS NUNO CASTRO PLOG 2016/2017
='),nl,

          write('=================================================').

display_options:- write('          1. Play Game              '),nl,
                                  write('          2. Intructions            '),nl,
                                  write('          3. Exit                '),nl.


main_menu:- clear_screen,
                          display_title,
                          nl,
                          nl,
                          display_options,
                          read_main_option.

read_main_option:- read(Option), select_main_option(Option).
select_main_option(1):- clear_screen, play_menu.
select_main_option(2):- clear_screen, display_instructions.
select_main_option(3).



% ******************* %
% PLAY MENU FUNCTIONS %
% ******************* %
play_menu:- clear_screen,
                          display_title,nl,nl,
                          write('          1. Player vs Player            '),nl,
                          write('          2. Player vs Computer          '),nl,
                          write('          3. Computer vs Computer        '),nl,
                          read_play_option.
```

```
read_play_option:- read(PlayOption), select_play_option(PlayOption).
select_play_option(1):- clear_screen,initial_board(Board), play_game(Board).
select_play_option(2):- clear_screen,initial_board(Board), play_game_computer(Board).
select_play_option(3):- clear_screen, initial_board(Board),
play_computer_computer(Board).
select_play_option(N):- clear_screen, play_menu.



% ************************** %
% INSTRUCTIONS MENU FUNCTIONS %
% ************************** %
display_instructions:-
        write('====================='),nl,
        write('=   INSTRUCTIONS   ='),nl,
        write('====================='), nl,nl,
        write('The objective is for the winner to have their crabs on top of all the oponent
ones, making impossible any move.'),nl,
        write(' Wich player has three crabs of each size: small can move up to three
spaces, medium can move two spaces'), nl,
        write('and big can move one space. The crabs can only move to spaces with other
crabs.'),nl,nl,
        write('1-Return to Main Menu'),nl,
        read(Op), select_inst_option(Op).
select_inst_option(1):-clear_screen, main_menu.
select_inst_option(N):- display_instructions.

% ********************* %
% WINNER MENU FUNCTIONS %
% ********************* %

player1_won_menu:-
        clear_screen,
        write('      ====================='),nl,
        write('      =    PLAYER 1 WON   ='),nl,
        write('      ====================='),nl,nl,
        write(' 1- Return to main menu'),nl,
        read(OP), main_menu.

player2_won_menu:-
        clear_screen,
        write('      ====================='),nl,
        write('      =    PLAYER 2 WON   ='),nl,
        write('      ====================='),nl,nl,
        write(' 1- Return to main menu'),nl,
```

```
        read(OP), main_menu.
```

UTILITIES.PL

```
% Clears the screen
clear_screen :- new_line(50), !.
new_line(Nlines) :-
        new_line(0, Nlines).
new_line(Line, MaxLines) :-
        Line < MaxLines,
        NextLine is Line + 1,
        nl,
        new_line(NextLine, MaxLines).
new_line(_,_).

% Displays Player Crabs
display_players_info:-
                write('Player 1 Crabs: a A * (small, medium, big)'),nl,
                write('Player 2 Crabs: b B + (small, medium, big)').

% Prints the separation between turns
print_separador:-
        write('************************************************************************
************'),nl,nl.

% Checks if a given list is empty
list_empty([], true).
list_empty([_|_], false).

% Displays the the available moves lists
display_available_moves([],[],N).
display_available_moves([AR|ARs],[AC|ACs],N):-
        N1 is N + 1,
        write(N1),
        write('-  Row: '), write(AR),
        write('  Column: '), write(AC),nl,
        display_available_moves(ARs,ACs,N1).
```