



## Relatório



## Mestrado Integrado em Engenharia Informática e Computação Programação em Logica

### Grupo 2:

Inês Alexandra dos Santos Carneiro - up201303501  
Filipa Marília Monteiro Ramos - up201305378

Faculdade de Engenharia da Universidade do Porto  
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

## Resumo

Na unidade curricular de Programação em Lógica foi apresentado o problema de realizar uma simulação do jogo Q!nto em PROLOG. Inicialmente, a atenção foi focada no tabuleiro, sendo construídos os predicados de criação da lista de listas (tabuleiro) e representação do mesmo. No presente relatório é apresentado todo o trabalho realizado após esses predicados. Estando dividido em variadas secções, é dada a maior importância à lógica desenvolvida na aplicação do jogo. São apresentados tanto os estados de jogo como os predicados implementados para a execução e validação de jogadas. Para além disto, é referenciada a interface com o utilizador incluindo imagens que demonstram a representação dos objetos e da comunicação com o utilizador. Finalmente, apresentam-se as conclusões atingidas após a implementação.

Entre os variados objetivos deste projeto destaca-se o aumento da capacidade de raciocínio lógico característico de um programador. Na verdade, a linguagem PROLOG necessita da aplicação de lógica em cada momento. É também de referência a importância do projeto na aprofundação do conhecimento da linguagem apresentada.

## 1. Introdução

No âmbito da unidade curricular de Programação em Lógica, foi proposta a realização de uma simulação do jogo Q!nto em PROLOG. Pretende-se com este projeto aprofundar o conhecimento da linguagem referida.

O presente relatório está dividido em 3 grandes subsecções - a primeira refere-se ao jogo em si, explicando como o mesmo funciona. A segunda secção aprofunda a lógica da implementação escolhida, desde estados de jogo até à validação e execução de jogadas. A última secção apresenta a interface fornecida ao utilizador.

## 2. Q!nto

### 2.1 História

Q!nto é um jogo de estratégia abstrata que foi desenvolvido pelo designer Gene Mackles e publicado pela PDG games. O seu lançamento no mercado data do ano 2014. É adequado para todas as idades a partir dos 8 anos podendo ser jogado por um mínimo de 2 e um máximo de 4 jogadores. Existem 3 variações de Q!nto: Q!nto clássico; Q!nto Plus que permite a contagem de pontos numa diagonal de 3 ou mais cartas e Q!nto Light no qual as cartas são divididas igualmente pelos jogadores e o vencedor é o que esvazia a sua mão primeiro.

## 2.2 Regras

Cada jogador possui 30 cartas sendo que existem 5 formas e 5 cores possíveis. Existem cartas, em menor número, que permitem escolher ou a sua forma ou a sua cor ou ambas.

- Carta que permite escolher a cor: tem uma forma específica e o utilizador escolhe a cor que esta possui.
- Carta que permite escolher a forma: tem uma cor definida e permite a escolha da forma da mesma.
- Carta universal: forma e cor definível.

Uma jogada é válida se for feita uma coluna ou linha com cartas nas seguintes condições:

- cores iguais ou cores diferentes.
- símbolos iguais ou símbolos diferentes.

O jogo termina quando um dos jogadores não tem mais cartas para jogar. O que esvazia a sua mão primeiro vence.

## 3. Lógica

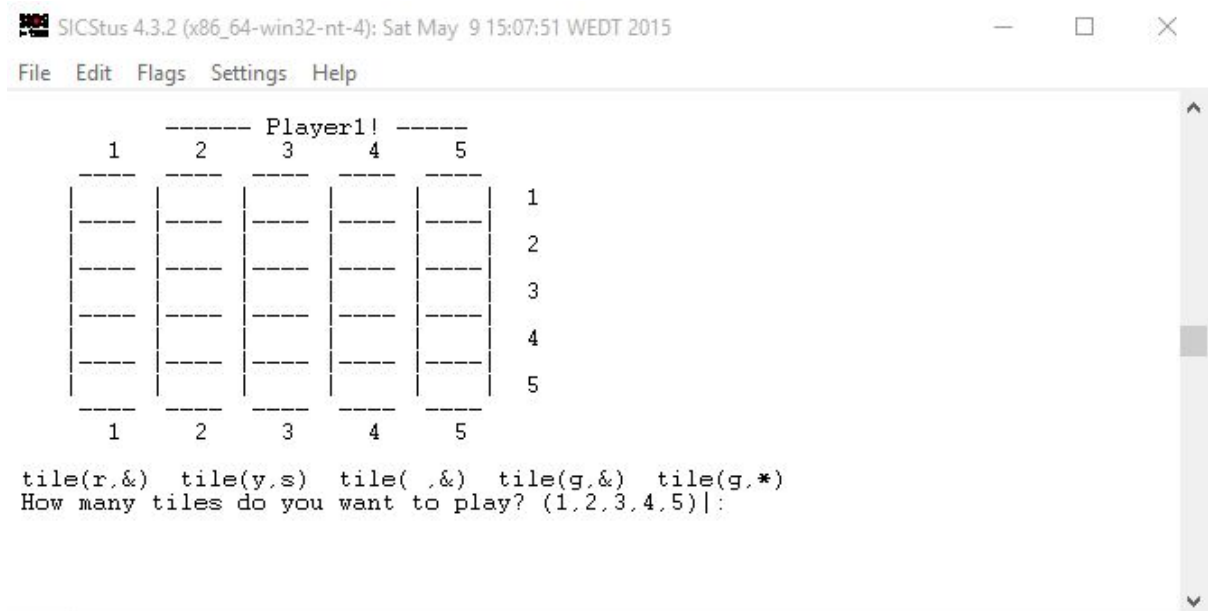
A lógica do jogo foi implementada com base numa variedade de predicados auxiliares que permitem a representação do tabuleiro, execução e validação de jogadas, terminação do jogo, etc...

### 3.1 Estado de jogo

Existem dois estados principais: o estado do jogador 1 e o do jogador 2. Cada estado guarda o jogador, isto é as cartas que cada um possui e o tabuleiro na altura da jogada. Assim, o estado é definido da seguinte maneira: :- dynamic state/2. O estado é alterado com a ajuda dos predicados assert e retract.

### 3.2 Visualização

O tabuleiro é representado na interface textual com a ajuda de barras e traços. Na consola visualiza-se o seguinte:



```
SICStus 4.3.2 (x86_64-win32-nt-4): Sat May 9 15:07:51 WEDT 2015
File Edit Flags Settings Help

      ----- Player1! -----
      1      2      3      4      5
      |      |      |      |      |
      |      |      |      |      |  1
      |      |      |      |      |  2
      |      |      |      |      |  3
      |      |      |      |      |  4
      |      |      |      |      |  5
      |      |      |      |      |
      1      2      3      4      5

tile(r,&) tile(y,s) tile(.,&) tile(g,&) tile(g,*)
How many tiles do you want to play? (1,2,3,4,5)|:
```

Em cima é visível o tabuleiro com linhas na vertical e na horizontal para melhor identificar as posições do mesmo. Por baixo é visível a mão do jogador que está a efetuar uma jogada. O tabuleiro é expandido 1 casa em cada direção a cada jogada. Desta forma, a visualização está sempre a ser alterada.

### 3.3 Jogadas Válidas

As jogadas válidas são testadas através do predicado `valid_ListMoves` que recebe o tabuleiro, uma lista de movimentos, a mão. Este predicado chama outros 2 que verificam as jogadas válidas tanto na vertical como na horizontal e ainda verifica se a peça a ser jogada pertence a lista de cartas do utilizador. Primeiro, começa-se por avaliar se a peça pertence as cartas do utilizador. Depois, é avaliada a peça a jogar com as peças próximas. Se for uma jogada válida, então este predicado retorna verdadeiro.

Este predicado faz uso de uma variedade de predicados auxiliares como o `verify_Hor` e o `verify_Vert` que por sua vez fazem uso de predicados como `verify_Down`, `verify_Up`, `verify_Right`, `verify_Left`, `verify_Down_aux`, `verify_Up_aux`, `verify_Right_aux`, `verify_Left_aux`.

### 3.3 Execução de Jogadas

A execução de jogadas é realizada com o auxílio do predicado `apply_moves` que recebe uma lista de jogadas a efetuar e as aplica no tabuleiro recebido. Este retorna um novo tabuleiro com as peças colocadas nos respetivos locais. Este chama o predicado

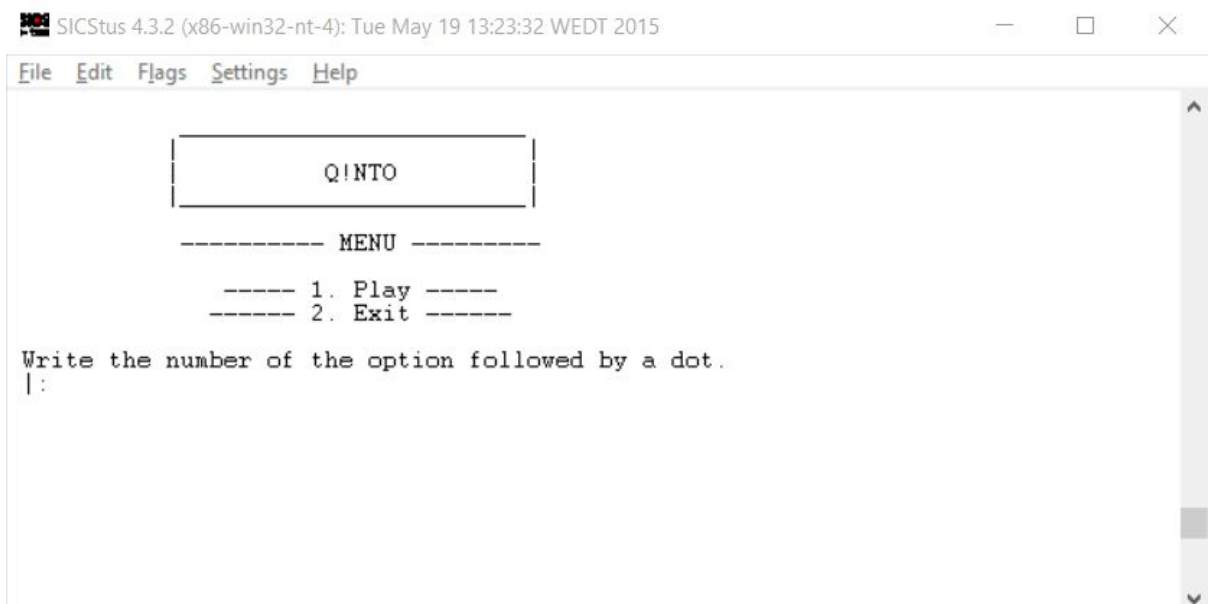
move que coloca na lista de listas (tabuleiro) na posição escolhida a peça. Este retorna o novo tabuleiro com a peça colocada.

### 3.5 Final do Jogo

O jogo termina quando um dos utilizadores tem a mão vazia. O predicado que verifica esta condição é o done. Se a lista está vazia, o jogador que tem esta mão é o vencedor do jogo.

## 4. Interface

Existem dois menus de interface textual para comunicar com o utilizador. O menu inicial tem o seguinte aspeto:



Tem duas opções: jogar e sair. A opção sair aborta a execução do programa. A opção "Play" leva a outro menu que permite a escolha do modo de jogo.

```
SICStus 4.3.2 (x86-win32-nt-4): Tue May 19 13:23:32 WEDT 2015
File Edit Flags Settings Help

      PLAY!

----- PLAY MENU -----
----- 1. Player vs Player -----
----- 2. Player vs Computer -----
----- 3. Computer vs Computer -----
----- 4. Go Back -----
----- 5. Exit -----

Write the number of the option followed by a dot.
|:
```

Este menu permite ainda voltar ao menu inicial ou sair do jogo. As outras opções levam à execução de jogo correspondente ao modo escolhido.

O tabuleiro e a mão são representados como o apresentado na secção 3.2. Para proceder à execução de uma jogada é perguntado ao utilizador quantas peças quer mover e qual a cor, forma e posição de cada peça a ser jogada.

```
How many tiles do you want to play? 3.
Choose the tile to play. Color: |: b.
Shape: |: '#'.
Choose where to place it. Number? |: 2.
Letter ?|: A.
```

## 5. Conclusões

Em suma, no início a programação em PROLOG revelou-se um desafio visto que não estávamos familiarizadas com o processo de raciocínio da linguagem. Porém, após o desenvolvimento do jogo descrito fizemos um grande processo de aprendizagem da linguagem, tornando-se cada vez mais fácil a compreensão da mesma. PROLOG revela-se interessante no ponto de vista de estimular o raciocínio lógico de quem o implementa.

O jogo escolhido pareceu-nos interessante e diferente, razões pelas quais o escolhemos, porém revelou-se um desafio a sua implementação. Devido ao seu elevado número de restrições nas jogadas válidas, houveram predicados de difícil implementação. Porém, no fim estas dificuldades foram ultrapassadas apesar de haver muito espaço para melhorar o projeto final, o que será feito para a próxima entrega.

## Anexos A

```
:-use_module(library(lists)).
:-use_module(library(between)).
:- use_module(library(random)).
/*PROLOG Q!NTO SIMULATION*/
```

```
/* represents the possible colors */
color(r).
color(b).
color(g).
color(y).
color(' ').
color(c).
```

```
/* represents the possible shapes */
shape('*').
shape('!').
shape('#').
shape('+').
shape('&').
shape(s).
```

```
/*return a tile*/
oneTile(X) :- color(C), shape(F), X = tile(C,F).
```

```
tile(' ',' ').
tile(C, F) :- color(C), shape(F).
```

```
%//////////////////////////////////// DECK
////////////////////////////////////
```

```
/*Creating Deck*/
```

```
wildQuintoTile(C, S) :- C = c, S = s.
```

```
wildColorTile(C, _S) :- C = c.
wildShapeTile(_C, S) :- S = s.
```

```
deckWithDuplicates(X) :- findall(T, oneTile(T), X1), findall(T, oneTile(T), X2), append(X1, X2, X).
```

```
rmvElem(NewDeck, [], NewDeck).
```

```
rmvElem(Deck, [LElem|T], NewDeck) :- delete(Deck, LElem, Deck1), rmvElem(Deck1, T,
NewDeck).
```

```
rmvWildTiles(Deck, NewDeck) :- CT1 = tile(c, '*'), CT2 = tile(c, '#'), CT3 = tile(c, '+'), CT4 =
tile(c, '&'),
                        ST1 = tile(r, s), ST2 = tile(g, s), ST3 = tile(b, s), ST4 = tile(y, s), ST5 =
tile(c, s),
                        ExclamationTile = tile(c, '!'), BlackTile = tile(' ', s),
                        L = [CT1, CT2, CT3, CT4, ST1, ST2, ST3, ST4, ST5, ExclamationTile,
BlackTile], rmvElem(Deck, L, NewDeck).
```

```
makeDeck(Deck) :- deckWithDuplicates(L1), rmvWildTiles(L1, L2),
                        CT1 = tile(c, '*'), CT2 = tile(c, '#'), CT3 = tile(c, '+'), CT4 = tile(c, '&'), CT5 = tile(c,
s),
                        ST1 = tile(r, s), ST2 = tile(g, s), ST3 = tile(b, s), ST4 = tile(y, s), ST5 = tile(c, s),
                        L = [CT1, CT2, CT3, CT4, CT5, ST1, ST2, ST3, ST4, ST5], append(L2, L, Deck).
```

```
%////////////////////////////////////// HAND
//////////////////////////////////////
```

```
/*Creating Hands - Mixing Deck and Dividing it in two hands*/
```

```
mixingElemntsDeck(Deck, NewDeck) :- random_permutation(Deck, NewDeck).
```

```
div(L, A, B) :-
    append(A, B, L),
    length(A, N),
    length(B, N).
```

```
creatingHand(Deck, Hand1, Hand2) :- div(Deck, Hand1, Hand2).
```

```
/* Display hand */
```

```
displayHand([_HHand|_T], 5).
displayHand([HHand|T], CC) :- CC < 5, CC1 is CC+1, write(HHand), write(' '),
displayHand(T, CC1).
```

```
displayTile(C,S) :- write(C), write(S), write(' | ').
```

```
%//////////////////////////////////BOARD -
DISPLAY//////////////////////////////////
```

```
writetile(tile(C,F)) :- write(' '), print(C), print(F), write(' ').
```



```

/* 1 2 3 4 .... */
fguideLine(N, CC) :- CC >= N, write(' ').
fguideLine(N, CC) :- CC < N, CC2 is CC+1, write(' '), write(CC2), write(' '), fguideLine(N,
CC2).

/* ---- ---- */
fHorizontalLine(0).
fHorizontalLine(N) :- N>0, write(' '), write('----'), N1 is N-1, fHorizontalLine(N1).

/* | | | | */
sHorizontalLine(0, []) :- write('|'), write(' ').
sHorizontalLine(N, [L|R]) :- N>0, write(' '), write('|'), writetile(L), N1 is N-1,
sHorizontalLine(N1, R).

/* |---|---|---| */
tHorizontalLine(0) :- write('|').
tHorizontalLine(N) :- N>0, write(' '), write('|'), write('----'), N1 is N-1, tHorizontalLine(N1).

displayBoardaux([], _C).
displayBoardaux([L1], C) :- write(' '), length(L1, N1), sHorizontalLine(N1, L1), write(' '),
write(C), nl.
displayBoardaux([L1|R], C) :- R \= [], length(L1, N1), C < N1, C2 is C+1, write(' '),
sHorizontalLine(N1, L1), write(' '), write(C), nl, write(' '), tHorizontalLine(N1), nl,
displayBoardaux(R, C2).

displayBoard([L1|R]) :- length(L1, N1), nl, write(' '), fguideLine(N1, 0), nl, write(' '),
fHorizontalLine(N1), nl,
                        displayBoardaux([L1|R], 1), write(' '), fHorizontalLine(N1), nl,
write(' '), fguideLine(N1, 0), nl.

%/////////////////////////////////////////BOARD -
CREATION/////////////////////////////////////////

/* Lista de Listas..Cria Board Matrix tudo com espaços vazios...*/
createBoard(W, H, Matrix) :- listElement(L, W, tile(' ', ' ')), listElement(Matrix, H, L).

randomCentre(T, Hand) :- length(Hand, N), random(0, N, Num), nth0(Num, Hand, T).
createCenter(B, W, H, Hand, Bnew, NHand) :- W1 is W/2, H1 is H/2, W2 is ceiling(W1), H2
is ceiling(H1), randomCentre(T, Hand),
                        move(B, H2, W2, T, Bnew), deleteElemHand(Hand, T, NHand).

/*expand matrix*/
matrix_width(Matrix, W) :- nth0(0, Matrix, Elem), length(Elem, W).
empty_tile(tile(' ', ' ')).

```

```

expand_matrix_up(Matrix, [L|Matrix]) :- empty_tile( E ), matrix_width(Matrix, W),
listElement(L, W, E).
expand_matrix_down(Matrix, NewMatrix) :- empty_tile( E ), matrix_width(Matrix, W),
listElement(L, W, E), append(Matrix, [L], NewMatrix).
expand_matrix_left([], []).
expand_matrix_left([L|Matrix], [NL|NewMatrix]) :- empty_tile( E ), append([E], L, NL),
expand_matrix_left(Matrix, NewMatrix).
expand_matrix_right([], []).
expand_matrix_right([L|Matrix], [NL|NewMatrix]) :- empty_tile( E ), append(L, [E], NL),
expand_matrix_right(Matrix, NewMatrix).
listElement([], 0, _X).
listElement([X|Xs], N, X) :- N1 is N - 1, listElement(Xs, N1, X).

```

```

/*Expand 5 tiles in each direction*/

```

```

expand_matrix_5left(5, _NM, _NM1).
expand_matrix_5left(CC, M, NM) :- CC < 5, CC > 0, expand_matrix_left(M, NM), CC1 is
CC+1, expand_matrix_5left(CC1, NM, _NM1).

```

```

/* Get Tiles from the board */

```

```

getTile(B, Px, Py, T) :- nth1(Px, B, L), nth1(Py, L, T).

```

```

%////////////////////////////////////MOVEMENT
FUNCTIONS////////////////////////////////////

```

```

deleteElemHand(Hand, Tile, NewHand) :- select(Tile, Hand, NewHand), !.

```

```

deleteElemHand(Hand, _Tile, Hand).

```

```

/*REPLACES AN ELEMNT */

```

```

replace([_|T], 0, X, [X|T]).

```

```

replace([H|T], I, X, [H|R]) :- I > -1, NI is I-1, replace(T, NI, X, R), !.

```

```

replace(L, _, _, L).

```

```

move(B, Px, Py, T, Hand, Bnew, NewHand) :- nth1(Px, B, L), I is Py-1, replace(L, I, T, B1),
P is Px-1, replace(B, P, B1, Bnew), deleteElemHand(Hand, T, NewHand).

```

```

move(B, Px, Py, T, Bnew) :- nth1(Px, B, L), I is Py-1, replace(L, I, T, B1),
P is Px-1, replace(B, P, B1, Bnew).

```

```

/*Applies a List of moves*/

```

```

apply_moves(NewBoard, [], NewHand, NewHand, NewBoard) :- !.

```

```

apply_moves(B, [[T,X,Y]|TM], Hand, NewHand, NewBoard) :- move(B, X, Y, T, Hand, NB,
NewHand1), !, apply_moves(NB, TM, NewHand1, NewHand, NewBoard).

```

```

apply_moves(NewBoard, [], NewBoard) :- !.

```

```

apply_moves(B, [[T,X,Y]|TM], NewBoard) :- move(B, X, Y, T, NB), !, apply_moves(NB, TM,
NewBoard).

```

%//////////////////////////////////// NEW VALID MOVE  
////////////////////////////////////

empty\_Tile(T) :- T1 = tile(' ', ' '), T == T1.  
not\_empty\_Tile(T, Deck) :- member(T, Deck).

verify\_Line(\_B, \_Limite, \_Line, \_Count).  
verify\_Line(B, Limite, Line, Count) :- Count < Limite, nth0(Count, B, Elem),  
empty\_Tile(Elem), Count1 is Count + 1, verify\_Line(B, Limite, Line, Count1).

all\_Empty\_Board(\_B, \_CC).  
all\_Empty\_Board(B, CC) :- length(B, Altura), CC < Altura, nth0(0, B, Elem), length(Elem,  
Largura), nth0(CC, B, Line),  
verify\_Line(B, Largura, Line, 0), CC1 is CC+1, all\_Empty\_Board(B, CC1).

verify\_Left\_aux(B, Px, Py, [S|LS], [C|LC]) :- Py > 0,  
getTile(B, Px, Py, T), \+ empty\_Tile(T), T = tile(C, S),  
Py1 is Py - 1, verify\_Left\_aux(B, Px, Py1, LS, LC).  
verify\_Left\_aux(B, Px, Py, LS, LC) :- (Py == 0; (getTile(B, Px, Py, T), empty\_Tile(T))), LS =  
[], LC = [], !.

verify\_Left(B, Px, Py, LS, LC) :- Py1 is Py - 1, verify\_Left\_aux(B, Px, Py1, LS, LC).

verify\_Right\_aux(B, Px, Py, [S|LS], [C|LC]) :- nth0(0, B, Elem), length(Elem, Largura), Py ==  
Largura,  
getTile(B, Px, Py, T), \+ empty\_Tile(T), T = tile(C, S),  
Py1 is Py + 1, verify\_Right\_aux(B, Px, Py1, LS, LC).  
verify\_Right\_aux(B, Px, Py, LS, LC) :- nth0(0, B, Elem), length(Elem, Largura), (Py ==  
Largura+1; (getTile(B, Px, Py, T), empty\_Tile(T))), LS = [], LC = [], !.

verify\_Right(B, Px, Py, LS, LC) :- Py1 is Py + 1, verify\_Right\_aux(B, Px, Py1, LS, LC).

verify\_Down\_aux(B, Px, Py, [S|LS], [C|LC]) :- length(B, Altura), Py == Altura,  
getTile(B, Px, Py, T), \+ empty\_Tile(T), T = tile(C, S),  
Px1 is Px + 1, verify\_Down\_aux(B, Px1, Py, LS, LC).  
verify\_Down\_aux(B, Px, Py, LS, LC) :- length(B, Altura), (Px == Altura+1; (getTile(B, Px, Py,  
T), empty\_Tile(T))), LS = [], LC = [], !.

verify\_Down(B, Px, Py, LS, LC) :- Px1 is Px + 1, verify\_Down\_aux(B, Px1, Py, LS, LC).

```

verify_Up_aux(B, Px, Py, [S|LS], [C|LC]) :- Px > 0,
    getTile(B, Px, Py, T), \+ empty_Tile( T), T = tile(C, S),
    Px1 is Px - 1, verify_Up_aux(B, Px1, Py, LS, LC).
verify_Up_aux(B, Px, Py, LS, LC) :- (Px == 0; (getTile(B, Px, Py, T), empty_Tile( T ))), LS = [],
LC = [], !.

```

```

verify_Up(B, Px, Py, LS, LC):- Px1 is Px - 1, verify_Up_aux(B, Px1, Py, LS, LC).

```

```

inBounds(B, X, Y):- length(B, H), nth1(1, B, L), length(L, W),
    between(1, H, X), between(1, W, Y).

```

```

isEmpty(B, X, Y):- getTile(B, X, Y, Tile2), Tile2 = tile(' ', ' ').

```

```

verify_Vert(B, Px, Py, LS, LC) :- verify_Up(B, Px, Py, LS1, LC1), verify_Down(B, Px, Py,
LS2, LC2),
    append(LS1, LS2, LS), append(LC1, LC2, LC).

```

```

verify_Hor(B, Px, Py, LS, LC) :- verify_Right(B, Px, Py, LS1, LC1), verify_Left(B, Px, Py,
LS2, LC2),
    append(LS1, LS2, LS), append(LC1, LC2, LC).

```

```

valid(B, _T, _Px, _Py) :- all_Empty_Board(B, 0), !.
valid(B, T, Px, Py) :- inBounds(B, Px, Py), isEmpty(B, Px, Py), verify_Vert(B, Px, Py, LS, LC),
    T = tile(C, S), append(LC, [C], LC3), append(LS, [S], LS3),
    !, all_same_or_different(LS3), all_same_or_different(LC3).

```

```

valid(B, T, Px, Py) :- inBounds(B, Px, Py), isEmpty(B, Px, Py), verify_Hor(B, Px, Py, LS, LC),
    T = tile(C, S), append(LC, [C], LC3), append(LS, [S], LS3),
    !, all_same_or_different(LS3), all_same_or_different(LC3).

```

```

belong_toHand(Move, Hand) :- Move = [T, _Px, _Py], member(T, Hand).

```

```

valid_ListMoves(_B, [_Move|_T], _Hand).
valid_ListMoves(B, [Move|T], Hand) :- Move = [T, Px, Py], belong_toHand(Move, Hand),
valid(B,T,Px, Py), valid_ListMoves(B, T, Hand).

```

```

%////////////////////////////////////VALID-MOV////////////////////////////////////
////////////////////////////////////

```

```

hasNeighbour(_B, _X, _Y, []):-!, fail.

```

```

hasNeighbour(B, X, Y, [[Dx, Dy]|_Ds]):- X1 is X + Dx, Y1 is Y + Dy, inBounds(B, X1, Y1),
\+isEmpty(B, X1, Y1), !.
hasNeighbour(B, X, Y, [_Dx, _Dy]|Ds):- hasNeighbour(B, X, Y, Ds).

```

```

validPositions(_, []).
validPositions(B, [FirstMove|OtherMoves]):- FirstMove =[_Tile, X, Y], validPosition(B, X, Y),
apply_moves(B, [FirstMove], NewB), validPositions(NewB, OtherMoves).

```

/\*Verifies if a Given Tile belongs to the Hand\*/

```

inHand(List, Hand):- sublist(Hand, List, _, _, _), List \= [].
inHandPos(List, Hand):- var(List), subl(Hand, TileTemp), TileTemp \=
[], permutation(TileTemp, Tiles), extract_pos(List, Tiles, _).
inHandPos(List, Hand):- nonvar(List), extract_pos(List, Tiles, _), subl(Hand, Tiles).

```

/\*Extracts position from a list of moves formed by Tile, Px, Py\*/

```

extract_pos([], [], []).
extract_pos([Move|Moves], [Tile|Tiles], [Pos|Positions]):- Move = [Tile, X, Y], Pos = [X, Y], !,
extract_pos(Moves, Tiles, Positions).

```

/\*Dir code ser 01- UP 0-1- DOWN 10- LEFT -10- RIGHT \*/

```

validMovAux(B, [X, Y], _D, _T, [], [], []) :- \+((length(B, H), nth0(0, B, L), length(L, W) , X>0, X
=<H, Y>0, Y=<W)), !.
validMovAux(B, [X, Y], _D, _T, [], [], []) :- getTile(B, X, Y, Tile), Tile = tile(' ', ' '), !.

```

```

validMovAux(B, [X, Y], [DirX, DirY], T, [[X, Y]|Tseen], [S|LS], [C|LC]) :- member([X, Y], T), !,
getTile(B, X, Y, Tile), Tile = tile(C, S),

```

```

NextX is X + DirX,
NextY is Y + DirY,
validMovAux(B, [NextX, NextY], [DirX, DirY], T,

```

```

Tseen, LS, LC).

```

```

validMovAux(B, [X, Y], [DirX, DirY], T, Tseen, [S|LS], [C|LC]) :- getTile(B, X, Y, Tile), Tile =
tile(C, S),

```

```

NextX is X + DirX,
NextY is Y + DirY,
validMovAux(B, [NextX, NextY], [DirX, DirY], T,

```

```

Tseen, LS, LC).

```

```

validPosition(B, X, Y):- getTile(B, X, Y, Tile), Tile = tile(' ', ' '),
findall([X1, Y1], (between(-1, 1, X1), between(-1, 1, Y1), X1*Y1=:=0,
X1+Y1=:=0), L),
hasNeighbour(B, X, Y, L).

```

/\*Tests if a Move is valid or not, if it isnt returns a valid one...\*/

validMov(B,[H|T], Hand):- validMovVert(B,[H|T], Hand, \_Pont1) , validMovHor(B,[H|T], Hand, \_Pont2).

validMovVert(B,[H|T], Hand, Pontf) :-

    inHandPos([H|T], Hand),  
    extract\_pos([H|T], [\_FirstTile|\_Other], [FirstPosition|Positions]),  
    FirstPosition = [X,Y],  
    validPositions(B, [H|T]),  
    apply\_moves(B, [H|T], NewBoard),  
    validMovAux(NewBoard, [X,Y], [1,0] , Positions, Tseen1, LS1, LC1),  
    X1 is X - 1,  
    validMovAux(NewBoard, [X1,Y], [-1,0], Positions, Tseen2, LS2, LC2),  
    append(Tseen1, Tseen2, Tseen),  
    append(LS1, LS2, LS),  
    append(LC1, LC2, LC),  
    length(LC, N),  
    if((Tseen = Positions, all\_same\_or\_different(LS), all\_same\_or\_different(LC)),  
    (if(N =:= 1, Pont is 0,  
    if(N mod 5 =:= 0, Pont is N \*2, Pont is N ))),Pont is 0),  
    Pontf is Pont.

validMovHor(B,[H|T], Hand, Pontf) :-

    inHandPos([H|T], Hand),  
    extract\_pos([H|T], [\_FirstTile|\_Other], [FirstPosition|Positions]),  
    FirstPosition = [X,Y],  
    validPositions(B, [H|T]),  
    apply\_moves(B, [H|T], NewBoard),  
    validMovAux(NewBoard, [X,Y], [0,1] , Positions, Tseen1, LS1, LC1),  
    Y1 is Y - 1,  
    validMovAux(NewBoard, [X,Y1], [0,-1], Positions, Tseen2, LS2, LC2),  
    append(Tseen1, Tseen2, Tseen),  
    append(LS1, LS2, LS),  
    append(LC1, LC2, LC),  
    length(LC, N),  
    if((Tseen = Positions, all\_same\_or\_different(LS), all\_same\_or\_different(LC)),  
    (if(N =:= 1, Pont is 0,  
    if(N mod 5 =:= 0, Pont is N \*2, Pont is N ))),Pont is 0),  
    Pontf is Pont.

/\*Tests if in a given List all elements are either all the same or all different\*/

all\_same\_or\_different(L):- all\_same( L ), !.

all\_same\_or\_different(L):- all\_different( L ), !.

```

all_different([]).
all_different([H|T]):- \+member(H, T), all_different( T ).
all_same([]).
all_same([H|T]):- length(T,N), listElement(T, N, H).


%////////////////////////////////////BOT2 -
SMART////////////////////////////////////

/*return the best Move the one that uses more tiles...*/

evaluateMov(N, Lista) :- length(Lista, N).

subl_aux(N, List, Sub):- subset(N, List, Sub).
subl_aux(N, List, Sub):- N > 1, N1 is N - 1, subl_aux(N1, List, Sub).

subl(List, Sub):- length(List, N), subl_aux(N, List, Sub).

subset(Len, [E|Tail], [E|NTail]):- PLen is Len - 1, (PLen > 0 -> subset(PLen, Tail, NTail) ;
NTail=[]).
subset(Len, [_|Tail], NTail):- subset(Len, Tail, NTail).

best_Mov(B, Hand, Best) :- validMov(B, Best, Hand).


/*TENTATIVA DE BOT RANDOM...*/
random_bot_Mov(_B, _Hand, _ListBest, 10).
random_bot_Mov(B, Hand, [Best|L], CC) :- CC > 0, CC < 10, validMov(B, Best, Hand), CC1
is CC+1, random_bot_Mov(B, Hand, L, CC1).

apllly_random_Bot(B, Hand, Best) :- random(1,10, Num), random_bot_Mov(B, Hand,
ListBest, 1), nth0(Num, ListBest, Best).


%////////////////////////////////////PLAYER1 VS
PLAYER2////////////////////////////////////

/*ASKS COLER, LETTER, POSITION IN ORDER TO MOVE TILE... */
moveTile(C, S, Px, Py) :- write('Choose the tile to play. DO NOT PUT A DOT IN THE END!!!!
Color: '),read_line(_),
                        read_line(X1), name(C, X1), color( C ), write('Shape: '), read_line(X2),
name(S, X2), shape(S), write('Choose where to place it. Row? '),
                        read(Px), write('Column ?'), read(Py).

numberTiles(L) :- write('How many tiles do you want to play? (1,2,3,4,5)'), read(Count),
Count > 0, Count < 6, movement(Count, L).

```

```

movement(0, _L).
movement(Count, [Move|L]) :- Count > 0, moveTile(C, S, Px, Py), T = tile(C, S), Move = [T,
Px, Py],
                                N1 is Count - 1, movement(N1, L).

```

```

playPlayerMove(B, Hand, NewHand, NewBoard) :- displayBoard(B), nl,
displayHand(Hand,0), nl, numberTiles(L), validMov(B, L, Hand),nl,
                                apply_moves(B, L, Hand, NewHand, NewBoard),nl,
displayBoard(NewBoard), nl, nl.

```

```

%////////////////////////////////////MENUS////////////////////////////////////
////////////////////////////////////

```

```

/* menu */

```

```

logo :- write('_____'), nl,
        write(' |           |'), nl,
        write(' |   Q!NTO   |'), nl,
        write(' |_____|'), nl, nl.

```

```

playlogo :- write('_____'), nl,
            write(' |           |'), nl,
            write(' |   PLAY!   |'), nl,
            write(' |_____|'), nl, nl.

```

```

menu :- load, repeat, write('\33\[2J'), nl, logo, write('----- MENU -----'), nl, nl,
        write('----- '), write('1. Play'), write('-----'), nl,
        write('----- '), write('2. Exit'), write('-----'), nl, nl,
        write('Write the number of the option followed by a dot.'), nl,
        read(C), C>0, C=<2, number(C), choice(C).

```

```

menuPlay :- repeat, write('\33\[2J'), nl, playlogo, write('----- PLAY MENU -----'), nl,
nl,

```

```

        write('----- '), write('1. Player vs Player'), write('-----'), nl,
        write('----- '), write('2. Player vs Computer'), write('-----'), nl,
        write('----- '), write('3. Computer vs Computer'), write('-----'), nl,
        write('----- '), write('4. Go Back'), write('-----'), nl,
        write('----- '), write('5. Exit'), write('-----'), nl, nl,
        write('Write the number of the option followed by a dot.'), nl,
        read(P), P>0, P=<5, number(P), playOp(P).

```

```

/* Menu Options */

```

```

choice(1) :- menuPlay.
choice(2) :- abort.

```

```

/* Play Options */

```



```

playOp(1) :- write("\33[2J"), nl, createBoard(3,3,B), makeDeck(Deck),
mixingElemntsDeck(Deck, NewDeck),
                creatingHand(NewDeck, Hand1, Hand2), write('      ----- Player1! -----'), nl,
game(B, 0, Hand1, Hand2, 0, 0).

```

```

playOp(2) :- write("\33[2J"), nl, createBoard(3,3,B), makeDeck(Deck),
mixingElemntsDeck(Deck, NewDeck),
                creatingHand(NewDeck, Hand1, Hand2), write('      ----- Player1! -----'), nl,
gameHBot(B, 0, Hand1, Hand2, 0, 0).

```

```

playOp(3) :- write("\33[2J"), nl, createBoard(3,3,B), makeDeck(Deck),
mixingElemntsDeck(Deck, NewDeck),
                creatingHand(NewDeck, Hand1, Hand2), createCenter(B, 3, 3, Hand1, Bnew,
NHand1), write('      ----- Player1! -----'),nl, gameBot(Bnew, 0, NHand1, Hand2, 0, 1).

```

```

playOp(4) :- menu.
playOp(5) :- abort.

```

```

/* game loop */

```

```

done([]).

```

```

playMove(B, Hand, NewHand, NewBoard) :- displayBoard(B), nl, displayHand(Hand,0), nl,
numberTiles(L),valid_ListMoves(B, L, Hand),nl, apply_moves(B, L, Hand, NewHand,
NewBoard),nl, displayBoard(NewBoard), nl, nl.

```

```

playBotMov(B, Hand, NewHand, NewBoard) :- displayBoard(B), nl, displayHand(Hand,0), nl,
best_Mov(B, Hand, Best),nl,
                apply_moves(B, Best, Hand, NewHand, NewBoard), nl,
displayBoard(NewBoard), nl, nl.

```

```

game(Board,0, Hand1, Hand2, 0, 0):- \+done(Hand1), \+done(Hand2),
expand_matrix_up(Board, NB), expand_matrix_down(NB, NNB),
                expand_matrix_left(NNB, NNNB), expand_matrix_right(NNNB,
NNNNB), playMove(NNNNB, Hand1, NewHand, NewBoard),
                !, write('      ----- Player2! -----'), nl, game(NewBoard,1, NewHand,
Hand2, 1, 0).

```

```

game(Board,1, Hand1, Hand2, 1, 0):- \+done(Hand1), \+done(Hand2),
expand_matrix_up(Board, NB),expand_matrix_down(NB, NNB),
                expand_matrix_left(NNB, NNNB), expand_matrix_right(NNNB,
NNNNB) , playMove(NNNNB, Hand2, NewHand, NewBoard),
                !, write('      ----- Player1! -----'), nl, game(NewBoard,0, Hand1,
NewHand, 0, 0).

```

```

gameBot(Board,0, Hand1, Hand2, 0, 1):- \+done(Hand1), \+done(Hand2),
expand_matrix_up(Board, NB), expand_matrix_down(NB, NNB),
            expand_matrix_left(NNB, NNNB), expand_matrix_right(NNNB,
NNNNB),playBotMov(NNNNB, Hand1, NewHand, NewBoard),
            !, write('      ----- Player2! -----'), nl, gameBot(NewBoard,1,
NewHand, Hand2, 1, 1).
gameBot(Board,1, Hand1, Hand2, 1, 1):- \+done(Hand1), \+done(Hand2),
expand_matrix_up(Board, NB), expand_matrix_down(NB, NNB),
            expand_matrix_left(NNB, NNNB), expand_matrix_right(NNNB,
NNNNB), playBotMov(NNNNB, Hand2, NewHand, NewBoard),
            !, write('      ----- Player1! -----'), nl, gameBot(NewBoard,0, Hand1,
NewHand, 0, 1).

```

```

gameHBot(Board,0, Hand1, Hand2, 0, 0):- \+done(Hand1), \+done(Hand2),
expand_matrix_up(Board, NB), expand_matrix_down(NB, NNB),
            expand_matrix_left(NNB, NNNB), expand_matrix_right(NNNB,
NNNNB), playMove(NNNNB, Hand1, NewHand, NewBoard),
            !, write('      ----- Player2! -----'), nl, gameHBot(NewBoard,1,
NewHand, Hand2, 1, 0), print('sdds').
gameHBot(Board,1, Hand1, Hand2, 1, 0):- \+done(Hand1), \+done(Hand2),
expand_matrix_up(Board, NB), expand_matrix_down(NB, NNB),
            expand_matrix_left(NNB, NNNB), expand_matrix_right(NNNB,
NNNNB), playBotMov(NNNNB, Hand2, NewHand, NewBoard),
            !, write('      ----- Player1! -----'), nl, gameHBot(NewBoard,0,
Hand1, NewHand, 0, 0).

```

```

game(_,0,_,_,_,_):- nl, write('Player 1 Won!'), nl.
game(_,1,_,_,_,_):- nl, write('Player 2 Won!'),nl.
game(_,_,_,_,_,0):- nl, write('Bot 1 Won!'), nl.
game(_,_,_,_,_,1):- nl, write('Bot 2 Won!'),nl.

```

```

/* Load librarys */

```

```

load :- use_module(library(random)), use_module(library(lists)).

```