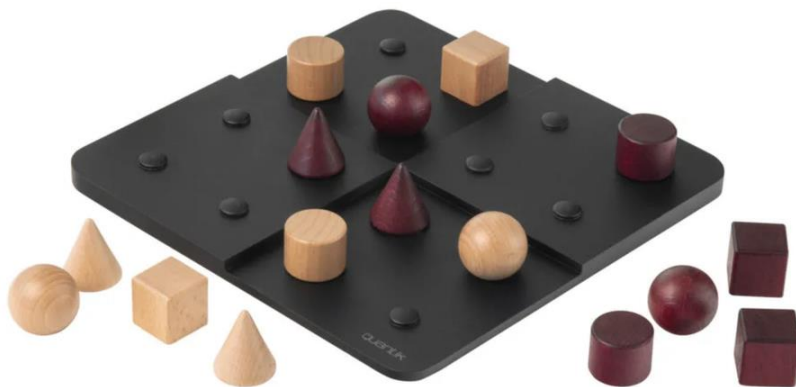


Programação Lógica

# Quantik

Aplicação em Prolog para um Jogo de Tabuleiro



**Grupo Quantik\_1:**

João Praça - up201704748

Lucas Ribeiro - up201705227

## 1. Introdução

- Iniciamos assim este projecto no âmbito da disciplina de Programação Lógica. Este baseia-se no desenvolvimento de um jogo de tabuleiro, no nosso caso o jogo em questão será o *Quantik*. O jogo deverá ter três modos de utilização, Humano contra Humano, Humano contra Computador e Computador contra Computador, tendo o computador dois níveis de dificuldade.

Neste relatório temos o objectivo de explicar o funcionamento do jogo, tal como o modo de procedimento aquando o desenvolvimento do mesmo.

## 2. O Jogo Quantik

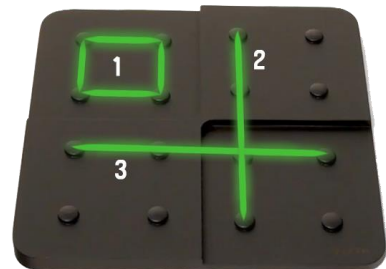
### História

- O *Quantik* é um jogo desenvolvido pela publicadora *Gigamic*, de origem francesa, esta publicadora foi criada por 3 irmãos, tendo já criado mais de 400 jogos de tabuleiro, tendo estes sido adaptados para mais de 45 países.
- O designer do jogo foi Nouri Khalifa, mesmo criador dos jogos *Damix* e *Ordo*.
- O jogo está indicado como pertencente ao género de estratégia abstrata, sendo descrito como um jogo para dois jogadores, com um tempo de duração média de 20 minutos e recomendado para qualquer pessoa com mais de 8 anos.



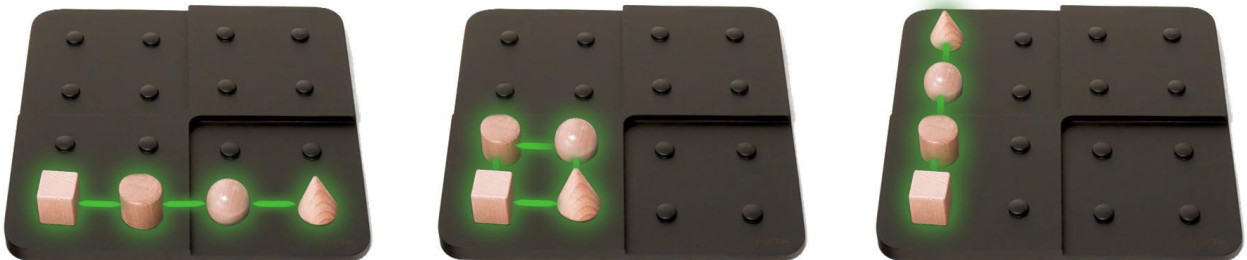
### Regras

- O tabuleiro do jogo divide-se em 4 quadrantes (1), sendo que em cada destes quadrantes existem 4 pontos onde se podem posicionar as peças, podendo-se assim dividir também o tabuleiro em 4 colunas (2) e 4 linhas (3).
- As peças do jogo são todas sólidos geométricos, podendo ser cones, cubos, cilindros ou esferas. Estas peças podem também ser claras ou escuras dependendo do jogador a que pertencem. Cada jogador tem direito a 8 peças, sendo estas 2 peças de cada tipo.



- O objetivo do jogo é formar uma linha (primeira figura abaixo), coluna (segunda figura abaixo) ou preencher um quadrante (terceira figura abaixo) com 4 peças diferentes, independentemente de quem as colocou, podendo

assim um jogador vencer ao colocar a 4 peça diferente numa linha em que as outras 3 peças previamente colocadas não lhe pertencem. Tendo os jogadores que à vez posicionar uma peça, contudo uma peça pode apenas ser posicionada numa linha, coluna ou quadrante em que o adversário ainda não colocou uma peça sua do mesmo tipo que a que o jogador pretende colocar.



### 3. Lógica do jogo

#### 3.1 Representação interna do estado do jogo

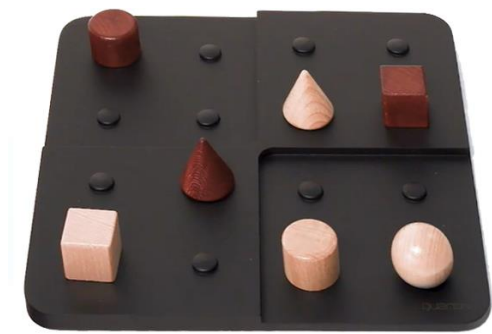
O jogo será representado em Prolog através de um termo, cujo primeiro elemento representa o jogador atual, o segundo elemento é uma lista de termos cell (que incluem uma linha, coluna e peça), o terceiro elemento é uma lista que representa as peças do jogador 1 e o quarto uma lista que representa as peças do jogador 2.

**Representação do estado inicial do tabuleiro:**



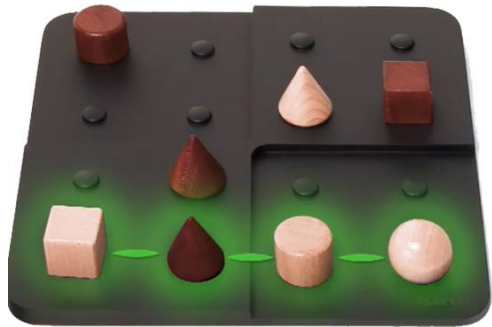
```
Board(p1, [cell(1,a,e), cell(1,a,e), cell(1,a,e),
cell(1,a,e), cell(1,a,e), cell(1,a,e), cell(1,a,e),
cell(1,a,e), cell(1,a,e), cell(1,a,e), cell(1,a,e),
cell(1,a,e)], [wo,wo,wy,wy,ws,ws,wc,wc],
[bo,bo,by,by,bs,bs,bc,bc])
```

Representação de um possível estado intermédio do tabuleiro:



Board(p2, [cell(1,a,by), cell(1,a,e), cell(1,a,e),  
cell(1,a,e), cell(1,a,e), cell(1,a,e), cell(1,a,wo),  
cell(1,a,bc), cell(1,a,e), cell(1,a,bo), cell(1,a,e),  
cell(1,a,e), cell(1,a,wc), cell(1,a,e), cell(1,a,wy),  
cell(1,a,ws)], [wo,wy,ws,wc], [bo,by,bs,bs,bc])

Representação de um possível estado final do tabuleiro:



Board(p2, [cell(1,a,by), cell(1,a,e), cell(1,a,e),  
cell(1,a,e), cell(1,a,e), cell(1,a,e), cell(1,a,wo),  
cell(1,a,bc), cell(1,a,e), cell(1,a,bo), cell(1,a,e),  
cell(1,a,e), cell(1,a,wc), cell(1,a,wo), cell(1,a,wy),  
cell(1,a,ws)], [wy,ws,wc], [bo,by,bs,bs,bc])

3.2 Visualização do tabuleiro em modo de texto

Estas são as 3 situações anteriormente referidas representadas do modo que são vistas na consola.

|   | a | b | c | d |
|---|---|---|---|---|
| 1 |   |   |   |   |
| 2 |   |   |   |   |
| 3 |   |   |   |   |
| 4 |   |   |   |   |











|   | a  | b  | c  | d  |
|---|----|----|----|----|
| 1 | BY |    |    |    |
| 2 |    |    | WO | BC |
| 3 |    | BO |    |    |
| 4 | WC |    | WY | WS |

|   | a  | b  | c  | d  |
|---|----|----|----|----|
| 1 | BY |    |    |    |
| 2 |    |    | WO | BC |
| 3 |    | BO |    |    |
| 4 | WC | BO | WY | WS |

### 3.3 Lista de jogadas válidas

Aquando a decisão de posicionamento de uma peça são determinadas as jogadas válidas sendo que no Quantik as jogadas válidas dependem da peça selecionada, mais concretamente das suas características como cor e forma.

Tomando o caso concreto representado à esquerda, em que o jogador das peças brancas é o atual, e terá selecionado o cone branco (wo). As células que contêm um círculo indicativo verde são células que representam jogadas válidas, ou seja, células onde o jogador poderia posicionar a peça selecionada, enquanto as vermelhas representam o inverso. O posicionamento da peça selecionada é considerado inválido nas células de coordenadas 1c e 2c sendo que estas células fazem parte da mesma coluna que a célula 3c, célula esta que contém uma peça (bo), que é da mesma forma mas de cor diferente da selecionada, ou seja o posicionamento da peça quebraria as regras do jogo. O mesmo se aplica para linhas, o que invalida o posicionamento em 3a e 3d, como também para quadrantes, invalidando-o em 4d e também de novo em 3d, que pertence à mesma linha e quadrante que a célula 3c.

|   | a   | b   | c   | d   |
|---|---|---|---|---|
| 1 | WO  |  |  |  |
| 2 | BC  |  |  | BY  |
| 3 |  | WS  | BO  |  |
| 4 |  |  | WS  |  |

Os seguintes são os predicados utilizados para a obtenção das jogadas válidas:

```
valid_moves(Board, Player, ListOfMoves) :-
    findall(Move, valid_move(Board, Player, Move), ListOfMoves).

valid_move(Board, Player, Move) :-
    getQuad(Row, Col, _Quad),
    piecePlayer(Player, Piece),
    hasPiece(Board, Player, Piece),
    verifyMove(Board, Row, Col, Piece),
    Move = [Row, Col, Piece].
```

**valid\_moves** - ao ser chamado preenche *ListofMoves* com todos as jogadas possíveis no momento, recorrendo aos predicados *findall* e *valid\_move*.

**valid\_move** - ao ser chamado testa todas as combinações de colunas, linhas, peças que um jogador tem ainda fora do tabuleiro, verificando se o movimento é valido através do predicado *verifyMove*, guardando a jogada na variável *Move*.

```
verifyMove(Board, Row, Col, Piece) :-
    verifyEmptyCell(Board, Row, Col),
    verifyRow(Board, Row, Piece),
    verifyColumn(Board, Col, Piece),
    getQuad(Row, Col, Quad),
    verifyQuad(Board, Quad, Piece).
```

**verifyMove** - verifica se uma peça pode ser colocada numa determinada célula do tabuleiro, esta verificação ocorre em 4 etapas, que são as seguintes:

```
verifyRow(Board, Row, Piece) :-
    oppositePiece(Piece, Opposite),
    \+getPiece(Board, Row, a, Opposite),
    \+getPiece(Board, Row, b, Opposite),
    \+getPiece(Board, Row, c, Opposite),
    \+getPiece(Board, Row, d, Opposite).
```

**verifyRow** - verifica se é possível colocar uma peça numa determinada linha do tabuleiro.

```
verifyColumn(Board, Col, Piece) :-
    oppositePiece(Piece, Opposite),
    \+getPiece(Board, 1, Col, Opposite),
    \+getPiece(Board, 2, Col, Opposite),
    \+getPiece(Board, 3, Col, Opposite),
    \+getPiece(Board, 4, Col, Opposite).
```

**verifyColumn** - verifica se é possível colocar uma peça numa determinada coluna do tabuleiro.

```
verifyQuad(Board, 1, Piece) :-
    oppositePiece(Piece, Opposite),
    \+getPiece(Board, 1, a, Opposite),
    \+getPiece(Board, 1, b, Opposite),
    \+getPiece(Board, 2, a, Opposite),
    \+getPiece(Board, 2, b, Opposite).
```

**verifyQuad** - verifica se é possível colocar uma peça num determinado quadrante do tabuleiro.

E por fim:

**verifyEmptyCell** - verifica se a célula onde se pretende colocar a peça seleccionada está ocupada ou não.

```
verifyEmptyCell(board(_CurrentPlayer, PiecesBoard, _PiecesPlayer1, _PiecesPlayer2), Row, Col) :-
    member(cell(Row, Col, e), PiecesBoard).
```

### 3.4 Execução de jogadas

```
move([Row, Col, Piece], Board, NewBoard) :-  
    setPiece(Board, Row, Col, Piece, NewBoard).
```

**move** - coloca uma peça numa célula com as coordenadas novas, criando um novo tabuleiro, representando esse movimento, para tal utiliza o predicado *setPiece*.

```
setPiece(board(FirstPlayer, PiecesBoard, PiecesPlayer1, PiecesPlayer2), Row,  
Col, Piece, board(FirstPlayer, NewPiecesBoard, NewPiecesPlayer1, PiecesPlayer2)) :-  
    playerNumber(FirstPlayer, 1),  
    updateBoardPieces(PiecesBoard, Row, Col, Piece, NewPiecesBoard),  
    updatePlayerPieces(PiecesPlayer1, 1, Piece, NewPiecesPlayer1).  
  
setPiece(board(SecondPlayer, PiecesBoard, PiecesPlayer1, PiecesPlayer2), Row,  
Col, Piece, board(SecondPlayer, NewPiecesBoard, PiecesPlayer1, NewPiecesPlayer2)) :-  
    playerNumber(SecondPlayer, 2),  
    updateBoardPieces(PiecesBoard, Row, Col, Piece, NewPiecesBoard),  
    updatePlayerPieces(PiecesPlayer2, 2, Piece, NewPiecesPlayer2).
```

**setPiece** - desenvolve as tarefas do predicado *move*, chamando os predicados *playerNumber*, *updateBoardPieces* e *UpdatePlayerPieces*.

```
playerNumber(p1, 1).  
playerNumber(p, 1).  
playerNumber(c1, 1).  
playerNumber(p2, 2).  
playerNumber(c, 2).  
playerNumber(c2, 2).
```

**playerNumber** - obtém o número do jogador atual em termos de jogo através da representação interna do jogador atual, representação esta que também inclui sobre se este é um jogador humano ou não.

```
updateBoardPieces(PiecesBoard, Row, Col, Piece, NewPiecesBoard) :-  
    delete_one(cell(Row, Col, e), PiecesBoard, NB),  
    append(NB, [cell(Row, Col, Piece)], NewPiecesBoard).
```

**updateBoardPieces** - atualiza o tabuleiro, removendo os dados sobre a célula onde se pretende colocar uma peça nova, e adicionando ao tabuleiro os dados novos referentes à peça que se pretende lá colocar.

```
updatePlayerPieces(PiecesPlayer1, 1, Piece, NewPiecesPlayer1) :-  
    delete_one(Piece, PiecesPlayer1, NewPiecesPlayer1).  
  
updatePlayerPieces(PiecesPlayer2, 2, Piece, NewPiecesPlayer2) :-  
    delete_one(Piece, PiecesPlayer2, NewPiecesPlayer2).
```

**updatePlayerPieces** - atualiza a lista de peças que o jogador ainda não posicionou ao remover a última a ser posicionada.

### 3.5 Final de Jogo

```
game_over(Board, Winner) :-  
    ... checkWin(Board),  
    ... getCurrentPlayer(Board, Player),  
    → Winner = Player.
```

**game\_over** - verifica se o jogador que fez a última jogada ganhou: se completou numa mesma linha, coluna ou quadrante quatro peças de forma diferente, mesmo que nem todas as peças colocadas anteriormente lhe pertençam; retorna o jogador atual

como Winner.

```
checkWin(Board) :-  
    ... checkRowWin(Board, 1);  
    ... checkRowWin(Board, 2);  
    ... checkRowWin(Board, 3);  
    ... checkRowWin(Board, 4);  
    ... checkColumnWin(Board, a);  
    ... checkColumnWin(Board, b);  
    ... checkColumnWin(Board, c);  
    ... checkColumnWin(Board, d);  
    ... checkQuadWin(Board, 1);  
    ... checkQuadWin(Board, 2);  
    ... checkQuadWin(Board, 3);  
    ... checkQuadWin(Board, 4).
```

**checkWin** - verifica para todas as linhas, colunas e quadrantes se têm um conjunto de quatro peças de forma diferente.

```
checkRowWin(Board, Row) :-  
    ... rowToList(Board, Row, Pieces),  
    ... checkListUnique(Pieces, 4).
```

**check(...)Win** - para cada linha/coluna/quadrante, converte todas as suas peças para as respetivas formas e verifica que tem 4 elementos distintos.



### 3.6 Avaliação do Tabuleiro

```
value(Board, Player, Value) :-
    (checkWin(Board) ->
        Value is 999999;
        (
            value_valid_moves(Board, Player, Moves, Valid),
            value_winning_moves(Board, Player, Moves, Winning),
            value_losing_moves(Board, Player, Moves, Losing),
            UpdatedValid is (Valid - Winning - Losing),
            Value is (UpdatedValid + Winning*100 - Losing*100)
        )
    ).
```

**value** - através dos dados de um tabuleiro e do jogador atual, a função obtém um valor que será quanto mais elevado quanto maior a probabilidade de se atingir a vitória a partir desse tabuleiro. Se o tabuleiro representar uma vitória, o valor é suficientemente grande de modo a que seja o preferido.

```
value_valid_moves(Board, Player, ListOfMoves, Value) :-
    valid_moves(Board, Player, ListOfMoves),
    length(ListOfMoves, Value).
```

**value\_valid\_moves** - calcula a quantidade de jogadas possíveis a partir de uma distribuição das peças tanto no tabuleiro como na posse do jogador. Cada jogada válida vale 1 ponto.

```
value_winning_moves(Board, Player, ListOfMoves, Value) :-
    winning_moves(Board, Player, ListOfMoves, [], ListOfWinningMoves),
    length(ListOfWinningMoves, Value).
```

**value\_winning\_moves** - calcula a quantidade de jogadas que levariam à vitória na sua jogada seguinte, a partir de uma distribuição das peças tanto no tabuleiro como na posse do jogador, isto é, verifica se cada jogada leva a uma vitória, e garante que o oponente não tem uma peça com a mesma forma da jogada vencedora. Cada jogada que garante a vitória vale 100 pontos.

```
value_losing_moves(Board, Player, ListOfMoves, Value) :-
    losing_moves(Board, Player, ListOfMoves, [], ListOfLosingMoves),
    length(ListOfLosingMoves, Value).
```

**value\_losing\_moves** - calcula a quantidade de jogadas que levariam a uma possível derrota na jogada seguinte, a partir de uma distribuição das peças tanto no tabuleiro como na posse do jogador, isto é, verifica se cada jogada leva a uma vitória, e que o oponente tem uma peça com a mesma forma da jogada vencedora e pode jogá-la. Cada jogada que pode levar à derrota vale -100 pontos.

### 3.7 Jogada do Computador

```
choose_move(Board, 1, Move) :-  
    ....getCurrentPlayer(Board, Player),  
    ....valid_moves(Board, Player, ListOfMoves),  
    ....random_member(Move, ListOfMoves).
```

**choose\_move** - encontra a lista de jogadas válidas para o jogador atual e escolhe uma aleatoriamente, caso o nível de jogo seja *Fácil*.

```
choose_move(Board, Level, Move) :-  
    ....getCurrentPlayer(Board, Player),  
    ....valid_moves(Board, Player, ListOfMoves),  
    ....NewLevel is Level-1,  
    ....best_move(Board, Player, NewLevel, ListOfMoves, -999999, _MaxValue, [], BestMove),  
    ....(BestMove \= 0 ->  
        .... Move = BestMove;  
        .... random_member(Move, ListOfMoves)  
    ....).
```

Caso o nível seja *Médio* ou *Difícil*, encontra a lista de jogadas válidas e escolhe a melhor jogada possível utilizando **best\_move**.

```
best_move(Board, Player, 1, [Move | ListOfMoves], Curr_MaxValue, MaxValue, Curr_BestMove, BestMove) :-  
    ....move(Move, Board, NewBoard),  
    ....value(NewBoard, Player, Value),  
    ....(Value > Curr_MaxValue ->  
        .... best_move(Board, Player, 1, ListOfMoves, Value, MaxValue, [Move], BestMove);  
        .... (Value <= Curr_MaxValue ->  
            .... best_move(Board, Player, 1, ListOfMoves, Value, MaxValue, [Move | Curr_BestMove], BestMove);  
            .... best_move(Board, Player, 1, ListOfMoves, Curr_MaxValue, MaxValue, Curr_BestMove, BestMove)  
            ....)  
    ....).
```

```
best_move(Board, Player, Depth, [Move | ListOfMoves], Curr_MaxValue, MaxValue, Curr_BestMove, BestMove) :-  
    ....Depth >= 1,  
    ....NewDepth is Depth - 1,  
    ....move(Move, Board, NewBoard),  
    ....write('.'),  
    ....(checkWin(NewBoard) ->  
        .... best_move(Board, Player, Depth, [], -999999, MaxValue, [Move], BestMove);  
        ....(  
            ....opponent_best_move(NewBoard, Player, NewDepth, OpponentBoard),  
            ....(checkWin(OpponentBoard) ->  
                .... best_move(Board, Player, Depth, ListOfMoves, Curr_MaxValue, MaxValue, Curr_BestMove, BestMove);  
                ....(  
                    .... valid_moves(OpponentBoard, Player, ListOfNextMoves),  
                    .... best_move(OpponentBoard, Player, NewDepth, ListOfNextMoves, -999999, Value, _Aux, _NextMove),  
                    ....(Value > Curr_MaxValue ->  
                        .... best_move(Board, Player, Depth, ListOfMoves, Value, MaxValue, [Move], BestMove);  
                        ....(Value <= Curr_MaxValue ->  
                            .... best_move(Board, Player, Depth, ListOfMoves, Value, MaxValue, [Move | Curr_BestMove], BestMove);  
                            .... best_move(Board, Player, Depth, ListOfMoves, Curr_MaxValue, MaxValue, Curr_BestMove, BestMove)  
                        ....)  
                    ....)  
                ....)  
            ....)  
        ....)  
    ....).
```

**best\_move** - determina a melhor jogada possível com base num tabuleiro, no jogador atual e nas suas jogadas válidas. Quando a profundidade de procura é maior que 1, para cada jogada válida avalia se obteve a vitória; caso tal não suceda, obtém a melhor jogada possível do oponente, determina se este venceu, ou caso não vença, calcula a melhor jogada do tabuleiro resultante e com profundidade decrementada; de seguida, avalia o valor do board obtido em relação ao máximo atual: se for maior, cria uma nova lista de jogadas de valor máximo e atualiza o valor máximo; se for igual acrescenta a jogada à lista, se for menor, continua a percorrer a lista de jogadas válidas sem alterar nenhum argumento.

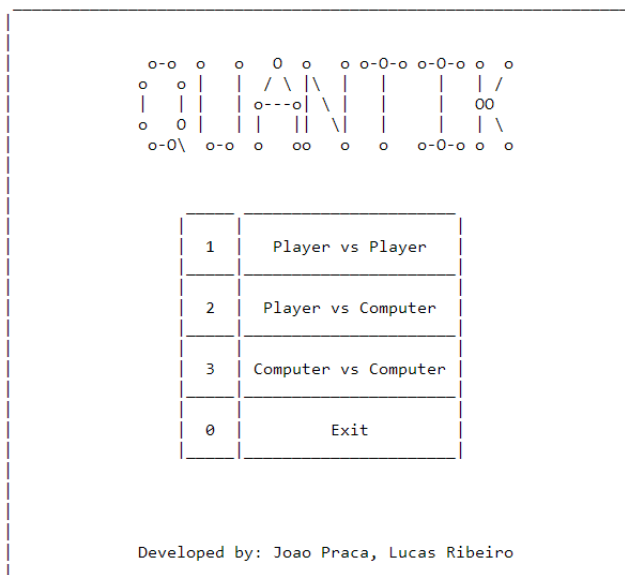
Quando a profundidade é 1, avalia apenas as jogadas válidas do tabuleiro e jogador atuais, e devolve uma lista com todas as jogadas que levam a tabuleiros cujo valor é máximo.

Por fim, quando a lista de jogadas válidas é totalmente percorrida, escolhe aleatoriamente uma jogada da lista de melhor jogadas, e devolve-a. No caso de não haver nenhuma melhor jogada, ou seja, terem todas o mesmo valor, escolhe aleatoriamente uma jogada válida.

### 3.8 Menu do jogo e Procedimento de utilização

No desenvolvimento do projeto foi utilizado o programa SICStus PROLOG, com o tipo de fonte Consolas, tamanho 12.

Para facilitar o envolvimento do utilizador com o jogo, tentamos desenvolver uma interface intuitiva, que procederemos a explicar aqui:



O menu do jogo contém 4 opções, correspondendo estas aos 3 modos de jogos e uma quarta que é a opção de abandonar o jogo. A primeira opção refere-se a um modo de jogo em que dois jogadores se confrontam na mesma plataforma, a segunda permite ao utilizador jogar sozinho, enfrentando o seu computador e por fim, a terceira, permite que o utilizador simule jogos entre computadores, podendo seleccionar o nível destes.

No final de cada ecrã é requerido ao utilizador que este tome algum tipo de escolha ou opção.

Please choose an option: █

|   |        |
|---|--------|
| 1 | Easy   |
| 2 | Medium |
| 3 | Hard   |

A segunda escolha do primeiro menu leva-nos a este ecrã, onde é requerido ao utilizador que escolha a dificuldade do computador que irá defrontar.

Este é também o ecrã a que nos leva a terceira escolha, porém teremos de fazer esta escolha duas vezes, uma por cada computador na simulação.

```
'Player 1'  
Pieces : 'WO' 'WO' 'WY' 'WY' 'WS' 'WS' 'WC' 'WC'
```

```
Player 1, what piece do you want to place? (use lower case) |: wc.
```

```
The piece wc has been selected.
```

```
Player 1, in what row do you want to place it? |: 1.
```

```
Player 1, in what column do you want to place it? |: a.
```

Por fim, durante o jogo em si, a cada ronda o jogador, para além do tabuleiro, poderá também visualizar as suas peças, será também questionado 3 vezes: primeiro sobre que peça pretende posicionar nesta ronda, segundo em que linha a pretende posicionar e por fim em que coluna.

```
Player 2, what piece do you want to place? (use lower case) |: ws.
```

```
The piece ws is not available.
```

Caso o utilizador introduza input não reconhecido ou que não respeite as regras do jogo, uma mensagem informativa será observada, sendo a da figura acima uma delas.

## 4. Conclusão

Este projeto teve o objetivo de nos levar a implementar o conhecimento adquirido através das aulas da unidade curricular de Programação Lógica, mais concretamente desenvolver competências de programação em PROLOG, sendo este projeto obviamente mais extenso e complexo que qualquer exercício que pudéssemos desenvolver nas aulas práticas.

Ao longo do projeto foram encontradas algumas dificuldades que tentámos ultrapassar, como a adaptação à linguagem PROLOG que estabelece à partida a sua diferença face às linguagens que estamos habituados a utilizar, que pelas diferentes estruturas de dados utilizadas quer pelo uso recorrente de recursão.

Evidentemente alguns aspectos do projeto poderão ser desenvolvidos mais extensamente, como aumentar a profundidade de pesquisa aquando a determinação da jogada mais favorável pelos jogadores não humanos e o melhoramento da eficácia destes métodos de determinação, ainda que consideremos que a versão atual é suficientemente boa e robusta.

Por fim, terminamos o projeto sabendo que desenvolvemos competências úteis para o melhor entendimento da linguagem PROLOG, entre outros âmbitos da unidade curricular.

## 5. Bibliografia

- <https://en.wikipedia.org/wiki/Gigamic>
- <https://www.boardgamegeek.com/boardgame/286295/quantik>
- <https://en.gigamic.com/game/quantik>
- <https://www.swi-prolog.org/>
- Slides presentes no Moodle