

**Programação em Lógica**

**Relatório Final**

**Barragoon 4**

****

**Leonardo Manuel Gomes Teixeira – up201502848**

**Maria Eduarda Santos Cunha – up201506524**

# Resumo

Este projeto é uma possível abordagem ao jogo Barragoon, descrito em detalhe na secção O Jogo Barragoon, em PROLOG.

Todo o processo de desenvolvimento verificou-se bastante difícil dada a novidade do paradigma da linguagem de programação em questão para nós, aliado ao facto de o próprio jogo ter uma lógica bastante complexa, pelo seu número elevado de regras e restrições.

Conseguimos implementar com sucesso todas as regras do jogo e completar os três modos de jogo – humano contra humano, humano contra computador e computador contra computador, e criamos 2 níveis de dificuldade.

Apesar do volume de regras, pensamos que a versão final é bastante *user friendly* e intuitiva de jogar.

De forma geral, o nosso feedback é bastante positivo. Este trabalho foi fundamental para a consolidação dos conteúdos lecionados nas aulas práticas e teóricas e sentimo-nos capazes de realizar os novos níveis ainda que não tenhamos tido tempo para tal.

Índice

[1. Resumo 2](#_Toc498293799)

[2. O Jogo Barragoon 4](#_Toc498293800)

[2.1. Tabuleiro e Peças 4](#_Toc498293801)

[2.2. Objetivo 4](#_Toc498293802)

[2.3. Movimentos 5](#_Toc498293803)

[2.4. Regras 5](#_Toc498293804)

[3. Lógica do Jogo 7](#_Toc498293805)

[3.1. Representação do Estado de Jogo 7](#_Toc498293806)

[3.2. Visualização do Tabuleiro 9](#_Toc498293807)

[3.3. Lista de Jogadas Válidas 11](#_Toc498293808)

[3.4. Execução de Jogadas 11](#_Toc498293809)

[3.5. Avaliação do Tabuleiro 12](#_Toc498293810)

[3.6. Final do Jogo 12](#_Toc498293811)

[3.7. Jogada do Computador 13](#_Toc498293812)

[4. Interface com o Utilizador 14](#_Toc498293813)

[5. Conclusões 16](#_Toc498293814)

[6. Bibliografia 16](#_Toc498293815)

[7. Anexos 17](#_Toc498293816)

# O Jogo Barragoon

O Barragoon é um jogo de estratégia sem qualquer fator de aleatoriedade para 2 jogadores. Foi publicado pela primeira vez a 3 de março de 2014 pela companhia WiWa Spiele UG. As suas regras foram atualizadas pela última vez a 30 de março de 2016.

## Tabuleiro e Peças

O jogo realiza-se num tabuleiro de 9x7 células e os jogadores jogam sempre à vez.

Existem 2 tipos principais de peças: as telhas de cada jogador, brancas ou castanhas, e os barragoons.

* As telhas dos jogadores possuem na sua face um símbolo com 2, 3 ou 4 círculos, relativo ao número de células que podem andar num só movimento (fig.1). Cada jogador começa com 7 telhas: 2 de 2 círculos, 3 de 3 círculos e 2 de 4 círculos.
* O barragoon é a peça central do jogo. É uma peça cúbica, em que cada uma das suas faces possui um símbolo que indica a permissão do jogador de mover a sua peça pela célula em que o barragoon se encontra (fig.2). O jogo começa com 8 barragoons, mas existem 32.



**Sem passagem**

**Uma direção**

**Duas direções**

**Virar à direita**

**Virar à esquerda**

**Todas as direções**

Figura 1: Faces de um Barragoon



**2 células**

**3 células**

**4 células**

Figura 2: Telhas Brancas ou Castanhas

## Objetivo

Ambos os jogadores têm de recorrer às suas aptidões táticas para mover as suas telhas e dispor os barragoons de forma a que lhes seja permitido capturar todas as telhas do outro jogador ou, pelo menos, impedir o seu progresso.

O jogo acaba quando um dos jogadores já não consegue mover telhas, porque não possui nenhuma ou por as que tem se encontrarem limitadas por barragoons. O outro é o vencedor.

## Movimentos

Existem 2 tipos de movimentos: full moves e short moves.

* Os full moves correspondem a percorrer x células, de acordo com o número de círculos na telha do jogador (2, 3 ou 4), respetivamente (fig.3);
* Os short moves correspondem a percorrer x-1 células, de acordo com o número de círculos na telha do jogador (2, 3 ou 4, logo movimentos de 1, 2 ou 3 células), respetivamente (fig.4).



Figura 4: Short Move com Telha de 3 Círculos



Figura 3: Full Move com Telha de 4 Círculos

## Regras

* Uma peça é capturada se a peça do oponente terminar na mesma célula que ela (fig.5);
* Só é possível capturar uma peça durante um full move;
* As telhas com 2 círculos não podem capturar barragoons com o símbolo “todas as direções” virado para cima (fig.6);
* Se um barragoon for capturado, tem de voltar a ser colocado no tabuleiro, numa posição livre à escolha do jogador, com a face levantada para cima que ele preferir;
* Sempre que uma telha é capturada, são adicionados 2 barragoons novos ao tabuleiro, um por cada jogador, e coloca primeiro no tabuleiro o jogador cuja telha foi capturada;
* Nunca se pode mudar um barragoon de posição uma vez colocado;
* Quando se toca numa telha para a mover, não se pode trocar por outra ou voltar atrás;
* Durante um movimento, só se pode efetuar uma mudança de direção uma vez de 90º (fig.7);
* Os movimentos só podem ser verticais ou horizontais, nunca na diagonal.

Figura 6: Peça de 2 Círculos Não Captura Barragoon com Face “Todas as Direções” Voltada para Cima



Figura 5: Captura de um Barragoon

Figura 7: Movimento Impossível com 2 Mudanças de Direção



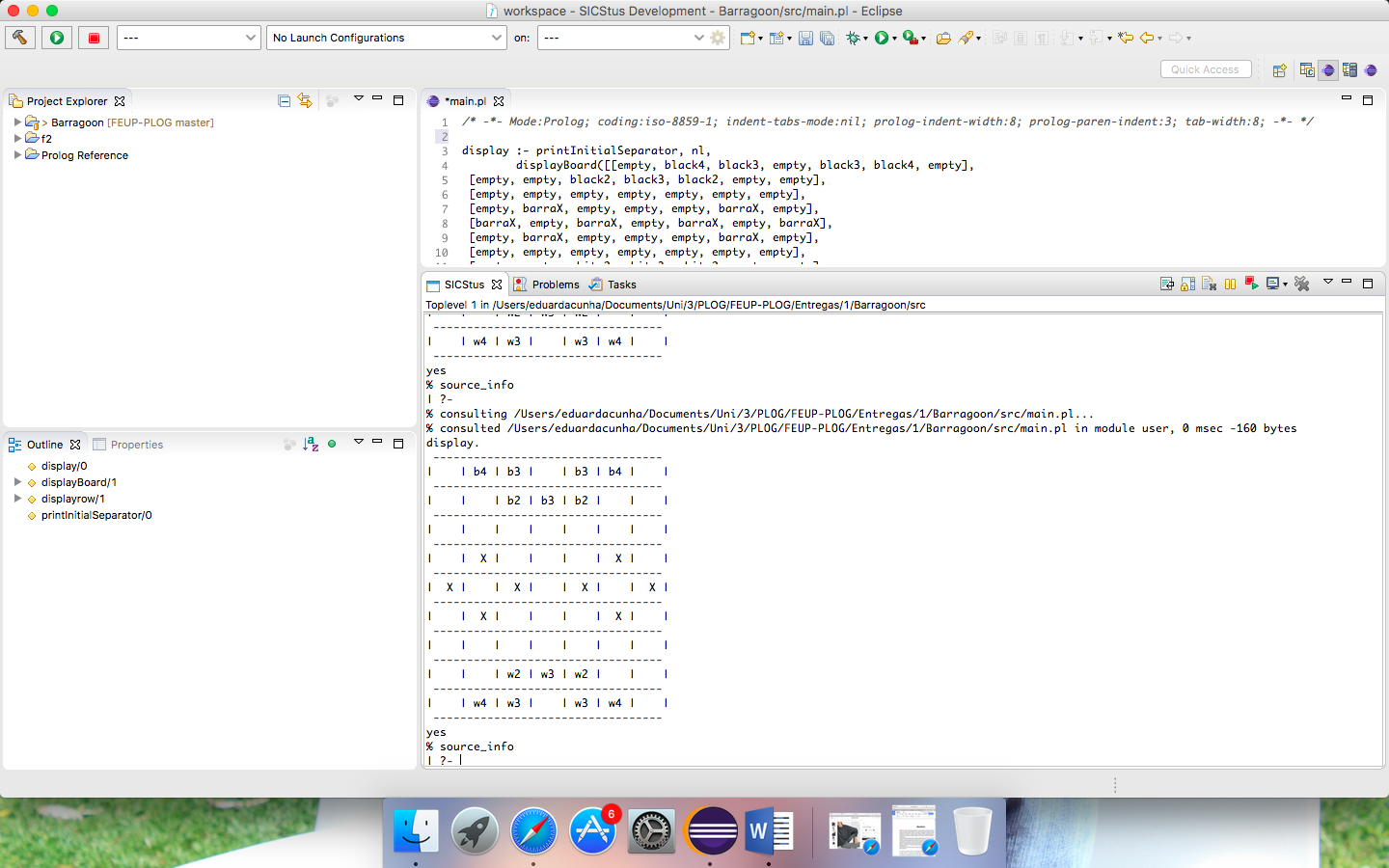
Ainda que só existam 32 barragoons e seria, por consequência, esperado que houvesse uma regra implementada para limitar a inserção de barragoons tendo em conta este valor, dado que sempre que um barragoon é comido é inserido o mesmo de volta e sempre que uma telha é comida são inseridos 2 e só há 7 telhas por jogador, concluímos que a situação em que se tenta inserir mais do que 32 nunca se vai verificar.

Por exemplo, tendo sido comidos 6 telhas de cada jogador, na situação final em que cada jogador possui 1 telha, quando um comer a do outro, o jogo acaba. E apenas nesse momento é que ocorreria a hipotética inserção do 33º.

# Lógica do Jogo

## Representação do Estado de Jogo

Por questões de simplificação, os barragoons encontram-se aqui representados apenas por, por exemplo, barraX, em vez de bg-'barraX'. Na prática e em todos os fragmentos de código, referimo-nos a cada barragoon por um par, cujo primeiro elemento é bg e o segundo é a face voltada para cima.

**Estado Inicial:**

[[empty, b-4, b-3, empty, b-3, b-4, empty],

[empty, empty, b-2, b-3, b-2, empty, empty],

[empty, empty, empty, empty, empty, empty, empty],

[empty, barraX, empty, empty, empty, barraX, empty],

[barraX, empty, barraX, empty, barraX, empty, barraX],

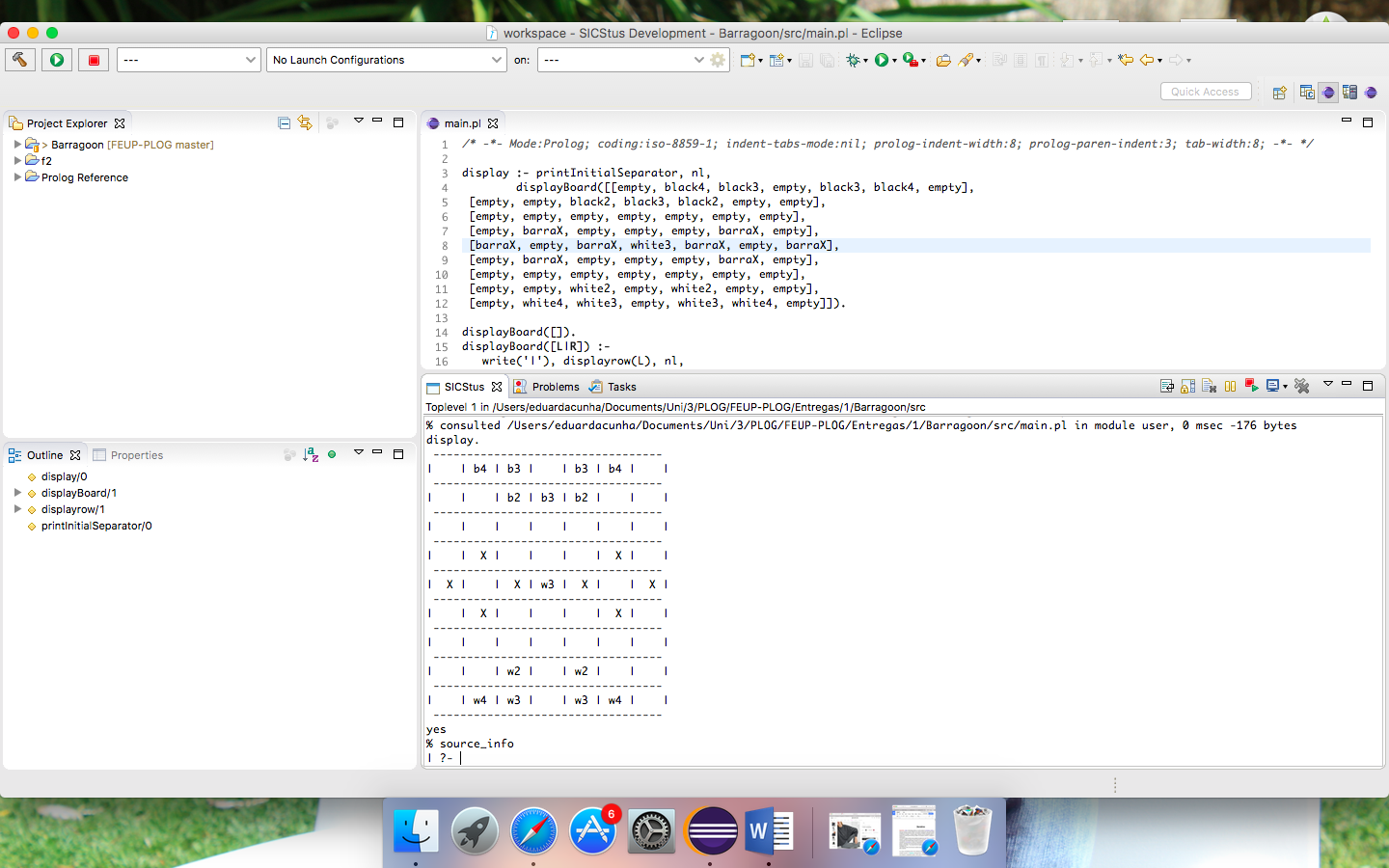
[empty, barraX, empty, empty, empty, barraX, empty],

[empty, empty, empty, empty, empty, empty, empty],

[empty, empty, w-2, w-3, w-2, empty, empty],

 [empty, w-4, w-3, empty, w-3, w-4, empty]]

**Estados intermédios:**

**1.**

[[empty, b-4, b-3, empty, b-3, b-4, empty],

[empty, empty, b-2, b-3, b-2, empty, empty],

[empty, empty, empty, empty, empty, empty, empty],

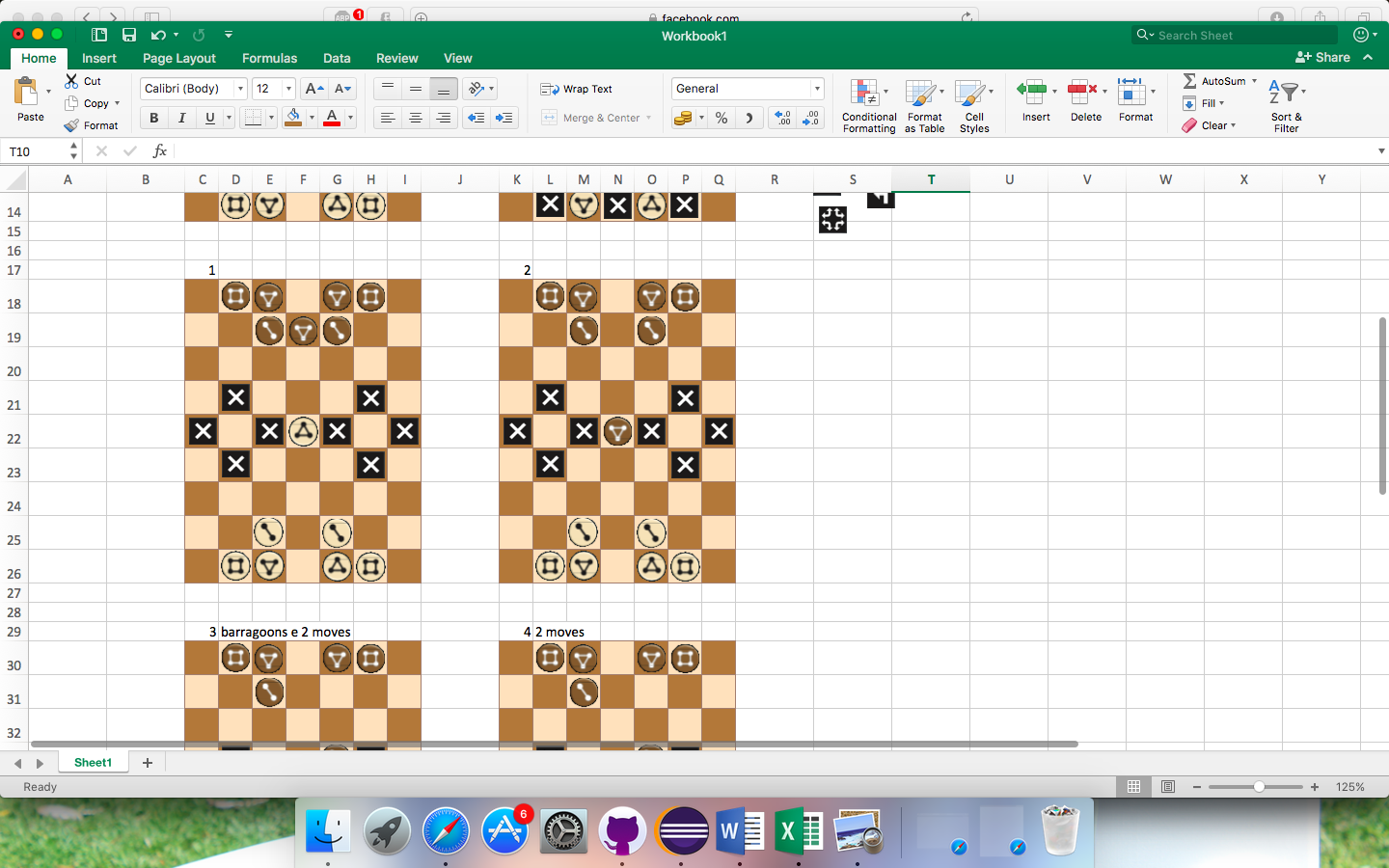
[empty, barraX, empty, empty, empty, barraX, empty],

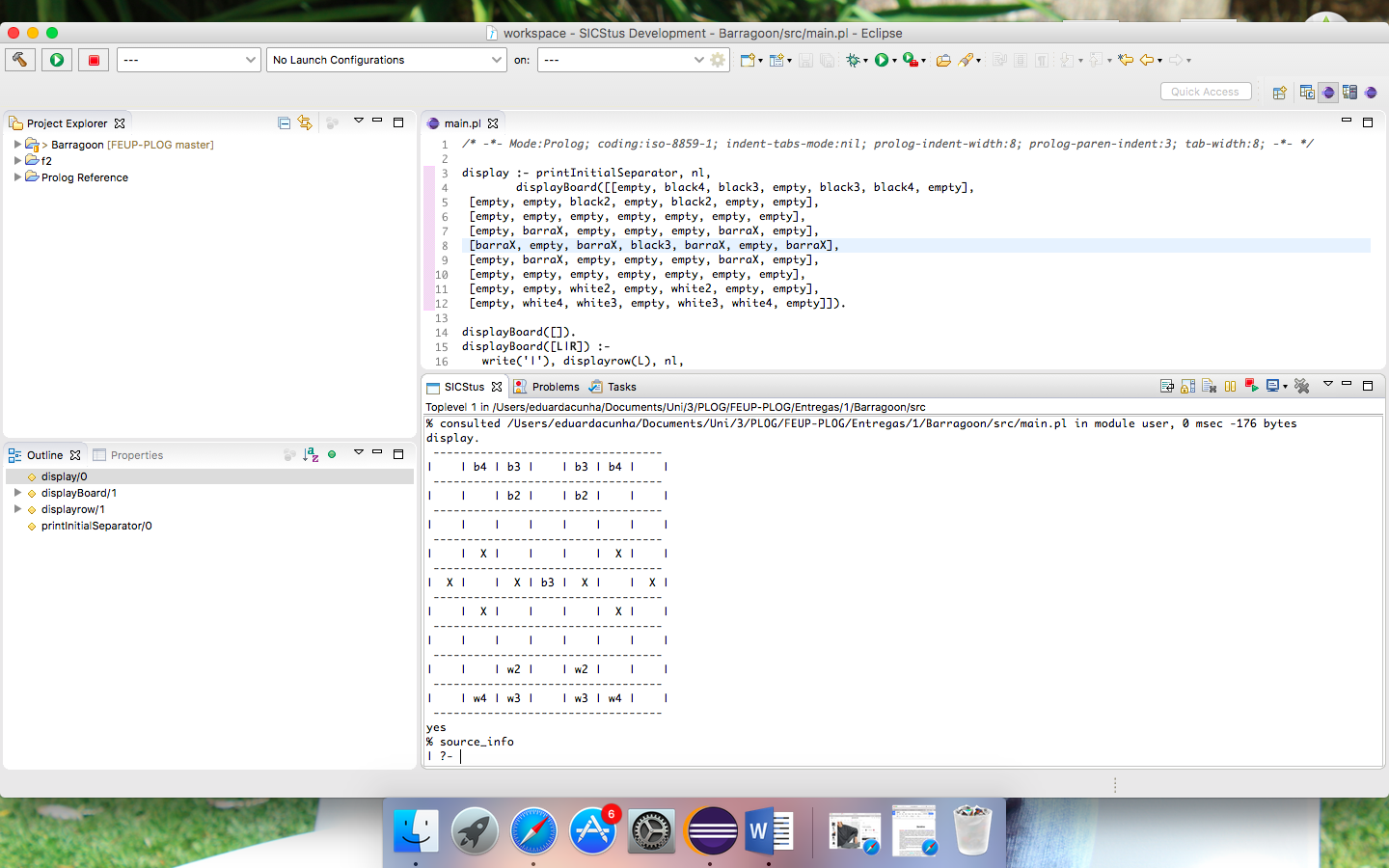
[barraX, empty, barraX, w-3, barraX, empty, barraX],

[empty, barraX, empty, empty, empty, barraX, empty],

[empty, empty, empty, empty, empty, empty, empty],

[empty, empty, w-2, empty, w-2, empty, empty],

 [empty, w-4, w-3, empty, w-3, w-4, empty]]

**2.**

[[empty, b-4, b-3, empty, b-3, b-4, empty],

[empty, empty, b-2, empty, b-2, empty, empty],

[empty, empty, empty, empty, empty, empty, empty],

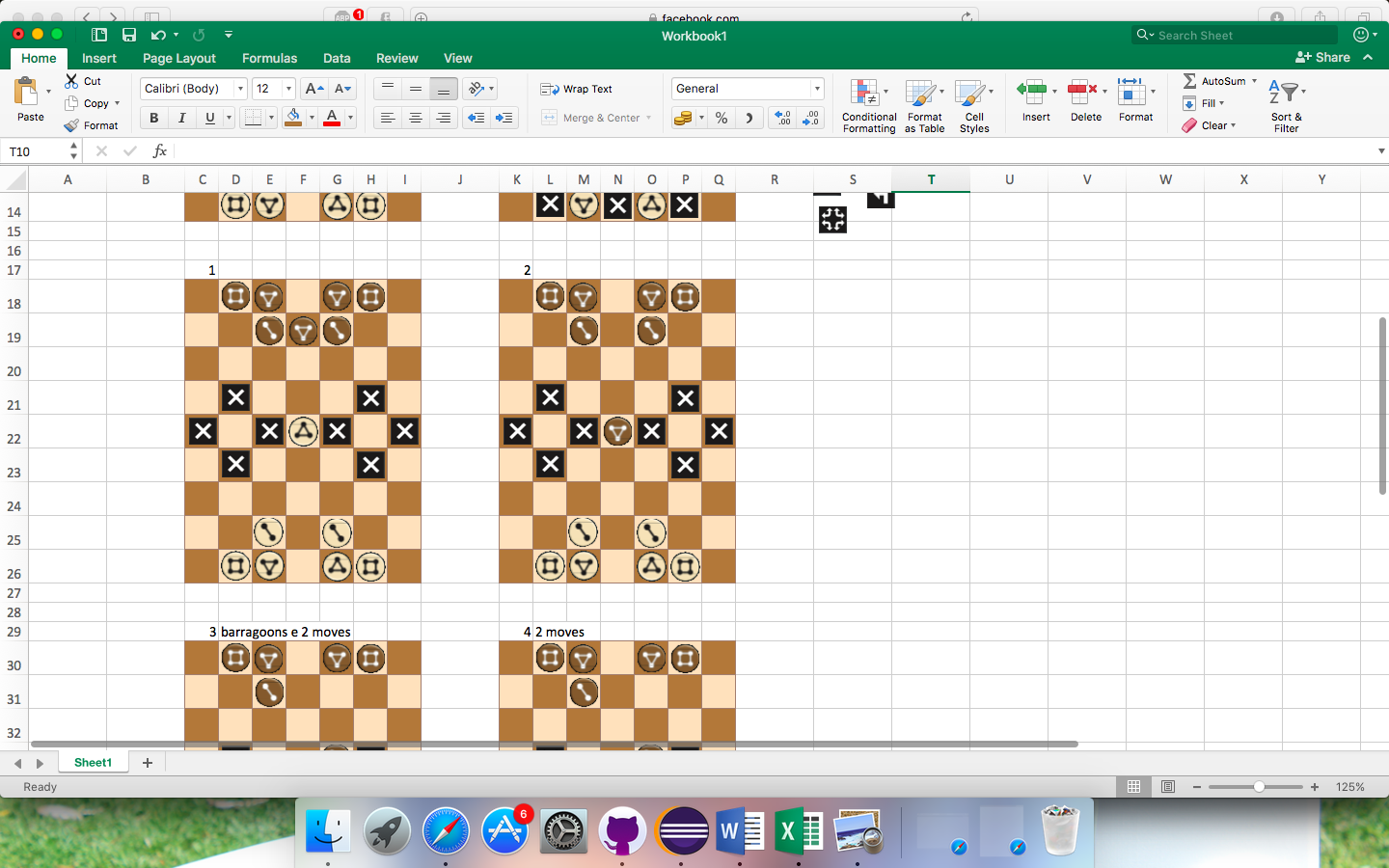
[empty, barraX, empty, empty, empty, barraX, empty],

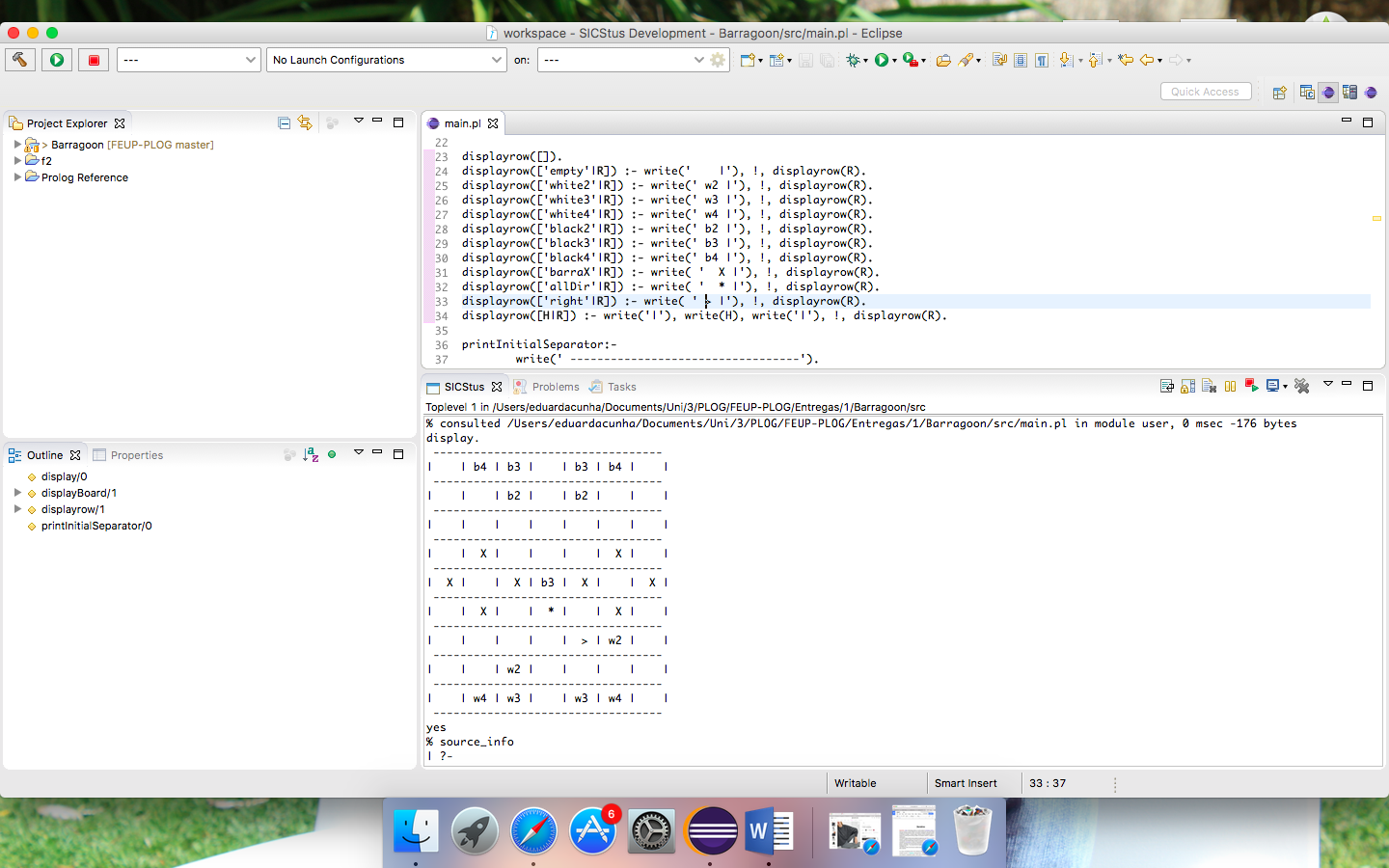
[barraX, empty, barraX, b-3, barraX, empty, barraX],

[empty, barraX, empty, empty, empty, barraX, empty],

[empty, empty, empty, empty, empty, empty, empty],

[empty, empty, w-2, empty, w-2, empty, empty],

**** [empty, w-4, w-3, empty, w-3, w-4, empty]]

**3.**

[[empty, b-4, b-3, empty, b-3, b-4, empty],

[empty, empty, b-2, empty, empty, empty, empty],

[empty, empty, empty, empty, empty, empty, empty],

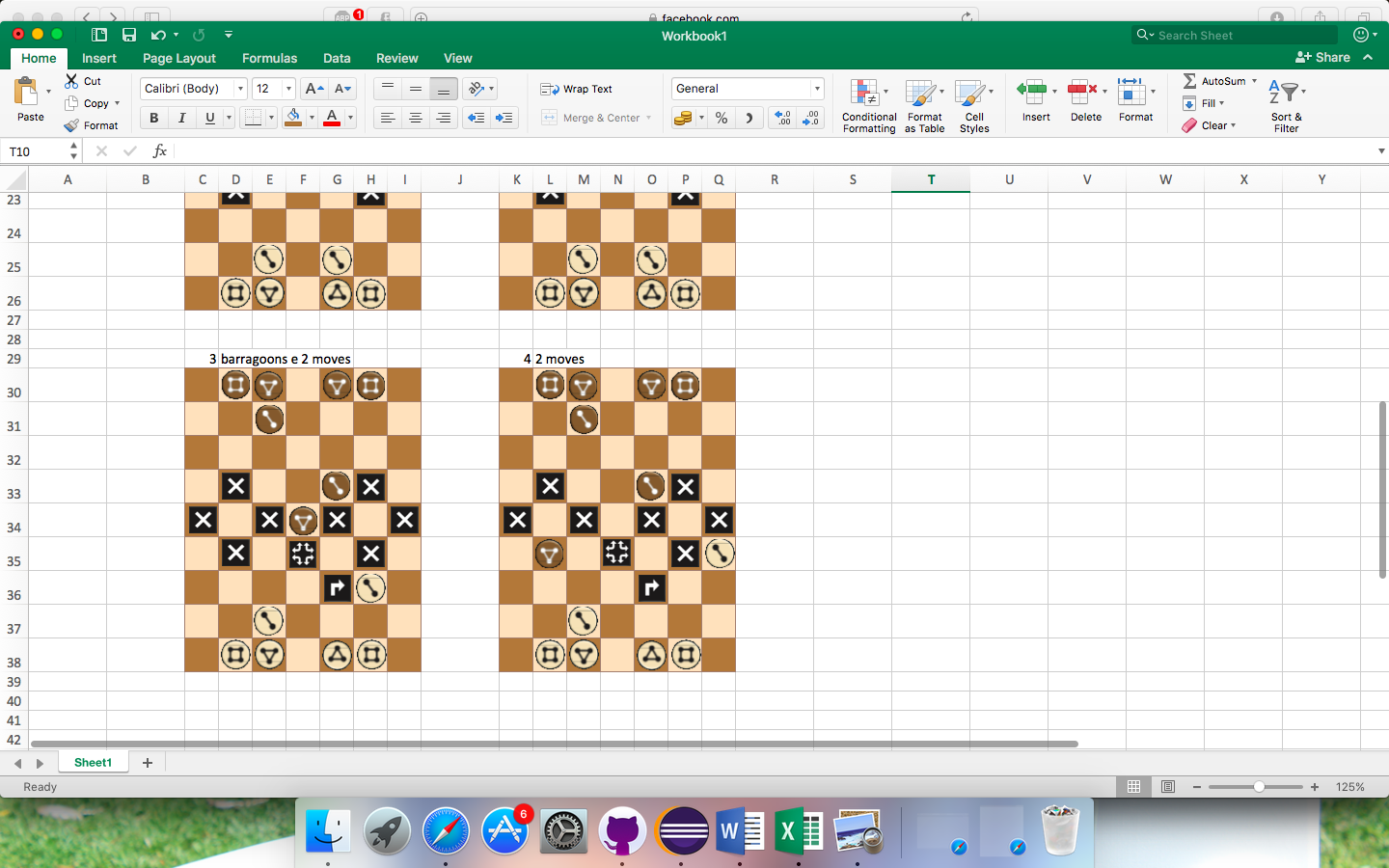
[empty, barraX, empty, empty, b-2, barraX, empty],

[barraX, empty, barraX, b-3, barraX, empty, barraX],

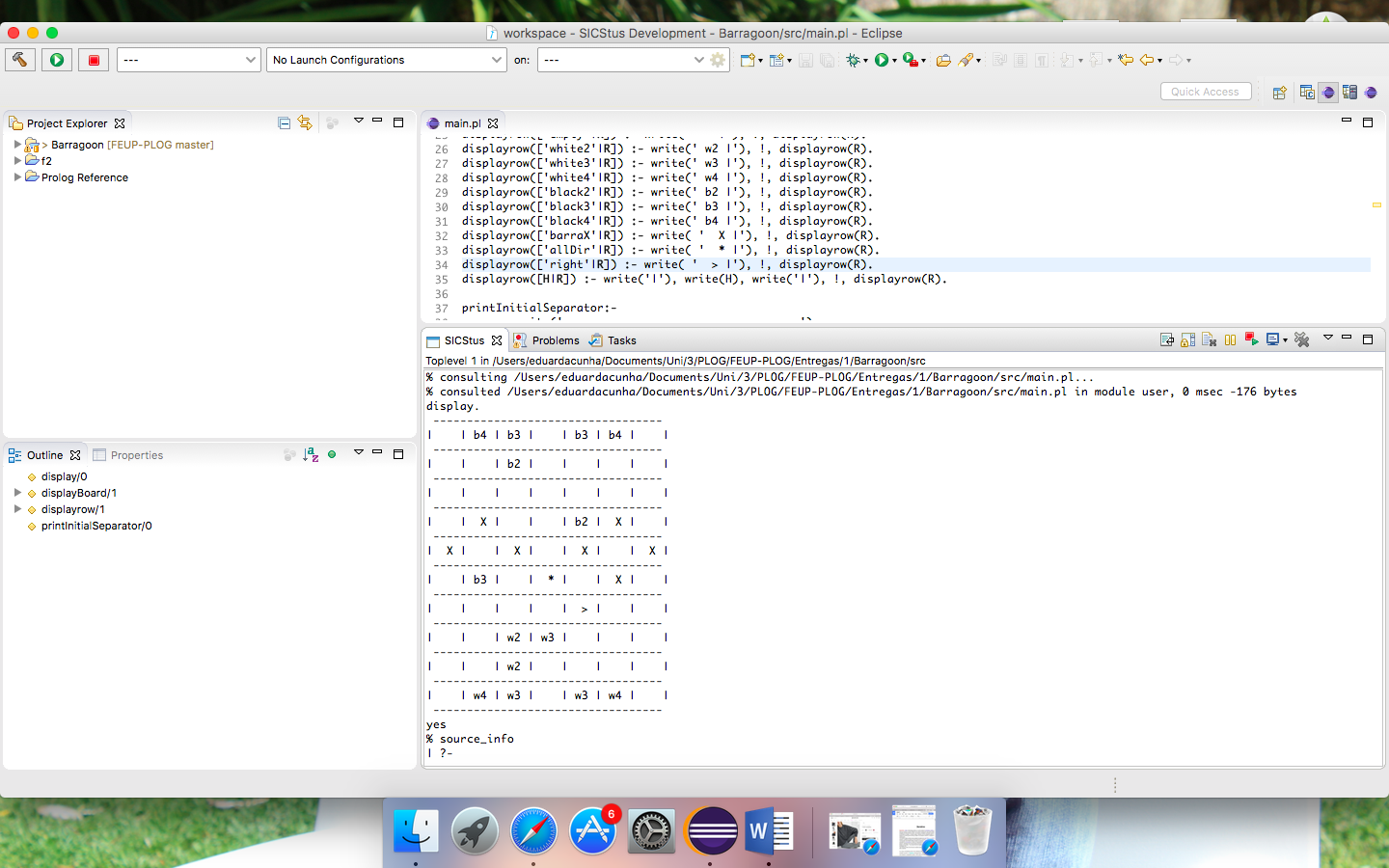
[empty, barraX, empty, allDir, empty, barraX, empty],

[empty, empty, empty, empty, right, w-2, empty],

[empty, empty, w-2, empty, empty, empty, empty],

 [empty, w-4, w-3, empty, w-3, w-4, empty]]

**4.**

****[[empty, b-4, b-3, empty, b-3, b-4, empty],

[empty, empty, b-2, empty, empty, empty, empty],

[empty, empty, empty, empty, empty, empty, empty],

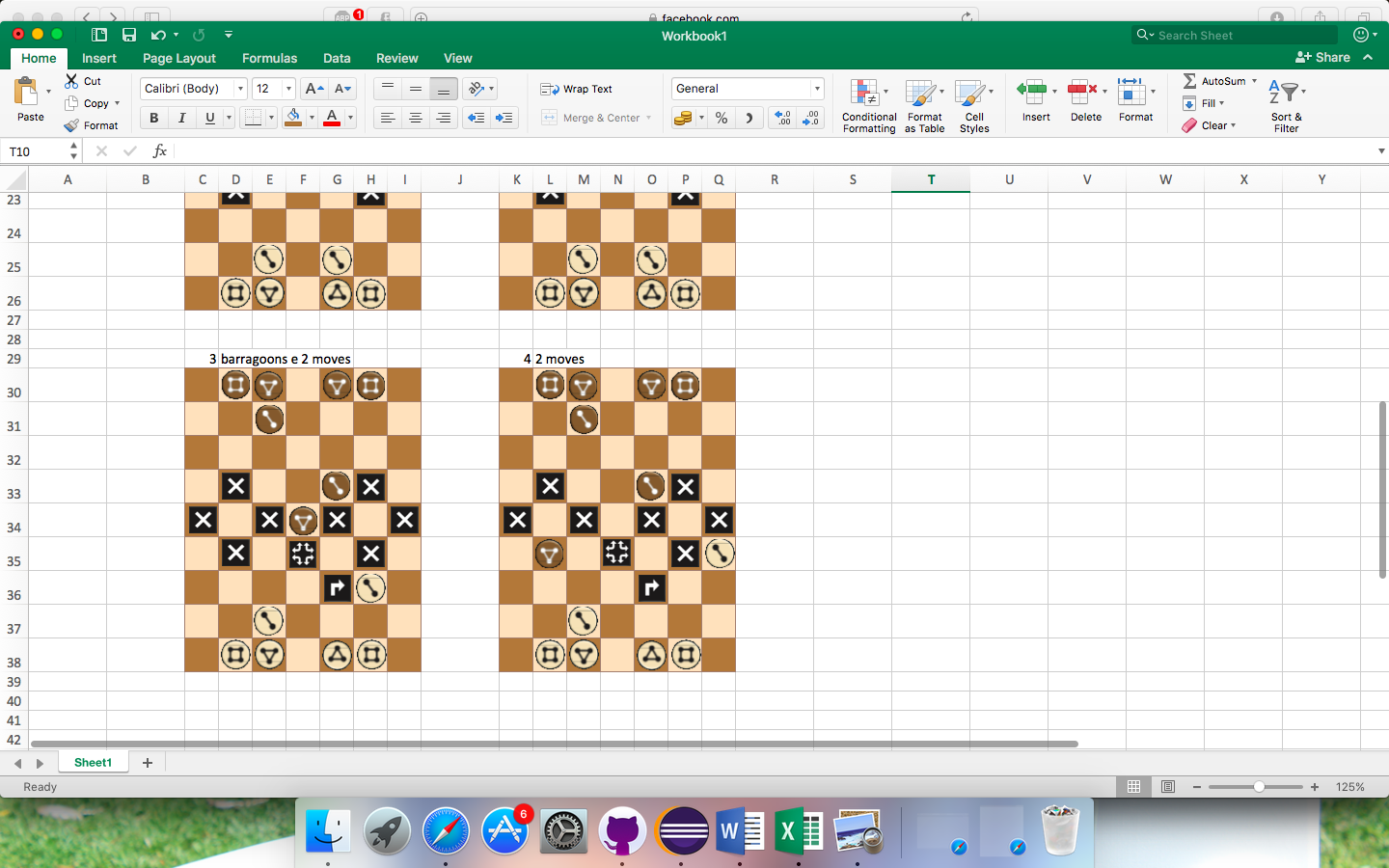
[empty, bg-'barraX', empty, empty, b-2, barraX, empty],

[barraX, empty, barraX, empty, barraX, empty, barraX],

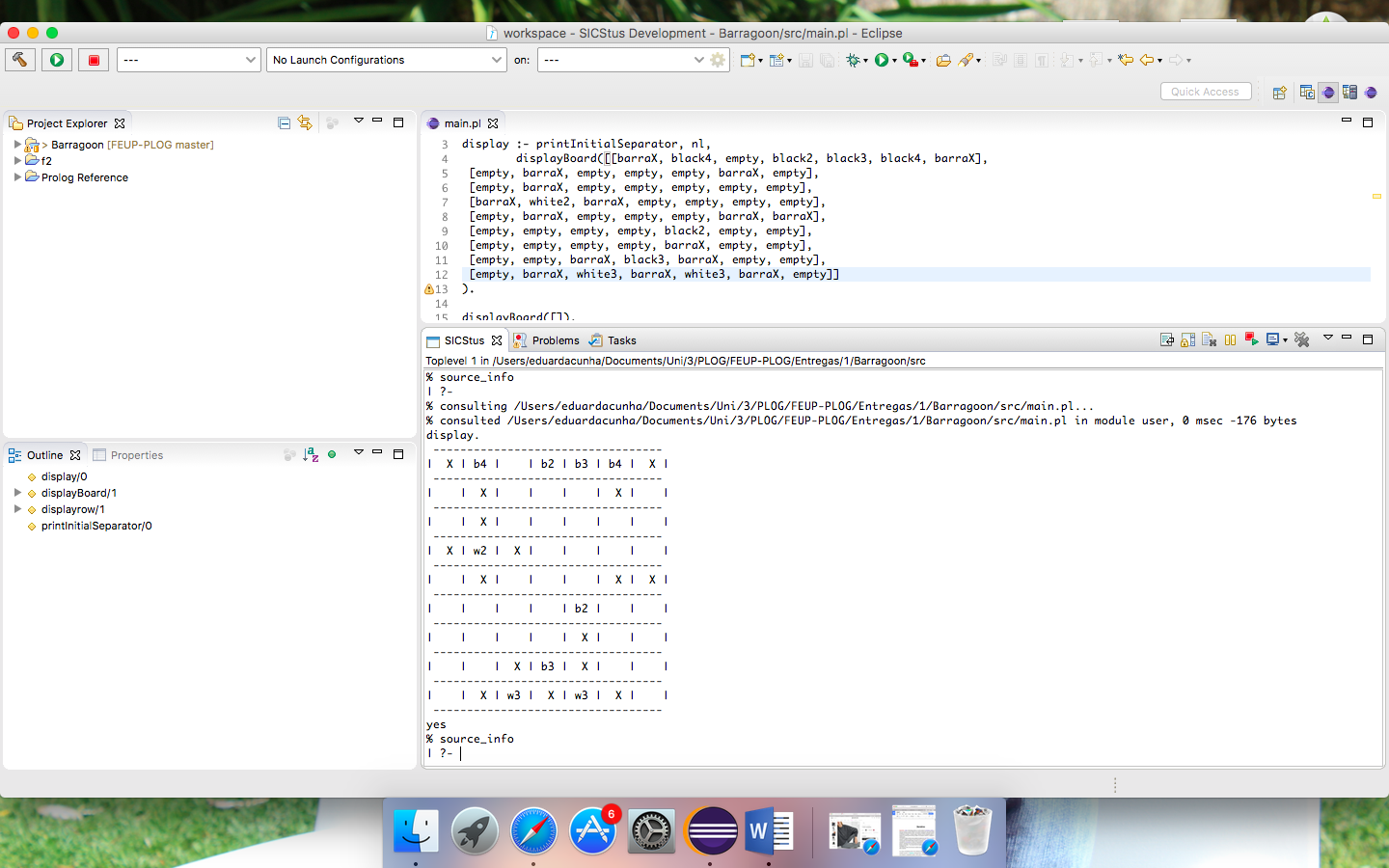
[empty, b-3, empty, allDir, empty, barraX, empty],

[empty, empty, empty, empty, right, empty, empty],

[empty, empty, w-2, w-3, empty, empty, empty],

 [empty, w-4, w-3, empty, w-3, w-4, empty]]

**Estado final:**

[[barraX, b-4, empty, b-2, b-3, b-4, barraX],

[empty, barraX, empty, empty, empty, barraX, empty],

[empty, barraX, empty, empty, empty, empty, empty],

[barraX, w-2, barraX, empty, empty, empty, empty],

[empty, barraX, empty, empty, empty, barraX, barraX],

[empty, empty, empty, empty, b-2, empty, empty],

[empty, empty, empty, empty, barraX, empty, empty],

[empty, empty, barraX, b-3, barraX, empty, empty],

 [empty, barraX, w-3, barraX, w-3, barraX, empty]]

## Visualização do Tabuleiro

1. initialBoard(
2. [[empty, b-*4*, b-*3*, empty, b-*3*, b-*4*, empty],
3. [empty, empty, b-*2*, b-*3*, b-*2*, empty, empty],
4. [empty, empty, empty, empty, empty, empty, empty],
5. [empty, bg-'barraX', empty, empty, empty, bg-'barraX', empty],
6. [bg-'barraX', empty, bg-'barraX', empty, bg-'barraX', empty, bg-'barraX'],
7. [empty, bg-'barraX', empty, empty, empty, bg-'barraX', empty],
8. [empty, empty, empty, empty, empty, empty, empty],
9. [empty, empty, w-*2*, w-*3*, w-*2*, empty, empty],
10. [empty, w-*4*, w-*3*, empty, w-*3*, w-*4*, empty]])**.**
11. displayGame(**Game**) **:-**
12. getBoard(Game, **Board**),
13. getCurrentPlayer(Game, **Player**),
14. clearScreen,
15. displayPlayerTurn(Player),
16. lettersAxis,nl,
17. horizontalBorder, nl,
18. numbersAxis(**RowNumbers**),
19. displayBoard(Board, RowNumbers), nl**.**
20. displayBoard([], [])**.**
21. displayBoard([**RowToDisplay**|**RemainingBoard**],
22. [**RowToDisplayNumber**|**RemainingRowNumbers**]) **:-**
23. translate([RowToDisplayNumber]),
24. translate(RowToDisplay),border, nl,
25. horizontalBorder, nl,
26. displayBoard(RemainingBoard, RemainingRowNumbers)**.**
27. *% -- Board Translation --*
28. translate([])**.**
29. translate(['empty'|**R**]) **:-** border, write(' '), !, translate(R)**.**
30. translate([w-*2*|**R**]) **:-** border, write(' w2 '), !, translate(R)**.**
31. translate([w-*3*|**R**]) **:-** border, write(' w3 '), !, translate(R)**.**
32. translate([w-*4*|**R**]) **:-** border, write(' w4 '), !, translate(R)**.**
33. translate([b-*2*|**R**]) **:-** border, write(' b2 '), !, translate(R)**.**
34. translate([b-*3*|**R**]) **:-** border, write(' b3 '), !, translate(R)**.**
35. translate([b-*4*|**R**]) **:-** border, write(' b4 '), !, translate(R)**.**
36. translate([bg-'barraX'|**R**]) **:-** border, write(' X '), !, translate(R)**.**
37. translate([bg-'allDir'|**R**]) **:-** border, write(' + '), !, translate(R)**.**
38. translate([bg-'oDirU'|**R**]) **:-** border, write(' V '), !, translate(R)**.**
39. translate([bg-'oDirD'|**R**]) **:-** border, write(' A '), !, translate(R)**.**
40. translate([bg-'oDirL'|**R**]) **:-** border, write(' <= '), !, translate(R)**.**
41. translate([bg-'oDirR'|**R**]) **:-** border, write(' => '), !, translate(R)**.**
42. translate([bg-'tDirH'|**R**]) **:-** border, write(' X '), !, translate(R)**.**
43. translate([bg-'tDirV'|**R**]) **:-** border, write(' X '), !, translate(R)**.**
44. translate([bg-'DtoR'|**R**]) **:-** border, write(' .> '), !, translate(R)**.**
45. translate([bg-'DtoL'|**R**]) **:-** border, write(' <. '), !, translate(R)**.**
46. translate([bg-'UtoR'|**R**]) **:-** border, write(' \'> '), !, translate(R)**.**
47. translate([bg-'UtoL'|**R**]) **:-** border, write(' <\' '), !, translate(R)
48. translate([bg-'LtoU'|**R**]) **:-** border, write(' -^ '), !, translate(R)**.**
49. translate([bg-'LtoD'|**R**]) **:-** border, write(' -v '), !, translate(R)**.**
50. translate([bg-'RtoU'|**R**]) **:-** border, write(' ^- '), !, translate(R)**.**
51. translate([bg-'RtoD'|**R**]) **:-** border, write(' v- '), !, translate(R)**.**
52. translate(['um'|**R**]) **:-** write('1'), !, translate(R)**.**
53. translate(['dois'|**R**]) **:-** write('2'), !, translate(R)**.**
54. translate(['tres'|**R**]) **:-** write('3'), !, translate(R)**.**
55. translate(['quatro'|**R**]) **:-** write('4'), !, translate(R)**.**
56. translate(['cinco'|**R**]) **:-** write('5'), !, translate(R)**.**
57. translate(['seis'|**R**]) **:-** write('6'), !, translate(R)**.**
58. translate(['sete'|**R**]) **:-** write('7'), !, translate(R)**.**
59. translate(['oito'|**R**]) **:-** write('8'), !, translate(R)**.**
60. translate(['nove'|**R**]) **:-** write('9'), !, translate(R)**.**
61. *% -- Board Axis –*
62. numbersAxis([um, dois, tres, quatro, cinco, seis, sete, oito, nove])**.**
63. lettersAxis **:-** write(' A B C D E F G')**.**
64. *% -- Board Borders –*
65. horizontalBorder **:-** write(' ----------------------------------')**.**
66. border **:-** write('|')**.**

## Lista de Jogadas Válidas

O predicado **getMovesAvailable(+Game, +Row, +Column, -List)** retorna uma lista com as várias opções de percurso que uma peça pode percorrer, já validadas.

Esta função processa-se através da obtenção da telha a ser jogada com **getCell(+Game, +Row, +Column, -List)**. Assim que sabemos o tipo de telha (2, 3 ou 4 círculos), obtemos em **availableMoves(+NDots, -allMovesAvailable)** todos os percursos existentes para uma telha dessas. Finalmente, com **getMovesAvailableAux(+Game, +Row, +Column, +AllMovesAvailable, -List)** retornamos a lista final com os percursos após serem validados por **validatePath(+Row, +Column, +Path, +false)** e **validateMove(+Game, +Row, +Column, +Path, \_, +false)**.

Em **validatePath** é verificado se o movimento não causa o ultrapassar dos limites do tabuleiro e se apenas se dá uma mudança de direção.

Em **validateMove** verifica-se se o caminho corresponde a um tipo de movimento full ou short e se as telhas estão a deslocar-se corretamente por cima dos barragoons dispostos no tabuleiro.

Este cálculo automático da lista de jogadas válidas só tem relevância aquando das jogadas do computador.

## Execução de Jogadas

O ciclo principal **playGame(+Game, -NewGame)** é caracterizado pela execução da jogada **playerTurn(+Game, -UpdatedGame)**, responsável por efetuar todos os pontos relativos à jogada de um jogador, **switchPlayer(+UpdatedGame, -NextPlayerGame)**, que trata de alternar os jogadores a cada jogada, e a chamada recursiva de **playGame**.

O predicado **playerTurn** é constituído por **playerMove(+Game, +RolSrc, +ColSrc, -Path)** ou **botMove(+Game, -Row, -Column, -Path)** dependendo do modo jogo que se trata, **validateMove(+Game, +RowSrc, +ColSrc, +Path, -PieceCaptured)**, **movePiece(+Game, +RowSrc, +ColSrc, +Path, -NewGame1)** e uma verificação de se alguma peça, telha ou barragoon, foi capturada.

Em **playerMove**, é pedido ao jogador que escolha a telha que pretende mover, através da inserção na consola da linha, representada por um número de 1 a 9, e da coluna, representada por uma letra de A a G (**chooseTile**), é validado que essa telha existe e lhe pertence (**validateTile**), pede-se o percurso que a peça vai efetuar, representado pelas teclas WASD (**choosePath**) e, por fim, verifica-se se esse caminho não ultrapassa os limites do tabuleiro (**validatePath**).

O predicado **botMove**, que existe em paralelo com **playerMove**, ao invés de receber do jogador as coordenadas e caminho, gera-os aleatoriamente (**getRandomElemFromList**) a partir de listas de posições (**getPlayerPieces)** e percursos (**getMovesAvailable**)que já foram selecionados por serem válidos. Isto torna a função seguinte, **validateMove**, algo desnecessária para esta jogada. Com algum tempo, podia haver alguma reestruturação de código de forma a eliminar esta redundância.

Em **validateMove**, trata-se de todas as questões relacionadas com o movimento da peça respeitar as regras do jogo. Verificando através de **getCell(+Board, +RowSrc, +ColSrc, -Piece)** qual é a telha que está a ser movida, com recurso a **isShortMove(+Piece, +Path)** e **isFullMove(+Piece, +Path)** determinamos se o percurso inserido pelo jogador é válido. Ainda, em **validateCrossMovements(+Game, +RowSrc, +ColSrc, +Path, +IsLongMove, +Piece, -PieceCaptured)** calcula-se se o movimento em questão obedece às regras de passagem em cima de barragoons e se não passa por cima de telhas se não for com o objetivo de as comer.

Em **movePiece**, é calculada a célula final em que a telha deve acabar através de **getDestCellFromPath(+RowSrc, +ColSrc, +Path, -RowDest, -ColDest)**, elimina-se o conteúdo dessa posição e procede-se à colocação da telha no novo destino com **moveFromSrcToDest(+Game, +RowSrc, +ColSrc, +RowDest, +ColDest, -NewGame)**.

## Avaliação do Tabuleiro

No primeiro nível de dificuldade, não há qualquer tipo de avaliação das jogadas favoráveis. O computador joga sempre aleatoriamente.

No segundo nível de dificuldade, as jogadas do computador são determinadas de acordo com um sistema de prioridades, sendo que, quando são calculados os movimentos possíveis para a telha selecionada, se resultarem em telhas comidas, são colocados na cabeça da lista. Caso contrário, vão para o fim da lista.

## Final do Jogo

O jogo termina quando um dos jogadores já não tem telhas ou as que possui não se conseguem deslocar.

De forma a lidar com o primeiro ponto, utilizamos o predicado **countPlayerPieces(+Board, +CurrentPlayer, -CountPieces)** que verifica no tabuleiro o número de telhas do jogador cuja vez for de jogar. Se **CountPieces** retornar a 0, o jogo acaba e esse jogador perde.

Para verificar se, existindo peças, estas possuem movimentos possíveis, recorremos a **countMovesAvailable(+Game, +Row, +Column, -Count)** para contar os movimentos de uma peça. Se **Count** retornar a 0 para todas as peças em tabuleiro do jogador, o jogo acaba e esse jogador perde. Esta solução é um pouco *hardcoded*, pois recorremos a todos os percursos possíveis para cada telha para efetuar esta verificação.

## Jogada do Computador

No primeiro nível de jogo, tanto no modo humano contra computador como computador contra computador, as jogadas são completamente aleatórias. A forma como a escolha de telha a mover e percurso a efetuar se processa está extensivamente descrita nas secções Lista de Jogadas Válidas e Execução de Jogadas.

No segundo nível de jogo, de acordo com diferentes consequências possíveis de um movimento, comer uma telha ou não, define-se uma lista com prioridades. Quando o caminho resulta em comer uma telha, passa para a cabeça da lista, caso contrário, vai para o fim. Em vez de escolher um percurso aleatoriamente da lista, escolhe sempre o primeiro, pois é garantido que resulta numa telha comida.

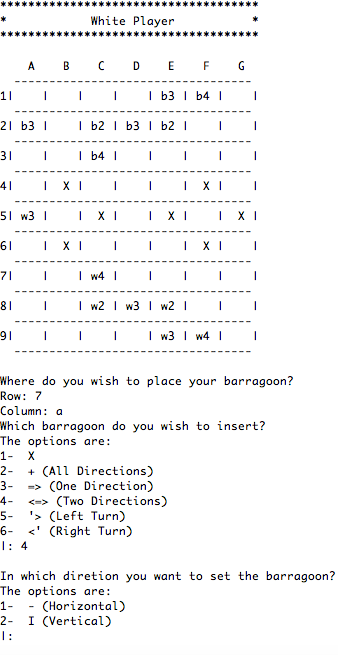
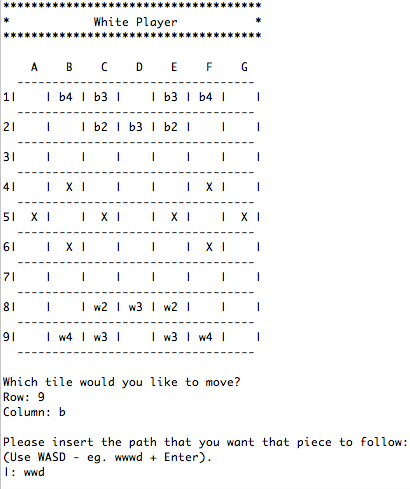
# Interface com o Utilizador

Quando iniciado com o comando *start*, é apresentado o menu principal com as opções de jogar em modo humano contra humano, humano contra computador e computador contra computador, apresentar as regras ou sair do jogo.

Se for escolhida a opção 4 - Regras, é apresentado o seguinte menu que enuncia as regras relevantes para o jogador.

Iniciado o jogo em modo humano contra humano, as jogadas apresentam-se com um cabeçalho que indica a vez do jogador – White Player ou Black Player, o tabuleiro de jogo com eixos de números para as linhas e letras para as colunas e, por fim, perguntas relativas ao *input* do utilizador, como coordenadas da peça a mover e caminho que a peça deve seguir. Caso alguma destas informações seja inserida de forma errada, é sempre apresentada uma mensagem de erro e é pedido de novo ao jogador que insira a informação em questão até que a insira corretamente.

Se for o caso de inserir um barragoon no tabuleiro, é pedido ao respetivo jogador que insira a informação relativa a essa colocação (coordenadas de destino do barragoon e a face que se pretende que fique voltada para cima).



# Conclusões

Concluído o projeto, achamos importante ressaltar aquilo que poderia ter sido melhorado e não foi apenas pela ausência de tempo. Destaca-se a existência de soluções ditas *hardcoded* (por exemplo, o cálculo do fim do jogo através da verificação de todos os paths possíveis) e a simplicidade do segundo nível de dificuldade.

Pretendíamos ter criado uma lista paralela à lista de percursos válidos do computador com valores do tipo 0, 1 ou 2 caso resultassem em não comer nenhuma peça, comer um barragoon ou comer uma telha, respetivamente. Acabamos por fazer apenas uma lista organizada com percursos que resultem em telha comida a serem enviados para a cabeça e o resto para o fim da lista.

Ainda, pensamos que a nossa implementação é bastante intuitiva e de fácil compreensão, mas a sua complexidade poderia ter sido sensivelmente reduzida com alguma reutilização de código e *layering*.

Concluindo, consideramos que realizamos com sucesso aquilo a que nos propusemos e com o tempo que nos foi dado. Os conceitos das aulas foram, sem dúvida, consolidados e temos agora uma nova compreensão da utilidade desta linguagem de programação.

# Bibliografia

1. https://www.youtube.com/watch?v=qG1i0\_sn\_FI
2. https://boardgamegeek.com/boardgame/157779/barragooN
3. http://www.barragoon.de/bsp/BARRAGOON\_en.pdf
4. https://stackoverflow.com
5. http://www.swi-prolog.org

# Anexos

main.pl

1. **:-** include('Utilities.pl')**.**
2. **:-** include('Interface.pl')**.**
3. **:-** include('Logic.pl')**.**
4. **:-** use\_module(library(system))**.**
5. *%--------------------------------%*
6. *%-----------Barragoon -----------%*
7. *%--------------------------------%*
8. *%--------- escreva start --------%*
9. *%---- na consola para correr ----%*
10. *%--------------------------------%*
11. *%--------------------------------%*
12. *% --- START ---*
13. start **:-**
14. clearScreen,
15. mainMenu**.**

Interface.pl

1. *%-------------------------------%*
2. *%----------Interface------------%*
3. *%-------------------------------%*
4. *% -------------------------------------------------------------------------*
5. *% ----------------------------------- MENUS -------------------------------*
6. *% -------------------------------------------------------------------------*
7. *% -- Menus --*
8. mainMenu **:-**
9. displayMainMenu,
10. getCharThenEnter(**Option**),
11. (
12. Option = '1' -> **startGamePvP**;
13. Option = '2';
14. Option = '3';
15. Option = '4' -> displayRules;
16. Option = '5';
17. clearScreen,
18. write('ERROR : invalid input...'), spacing(*1*),
19. mainMenu
20. )**.**
21. displayMainMenu **:-**
22. upperFrame,
23. titleFrame,
24. write('| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |'),nl,
25. write('| \* Main Menu \* |'),nl,
26. write('| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |'),nl,
27. write('| Play: |'),nl,
28. write('| 1 - Player vs Player |'),nl,
29. write('| 2 - Player vs Computer |'),nl,
30. write('| 3 - Computer vs Computer Easy |'),nl,
31. write('| 4 - Player vs Computer Hard |'),nl,
32. write('| |'),nl,
33. write('| 5 - Rules |'),nl,
34. write('| 6 - Quit |'),nl,
35. lowerFrame**.**
36. displayRules **:-**
37. clearScreen,
38. upperFrame,
39. titleFrame,
40. write('| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |'),nl,
41. write('| \* Game Rules \* |'),nl,
42. write('| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |'),nl,
43. write('| 1. A piece is captured if the oponent\'s piece finishes |'),nl,
44. write('| in the same cell. |'),nl,
45. write('| 2. Captures are only allowed during a full move. |'),nl,
46. write('| 3. Tiles with 2 circles can\'t capture barragoons with |'),nl,
47. write('| the "all directions" symbol turned up. |'),nl,
48. write('| 4. If a barragoon is captured, it must be placed back |'),nl,
49. write('| in the board, in a free place, with whichever symbol |'),nl,
50. write('| the player preffers. |'),nl,
51. write('| 5. When a tile is captured, 2 barragoons are added to |'),nl,
52. write('| the board, 1 per player and the first to place it is |'),nl,
53. write('| the one whose tile was captured. |'),nl,
54. write('| 6. Once a barragoon is placed, it can\'t be moved. |'),nl,
55. write('| 7. During a move, there can only be one change of |'),nl,
56. write('| direction. |'),nl,
57. write('| 8. Movements can only be vertical or horizontal, never |'),nl,
58. write('| diagonal. |'),nl,
59. write('| 9. During a move, there can only be one 90º turn. |'),nl,
60. lowerFrame,
61. spacing(*2*),
62. next, mainMenu, !**.**
63. displayPlayerTurn(**Player**) **:-**
64. ifelse(
65. Player = w ,
66. (
67. write('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'),nl,
68. write('\* White Player \*'),nl,
69. write('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'),nl,nl
70. ),
71. (
72. write('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'),nl,
73. write('\* Black Player \*'),nl,
74. write('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'),nl,nl
75. ))**.**
76. *% -- Logo --*
77. titleFrame **:-**
78. write('| \_\_\_\_ |'), nl,
79. write('| | \_ \\ |'), nl,
80. write('| | |\_) | \_\_ \_ \_ \_\_ \_ \_\_ \_\_ \_ \_\_ \_ \_\_\_ \_\_\_ \_ \_\_ |'), nl,
81. write('| | \_ < / \_` | \'\_\_| \'\_\_/ \_` |/ \_` |/ \_ \\ / \_ \\| \'\_ \\ |'), nl,
82. write('| | |\_) | (\_| | | | | | (\_| | (\_| | (\_) | (\_) | | | | |'), nl,
83. write('| |\_\_\_\_/ \\\_\_,\_|\_| |\_| \\\_\_,\_|\\\_\_, |\\\_\_\_/ \\\_\_\_/|\_| |\_| |'), nl,
84. write('| \_\_/ | |'), nl,
85. write('| |\_\_\_/ |'), nl,
86. write('| |'),nl**.**
87. *% -- Frames --*
88. upperFrame **:-**
89. write(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_'),nl,
90. write('| |'),nl**.**
91. lowerFrame **:-**
92. write('| |'),nl,
93. write('|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|'),nl**.**
94. *% -------------------------------------------------------------------------*
95. *% ----------------------------------- BOARD -------------------------------*
96. *% -------------------------------------------------------------------------*
97. *% -- BOARD --*
98. initialBoard( [[empty, b-*4*, b-*3*, empty, b-*3*, b-*4*, empty],
99. [empty, empty, b-*2*, b-*3*, b-*2*, empty, empty],
100. [empty, empty, empty, empty, empty, empty, empty],
101. [empty, bg-barraX, empty, empty, empty, bg-barraX, empty],
102. [bg-barraX, empty, bg-barraX, empty, bg-barraX, empty, bg-barraX],
103. [empty, bg-barraX, empty, empty, empty, bg-barraX, empty],
104. [empty, empty, empty, empty, empty, empty, empty],
105. [empty, empty, w-*2*, w-*3*, w-*2*, empty, empty],
106. [empty, w-*4*, w-*3*, empty, w-*3*, w-*4*, empty]])**.**
107. displayGame(**Game**) **:-**
108. getBoard(Game, **Board**),
109. getCurrentPlayer(Game, **Player**),
110. clearScreen,
111. displayPlayerTurn(Player),
112. lettersAxis,nl,
113. horizontalBorder, nl,
114. numbersAxis(**RowNumbers**),
115. displayBoard(Board, RowNumbers), nl**.**
116. displayBoard([], [])**.**
117. displayBoard([**RowToDisplay**|**RemainingBoard**],
118. [**RowToDisplayNumber**|**RemainingRowNumbers**]) **:-**
119. translate([RowToDisplayNumber]),
120. translate(RowToDisplay),border, nl,
121. horizontalBorder, nl,
122. displayBoard(RemainingBoard, RemainingRowNumbers)**.**
123. *% -- Board Translation --*
124. translate([])**.**
125. translate(['empty'|**R**]) **:-** border, write(' '), !, translate(R)**.**
126. translate([w-*2*|**R**]) **:-** border, write(' w2 '), !, translate(R)**.**
127. translate([w-*3*|**R**]) **:-** border, write(' w3 '), !, translate(R)**.**
128. translate([w-*4*|**R**]) **:-** border, write(' w4 '), !, translate(R)**.**
129. translate([b-*2*|**R**]) **:-** border, write(' b2 '), !, translate(R)**.**
130. translate([b-*3*|**R**]) **:-** border, write(' b3 '), !, translate(R)**.**
131. translate([b-*4*|**R**]) **:-** border, write(' b4 '), !, translate(R)**.**
132. translate([bg-'barraX'|**R**]) **:-** border, write(' X '), !, translate(R)**.**
133. translate([bg-'allDir'|**R**]) **:-** border, write(' + '), !, translate(R)**.**
134. translate([bg-'oDirU'|**R**]) **:-** border, write(' A '), !, translate(R)**.**
135. translate([bg-'oDirD'|**R**]) **:-** border, write(' V '), !, translate(R)**.**
136. translate([bg-'oDirL'|**R**]) **:-** border, write(' <= '), !, translate(R)**.**
137. translate([bg-'oDirR'|**R**]) **:-** border, write(' => '), !, translate(R)**.**
138. translate([bg-'tDirH'|**R**]) **:-** border, write(' - '), !, translate(R)**.**
139. translate([bg-'tDirV'|**R**]) **:-** border, write(' I '), !, translate(R)**.**
140. translate([bg-'DtoR'|**R**]) **:-** border, write(' .> '), !, translate(R)**.**
141. translate([bg-'DtoL'|**R**]) **:-** border, write(' <. '), !, translate(R)**.**
142. translate([bg-'UtoR'|**R**]) **:-** border, write(' \'> '), !, translate(R)**.**
143. translate([bg-'UtoL'|**R**]) **:-** border, write(' <\' '), !, translate(R)**.**
144. translate([bg-'LtoU'|**R**]) **:-** border, write(' -^ '), !, translate(R)**.**
145. translate([bg-'LtoD'|**R**]) **:-** border, write(' -v '), !, translate(R)**.**
146. translate([bg-'RtoU'|**R**]) **:-** border, write(' ^- '), !, translate(R)**.**
147. translate([bg-'RtoD'|**R**]) **:-** border, write(' v- '), !, translate(R)**.**
148. translate(['1'|**R**]) **:-** write('1'), !, translate(R)**.**
149. translate(['2'|**R**]) **:-** write('2'), !, translate(R)**.**
150. translate(['3'|**R**]) **:-** write('3'), !, translate(R)**.**
151. translate(['4'|**R**]) **:-** write('4'), !, translate(R)**.**
152. translate(['5'|**R**]) **:-** write('5'), !, translate(R)**.**
153. translate(['6'|**R**]) **:-** write('6'), !, translate(R)**.**
154. translate(['7'|**R**]) **:-** write('7'), !, translate(R)**.**
155. translate(['8'|**R**]) **:-** write('8'), !, translate(R)**.**
156. translate(['9'|**R**]) **:-** write('9'), !, translate(R)**.**
157. *% -- Board Axis --*
158. numbersAxis(['1', '2', '3', '4', '5', '6', '7', '8', '9'])**.**
159. lettersAxis **:-** write(' A B C D E F G')**.**
160. *% -- Board Borders --*
161. horizontalBorder **:-** write(' ----------------------------------')**.**
162. border **:-** write('|')**.**

Logic.pl

1. *%-------------------------------%*
2. *%-----------Game Logic----------%*
3. *%-------------------------------%*
4. *% -------------------------------------------------------------------------*
5. *% ----------------------------------- GAME -------------------------------*
6. *% -------------------------------------------------------------------------*
7. startGame(**Mode**)**:-**
8. initializeGame(**Game**, Mode),
9. playGame(Game)**.**
10. initializeGame(**Game**, **Mode**)**:-**
11. initialBoard(**Board**),
12. Game = [Board, w, Mode], !**.**
13. *% --- Game Loop ---*
14. playGame(**Game**) **:-**
15. ifelse(isGameOver(Game),
    1. (
       1. switchPlayer(Game, **EndGame**),
       2. gameOver(EndGame)
    2. ),
    3. (
       1. playerTurn(Game, **UpdatedGame**),
       2. switchPlayer(UpdatedGame, **NextPlayerGame**),
       3. playGame(NextPlayerGame)
    4. ))**.**
16. isGameOver(**Game**) **:-**
17. getBoard(Game, **Board**),
18. getCurrentPlayer(Game, **CurrentPlayer**),
19. *%check is current player still has any piece*
20. countPlayerPieces(Board, CurrentPlayer, **CountPieces**),
21. !,
22. ifelse(CountPieces = *0*,
    1. true,
    2. (
       1. countMovesAvailableAllPieces(Game, **CountMoves**),
       2. !,
       3. ifelse(CountMoves = *0*,
       4. true,
       5. fail
       6. )
    3. )
23. )**.**
24. gameOver(**Game**) **:-**
25. nl, write('And the Winner is...'), nl, nl,
26. displayGame(Game),
27. nl, write('Bye')**.**
28. playerTurn(**Game**, **NewGame**) **:-**
29. displayGame(Game),
30. *%do playerMove while not validateMove:*
31. getPlayerType(Game, **PlayerType**),
32. repeat,
33. (
    1. PlayerType = player -> playerMove(Game, **RowSrc**, **ColSrc**, **Path**);
    2. PlayerType = bot -> botMove(Game, **RowSrc**, **ColSrc**, **Path**)
34. ),
35. validateMove(Game, RowSrc, ColSrc, Path, **PieceCaptured**),
36. !,
37. movePiece(Game, RowSrc, ColSrc, Path, **NewGame1**),
38. (
    1. PieceCaptured = barragoonPiece -> barragoonCaptured(NewGame1,NewGame);
    2. PieceCaptured = playerPiece -> playerPieceCaptured(NewGame1, NewGame);
    3. PieceCaptured = empty -> NewGame = NewGame1
39. )**.**
40. playerMove(**Game**, **RowSrc**, **ColSrc**, **Path**) **:-**
41. *%do chooseTile while not validateTile:*
42. repeat,
43. (
44. chooseTile(RowSrc, ColSrc, 'Which tile would you like to move?')
45. ),
46. validateTile(Game, RowSrc, ColSrc),
47. !,
48. *%do choosePath while not validatePath:*
49. repeat,
50. (
51. choosePath(Path, 'Please insert the path that you want that piece to follow:\n(Use WASD - eg. wwwd + Enter).')
52. ),
53. validatePath(RowSrc, ColSrc, Path),
54. !**.**
55. botMove(**Game**, **Row**, **Column**, **Path**)**:-**
56. repeat,
57. (
    1. getPlayerPieces(Game, **Pieces**),
    2. getRandomElemFromList(Pieces, [Row, Column]),
    3. getMovesAvailable(Game, Row, Column, **MovesAvailable**)
58. ),
59. botChooseMove(Game, MovesAvailable, Path)**.**
60. botChooseMove(**Game**, **MovesAvailable**, **Path**) **:-**
61. getDifficulty(Game, **Difficulty**),
62. (
    1. Difficulty = hard -> (
       1. length(MovesAvailable, **N**),
       2. N > *0*,
       3. !,
       4. nth0(*0*, MovesAvailable, Path)
    2. );
    3. Difficulty = easy -> getRandomElemFromList(MovesAvailable, Path)
63. )**.**
64. *% -------------------------------------------------------------------------*
65. *% ------------------------------ MOVEMENTS --------------------------------*
66. *% -------------------------------------------------------------------------*
67. getDestCellFromPath(**RowSrc**, **ColSrc**, [], **RowDest**, **ColDest**)**:-**
68. RowDest = RowSrc,
69. ColDest = ColSrc**.**
70. getDestCellFromPath(**RowSrc**, **ColSrc**, [**Move**|**Tail**], **RowDest**, **ColDest**) **:-**
71. (
72. Move == 'w' -> (**RowDest1** is RowSrc-*1*, getDestCellFromPath(RowDest1,ColSrc,Tail,RowDest,ColDest));
73. Move == 's' -> (**RowDest1** is RowSrc+*1*, getDestCellFromPath(RowDest1,ColSrc,Tail,RowDest,ColDest));
74. Move == 'a' -> (**ColDest1** is ColSrc-*1*, getDestCellFromPath(RowSrc,ColDest1,Tail,RowDest,ColDest));
75. Move == 'd' -> (**ColDest1** is ColSrc+*1*, getDestCellFromPath(RowSrc,ColDest1,Tail,RowDest,ColDest))
76. )**.**
77. *% --- Move piece ---*
78. movePiece(**Game**, **RowSrc**, **ColSrc**, **Path**, **NewGame**) **:-**
79. getDestCellFromPath(RowSrc, ColSrc, Path, **RowDest**, **ColDest**),
80. moveFromSrcToDest(Game,RowSrc,ColSrc,RowDest,ColDest,NewGame)**.**
81. moveFromSrcToDest(**Game**, **RowSrc**, **ColSrc**, **RowDest**, **ColDest**, **NewGame**) **:-**
82. clearCell(Game, RowSrc, ColSrc, **Value**, **NewGame1**),
83. setCell(NewGame1,RowDest, ColDest, Value, NewGame)**.**
84. *% --- Check if it is a short move ---*
85. *%isShortMove(+Piece, +Path)*
86. isShortMove(\_Player-**NDots**, **Path**) **:-**
87. length(Path, **N**),
88. **N1** is N+*1*,
89. N1 == NDots**.**
90. *% --- Check if it is a long move ---*
91. *%isFullMove(+Piece, +Path)*
92. isFullMove(\_Player-**NDots**, **Path**) **:-**
93. length(Path, **N**),
94. N == NDots**.**
95. *% -------------------------------------------------------------------------*
96. *% ---------------------------- IDENTIFY PIECES ----------------------------*
97. *% -------------------------------------------------------------------------*
98. *% --- Check if it is a valid tile ---*
99. *%validateTile(+Game, +RowSrcPos, +ColSrcPos): make sure the position corresponds to a piece of the player*
100. validateTile(**Game**, **RowSrc**, **ColSrc**) **:-**
101. getCurrentPlayer(Game, **CurrentPlayer**),
102. getCell(Game, RowSrc, ColSrc, **Piece**),
103. Piece = CurrentPlayer-\_**.**
104. validateTile(\_Game, \_RowSrc, \_ColSrc) **:-**
105. write('There\'s not a movable piece in that position.'), nl,
106. write('Please, try another position.'), nl,
107. **fail.**
108. *%go\_back to repeat cycle*
109. *% -------------------------------------------------------------------------*
110. *% ---------------------------- MOVEMENT RULES -----------------------------*
111. *% -------------------------------------------------------------------------*
112. *% --- Check if it is a valid path ---*
113. validatePathValues([])**.**
114. validatePathValues([**H**|**T**]) **:-**
115. member(H,['w','a','s','d']),
116. validatePathValues(T)**.**
117. validatePath(**RowSrc**, **ColSrc**, **Path**) **:-**
118. validatePath(RowSrc, ColSrc, Path, true)**.**
119. validatePath(**RowSrc**, **ColSrc**, **Path**, \_) **:-**
120. *%verify if it ends inside the board*
121. getDestCellFromPath(RowSrc, ColSrc, Path, **RowDest**, **ColDest**),
122. RowDest < *9*, RowDest >= *0*,
123. ColDest < *9*, ColDest >= *0*,
124. *%verify if it turns just once*
125. verifyTurnsOnce(Path)**.**
126. validatePath(\_, \_, \_, **ErrorMessageFlag**) **:-**
127. (ErrorMessageFlag -> (
     1. write('That path is not valid!'), nl,
     2. write('Please, try another path.'), nl
128. ); true),
129. **fail.**
130. *%go\_back to repeat cycle*
131. validateMove(**Game**, **RowSrc**, **ColSrc**, **Path**, **PieceCaptured**) **:-**
132. validateMove(Game, RowSrc, ColSrc, Path, PieceCaptured, true)**.**
133. validateMove(**Game**, **RowSrc**, **ColSrc**, **Path**, **PieceCaptured**, **ErrorMessageFlag**) **:-**
134. getCell(Game, RowSrc,ColSrc, **Piece**),
135. (
     1. isShortMove(Piece, Path) -> **IsFullMove** is *0*;
     2. isFullMove(Piece, Path) -> **IsFullMove** is *1*;
     3. (ErrorMessageFlag -> write('Invalid number of movimentations.\nPlease introduce another move\n\n'); true),
     4. **fail**
136. ),
137. validateCrossMovements(Game, RowSrc, ColSrc, Path, IsFullMove, Piece, PieceCaptured, ErrorMessageFlag)**.**
138. *%validateCrossMovements -> verify capture issues.*
139. validateCrossMovements(**Game**, **RowSrc**, **ColSrc**, [**LastMove**], **IsFullMove**, **PieceMoved**, **PieceCaptured**, **ErrorMessageFlag**) **:-**
140. getCurrentPlayer(Game, **CurrentPlayer**),
141. getDestCellFromPath(RowSrc,ColSrc,[LastMove], **RowDest**, **ColDest**),
142. getCell(Game, RowDest, ColDest, **Piece**),
143. ifelse(Piece = 'empty',
     1. emptyTile(PieceCaptured),
     2. ifelse(Piece = bg-**BgType**,
        1. ifelse(IsFullMove == *1*,
        2. ifelse((PieceMoved = \_P-*2*, BgType = 'allDir'),
           1. ((ErrorMessageFlag -> write('You cannot capture an All Directions Barragoon piece with a Two Dotted Tile.\n\n'); true), fail),
           2. true),
        3. ((ErrorMessageFlag -> write('You cannot capture a piece during a short move.\n\n'); true), fail)),
        4. ifelse(Piece = CurrentPlayer-\_,
        5. ((ErrorMessageFlag -> write('You may not capture your own piece.\n\n'); true), fail),
        6. playerPieceCaptured(PieceCaptured))))**.**
144. *%validateCrossMovements -> verify cross issues.*
145. validateCrossMovements(**Game**, **RowSrc**, **ColSrc**, [**FirstMove**, **SecondMove** | **PathRest**], **IsFullMove**, **PieceMoved**,**PieceCaptured**, **ErrorMessageFlag**) **:-**
146. getDestCellFromPath(RowSrc,ColSrc,[FirstMove], **RowDest1**, **ColDest1**),
147. getDestCellFromPath(RowDest1,ColDest1,[SecondMove], **RowDest2**, **ColDest2**),
148. getCell(Game, RowDest1, ColDest1, **Piece1**),
149. ifelse(Piece1 = bg-**BgType**,
150. verifyBarragoonCrossability(RowSrc,ColSrc,RowDest1,ColDest1,RowDest2,ColDest2,BgType,ErrorMessageFlag),
     1. ifelse(Piece1 = 'empty',
        1. true,
        2. ((ErrorMessageFlag -> write('You cannot cross any player piece.\n\n'); true), fail))),
151. validateCrossMovements(Game, RowDest1, ColDest1, [SecondMove | PathRest], IsFullMove, PieceMoved, PieceCaptured, ErrorMessageFlag)**.**
152. *% --- Check if it turns just once ---*
153. verifyTurnsOnce([**H**|**T**]) **:-**
154. complementary(H,**C**),
155. verifyTurnsOnceAux(T,H,*0*,C)**.**
156. verifyTurnsOnceAux([],\_,**N**,\_)**:-**
157. N < *2***.**
158. verifyTurnsOnceAux([**H**|**T**], H, **N**, **C**) **:-**
159. verifyTurnsOnceAux(T, H, N, C)**.**
160. verifyTurnsOnceAux([**H**|**T**], **Z**, **N**, **C**) **:-**
161. Z \= H,
162. H \= C,
163. complementary(H, **C1**),
164. **N1** is N+*1*,
165. verifyTurnsOnceAux(T, H, N1, C1)**.**
166. *% --- Verify crossability for each barragoon piece ---*
167. verifyBarragoonCrossability(**RowSrc**,**ColSrc**,**RowBg**,**ColBg**,**RowDest**,**ColDest**,**BgType**,**ErrorMessageFlag**)**:-**
168. (
     1. BgType = 'barraX' -> (
        1. (ErrorMessageFlag -> (nl, write('You cannot cross an X barragoon piece.', nl, nl )); true),
        2. **fail**
     2. );
     3. BgType = 'allDir' -> (
        1. (
        2. RowSrc \= RowDest,
        3. ColSrc \= ColDest
        4. );
        5. (ErrorMessageFlag -> (nl, write('You are crossing an All Direction Barragoon in a wrong way.'), nl, nl); true),
        6. **fail**
     4. );
     5. *%One direction*
     6. BgType = 'oDirU' -> (
        1. (
        2. **RowSrc2** is RowSrc-*2*,
        3. RowSrc2 = RowDest,
        4. ColSrc = ColDest
        5. );
        6. (ErrorMessageFlag -> (nl, write('You are crossing a Single Direction Barragoon in a wrong way.'), nl, nl); true),
        7. **fail**
     7. );
     8. BgType = 'oDirD' -> (
        1. (
        2. **RowSrc2** is RowSrc+*2*,
        3. RowSrc2 = RowDest,
        4. ColSrc = ColDest
        5. );
        6. (ErrorMessageFlag -> (nl, write('You are crossing a Single Direction Barragoon in a wrong way.'), nl, nl); true),
        7. **fail**
     9. );
     10. BgType = 'oDirR' -> (
         1. (
         2. **ColSrc2** is ColSrc+*2*,
         3. ColSrc2 = ColDest,
         4. RowSrc = RowDest
         5. );
         6. (ErrorMessageFlag -> (nl, write('You are crossing a Single Direction Barragoon in a wrong way.'), nl, nl); true),
         7. **fail**
     11. );
     12. BgType = 'oDirL' -> (
         1. (
         2. **ColSrc2** is ColSrc-*2*,
         3. ColSrc2 = ColDest,
         4. RowSrc = RowDest
         5. );
         6. (ErrorMessageFlag -> (nl, write('You are crossing a Single Direction Barragoon in a wrong way.'), nl, nl); true),
         7. **fail**
     13. );
     14. *%Two directions*
     15. BgType = 'tDirH' -> (
         1. (
         2. (
            1. **ColSrc2** is ColSrc-*2*,
            2. ColSrc2 = ColDest,
            3. RowSrc = RowDest
         3. );
         4. (
            1. **ColSrc2** is ColSrc+*2*,
            2. ColSrc2 = ColDest,
            3. RowSrc = RowDest
         5. )
         6. );
         7. (ErrorMessageFlag -> (nl, write('You are crossing a Double Direction Barragoon in a wrong way.'), nl, nl); true),
         8. **fail**
     16. );
     17. BgType = 'tDirV' -> (
         1. (
         2. (
            1. **RowSrc2** is RowSrc-*2*,
            2. RowSrc2 = RowDest,
            3. ColSrc = ColDest
         3. );
         4. (
            1. **RowSrc2** is RowSrc+*2*,
            2. RowSrc2 = RowDest,
            3. ColSrc = ColDest
         5. )
         6. );
         7. (ErrorMessageFlag -> (nl, write('You are crossing a Double Direction Barragoon in a wrong way.'), nl, nl); true),
         8. **fail**
     18. );
     19. *%Turn to right*
     20. BgType = 'DtoR' -> (
         1. (
         2. **RowSrcU** is RowSrc-*1*,
         3. RowBg = RowSrcU,
         4. **ColBgR** is ColBg+*1*,
         5. ColDest = ColBgR
         6. );
         7. (ErrorMessageFlag -> (nl, write('You are crossing a Right Turn Barragoon in a wrong way.'), nl, nl); true),
         8. **fail**
     21. );
     22. BgType = 'RtoU' -> (
         1. (
         2. **ColSrcL** is ColSrc-*1*,
         3. ColBg = ColSrcL,
         4. **RowBgU** is RowBg-*1*,
         5. RowDest = RowBgU
         6. );
         7. (ErrorMessageFlag -> (nl, write('You are crossing a Right Turn Barragoon in a wrong way.'), nl, nl); true),
         8. **fail**
     23. );
     24. BgType = 'UtoL' -> (
         1. (
         2. **RowSrcD** is RowSrc+*1*,
         3. RowBg = RowSrcD,
         4. **ColBgL** is ColBg-*1*,
         5. ColDest = ColBgL
         6. );
         7. (ErrorMessageFlag -> (nl, write('You are crossing a Right Turn Barragoon in a wrong way.'), nl, nl); true),
         8. **fail**
     25. );
     26. BgType = 'LtoD' -> (
         1. (
         2. **ColSrcR** is ColSrc+*1*,
         3. ColBg = ColSrcR,
         4. **RowBgD** is RowBg+*1*,
         5. RowDest = RowBgD
         6. );
         7. (ErrorMessageFlag -> (nl, write('You are crossing a Right Turn Barragoon in a wrong way.'), nl, nl); true),
         8. **fail**
     27. );
     28. *%Turn to left*
     29. BgType = 'DtoL' -> (
         1. (
         2. **RowSrcU** is RowSrc-*1*,
         3. RowBg = RowSrcU,
         4. **ColBgL** is ColBg-*1*,
         5. ColDest = ColBgL
         6. );
         7. (ErrorMessageFlag -> (nl, write('You are crossing a Left Turn Barragoon in a wrong way.'), nl, nl); true),
         8. **fail**
     30. );
     31. BgType = 'LtoU' -> (
         1. (
         2. **ColSrcR** is ColSrc+*1*,
         3. ColBg = ColSrcR,
         4. **RowBgU** is RowBg-*1*,
         5. RowDest = RowBgU
         6. );
         7. (ErrorMessageFlag -> (nl, write('You are crossing a Left Turn Barragoon in a wrong way.'), nl, nl); true),
         8. **fail**
     32. );
     33. BgType = 'UtoR' -> (
         1. (
         2. **RowSrcD** is RowSrc+*1*,
         3. RowBg = RowSrcD,
         4. **ColBgR** is ColBg+*1*,
         5. ColDest = ColBgR
         6. );
         7. (ErrorMessageFlag -> (nl, write('You are crossing a Left Turn Barragoon in a wrong way.'), nl, nl); true),
         8. **fail**
     34. );
     35. BgType = 'RtoD' -> (
         1. (
         2. **ColSrcL** is ColSrc-*1*,
         3. ColBg = ColSrcL,
         4. **RowBgD** is RowBg+*1*,
         5. RowDest = RowBgD
         6. );
         7. (ErrorMessageFlag -> (nl, write('You are crossing a Left Turn Barragoon in a wrong way.'), nl, nl); true),
         8. **fail**
     36. )
169. )**.**
170. *% -------------------------------------------------------------------------*
171. *% ----------------------------- CONSEQUENCES ------------------------------*
172. *% -------------------------------------------------------------------------*
173. *% --- When a barragoon is captured ---*
174. barragoonCaptured(**Game**, **NewGame**) **:-**
175. insertBarragoon(Game, NewGame)**.**
176. *% --- When a player piece is captured ---*
177. playerPieceCaptured(**Game**, **NewGame**) **:-**
178. switchPlayer(Game, **NewGame1**),
179. insertBarragoon(NewGame1, **FirstBarragoonAddedGame**),
180. switchPlayer(FirstBarragoonAddedGame,**NewGame2**),
181. insertBarragoon(NewGame2, NewGame)**.**
182. *% --- Insert new barragoon ---*
183. insertBarragoon(**Game**, **NewGame**) **:-**
184. *%displayGame(Game),*
185. getPlayerType(Game, **PlayerType**),
186. repeat,
187. (
     1. PlayerType = player -> chooseTile(**Row**, **Column**, 'Where do you wish to place your barragoon?');
     2. PlayerType = bot -> botChooseTile(Game, **Row**, **Column**)
188. ),
189. getCell(Game, Row, Column, empty),
190. !,
191. write('Which barragoon do you wish to insert?'),nl,
192. write('The options are:'),nl,
193. write( '1- X'),nl,
194. write('2- + (All Directions)'),nl,
195. write('3- => (One Direction)'),nl,
196. write('4- <=> (Two Directions)'),nl,
197. write('5- \'> (Left Turn)'),nl,
198. write('6- <\' (Right Turn)'),nl,
199. getPlayerType(Game, PlayerType),
200. (
     1. PlayerType = player -> getCharThenEnter(**Option**);
     2. PlayerType = bot -> (
        1. random(*1*,*7*, **OptionInt**),
        2. numbersAxis(**Numbers**),
        3. nth1(OptionInt, Numbers, **Option**)
     3. )
201. ),
202. (
203. Option = '1' -> setCell(Game, Row, Column, bg-'barraX', NewGame);
204. Option = '2' -> setCell(Game, Row, Column, bg-'allDir', NewGame);
205. Option = '3' -> insertOneDirectionBg(Game, Row, Column, NewGame);
206. Option = '4' -> insertTwoDirectionsBg(Game, Row, Column, NewGame);
207. Option = '5' -> insertTurnToTheLeft(Game, Row, Column, NewGame);
208. Option = '6' -> insertTurnToTheRight(Game, Row, Column, NewGame)
209. )**.**
210. insertOneDirectionBg(**Game**, **Row**, **Column**, **NewGame**)**:-**
211. nl, write('In which diretion you want to set the barragoon?'),nl,
212. write('The options are:'),nl,
213. write( '1- A (Up)'),nl,
214. write('2- V (Down)'),nl,
215. write('3- => (Right)'),nl,
216. write('4- <= (Left)'),nl,
217. getPlayerType(Game, **PlayerType**),
218. (
     1. PlayerType = player -> getCharThenEnter(**Option**);
     2. PlayerType = bot -> (
        1. random(*1*,*5*, **OptionInt**),
        2. numbersAxis(**Numbers**),
        3. nth1(OptionInt, Numbers, **Option**)
     3. )
219. ),
220. (
     1. Option = '1' -> setCell(Game, Row, Column, bg-'oDirU', NewGame);
     2. Option = '2' -> setCell(Game, Row, Column, bg-'oDirD', NewGame);
     3. Option = '3' -> setCell(Game, Row, Column, bg-'oDirR', NewGame);
     4. Option = '4' -> setCell(Game, Row, Column, bg-'oDirL', NewGame)
221. )**.**
222. insertTwoDirectionsBg(**Game**, **Row**, **Column**, **NewGame**)**:-**
223. nl, write('In which diretion you want to set the barragoon?'),nl,
224. write('The options are:'),nl,
225. write( '1- <> (Horizontal)'),nl,
226. write('2- I (Vertical)'),nl,
227. getPlayerType(Game, **PlayerType**),
228. (
     1. PlayerType = player -> getCharThenEnter(**Option**);
     2. PlayerType = bot -> (
        1. random(*1*,*3*, **OptionInt**),
        2. numbersAxis(**Numbers**),
        3. nth1(OptionInt, Numbers, **Option**)
     3. )
229. ),
230. (
     1. Option = '1' -> setCell(Game, Row, Column, bg-'tDirH', NewGame);
     2. Option = '2' -> setCell(Game, Row, Column, bg-'tDirV', NewGame)
231. )**.**
232. insertTurnToTheLeft(**Game**, **Row**, **Column**, **NewGame**)**:-**
233. nl, write('In which diretion you want to set the barragoon?'),nl,
234. write('The options are:'),nl,
235. write('1- <. (Down to Left)'),nl,
236. write('2- -^ (Left to Up)'),nl,
237. write('3- \'> (Up to Right)'),nl,
238. write('4- v- (Right to Down)'),nl,
239. getPlayerType(Game, **PlayerType**),
240. (
     1. PlayerType = player -> getCharThenEnter(**Option**);
     2. PlayerType = bot -> (
        1. random(*1*,*5*, **OptionInt**),
        2. numbersAxis(**Numbers**),
        3. nth1(OptionInt, Numbers, **Option**)
     3. )
241. ),
242. (
     1. Option = '1' -> setCell(Game, Row, Column, bg-'DtoL', NewGame);
     2. Option = '2' -> setCell(Game, Row, Column, bg-'LtoU', NewGame);
     3. Option = '3' -> setCell(Game, Row, Column, bg-'UtoR', NewGame);
     4. Option = '4' -> setCell(Game, Row, Column, bg-'RtoD', NewGame)
243. )**.**
244. insertTurnToTheRight(**Game**, **Row**, **Column**, **NewGame**)**:-**
245. nl, write('In which diretion you want to set the barragoon?'),nl,
246. write('The options are:'),nl,
247. write( '1- .> (Down to Right)'),nl,
248. write('2- ^- (Right to Up)'),nl,
249. write('3- <\' (Up to Left)'),nl,
250. write('4- -v (Left to Down)'),nl,
251. getPlayerType(Game, **PlayerType**),
252. (
     1. PlayerType = player -> getCharThenEnter(**Option**);
     2. PlayerType = bot -> (
        1. random(*1*,*5*, **OptionInt**),
        2. numbersAxis(**Numbers**),
        3. nth1(OptionInt, Numbers, **Option**)
     3. )
253. ),
254. (
     1. Option = '1' -> setCell(Game, Row, Column, bg-'DtoR', NewGame);
     2. Option = '2' -> setCell(Game, Row, Column, bg-'RtoU', NewGame);
     3. Option = '3' -> setCell(Game, Row, Column, bg-'UtoL', NewGame);
     4. Option = '4' -> setCell(Game, Row, Column, bg-'LtoD', NewGame)
255. )**.**
256. *% --- Valid coordinates ---*
257. validColumns(['a','b','c','d','e','f','g','A','B','C','D','E','F','G'])**.**
258. validRow(**Y**)**:-** Y > *48*, Y < *60***.**
259. validBarragoons([bg-barraX, bg-allDir, bg-oDirU, bg-oDirD, bg-oDirL, bg-oDirR, bg-oDirH, bg-oDirV, bg-left, bg-right])**.**
260. complementary('w','s')**.**
261. complementary('s','w')**.**
262. complementary('a','d')**.**
263. complementary('d','a')**.**
264. barragoonCaptured(barragoonPiece)**.**
265. playerPieceCaptured(playerPiece)**.**
266. emptyTile(empty)**.**
267. *% --- Get Player Pieces ---*
268. getPlayerPieces(**Game**, **List**)**:-**
269. getCurrentPlayer(Game, **Player**),
270. getPlayerPiecesAux(Game, Player, *0*, *0*, List)**.**
271. getPlayerPiecesAux(\_,\_,*9*,\_,[])**.**
272. getPlayerPiecesAux(**Game**,**CurrentPlayer**,**Row**,*7*,**List**) **:-**
273. **Row1** is Row+*1*,
274. getPlayerPiecesAux(Game,CurrentPlayer,Row1,*0*,List)**.**
275. getPlayerPiecesAux(**Game**, **CurrentPlayer**, **Row**, **Column**, **List**) **:-**
276. **Column1** is Column+*1*,
277. ifelse( getCell(Game, Row, Column, CurrentPlayer-\_),
     1. (
        1. getPlayerPiecesAux(Game, CurrentPlayer, Row, Column1, **List1**),
        2. List = [[Row, Column] | List1]
     2. ),
     3. getPlayerPiecesAux(Game, CurrentPlayer, Row, Column1, List)
278. )**.**
279. *% --- Count player pieces ---*
280. countPlayerPieces([], \_, *0*)**.**
281. countPlayerPieces([[] | **BoardRest**], **CurrentPlayer**, **Count**) **:-**
282. countPlayerPieces(BoardRest, CurrentPlayer, Count)**.**
283. countPlayerPieces([[**CurrentPlayer**-\_ | **RowRest**] | **BoardRest**], CurrentPlayer, **Count**) **:-**
284. countPlayerPieces([RowRest | BoardRest], CurrentPlayer, **N**),
285. Count is N+*1* **.**
286. countPlayerPieces([[\_Piece | **RowRest**] | **BoardRest**], **CurrentPlayer**, **Count**) **:-**
287. countPlayerPieces([RowRest | BoardRest], CurrentPlayer, Count)**.**
288. *% --- Count Moves Available ---*
289. countMovesAvailable(**Game**, **Row**, **Column**, **Count**)**:-**
290. getMovesAvailable(Game, Row, Column, **List**),
291. length(List, Count)**.**
292. getMovesAvailable(**Game**, **Row**, **Column**, **List**) **:-**
293. getCell(Game, Row, Column, \_Player-**NDots**),
294. availableMoves(NDots, **AllMovesAvailable**),
295. getMovesAvailableAux(Game, Row, Column, AllMovesAvailable, List)**.**
296. getMovesAvailableAux(\_,\_,\_,[],[])**.**
297. getMovesAvailableAux(**Game**, **Row**, **Column**, [**Path** | **Tail**], **List**) **:-**
298. ifelse( (validatePath(Row, Column, Path, false), validateMove(Game, Row, Column, Path, **PieceCaptured**, false)),
     1. (
        1. getMovesAvailableAux(Game, Row, Column, Tail, **List1**),
        2. ifelse( PieceCaptured = playerPiece,
        3. List = [Path | List1],
        4. append(List1, [Path], List)
        5. )
     2. ),
     3. (
        1. getMovesAvailableAux(Game, Row, Column, Tail, List)
     4. )
299. )**.**
300. *% --- Count Moves Available for All Pieces of Player ---*
301. countMovesAvailableAllPieces(**Game**, **Count**) **:-**
302. getPlayerPieces(Game, **PlayerPieces**),
303. countMovesAvailableAllPiecesAux(Game, PlayerPieces, Count)**.**
304. countMovesAvailableAllPiecesAux(\_, [], *0*)**.**
305. countMovesAvailableAllPiecesAux(**Game**, [[**Row**, **Column**] | **RemainingPlayerPieces**], **Count**)**:-**
306. countMovesAvailable(Game, Row, Column, **Count1**),
307. countMovesAvailableAllPiecesAux(Game, RemainingPlayerPieces, **Count2**),
308. Count is Count1+Count2**.**

Utilities.pl

1. *%-------------------------------%*
2. *%--------Useful functions-------%*
3. *%-------------------------------%*
4. *% -------------------------------------------------------------------------*
5. *% ----------------------------- VISUALIZATION -----------------------------*
6. *% -------------------------------------------------------------------------*
7. *% -- Visualization --*
8. spacing(**Lines**) **:-**
9. spacing(*0*, Lines)**.**
10. spacing(**Line**, **Limit**) **:-**
11. Line < Limit,
12. **LineInc** is Line + *1*,
13. nl,
14. spacing(LineInc, Limit)**.**
15. spacing(\_,\_)**.**
16. clearScreen **:-** spacing(*50*), !**.**
17. *% -------------------------------------------------------------------------*
18. *% ----------------------------- RECEIVE INPUT -----------------------------*
19. *% -------------------------------------------------------------------------*
20. *% --- Receive input ---*
21. next **:-**
22. write('Press any key to continue...'),
23. get\_code(\_), clearScreen**.**
24. getCharThenEnter(**X**) **:-**
25. get\_char(X),
26. get\_char(\_), !**.**
27. getCodeThenEnter(**X**) **:-**
28. get\_code(X),
29. get\_char(\_), !**.**
30. *% --- Get collumns and rows ---*
31. readCharFromUser(**X**, **Possibilities**, **Message**, \_Error) **:-**
32. write(Message),
33. getCharThenEnter(X),
34. member(X, Possibilities)**.**
35. readCharFromUser(**X**, **Possibilities**, **Message**, **Error**) **:-**
36. !,
37. write(Error),
38. readCharFromUser(X, Possibilities, Message, Error)**.**
39. readCodeFromUser(**X**, **Condition**, **Message**, \_Error) **:-**
40. write(Message),
41. getCodeThenEnter(X),
42. Condition**.**
43. readCodeFromUser(**X**, **Condition**, **Message**, **Error**) **:-**
44. !,
45. write(Error),
46. readCodeFromUser(X, Condition, Message, Error)**.**
47. *% --- Get position ---*
48. getPositionFromUser(**Row**, **Column**) **:-**
49. readCodeFromUser(Row, validRow(Row), 'Row: ', 'Invalid Row! Rows must be between 1 and 9.\n'),
50. !,
51. validColumns(**ValidCols**),
52. readCharFromUser(Column, ValidCols, 'Column: ', 'Invalid Column! Columns must be between A and G.\n'),
53. !**.**
54. chooseTile(**Row**, **Column**, **Message**) **:-**
55. write(Message), nl,
56. getPositionFromUser(**RowCode**, **ColumnChar**),
57. validColumns(**Possibilities**),
58. nth0(**ColPosInPossibilities**, Possibilities, ColumnChar),
59. Column is ColPosInPossibilities mod *7*,
60. Row is RowCode-*49***.**
61. choosePath(**Path**, **Message**) **:-**
62. nl, write(Message), nl,
63. readCharUntilEnter(Path),
64. validatePathValues(Path)**.**
65. choosePath(\_,\_) **:-**
66. nl, write('Wrong path values. Please try again.'), nl,
67. **fail.**
68. botChooseTile(**Game**, **Row**, **Column**)**:-**
69. getEmptyPlaces(Game, **EmptyPlaces**),
70. getRandomElemFromList(EmptyPlaces, [Row, Column])**.**
71. *% --- Get path ---*
72. readCharUntilEnter(**List**) **:-**
73. get\_char(**Char**),
74. readCharUntilEnterAux([Char|**Tail**],Char),
75. List = [Char|Tail],
76. !**.**
77. readCharUntilEnterAux([],'\n')**.**
78. readCharUntilEnterAux([**Char**|**Tail**], Char) **:-**
79. get\_char(**NewChar**),
80. readCharUntilEnterAux(Tail,NewChar)**.**
81. *% -------------------------------------------------------------------------*
82. *% ----------------------------- CONDITIONS --------------------------------*
83. *% -------------------------------------------------------------------------*
84. *% --- operators ---*
85. ifelse(**Condition**, **If**, \_Else) **:-** Condition, !, If**.**
86. ifelse(\_, \_, **Else**) **:-** Else**.**
87. not(**X**) **:-** X, !, **fail.**
88. not(\_X)**.**
89. *% -------------------------------------------------------------------------*
90. *% ----------------------------- READ BOARD --------------------------------*
91. *% -------------------------------------------------------------------------*
92. *% --- Get cell value ---*
93. getCell(**Game**, **Row**, **Column**, **Value**) **:-**
94. getBoard(Game, **Board**),
95. nth0(Row,Board,**RowList**),
96. nth0(Column,RowList,Value)**.**
97. *% --- Get board ---*
98. getBoard(**Game**, **GameBoard**)**:-**
99. nth0(*0*, Game, GameBoard)**.**
100. *% --- Get player ---*
101. getCurrentPlayer(**Game**, **CurrentPlayer**)**:-**
102. nth0(*1*, Game, CurrentPlayer)**.**
103. *% --- Get game mode ---*
104. getMode(**Game**, **Mode**)**:-**
105. nth0(*2*, Game, Mode)**.**
106. *% --- Get Difficulty ---*
107. getDifficulty(**Game**, **Difficulty**)**:-**
108. nth0(*2*, Game, \_-Difficulty)**.**
109. *% -- Get Empty Cells*
110. getEmptyPlaces(**Game**, **List**)**:-**
111. getEmptyPlacesAux(Game, *0*, *0*, List)**.**
112. getEmptyPlacesAux(\_,*9*,\_,[])**.**
113. getEmptyPlacesAux(**Game**,**Row**,*7*,**List**) **:-**
114. **Row1** is Row+*1*,
115. getEmptyPlacesAux(Game,Row1,*0*,List)**.**
116. getEmptyPlacesAux(**Game**, **Row**, **Column**, **List**) **:-**
117. **Column1** is Column+*1*,
118. ifelse( getCell(Game, Row, Column, empty),
119. (
120. getEmptyPlacesAux(Game, Row, Column1, **List1**),
121. List = [[Row, Column] | List1]
122. ),
123. getEmptyPlacesAux(Game, Row, Column1, List)
124. )**.**
125. getPlayerType(**Game**, **PlayerType**) **:-**
126. getCurrentPlayer(Game, **CurrentPlayer**),
127. getMode(Game, **Mode**),
128. (
129. Mode = pvp -> PlayerType = player;
130. Mode = pvc-\_ -> (
131. CurrentPlayer = w -> PlayerType = player;
132. CurrentPlayer = b -> PlayerType = bot
133. );
134. Mode = cvc-\_ -> PlayerType = bot
135. )**.**
136. *% -------------------------------------------------------------------------*
137. *% ---------------------------- CHANGE BOARD -------------------------------*
138. *% -------------------------------------------------------------------------*
139. *% --- Replace ---*
140. replaceInList([\_|**T**], *0*, **X**, [X|T])**.**
141. replaceInList([**H**|**T**], **I**, **X**, [H|**R**])**:-** I > *-1*, **NI** is I-*1*, replaceInList(T, NI, X, R), !**.**
142. replaceInList(**L**, \_, \_, L)**.**
143. *% --- Get random element from a List ---*
144. getRandomElemFromList(**List**, **Value**) **:-**
145. length(List, **ListLength**),
146. random(*0*,ListLength, **ElemIndex**),
147. nth0(ElemIndex, List, Value)**.**
148. *% --- Clear a cell ---*
149. clearCell(**Game**, **NRow**, **NColumn**, **Value**, **NewGame**) **:-**
150. getBoard(Game, **Board**),
151. */\*Get Value\*/*
152. nth0(NRow, Board, **Row**),
153. nth0(NColumn, Row, Value),
154. */\*Clear cell\*/*
155. replaceInList(Row, NColumn, empty, **NewRow**),
156. replaceInList(Board, NRow, NewRow, **NewBoard**),
157. setBoard(Game, NewBoard, NewGame)**.**
158. *% --- Change cell value ---*
159. setCell(**Game**, **NRow**, **NColumn**, **Value**, **NewGame**) **:-**
160. getBoard(Game, **Board**),
161. nth0(NRow, Board, **Row**),
162. replaceInList(Row, NColumn, Value, **NewRow**),
163. replaceInList(Board, NRow, NewRow, **NewBoard**),
164. setBoard(Game, NewBoard, NewGame)**.**
165. *% --- Change board ---*
166. setBoard(**Game**, **NewBoard**, **NewGame**)**:-**
167. replaceInList(Game, *0*, NewBoard, NewGame)**.**
168. setCurrentPlayer(**Game**, **NewPlayer**, **NewGame**)**:-**
169. replaceInList(Game, *1*, NewPlayer, NewGame)**.**
170. setMode(**Game**, **NewMode**, **NewGame**)**:-**
171. replaceInList(Game, *2*, NewMode, NewGame)**.**
172. *% --- Change player ---*
173. switchPlayer(**Game**, **NextPlayerGame**)**:-**
174. getBoard(Game, **Board**),
175. getCurrentPlayer(Game, **CurrentPlayer**),
176. getMode(Game, **Mode**),
177. ifelse( CurrentPlayer == w ,
178. **NextPlayer** = b,
179. NextPlayer = w),
180. NextPlayerGame = [Board, NextPlayer, Mode]**.**
181. availableMoves(*2*, [ *%fullMove*
182. ['w','w'], ['s','s'], ['d','d'], ['a','a'],
183. ['w','d'], ['d','w'],
184. ['d','s'], ['s','d'],
185. ['s','a'], ['a','s'],
186. ['a','w'], ['w','a'],
187. *%shortMove*
188. ['w'], ['d'], ['a'],['s']
189. ])**.**
190. availableMoves(*3*, [ *%fullMove*
191. ['w','w','w'], ['s','s','s'], ['d','d','d'], ['a','a','a'],
192. ['w','w','d'], ['d','w','w'],
193. ['w','d','d'], ['d','d','w'],
194. ['d','d','s'], ['s','d','d'],
195. ['d','s','s'], ['s','s','d'],
196. ['s','s','a'], ['a','s','s'],
197. ['s','a','a'], ['a','a','s'],
198. ['a','a','w'], ['w','a','a'],
199. ['a','w','w'], ['w','w','a'],
200. *%shortMove*
201. ['w','w'], ['s','s'], ['d','d'], ['a','a'],
202. ['w','d'], ['d','w'],
203. ['d','s'], ['s','d'],
204. ['s','a'], ['a','s'],
205. ['a','w'], ['w','a']
206. ])**.**
207. availableMoves(*4*, [ *%fullMove*
208. ['w','w','w','w'], ['s','s','s','s'],
209. ['d','d','d','d'], ['a','a','a','a'],
210. ['w','w','w','d'], ['d','w','w','w'],
211. ['w','w','d','d'], ['d','d','w','w'],
212. ['w','d','d','d'], ['d','d','d','w'],
213. ['d','d','d','s'], ['s','d','d','d'],
214. ['d','d','s','s'], ['s','s','d','d'],
215. ['d','s','s','s'], ['s','s','s','d'],
216. ['s','s','s','a'], ['a','s','s','s'],
217. ['s','s','a','a'], ['a','a','s','s'],
218. ['s','a','a','a'], ['a','a','a','s'],
219. ['a','a','a','w'], ['w','a','a','a'],
220. ['a','a','w','w'], ['w','w','a','a'],
221. ['a','w','w','w'], ['w','w','w','a'],
222. *%shortMove*
223. ['w','w','w'], ['s','s','s'], ['d','d','d'], ['a','a','a'],
224. ['w','w','d'], ['d','w','w'],
225. ['w','d','d'], ['d','d','w'],
226. ['d','d','s'], ['s','d','d'],
227. ['d','s','s'], ['s','s','d'],
228. ['s','s','a'], ['a','s','s'],
229. ['s','a','a'], ['a','a','s'],
230. ['a','a','w'], ['w','a','a'],
231. ['a','w','w'], ['w','w','a']
232. ])**.**