

# Programação em Lógica

## Damas3D

Nelson Jorge Silva Rodrigues  
Ricardo Jorge Marques Veloso



Universidade do Porto  
Faculdade de Engenharia

**FEUP**

Faculdade de Engenharia da Universidade do Porto  
Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal

**Novembro de 2002**



# Programação em Lógica:

## Damas3D

Nelson Jorge Silva Rodrigues

Ricardo Jorge Silva Rodrigues

Trabalho realizado no âmbito da disciplina de Programação em Lógica, do 1º semestre, do 3º ano, da Licenciatura em Eng. Informática e Computação da Faculdade de Engenharia da Universidade do Porto, leccionada por Maria Cristina Ribeiro e Luís Paulo Reis.

Faculdade de Engenharia da Universidade do Porto  
Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal

Novembro de 2002

## Resumo

Relatório da execução da primeira proposta de trabalho da cadeira de Programação em Lógica do primeiro semestre do terceiro ano da Licenciatura em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto.

Este relatório descreve de forma pormenorizada o sistema desenvolvido, as opções tomadas durante a implementação, bem como as razões das opções escolhidas, descreve os pormenores mais relevantes da implementação e finalmente apresenta as conclusões que os autores retiram da execução deste projecto.

*Aos nossos pais*  
*À Ana*  
*À Xana*

## **Agradecimentos**

Os autores deste trabalho gostariam de agradecer a todas as pessoas que directa ou indirectamente ajudaram a tornar este trabalho possível, em especial aos nossos pais por não partirem do princípio que tínhamos fugido de casa, às nossas namoradas por nos aturarem, ao nosso amigo Luís por toda a ajuda que nos prestou nas alturas em que necessitamos, e por fim um obrigado muito especial aos professores que leccionam a cadeira de Programação em Lógica sem os quais nada disto seria possível.

# Índice

<b>RESUMO .....</b>	<b>D</b>
<b>AGRADECIMENTOS .....</b>	<b>F</b>
<b>ÍNDICE.....</b>	<b>I</b>
<b>ÍNDICE DE FIGURAS .....</b>	<b>III</b>
<b>GLOSSÁRIO .....</b>	<b>IV</b>
<b>CAPÍTULO 1.....</b>	<b>1</b>
1. INTRODUÇÃO .....	1
1.1 Enquadramento.....	1
1.2 Objectivos .....	1
1.3 Estrutura do relatório .....	1
<b>CAPÍTULO 2.....</b>	<b>2</b>
2. DESCRIÇÃO DO PROBLEMA .....	2
2.1 Descrição e regras do jogo.....	2
2.2 Descrição do problema.....	2
2.3 Eventuais dificuldades .....	2
<b>CAPÍTULO 3.....</b>	<b>4</b>
3. ARQUITECTURA DO SISTEMA .....	4
3.1 Descrição sumária do sistema .....	4
3.2 Módulo de jogo .....	4
3.3 Módulo de visualização .....	4
3.4 Comunicação entre os módulos do sistema .....	4
<b>CAPÍTULO 4.....</b>	<b>5</b>
4. REPRESENTAÇÃO DO ESTADO DO MUNDO .....	5
4.1 Representação do estado do tabuleiro .....	5
4.2 Representação do jogador actual .....	5
<b>CAPÍTULO 5.....</b>	<b>6</b>
5. MÓDULO DE LÓGICA DE JOGO .....	6
5.1 Descrição do projecto.....	6
5.2 Implementação em Prolog .....	6
5.2.1 Descrição dos predicados .....	6
5.2.2 Implementação .....	8
<b>CAPÍTULO 6.....</b>	<b>10</b>
6. INTERFACE COM O UTILIZADOR .....	10
6.1 Módulo de lógica .....	10
6.2 Módulo de visualização .....	10
<b>CAPÍTULO 7.....</b>	<b>13</b>

7. CONCLUSÕES .....	13
7.1 Resultados .....	13
7.2 Perspectivas de desenvolvimento .....	13
7.3 Comentários ao desenvolvimento .....	13
7.4 Conclusões finais .....	14
<b>BIBLIOGRAFIA .....</b>	<b>15</b>
<b>ANEXO.....</b>	<b>16</b>

## Índice de Figuras

FIGURA 1 - EXEMPLO DE INTERFACE EM PROLOG.....	10
FIGURA 2 - EXEMPLO DE AMBIENTE DE JOGO - MÁRMORE .....	11
FIGURA 3 - EXEMPLO DE AMBIENTE DE JOGO - MADEIRA .....	11
FIGURA 4 - EXEMPLO DE INTERACÇÃO .....	12

## Glossário

Durante este documento os seguintes termos irão ser utilizados com os significados a seguir apresentados:

<i>Jogador</i>	elemento do jogo que efectua as jogadas, um de brancas ou pretas.
<i>Peça</i>	elemento de jogo de um determinado jogador.
<i>Pedra</i>	uma peça simples, peça básica com que se inicia o jogo.
<i>Dama</i>	peça composta, pedra que após ter atingido o limite do tabuleiro é elevada de categoria.
<i>Casa</i>	posição no tabuleiro, possui como característica as suas coordenadas, e caso a soma das coordenadas seja número ímpar é uma casa preta, casas pretas são as únicas casas válidas de jogo.
<i>Porquinho</i>	peça que não possui mais jogadas possíveis.
<i>Jogada simples</i>	jogada em que apenas se movimenta a peça.
<i>Jogada de comer</i>	jogada em que se elimina uma peça do adversário.
<i>Jogada complexa</i>	jogada em que se eliminam várias peças do adversário.
<i>Jogada múltipla</i>	o mesmo que <i>Jogada complexa</i> .
<i>Rio</i>	linha de casas pretas à direita do jogador.

# Capítulo 1

## 1. Introdução

### 1.1 Enquadramento

O trabalho descrito neste relatório enquadra-se na proposta de desenvolvimento feita pela regente da cadeira de Programação em Lógica.

### 1.2 Objectivos

O objectivo deste trabalho consiste na realização de um jogo de tabuleiro para dois jogadores. O módulo de jogo foi implementado em Prolog tendo sido complementado com um visualizador em Java3D, contendo também este módulo a interface com o utilizador.

### 1.3 Estrutura do relatório

Este relatório encontra-se dividido em sete capítulos, sendo este primeiro capítulo dedicado à introdução ao trabalho.

No segundo capítulo discute-se a natureza do problema e são expostas todas as particularidades inerentes a este trabalho.

No terceiro capítulo é apresentada a arquitectura do sistema desenvolvido.

O quarto capítulo é dedicado à representação do estado do mundo.

O quinto capítulo é dedicado à implementação do módulo de lógica de jogo.

No sexto capítulo é apresentada a interface com o utilizador.

No último capítulo são apresentadas as conclusões gerais, e são analisados os seus principais resultados.

## Capítulo 2

### 2. Descrição do Problema

#### 2.1 Descrição e regras do jogo

O jogo resume-se a um tabuleiro em xadrez de 8x8 casas ordenadas alfabeticamente nas colunas e numeradas nas linhas, as casas serão alternadamente brancas ou pretas, sendo as peças (“pedras”) colocadas nas casas pretas. Cada jogador inicia o jogo com um conjunto de 12 pedras de uma determinada cor, brancas ou pretas. O objectivo do jogo é impedir que o jogador adversário tenha jogadas possíveis a efectuar, sendo a maneira mais simples de o conseguir “comer” todas as peças do adversário. Caso nenhum jogador atinja o objectivo o jogo é considerado empatado.

Existem dois tipos de jogadas possíveis: jogada de movimentação, em que um jogador movimenta a pedra de uma casa para outra casa que esteja na diagonal da primeira, sempre na direcção em que o jogador esteja voltado; jogada de comer em que o jogador cumpre a regra da diagonal e da direcção, mas que pode avançar sobre uma peça da cor oposta, avançando assim duas casas, este tipo de jogada pode ser encadeada, sendo possível efectuar várias jogadas de comer numa única movimentação.

No caso de uma peça de um determinado jogador atingir a última linha do lado oposto em que iniciou o jogo, essa peça recebe a denominação de “dama”, e representa-se pela sobreposição de duas pedras normais. A dama além de poder efectuar todas as jogadas de uma pedra normal, pode ainda movimentar-se mais que uma casa de cada vez, e no sentido inverso à movimentação normal, tendo no entanto de cumprir a regra da diagonal.

#### 2.2 Descrição do problema

Implementar um sistema que utilize duas linguagens de programação (Prolog e Java) e que simule um jogo do tipo damas.

#### 2.3 Eventuais dificuldades

A maior preocupação ao entrar na fase de desenvolvimento deste sistema é a necessidade de aprender uma nova linguagem de programação

Uma dificuldade deste trabalho é a própria natureza do jogo em questão e a quantidade de casos que decorrem das inúmeras regras que requerem tratamento diferenciado, sendo um exemplo desta situação a necessidade de tratar as jogadas complexas, bem como a necessidade de resolver a situação de fazer dama.

## Capítulo 3

### 3. Arquitectura do Sistema

#### 3.1 Descrição sumária do sistema

O sistema de jogo desenvolvido subdivide-se em dois módulos separados: o módulo de jogo implementado em Prolog, e o módulo de visualização implementado em Java3D.

#### 3.2 Módulo de jogo

Programa desenvolvido em Prolog e que implementa todos os predicados necessários à validação e efectuação de uma jogada, ficando assim o núcleo de operações necessárias ao normal progresso do jogo completamente contido num único módulo.

#### 3.3 Módulo de visualização

Aplicação desenvolvida utilizando Java3D, com o propósito de proporcionar a interacção entre o utilizador e o módulo de jogo.

#### 3.4 Comunicação entre os módulos do sistema

A comunicação entre os dois módulos que constituem o sistema de jogo efectua-se através da utilização do Jasper, uma interface fornecida pelo ambiente de desenvolvimento SICStus Prolog, que permite de uma forma praticamente indolor construir uma aplicação que facilmente integra duas linguagens de programação: Prolog e Java.

## Capítulo 4

### 4. Representação do Estado do Mundo

#### 4.1 Representação do estado do tabuleiro

Para representar o tabuleiro optou-se por uma representação do tipo listas de listas, em que cada lista representa uma linha do tabuleiro e a lista de linhas representa o estado do tabuleiro.

Para representar casas que estejam vazias escolheu-se o carácter ‘\_’ (underscore), para representar pedras e damas brancas – ‘O’ e ‘B’, para representar pedras e damas pretas – ‘@’ e ‘P’. Foram escolhidos estes caracteres pela maneira como se assemelham com as peças verdadeiras, quando impressos numa janela de execução puramente em Prolog.

#### 4.2 Representação do jogador actual

A representação do jogador actual é feita através de uma variável que é passada aos predicados que necessitam de saber qual o jogador actual. Esta representação é simplesmente uma string que representa o jogador: “brancas” para o jogador que controla as peças brancas, e “pretas” para o jogador que controla as peças pretas.

## Capítulo 5

### 5. Módulo de Lógica de Jogo

#### 5.1 Descrição do projecto

Este projecto incide sobre a implementação em Prolog de um jogo de tabuleiro para dois jogadores, devendo este módulo conter toda a lógica necessária que permita validar e efectuar uma determinada jogada, bem como implementar um modo em que o computador actue como um ou ambos os jogadores.

Para isso foram implementados variados predicados que se descrevem a seguir.

#### 5.2 Implementação em Prolog

##### 5.2.1 Descrição dos predicados

Seguem-se as descrições dos mais importantes predicados desenvolvidos:

*validaJogada/5* – Predicado que valida uma jogada, é constituído por duas cláusulas, uma que valida imediatamente uma jogada dadas as coordenadas que a constituem, e outra que utiliza recursivamente a primeira para validar jogadas múltiplas.

*movimentoDama/9* – Predicado que verifica se uma determinada jogada realizada por uma dama é válida, chamado exclusivamente pelo predicado *validaJogada*. Predicado recursivo que verifica se uma jogada longa (uma dama pode-se mover por distâncias superiores a uma casa) é válida à custa de várias jogadas de distância 1.

*getPeca/4* – Predicado que testa a existência da peça Peca na posição Col, Lin, no tabuleiro Board. Pode ser usado também no modo de pesquisa, dando Col, Lin devolve a peça existente nessas coordenadas, ou dando Peca, devolve um par Col, Lin onde essa peça exista.

*getLinha/4* – Predicado que devolve a posição N de uma determinada lista.

*verificaSeFazDama/7* – Predicado que verifica se uma determinada jogada causa a criação de uma dama.

*verificaLinha/3* – Predicado que verifica se uma pedra avança na direcção correcta, isto é pretas avançam para baixo, e brancas para cima. Este predicado só é utilizado na verificação de jogadas simples.

*verificaColuna/3* – Predicado que verifica se uma pedra se move na diagonal, e apenas uma casa. Este predicado só é utilizado na verificação de jogadas simples.

*verificaJogadaDama/4* – Predicado que verifica se uma dama se move na diagonal.

*determinaIntermedia/7* - Predicado que determina a casa intermédia no caso de uma jogada de comer.

*fazJogada/8* – Predicado para efectuar uma determinada jogada, depois desta ter sido validada.

*mudaBoard/5* – Predicado “driver” do predicado *mudaBoard2/6*.

*mudaBoard2/6* – Predicado que percorre o tabuleiro à procura da posição a ser modificada.

*mudaLinha/5* – Predicado que efectivamente realiza a modificação no tabuleiro.

*acabou/2* – Predicado para verificar se o jogo terminou, este predicado apenas verifica se o jogador não tem mais peças no tabuleiro, caso ainda haja peças deste jogador é chamado *acabou/5*.

*acabou/5* – Predicado que verifica se as peças de um determinado jogador não têm mais jogadas possíveis, ou seja, peças que se encontrem no estado de porquinho.

*obrigaComer/3* – Predicado “driver” do predicado *obrigaComer/6*.

*obrigaComer/6* – Predicado que verifica se um jogador é obrigado a comer em alguma casa do tabuleiro, a lista devolvida contém todas estas casas, lista vazia significa que o jogador não é obrigado a comer em nenhuma casa.

*obrigaEsquerda/7* – Predicado que verifica se uma peça é obrigada a comer numa casa à esquerda da actual.

*obrigaDireita/7* – Predicado que verifica se uma peça é obrigada a comer numa casa à direita da actual.

*intelectual/4* – Predicado driver do predicado *intelectual/7*.

*intelectual/7* – Predicado que faz a primeira geração de jogadas possíveis, e lança o predicado que implementa o algoritmo min – Max.

*best/7*- Predicado que conjuntamente com o predicado *minimax/7* implementa o algoritmo minimax, usado para implementar a inteligência artificial necessária aos modos de jogo que envolvam o computador.

*minimax/7* - Predicado que conjuntamente com o predicado *best/7* implementa o algoritmo minimax, usado para implementar a inteligência artificial necessária aos modos de jogo que envolvam o computador.

*betterof/8* – Predicado que decide qual a melhor de entre duas jogadas, quer na perspectiva do Max quer na perspectiva do Min.

*evaluate\_board/3* – Predicado que atribui um valor a um tabuleiro, este valor é sempre atribuído tendo em consideração o jogador que vai efectuar a jogada, isto é, o tabuleiro é sempre avaliado no ponto de vista do Max.

*geraJogadas/3* – Predicado que procura por todas as peças do jogador para o qual se querem gerar jogadas, procura por todas as casas vazias que sejam casas válidas de jogo, e chama o predicado *geraJogadas/7*.

*geraJogadas/7* – Predicado que para cada peça do jogador chama o predicado *geraJogadas2/7* para gerar as jogadas válidas.

*geraJogadas2/7* – Predicado que recebe uma casa inicial, uma lista de casas vazias e devolve uma lista composta por todas as combinações que são jogadas válidas.

*casaPreta/2* – Predicado que verifica se uma posição no tabuleiro é uma casa válida de jogo.

## 5.2.2 Implementação

Segue-se a descrição da implementação dos predicados mais importantes.

*validaJogada/5* – Este predicado foi implementado utilizando duas cláusulas separadas:

- Cláusula não recursiva – trata da validação de uma jogada simples, esta jogada simples vai decomposta e todas os seus componentes serão validados por predicados mais simples, por exemplo: a movimentação na horizontal vai ser validada pelo predicado *verificaColuna/3*, a movimentação vertical vai ser validada pelo predicado *verificaLinha/3* e só no caso de ambas as movimentações serem válidas é que podemos considerar que a movimentação se efectua numa diagonal válida. Só depois de ser terem validado todas as condições é que a jogada é declarada válida.

- Cláusula recursiva – trata da validação de uma jogada múltipla, a validação deste tipo de jogadas é efectuada a custa de percorrer uma lista de casas que a peça deverá percorrer, de cada vez que a cláusula é chamada a

cláusula não recursiva é utilizada para validar 4 elementos da lista, sendo a seguir gastos os primeiros 2 elementos da lista, e reutilizados os 2 elementos seguintes. A cláusula irá deste modo iterar recursivamente ao longo da lista até ao final desta.

*geraJogadas/3* – Este é o predicado que vai devolver as jogadas possíveis a partir de um determinado tabuleiro. Isto é conseguido determinando todas as casas onde o jogador possui uma peça, determinar todas as casas vazias do tabuleiro que sejam casas válidas de jogo, e fazendo combinações destas duas listas, e depois de realizar mais alguns testes que eliminam várias das jogadas inválidas à partida, tentar validar a jogada resultante. Esta não será com certeza a maneira mais eficiente de realizar esta tarefa, mas devido ao facto do nosso predicado *validaJogada* só permitir o seu uso no modo de teste, com todas as variáveis que constituem a jogada instanciadas, este tornou-se o único método viável para a geração de jogadas válidas.

*evaluate\_board/3* – Este predicado faz a avaliação do tabuleiro para o algoritmo minimax. Caso o nível de dificuldade seja menor ou igual a 2 este predicado devolve um valor apenas baseado na contagem de peças do jogador, se o nível de dificuldade for maior que 2 além da contagem das peças também é efectuada uma avaliação posicional, embora rudimentar: as damas serão mais beneficiadas se estiverem no rio, as peças serão bastante beneficiadas se estiverem no rio, e moderadamente beneficiadas se não estiverem nas colunas 1 ou 8, caso em que não recebem nenhum aumento de pontuação.

## Capítulo 6

### 6. Interface com o Utilizador

#### 6.1 Módulo de lógica

Incluído com o módulo de lógica está também um predicado que permite utilizar o jogo utilizando apenas Prolog. Este modo de interacção foi primariamente desenvolvido para ser utilizado durante o período de desenvolvimento do trabalho, não devendo ser considerado a verdadeira interface com o utilizador.

```

1 @ _ @ _ @ _ @ _
2 _ @ _ @ _ @ _ @
3 @ _ @ _ @ _ @ _
4 _ _ _ _ _ _ _
5 _ _ _ _ _ _ _
6 _ 0 _ 0 _ 0 _ 0
7 0 _ 0 _ 0 _ 0 _
8 _ 0 _ 0 _ 0 _ 0
  A B C D E F G H
Jogada brancas - a6b5.■

```

Figura 1 - Exemplo de interface em Prolog

Na imagem pode-se observar um exemplo de interacção com o utilizador utilizando apenas o módulo de Prolog, destaca-se também o modo como são introduzidas as jogadas, no formato ColunaInicialLinhaInicialColunaFinalLinhaFinal.

#### 6.2 Módulo de visualização

É com este módulo que se pretende efectuar toda a interacção com o utilizador final da aplicação. Este módulo foi desenvolvido utilizando a linguagem Java com as extensões que possibilitam a criação e visualização de mundos 3D (Java3D).

O ambiente de jogo consiste essencialmente de um tabuleiro de damas em 3 dimensões, um marcador do número de jogos ganhos por cada jogador, e um temporizador que marca o tempo de jogo.

Existe também a possibilidade de mudar o aspecto do ambiente de jogo tendo sido implementados vários ambientes predefinidos.

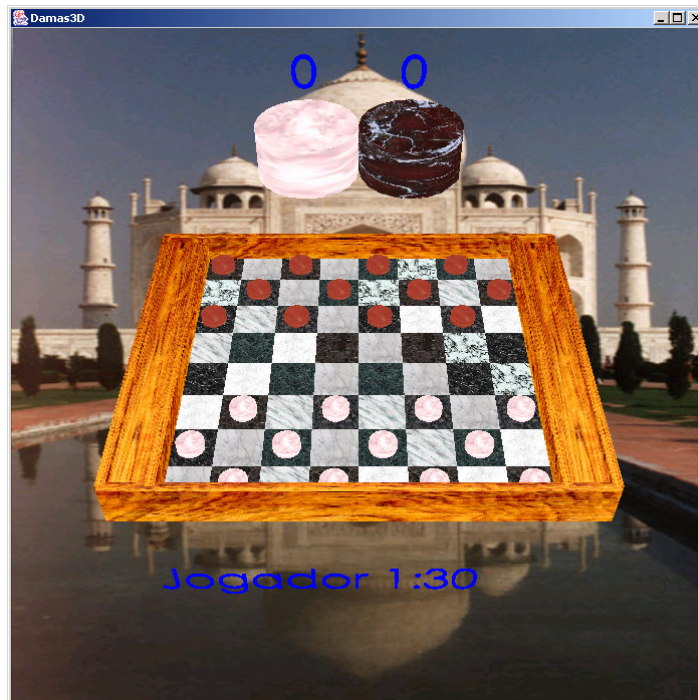


Figura 2 - Exemplo de ambiente de jogo - Mármore

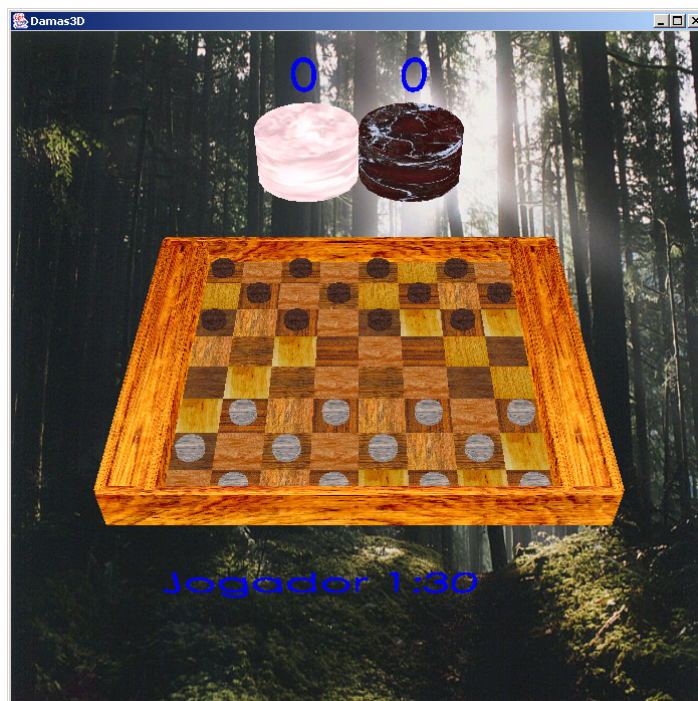
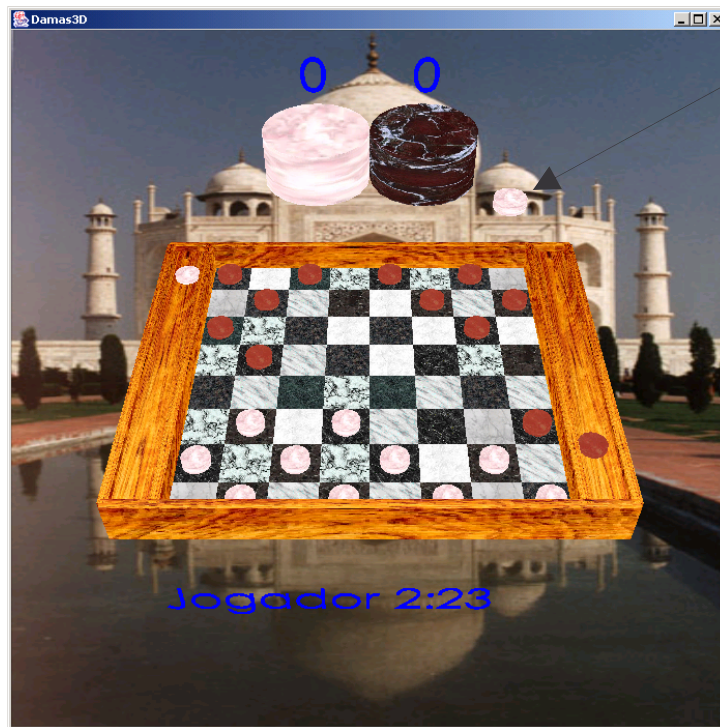


Figura 3 - Exemplo de ambiente de jogo - Madeira

A interação com o utilizador baseia-se na utilização do rato para movimentar as peças no tabuleiro de jogo, devendo no caso de jogadas compostas o utilizador passar por todas as casas que compõem a jogada.



Peça comida a ser retirada do tabuleiro de jogo.

**Figura 4 - Exemplo de interação**

Na Figura 4 está apresentado um exemplo de interação com o jogo, onde o jogador 2 (pedras vermelhas) avançou a pedra em F4 para a casa H6 comendo a pedra do jogador 1 (pedras brancas) em G5, na figura realça-se também a pedra comida a ser retirada do tabuleiro.

## Capítulo 7

### 7. Conclusões

#### 7.1 Resultados

O resultado da execução deste trabalho foi uma aplicação que foi desenvolvida utilizando duas linguagens de programação, e que modela um jogo de damas de forma bastante precisa.

#### 7.2 Perspectivas de desenvolvimento

Embora os autores deste projecto não tenham em mente voltar a trabalhar no mesmo, achamos que existe uma miríade de melhoramentos possíveis nomeadamente a introdução de uma inteligência artificial melhorada, possivelmente com a utilização de cortes alfa-beta. Também não seria de descartar a hipótese de mudar a interface de ligação entre as duas linguagens.

#### 7.3 Comentários ao desenvolvimento

O tempo de desenvolvimento do trabalho estendeu-se algo para além daquele que os autores consideravam necessário, sendo a causa disto grandemente a falta de conhecimento da linguagem de programação Prolog o que influenciou as nossas estimativas aquando do planeamento do projecto. Esta extensão causou um encurtamento significativo no tempo de desenvolvimento que tínhamos alocado para a fase de testes, isto poderá eventualmente significar que existe a possibilidade de algum bug mais escondido ter passado sem ser notado.

Os autores gostariam também de expressar o seu descontentamento em relação ao ambiente de desenvolvimento recomendado, SICStus Prolog, após apenas alguns dias a trabalhar com este ambiente chegamos a conclusão que as facilidades fornecidas para ajudar no debugging são pouco mais do que inexistentes, falta essencialmente uma maneira eficaz de visualizar o código enquanto o debug esta a decorrer, isto leva-nos directamente a outra falha fundamental, a falta de um editor de código dentro do próprio ambiente de desenvolvimento.

Em último lugar gostaríamos de nos referir à interface utilizada para efectuar a ligação entre ambas as linguagens de programação: o Jasper. Gostaríamos de poder dizer que foi fácil trabalhar com esta interface, mas esse não é o caso, demorou bastante tempo para apenas conseguir correr os métodos de teste que a própria interface fornece, tendo demorado ainda mais tempo para conseguir efectivamente executar uma chamada a algum predicado Prolog. Descobrimos que esta interface tem alguns problemas quando executada num programa que possua vários threads de execução, tendo dado bastante luta para conseguir ultrapassar esta limitação. Também descobrimos que esta interface é implementada com base em chamadas a métodos nativos, o que implica a impossibilidade de executar qualquer aplicação que use a interface como um applet. Outro aspecto a apontar é a falta de estabilidade da interface, bem como a falta de informação que a mesma disponibiliza quando algo de mau acontece, não são raras as vezes que a excepção `UnknownHostException` é atirada sem nenhuma razão aparente, mesmo em alturas em que não estão a ser chamados facilidades oferecidas pela interface. Isto contrasta bastante com a qualidade da documentação fornecida, concisa e bastante explicativa.

#### **7.4 Conclusões finais**

Após uma cuidada análise dos pontos que constituem este capítulo não será certamente difícil de concluir que este foi um projecto difícil de executar e cujo andamento não foi certamente linear, mas antes aos solavancos, não foram raras as vezes em que predicados que tínhamos dado como finalizados tiveram de ser revistos.

Apesar dos contratemplos e das dificuldades inerentes à aprendizagem de uma linguagem de programação nova, já para não referir todo um paradigma completamente diferente, conseguimos realizar um trabalho que está num nível que podemos considerar bastante bom.

## **Bibliografia**

- Leon Sterling, Ehud Shapiro. The Art of Prolog, 2a edição. MIT Press, 1994.
- Adventure In Prolog by AMZI –  
<http://oopweb.com/Prolog/Documents/AdventureInProlog/VolumeFrames.html>

## Anexo

```

%'@' - black
%'O' - white
%' ' - espaco
%'B' - Dama preta
%'P' - Dama branca

tabuleiro([
    ['@', '@', '@', '@', '@', '@', '@', '@'],
    ['@', '@', '@', '@', '@', '@', '@', '@'],
    ['@', ' ', '@', ' ', '@', ' ', '@', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', 'O', ' ', 'O', ' ', 'O', ' ', 'O'],
    ['O', ' ', 'O', ' ', 'O', ' ', 'O', ' '],
    [' ', 'O', ' ', 'O', ' ', 'O', ' ', 'O']
]).

tabuleiro2([
    ['@', '@', '@', ' ', 'B', ' ', ' ', '@'],
    [' ', '@', ' ', ' ', ' ', '@', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', 'B', ' '],
    [' ', 'O', ' ', '@', ' ', ' ', ' ', ' '],
    [' ', ' ', 'O', ' ', 'O', ' ', ' ', ' '],
    [' ', 'P', ' ', ' ', ' ', ' ', ' ', ' '],
    ['@', ' ', ' ', ' ', ' ', 'O', ' ', ' '],
    [' ', 'O', ' ', 'O', ' ', 'O', ' ', ' '']
]).

/*****
/*      Predicados de entrada de dados      */
/*****/

inicio :-
    retractall( computador( X, Y ) ), retractall( nivel( X ) ),
    write('1- Humano/Humano'),nl,
    write('2- Humano/Computador'),nl,
    write('3- Computador/Humano'),nl,
    write('4- Computador/Computador'),nl,
    read(Opcao),
    (
        Opcao == 1, assert( computador( X, fail ) )
        ;
        write( '1-Easy' ),nl,
        write( '2-Medium' ),nl,
        write( '3-Hard' ),nl,

```

```

        write( '4-Very Hard (nao recomendado)' ),nl,
        read( Dificuldade ), assert( nivel( Dificuldade ) ),
        (
        Opcao == 2, assert( computador( 'pretas', true ) )
        ;
        Opcao == 3, assert( computador( 'brancas', true ) )
        ;
        Opcao == 4, assert( computador( 'brancas', true ) ), assert(
computador( 'pretas', true ) )
        )
        ;
        true
    ),
    tabuleiro(Board),
    joga('brancas', Board),
    retractall( computador( X, Y ) ), retractall( nivel( X ) ).

/*****
/*          Predicado de de jogo          */
*****/

joga(Jogador, Board) :-
    showBoard(Board),
    (
        computador( Jogador, Value ), Value,
        nivel( Dificuldade ), intelectual( Jogador, Dificuldade,
Board, Mov-NewBoard ), write(Mov),nl
        ;
        recebeJogada(Jogador, Board, NewBoard )
    ),
    mudaJogador(Jogador, Jogador2),
    (acabou(Jogador2,
NewBoard),nl, showBoard(NewBoard), assert( final(NewBoard) ),nl, write('Jogo
terminou. Ganharam as '), write(Jogador), nl
    ;
    joga(Jogador2, NewBoard)).

mudaJogador('brancas', 'pretas').
mudaJogador('pretas', 'brancas').

/*****
/*          Predicados do tabuleiro          */
*****/

showBoard(Board) :-
    nl,
    showLines( Board, 1 ),
    write(' A B C D E F G H'),nl.

showLines( [], _ ).
showLines( [Cabeca | Resto], N ) :-
    write(N),
    write(' '),
    N1 is N + 1,
    showLine(Cabeca),nl,
    showLines(Resto, N1).

```

```

showLine( [] ).
showLine( [Cabeca | Resto] ) :-
    write(Cabeca),
    write(' '),
    showLine(Resto).

/*****
/*          Predicados de jogada          */
*****/

recebeJogada(Jogador, Board, NewBoard) :-
    repeat,
    write('Jogada '),write(Jogador),write(' - '),
    read(X),name(X,Lista),modificaLista( Lista, ListaFinal),
    validaJogada( Jogador, ListaFinal, Board, NewBoard, TipoJogada ).

/*****
/*          Predicado para validar jogada          */
*****/

validaJogada( Jogador, [ColIni, LinIni, ColFim, LinFim], Board,
NewBoard, TipoJogada ) :-!,
    (
        getPeca( ColFim, LinFim, Board, '_' ),
        getPeca( ColIni, LinIni, Board, Peca ), pecas( Jogador, Pc, PcDama
), (Peca == Pc ; Peca == PcDama ),
        obrigaComer( Jogador, Board, Lista ),!,
        length( Lista, N ),
        (
            ( N == 0 )
            ;
            member( [ColIni, LinIni, ColFim, LinFim], Lista )
        ),
        (
            ( Peca == 'B'; Peca == 'P' ),
            verificaJogadaDama( ColIni, LinIni, ColFim, LinFim ),
            mudaJogador( Jogador, Adv ),
            movimentoDama( Adv, Peca, ColIni, LinIni, ColFim, LinFim,
Board, NB, TJ )
        );
        verificaLinha( Jogador, LinIni, LinFim ),
        verificaColuna( ColIni, ColFim ),
        TJ = 'simples',
        fazJogada( Peca, TJ, ColIni, LinIni, ColFim, LinFim, Board, NB )
        ;
        determinaIntermedia( Jogador, ColIni, LinIni, ColFim, LinFim, X, Y
),
        member( [ColIni, LinIni, ColFim, LinFim], Lista ),
/*
        verificaLinha( Jogador, LinIni, Y ),
        verificaLinha( Jogador, Y, LinFim ),
        verificaColuna( ColIni, X ),
        verificaColuna( X, ColFim ),

        getPeca( X, Y, Board, PecaIntermedia ),

```

```

        (( Jogador == 'brancas', (PecaIntermedia == '@'; PecaIntermedia ==
'P') ) ; ( Jogador == 'pretas', (PecaIntermedia == '0'; PecaIntermedia
== 'B'))),*/

        TJ = 'come',
        fazJogada( Peca, TJ, ColIni, LinIni, X, Y, ColFim, LinFim, Board,
NB )
    )
    ;
    fail), verificaSeFazDama( Peca, ColFim, LinFim, NB, NewBoard, TJ,
TipoJogada ).

validaJogada( Jogador, [ColIni, LinIni, ColFim, LinFim | Resto ], Board,
NewBoard, TipoJogada ) :-
    validaJogada( Jogador, [ColIni, LinIni, ColFim, LinFim], Board,
NewBoard1, TipoJogada2 ),
    validaJogada( Jogador, [ColFim, LinFim | Resto ], NewBoard1,
NewBoard, TipoJogada ).

/*****
/*   Predicado para validar movimentos de uma dama   */
*****/

movimentoDama( Adv, Peca, ColFim, LinFim, ColIni, LinIni, NewBoard,
NewBoard, TipoJogada ) :- TipoJogada = 'damaSimples'.
movimentoDama( Adv, Peca, ColIni, LinIni, ColFim, LinFim, Board,
NewBoard, TipoJogada ) :-
    ((ColFim > ColIni) -> X is ColIni + 1, X1 is X + 1 ; X is ColIni -
1, X1 is X - 1),
    ((LinFim > LinIni) -> Y is LinIni + 1, Y1 is Y + 1 ; Y is LinIni -
1, Y1 is Y - 1),
    (
        getPeca( X, Y, Board, '_' ),
        TipoJogada2 = 'damaSimples',
        fazJogada( Peca, 'simples', ColIni, LinIni, X, Y, Board, NB ),
        Z = X, Z1 = Y
    )
    ;
    getPeca( X1, Y1, Board, '_' ),
    TipoJogada2 = 'damaCome',
    fazJogada( Peca, 'come', ColIni, LinIni, X, Y, X1, Y1, Board, NB
),
    Z = X1, Z1 = Y1
),
    movimentoDama( Adv, Peca, Z, Z1, ColFim, LinFim, NB, NewBoard,
TipoJogada3 ),
    ( TipoJogada3 == 'damaSimples' -> TipoJogada = TipoJogada2 ;
TipoJogada = 'damaCome' ).

descobrePeca( 'brancas', '0' ).
descobrePeca( 'brancasDama', 'B' ).
descobrePeca( 'pretas', '@' ).
descobrePeca( 'pretasDama', 'P' ).

/*****
/*   Predicado para descobrir qual a peca numa posicao   */
*****/

```

```

getPeca( Col, Lin, Board, Peca ) :-
    getLinha( Linha, Lin, 1, Board ),
    getLinha( Peca, Col, 1, Linha ).

getLinha(Member, N, N, [Member | R]).
getLinha(Member, N, P, [Cabeca | R]) :-
    P1 is P + 1,
    getLinha(Member, N, P1, R).

/*****
/*   Predicado para verificar se a jogada origina dama   */
*****/
verificaSeFazDama( Peca, ColFim, LinFim, Board, NewBoard, TJ, TipoJogada
) :-
    mudaPeca( Peca, NovaPeca ),
    (Peca == '0', LinFim == 1 ; Peca == '@', LinFim == 8)
    ,
    mudaBoard( NovaPeca, ColFim, LinFim, Board, NewBoard ), TipoJogada
= 'fazDama'
    ;
    NewBoard = Board, TipoJogada = TJ,
    true.

mudaPeca( '0', 'B' ).
mudaPeca( '@', 'P' ).

verificaLinha('brancas', LinIni, LinFim ) :-
    X is LinIni - 1,
    X == LinFim.
verificaLinha( 'pretas', LinIni, LinFim ) :-
    X is LinIni + 1,
    X == LinFim.

verificaColuna( 1, 2 ).
verificaColuna( 8, 7 ).
verificaColuna( ColIni, ColFim ) :-
    Z is ColIni + 1, ColFim == Z
    ;
    Z is ColIni - 1, ColFim == Z .

/*****
/*   Verifica se a movimentacao da dama é valida   */
*****/
verificaJogadaDama( ColIni, LinIni, ColFim, LinFim ) :-
    (ColIni \= ColFim, LinIni \= LinFim),
    X is abs(ColIni - ColFim),
    Y is abs(LinIni - LinFim),
    X == Y.

determinaIntermedia( 'brancas', ColIni, LinIni, ColFim, LinFim, X, Y )
:-
    (ColFim > ColIni, X is ColFim - 1 ; X is ColIni - 1),

    (Y is LinIni - 1 ).

```

```

determinaIntermedia( 'pretas', ColIni, LinIni, ColFim, LinFim, X, Y ) :-
    (ColFim > ColIni, X is ColFim - 1 ; X is ColIni - 1),
    (Y is LinIni + 1 ).

/*****
/*      Predicado para efectuar jogada      */
*****/

fazJogada( Peca, 'simples', ColIni, LinIni, ColFim, LinFim, Board,
NewBoard ) :-
    mudaBoard( '_', ColIni, LinIni, Board, NewBoard1 ),
    mudaBoard( Peca, ColFim, LinFim, NewBoard1, NewBoard ).

fazJogada( Peca, 'come', ColIni, LinIni, X, Y, ColFim, LinFim, Board,
NewBoard ) :-
    fazJogada( Peca, 'simples', ColIni, LinIni, X, Y, Board, NewBoard1
),
    fazJogada( Peca, 'simples', X, Y, ColFim, LinFim, NewBoard1,
NewBoard ).

mudaBoard( Novo, Col, Lin, Board, NewBoard ) :-
    mudaBoard2( 1, Novo, Col, Lin, Board, NewBoard ).

mudaBoard2( _,_,_,_, [],[] ).
mudaBoard2( Y, Novo, X, Y, [Lin|Resto], [NovLin|NovResto] ) :-
    mudaLinha( 1, Novo, X, Lin, NovLin ),
    N2 is Y + 1,
    mudaBoard2( N2, Novo, X, Y, Resto, NovResto ).
mudaBoard2( N, Novo, X, Y, [Lin|Resto], [Lin|NovResto] ) :-
    Y\=N, N2 is N + 1,
    mudaBoard2( N2, Novo, X, Y, Resto, NovResto ).

mudaLinha( _,_,_, [], [] ).
mudaLinha( X, Novo, X, [Peca|Resto], [Novo|NovResto] ) :-
    N2 = X + 1,
    mudaLinha( N2, Novo, X, Resto, NovResto ).
mudaLinha( N, Novo, X, [Peca|Resto], [Peca|NovResto] ) :-
    N \= X, N2 is N + 1,
    mudaLinha( N2, Novo, X, Resto, NovResto ).

/*****
/*      Predicado para traduzir ascis em numeros      */
*****/

modificaLista( [], [] ).
modificaLista( [Col, Lin |Resto], [NovaCol, NovaLinha|NovoResto] ) :-
    modifica( Col, Lin, NovaCol, NovaLinha ),
    modificaLista( Resto, NovoResto ).

modifica( X, Y, Col, Lin ) :-
    Col is X - 96,
    Lin is Y - 48.

/*****
/*      Predicado para ver se o jogo acabou      */
*****/

```

```

/*****/

acabou( Jogador, Board ) :-
    pecas( Jogador, PecaNormal, PecaDama ),
    findall( PecaNormal, getPeca(_,_, Board, PecaNormal), ListaNormal
),
    findall( PecaDama, getPeca(_,_, Board, PecaDama), ListaDama ),
    append(ListaNormal, ListaDama, Lista),
    length( Lista, N),
    (
        (N == 0)
        ;
        acabou( Jogador, 1, 1, Board, M ),!(, ( N == M )
    ).

acabou( _, 1, 9, _, 0).
acabou( Jogador, Col, Lin, Board, N ) :-
    (
        (Col < 8) -> C is Col + 1, L is Lin ; C is 1, L is Lin + 1
    ),
    acabou( Jogador, C, L, Board, N2 ),
    getPeca(Col, Lin, Board, Peca),
    (
        (Peca == '_'), N = N2
        ;
        ((Jogador == 'brancas', Peca == '0'),
            (
                (Col == 8, Lin == 2),getPeca(7,1,Board,P), (P == '@' ; P ==
'P')
                ;
                ((Col == 1 ),(C1 is Col + 1, C2 is Col + 2, L1 is Lin - 1,
L2 is Lin - 2),( L2 >= 1, L2 =< 8 )
                ;
                (Col == 8),(C1 is Col - 1, C2 is Col - 2, L1 is Lin - 1, L2
is Lin - 2),( L2 >= 1, L2 =< 8 )),
                getPeca(C1,L1,Board, P1), getPeca(C2,L2,Board,P2), (P1 ==
'@' ; P1 == 'P'), (P2 == '@' ; P2 == 'P')
                ), N is N2 + 1)
            ;
            ((Jogador == 'pretas', Peca == '@'),
                (
                    (Col == 1, Lin == 7),getPeca(2,8,Board,P), (P == '0' ; P ==
'B')
                    ;
                    ((Col == 1 ),(C1 is Col + 1, C2 is Col + 2, L1 is Lin + 1,
L2 is Lin + 2),( L2 >= 1, L2 =< 8 )
                    ;
                    ( Col == 8 ), (C1 is Col - 1, C2 is Col - 2, L1 is Lin + 1,
L2 is Lin + 2),( L2 >= 1, L2 =< 8 )),
                    getPeca(C1,L1,Board, P1), getPeca(C2,L2,Board,P2), (P1 ==
'0' ; P1 == 'B'), (P2 == '0' ; P2 == 'B')
                    ), N is N2 + 1)
                ;
                true, N = N2
            ).
/*****/
/*    Predicados uteis na manipulacao de listas    */

```

```

/*****/

append( [], B, B ).
append( [A|B], C, [A|D] ) :- append( B, C, D ).

member( [A, B, C, D], [A, B, C, D | T] ).
member( X, [_|T] ):-member(X,T).

/*****/
*****/
/*      Predicado para ver se existe alguma posicao em que e obrigado a
comer      */
/*****/
*****/

obrigaComer( Jogador, Board, Lista ) :- !,
        obrigaComer( Jogador, 1, 1, Board, [], Lista ).

obrigaComer( _, 1, 9, _, Lista, Lista).
obrigaComer( Jogador, Col, Lin, Board, Lst, Lista ) :-
        getPeca( Col, Lin, Board, Peca ),
        (
            Peca == ' ' ,
            List = Lst
        );
        ((Col == 1) ; (Col == 2)),obrigaDireita( Jogador, Col, Lin, ColF,
LinF, Board, Peca ),
        Move = [Col, Lin, ColF, LinF],
        append( Lst, Move, List)
        ;
        ((Col == 8) ; (Col == 7)),obrigaEsquerda( Jogador, Col, Lin, ColF,
LinF, Board, Peca ),
        Move = [Col, Lin, ColF, LinF],
        append( Lst, Move, List)
        ;
        obrigaDireita( Jogador, Col, Lin, ColF1, LinF1, Board, Peca ),
        Move1 = [Col, Lin, ColF1, LinF1],
        append( Lst, Move1, List1),
        obrigaEsquerda( Jogador, Col, Lin, ColF2, LinF2, Board, Peca ),
        Move2 = [Col, Lin, ColF2, LinF2],
        append( List1, Move2, List)
        ;
        obrigaDireita( Jogador, Col, Lin, ColF, LinF, Board, Peca ),
        Move = [Col, Lin, ColF, LinF],
        append( Lst, Move, List)
        ;
        obrigaEsquerda( Jogador, Col, Lin, ColF, LinF, Board, Peca ),
        Move = [Col, Lin, ColF, LinF],
        append( Lst, Move, List)
        ;
        List = Lst,
        true
    ),
    (
        (Col < 8) -> C is Col + 1, L is Lin ; C is 1, L is Lin + 1
    ),
    obrigaComer(Jogador, C, L, Board, List, Lista).

```

```

obrigaDireita( Jogador, Col, Lin, ColF, LinF, Board, P ) :-
(
    (Jogador == 'brancas', (P == '0' ; P == 'B')),
    (Lin \= 1), (Lin \= 2),
    C1 is Col + 1, L1 is Lin - 1, getPeca( C1, L1, Board, Peca),
    (Peca == '@' ; Peca == 'P'),
    C2 is Col + 2, L2 is Lin - 2, getPeca( C2, L2, Board, '_' ),
    ColF = C2, LinF= L2
    ;
    (Jogador == 'pretas', (P == '@' ; P == 'P')),
    (Lin \= 7), (Lin \= 8),
    C1 is Col + 1, L1 is Lin + 1, getPeca( C1, L1, Board, Peca),
    (Peca == '0' ; Peca == 'B'),
    C2 is Col + 2, L2 is Lin + 2, getPeca( C2, L2, Board, '_' ),
    ColF = C2, LinF= L2).

obrigaEsquerda( Jogador, Col, Lin, ColF, LinF, Board, P ) :-
(
    (Jogador == 'brancas', (P == '0' ; P == 'B')),
    (Lin \= 1), (Lin \= 2),
    C1 is Col - 1, L1 is Lin - 1, getPeca( C1, L1, Board, Peca),
    (Peca == '@' ; Peca == 'P'),
    C2 is Col - 2, L2 is Lin - 2, getPeca( C2, L2, Board, '_' ),
    ColF = C2, LinF= L2
    ;
    (Jogador == 'pretas', (P == '@' ; P == 'P')),
    (Lin \= 7), (Lin \= 8),
    C1 is Col - 1, L1 is Lin + 1, getPeca( C1, L1, Board, Peca),
    (Peca == '0' ; Peca == 'B'),
    C2 is Col - 2, L2 is Lin + 2, getPeca( C2, L2, Board, '_' ),
    ColF = C2, LinF= L2).

obrigaDama( Jogador, ColI, LinI, Board, Lista ) :-
    findall( [ColI, LinI, ColF, LinF],
        (getPeca(X, Y, Board, Peca), (Peca == '@' ; Peca ==
'P'),write(X),write(Y),nl,
            verificaJogadaDama( ColI, LinI, X, Y ),write('passei'),nl,
            livres( ColI, LinI, X, Y, ColF, LinF, Board )
        ), Lst), Lista = Lst.

livres(ColIni, LinIni, W, Z, ColFim, LinFim, Board) :-
    ((W > ColIni) -> X is W + 1 ; X is W - 1),
    ((Z > LinIni) -> Y is Z + 1 ; Y is Z - 1),
    getPeca( X, Y, Board, '_' ),
    ColFim = X, LinFim = Y.

/*****
/*
MiniMax
*/
*****/

intelectual( Jogador, MaxDepth, Board, BestMov) :-
    write( 'Computador a pensar.....' ), nl,
    intelectual( 0, MaxDepth, Jogador, Board, BestMov, Val, Jogador ),
    !.

```

```

intelectual(Depth, MaxDepth, Jogador, Board, BestMov, Val, JogActual )
:-
    geraJogadas( Jogador, Board, Lista ),!,
    Depth2 is Depth + 1,
    best( Depth2, MaxDepth, Jogador, Lista, BestMov, Val, JogActual ).

best( Depth, MaxDepth, Jogador, [Mov-Board], Mov-Board, Val, JogActual )
:-
    minimax( Depth, MaxDepth, Jogador, Board, _, Val, JogActual ), !.
best( Depth, MaxDepth, Jogador, [Mov1-Board1|BoardList], Mov-BestMov,
BestVal, JogActual ) :- !,
    minimax( Depth, MaxDepth, Jogador, Board1, _, Val1, JogActual ),
    !,
    best( Depth, MaxDepth, Jogador, BoardList, Mov2, Val2, JogActual
),
    betterof( Jogador, Mov1-Board1, Val1, Mov2, Val2, Mov-BestMov,
BestVal, JogActual ).

minimax( Depth, MaxDepth, Jogador, Board, BestMov, Val, JogActual ) :-
%   write(Depth),write(' - '),write(Jogador),nl,
    Depth < MaxDepth,
    mudaJogador( Jogador, Adv),
    Depth2 is Depth + 1,
    geraJogadas( Adv, Board, Lista ),!,
    length( Lista, Len ), (Len \= 0,best( Depth2, MaxDepth, Adv,
Lista, BestMov, Val, JogActual ) ; Val = 1000).
minimax( Depth, MaxDepth, Jogador, Board, BestMov, Val, JogActual ) :-
    !,
    evaluate_board( JogActual, Board, Val ).

betterof( Jogador, Mov0, Val0, Mov1, Val1, Mov0, Val0, JogActual ) :-
    mudaJogador( JogActual, Adv ),
    (
    Jogador == JogActual, Val0 > Val1, !
    ;
    Jogador == Adv, Val0 < Val1, !
    ).
betterof( Jogador, Mov0, Val0, Mov1, Val1, Mov1, Val1, JogActual ) :-!.

evaluate_board( Jogador, Board, Val ) :-
    nivel(X), X =< 2 -> evaluate_board1( Jogador, Board, Val ) ;
evaluate_board2( Jogador, Board, Val ).

evaluate_board1( Jogador, Board, Val ) :-
    pecas( Jogador, Peca, PecaDama ),
    findall( Peca, getPeca(_,_,Board,Peca), Bag), length( Bag, Z1 ),
    findall( PecaDama,getPeca(_,_,Board,PecaDama), Bag2), length(
Bag2, Z2 ),
    mudaJogador( Jogador, Adv ), pecas( Adv, PecaAdv, PecaDamaAdv ),
    findall( PecaAdv, getPeca(_,_,Board,PecaAdv), Bag3), length( Bag3,
Z3 ),
    findall( PecaDamaAdv, getPeca(_,_,Board,PecaDamaAdv), Bag4),
length( Bag4, Z4 ),

    Pecas is Z1 * 10, Damas is Z2 * 15, ValorJogador is Pecas + Damas,

```

```

    PecasAdv is Z3 * 10, DamasAdv is Z4 * 15, ValorJogadorAdv is
PecasAdv + DamasAdv,
    Val is ValorJogador - ValorJogadorAdv.

evaluate_board2( Jogador, Board, Val ) :-
    pecas( Jogador, Pc, PcDama ),
    findall( [X,Y,Peca], (getPeca( X, Y, Board, Peca), (Peca == Pc ;
Peca == PcDama)), Bag ),
    processaLista( Bag, Valor ),
    mudaJogador( Jogador, Adv ),
    pecas( Adv, PcAdv, PcDamaAdv ),
    findall( [X2,Y2,Peca2], (getPeca( X2, Y2, Board, Peca2), (Peca2 ==
PcAdv ; Peca2 == PcDamaAdv)), BagAdv ),
    processaLista( BagAdv, ValorAdv ),
    Val is Valor - ValorAdv.

processaLista( Lista, Valor ) :-
    processaLista( Lista, Valor, 0 ).

processaLista( [], Valor, Valor ).
processaLista( [[X,Y,Peca]|Resto], Valor, Acumulador ) :-
    (
        ((Peca == 'B' ; Peca == 'P'), valorDama(X,Y,Z), X2 is
Acumulador + 15, X3 is X2 + Z)
        ;
        ((Peca == 'O' ; Peca == '@'), valorPeca(X,Y,Z), X2 is
Acumulador + 10, X3 is X2 + Z)
    )
    ,
    processaLista( Resto, Valor, X3 ).

% Dama no rio e muito valiosa
valorDama(X,X,10).
% Dama noutra casa qualquer
valorDama(_,_,5).

% Peca na defesa, valor elevado
valorPeca(_,1,10).
valorPeca(_,8,10).
% Peca nas colunas 1 ou 8 tem poucos movimentos, valor minimo
valorPeca(1,_,0).
valorPeca(8,_,0).
% Peca em qualquer outra posicao
valorPeca(_,_,5).

/*****
*****/
/*    Predicado para gerar todas as jogadas validas de um determinado
jogador    */
/*****
*****/

geraJogadas( Jogador, Board, Lista ) :-
    pecas( Jogador, Peca, PecaDama ),
    findall( [Y,X], (getPeca( Y,X, Board, Pc), (Pc == Peca ; Pc ==
PecaDama )), Bag ),

```

```

        findall( [Y2,X2], (getPeca( Y2,X2, Board, '_' ), casaPreta( Y2,X2
    )), Bag2 ),
        geraJogadas( Jogador, Bag, Bag2, Board, Lista, [] ).

geraJogadas( _, [], _, _, Lista, Lista ) :- !.
geraJogadas( Jogador, [[ColIni, LinIni] | Resto ], Bag, Board, Lista,
Lst ) :-
    geraJogadas2( Jogador, ColIni, LinIni, Bag, Board, Lst2, [] ),
    append( Lst, Lst2, Lista2 ),
    geraJogadas( Jogador, Resto, Bag, Board, Lista, Lista2 ).

geraJogadas2( Jogador, _, _, [], _, Lista, Lista ) :- !.
geraJogadas2( Jogador, ColIni, LinIni, [[ColFim, LinFim] | Resto ],
Board, Lista, Lst ) :-
    pecas( Jogador, Peca, PecaDama ),
    ( getPeca( ColIni, LinIni, Board, Pc ), (
        (( Pc == Peca, X is ColFim - ColIni,
        2 >= abs(X), Y is LinFim - LinIni, 2 >= abs(Y)), (Jogador == 'brancas',
        LinIni > LinFim ; Jogador == 'pretas', LinFim > LinIni ) )
        ;
        Pc == PecaDama )
    ),
    validaJogada( Jogador, [ ColIni, LinIni, ColFim, LinFim ], Board,
NewBoard, _),
    append( Lst, [[ColIni, LinIni, ColFim, LinFim] - NewBoard], Lista2
),
    geraJogadas2( Jogador, ColIni, LinIni, Resto, Board, Lista, Lista2
)
    ;
    Lista2 = Lst,
    geraJogadas2( Jogador, ColIni, LinIni, Resto, Board, Lista, Lista2
).

pecas( 'brancas', 'O', 'B' ).
pecas( 'pretas', '@', 'P' ).

/*****
*****/
/* Predicado para verificar se uma determinada casa e uma casa valida
de jogo */
/*****
*****/

casaPreta( X, Y ) :- odd(X),odd(Y) ; even(X),even(Y).
even( X ) :- Y is X mod 2, Y == 0.
odd( X ) :- Y is X mod 2, Y \= 0.

```