**Faculdade de Engenharia da Universidade do Porto**

**Mestrado Integrado em Engenharia Informática e Computação**

**3º ano, 1º semestre**



Projeto 1 de Programação em Lógica:

Adaptoid

Relatório Final

**13 de novembro de 2016**

**Grupo Adaptoid\_1:**

|  |  |
| --- | --- |
| Andreia Cristina de Almeida Rodrigues | up201404691@fe.up.pt |
| Gonçalo da Mota Laranjeira Torres Leão | up201406036@fe.up.pt |

# **Resumo**

No âmbito da unidade curricular de Programação em Lógica, foi-nos proposto o desenvolvimento de um jogo de tabuleiro chamado Adaptoid, um jogo para dois jogadores sem a intervenção de nenhum fator sorte.

O principal objetivo deste projeto é a familiarização com a linguagem de programação Prolog, e com o principal paradigma que lhe está associado, o paradigma de programação em lógica.

Para cumprir com este objetivo, foi utilizado o sistema SICStus para desenvolvimento de programação em Prolog. Foi também necessário esforço, trabalho em equipa e paciência para produzir um trabalho que cumpra com os requisitos de qualidade esperados.

Como resultado, foi desenvolvido um jogo divertido e estimulante com interface em linha de comandos, apoiado numa implementação eficiente e robusta.

Quanto às principais conclusões, é do entendimento de ambos os membros do grupo que este trabalho permitiu-nos expandir os nossos horizontes enquantos programadores, ao ficarmos a conhecer uma nova linguagem de programação e, sobretudo, um novo paradigma de programação. Em geral, a equipa sente orgulho pelos frutos do seu trabalho e esforço.

# **Índice**

[**Resumo**](#_ydptfwstujeh)

[**Índice**](#_yel9vkuturga)

[**Introdução**](#_99xirpokf8gc)

[**O Jogo Adaptoid**](#_xx07k0wmuh35)

[**Informações gerais**](#_4aqrwdxw7cer)

[**Ações a realizar em cada turno**](#_juwl39ql977z)

[**Movimento e captura**](#_6a1u79807xy9)

[**Criação de um novo adaptoid**](#_91st0u6gl8v7)

[**Mutações dos adaptoids**](#_1fv1ia7thocq)

[**Alimentação dos adaptoids**](#_bple9hasdwjx)

[**Condições para terminar a partida**](#_rda4dhymig48)

[**Lógica de Jogo**](#_ure34j4g7qz7)

[**Representação do estado do jogo**](#_4xxcj1u8ul9)

[**Visualização do Tabuleiro**](#_5nxphosqnmim)

[**Execução das Jogadas**](#_4l70e7sl3pgz)

[**Movimentação de toids**](#_driya2l85d5z)

[**Adição de toids ao tabuleiro**](#_qlia6xsd8ov0)

[**Adição de pinças a toids**](#_qfwgxl6o9k3c)

[**Adição de pernas a toids**](#_jdh7689f68mf)

[**Atualizações do estado de jogo no final do turno**](#_dfkxoijkm56b)

[**Final do Jogo**](#_r1a51qu19d2k)

[**Jogada do Computador**](#_1elgss571gkc)

[**Listagem das Jogadas Válidas**](#_63j3cdkuveub)

[**Avaliação do Tabuleiro**](#_yz23iots68e7)

[**Interface com o Utilizador**](#_rhfgrxyro051)

[**Conclusões**](#_kx0d4dsbmad0)

[**Bibliografia**](#_fo921v6ldnbq)

[**Anexo 1: Arquitetura da programa**](#_ml3hfap4f173)

# **Introdução**

Como primeiro trabalho prático da unidade curricular de Programação em Lógica (PLOG), do primeiro semestre do terceiro ano do Mestrado em Engenharia Informática e Computação (MIEIC), foi-nos proposto o desenvolvimento de um jogo de tabuleiro onde não intervém nenhum fator de sorte, com interface através da linha de comandos.

Da gama de vinte jogos propostos pelos docentes, o grupo optou pelo Adaptoid pela sua simplicidade em termos de regras de jogo, por ser um jogo que promove o raciocínio lógico e por não se tornar aborrecido após algumas partidas, dada a sua vasta combinação de jogadas possíveis.

O maior objetivo deste trabalho prático foi a interiorização dos conteúdos abordados nas aulas teóricas e teórico-práticas acerca da linguagem de programação Prolog, e do principal paradigma que lhe está associado: o paradigma de programação em lógica. Este objetivo é de grande importância para ambos os membros do grupo, pois, dado o plano de estudos do MIEIC, estes ainda não foram muito exercitados com este novo paradigma, estando ambos mais habituados ao paradigma de programação imperativa, onde o programador descreve como o programa deve operar, especificando a sua sequência de ações e mudanças de estado. Por sua vez, o paradigma de programação em lógica herda o modelo da teoria e demonstração da lógica de 1ª ordem.

O seguimento deste relatório subdivide-se nas seguintes partes:

* **O Jogo Adaptoid**, onde é apresentada a história e as regras do jogo implementado.
* **Lógica do Jogo**, onde é descrita a implementação da lógica do jogo em Prolog. Tendo como foco os seguintes pontos:
  + Representação do Estado do Jogo.
  + Visualização do Tabuleiro.
  + Execução de Jogadas.
  + Final do Jogo.
  + Jogada do Computador.
* **Interface com o Utilizador**, onde é descrito o módulo com o utilizador em modo texto.
* **Conclusões**.
* **Bibliografia**.

# **O Jogo Adaptoid**

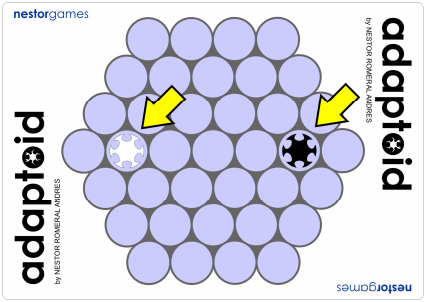
## Informações gerais

O jogo Adaptoid foi criado entre 2008 e 2009 por Néstor Romeral Andrés, um criador de jogos de tabuleiro espanhol [(«Néstor Romeral Andrés» 2016)](https://paperpile.com/c/lZgRxP/2tJ5). Trata-se de um jogo de tabuleiro para dois jogadores em que não existe a influência de nenhum fator sorte [(Andrés 2008; Sandler 2016; Valentine 2016)](https://paperpile.com/c/lZgRxP/8bhM+G8fs+bHVn).

O jogo é baseado em criaturas chamadas *adaptoids* que sofrem mutações sucessivas para se adaptarem melhor ao seu ambiente.

Cada jogador controla *adaptoids* de uma dada cor. Um jogador joga com peças brancas, o outro, com peças pretas. O jogo joga-se por turnos e ocorre num tabuleiro hexagonal de 37 casas (conferir a figura 1 abaixo para o esquema do tabuleiro).

No início da partida, cada jogador posiciona um *adaptoid* (sem pinças, nem pernas) no tabuleiro segundo a configuração apresentada na figura 1 abaixo. O jogador com as peças brancas é quem joga o primeiro turno.



*Figura 1: Configuração inicial do jogo  
(Fonte: Manual de instruções disponível em nestorgames.com)*

## Ações a realizar em cada turno

Em cada turno, o jogador ativo deve efetuar as seguintes ações, respeitando a ordem em que se apresentam:

* Se quiser e se for possível, pode mover um dos seus *adaptoids*. Este movimento pode gerar a captura de um *adaptoid*.
* Se quiser e se for possível, pode posicionar uma nova peça da sua cor no tabuleiro ou acrescentar a um dos seus *adaptoids* (pode ser ou não o que foi movido neste turno) uma pinça ou uma perna.

Ao terminar o turno, todos os *adaptoids* do adversário que não estão alimentados são capturados.

## Movimento e captura

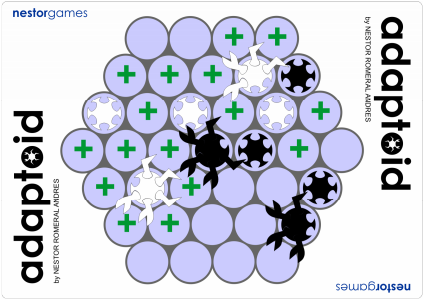
Em cada turno, um *adaptoid* pode-se mover para uma casa livre adjacente até tantas vezes quanto o número de pernas que possui. Assim, uma criatura sem pernas não pode efetuar nenhum movimento. A meio do movimento, não pode passar por casas ocupadas, mas pode terminar o movimento numa casa ocupada por um *adaptoid* inimigo.

Quando dois *adaptoids* inimigos se encontram na mesma casa, é capturado aquele que tiver o menor número de pinças. Em caso de empate, ambos as peças são capturadas.

Sempre que um *adaptoid* é capturado, o jogador adversário ganha um ponto.

## Criação de um novo *adaptoid*

Um novo *adaptoid* apenas pode ser posicionado no tabuleiro numa casa adjacente a outro *adaptoid* da mesma cor. Os novos *adaptoids* começam sempre sem pinças nem pernas.



*Figura 2: Posições (marcadas com uma cruz verde) onde é possível adicionar um “adaptoid” branco*

Cada jogador dispõe de apenas 12 peças do tipo *adaptoid*, pelo que só pode ter até esse número de criaturas no tabuleiro a qualquer momento. Quando um *adaptoid* é capturado, a peça correspondente pode ser novamente usada para criar um novo *adaptoid* mais à frente no jogo.

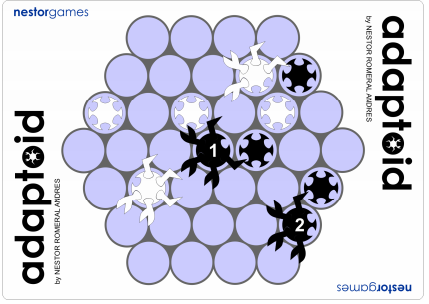
## Mutações dos *adaptoids*

Cada *adaptoid* pode ter até no máximo seis membros. Estes membros podem tanto ser pinças como pernas, e não há nenhuma imposição sobre a proporção entre estes dois tipos de membros. Assim, existem 28 estados em que um *adaptoid* se pode encontrar.

Tal para as peças do tipo *adaptoid*, cada jogador dispõe de apenas 12 pinças e 12 peças para poder usar nas suas peças. Quando um *adaptoid* é capturado, qualquer pinça ou perna que tinha volta para a posse do jogador correspondente, e poderá de novo ser usada.

## Alimentação dos *adaptoids*

Para se manterem vivos, os *adaptoids* têm de ser constantemente alimentados. Um *adaptoid* está alimentado se o número de casas livres adjacentes à sua posição é maior ou igual ao seu número de membros. Assim, quanto mais membros tiver uma criatura, mais são os seus requisitos para estar alimentado.



*Figura 3: Uma configuração possível do jogo para ilustrar o conceito de um “adaptoid” estar alimentado*

Considere-se a figura acima e que vai terminar o turno do jogador das peças brancas. O *adaptoid* preto marcado com um “1” tem 5 membros (3 pinças e 2 pernas) e tem apenas 4 casas livres à sua volta, pelo que é capturado. O mesmo ocorre com a criatura marcada com um “2” que tem 4 membros (3 pinças e 1 perna) mas apenas 3 casas livres à sua volta. As três restantes peças pretas nunca poderiam ser capturadas, independentemente do estado das suas casas vizinhas, pois não têm membros. Assim, no final do turno do jogador das peças brancas, este ganha 2 pontos, pois efetuou 2 capturas.

## Condições para terminar a partida

O objetivo do jogo é ganhar 5 pontos, o que é possível capturando 5 *adaptoids* do jogador adversário. O jogador que atingir este objetivo primeiro vence a partida. Além disso, um jogador perde, se, a um dado momento da partida, não tem nenhum *adaptoid* da sua cor no tabuleiro. Em caso de empate (se um *adaptoid* se mover para uma casa com uma peça inimiga com o mesmo número de pinças, e isto permitir a ambos jogadores atingir os 5 pontos), vence quem jogou mais recentemente.

# **Lógica de Jogo**

Nota: na escolha de nomes de predicados e de variáveis, e em comentários, para ser mais conciso, referimo-nos frequentemente a *adaptoids* como sendo *toids*.

## Representação do estado do jogo

Para representar o jogo, recorremos ao predicado game/5 que tem como campos, em ordem:

* o número do turno,
* a cor (identificador) do jogador ativo (0 - jogador branco, 1- jogador preto),
* uma lista com os dados de cada jogador
* o tabuleiro atual (uma lista de listas).
* os dados relativos à última jogada, sob a forma de um par de termos, onde cada um descreve a decisão tomada para cada uma das ações descritas na secção 2.2., respetivamente. Por exemplo: se o jogador não mover uma peça e depois adicionar um *toid* à casa na primeira linha, segunda coluna, obtemos o par: none-addToid(1-2).

game(+TurnNo,+ActivePlayerColor,+Players,+Board,+LastFirstMove-LastSecondMove).

Para representar um jogador, recorremos ao predicado que tem como campos, em ordem:

* a pontuação,
* o número de peças de *toids*,
* o número de peças de pinças,
* o número de peças de pernas.

player(+Points,+Toids,+Pincers,+Legs)

Para o tabuleiro, usamos uma lista de listas em que cada elemento de uma sub-lista representa uma célula, que pode estar vazia (none) ou conter um *adaptoid* (cujos dados estão contidos num predicado *toid*). Quanto à indexação de elementos do tabuleiro, a primeira coordenada corresponde ao índice da linha (cuja numeração aumenta de cima para baixo), e a segunda corresponde ao índice da coluna (cuja numeração aumenta da esquerda para a direita). O elemento do canto superior esquerdo é [1,1] e não [0,0]. Dependendo da linha, dada a forma hexagonal do tabuleiro, o índice da coluna começa sempre em 1 mas o seu valor máximo varia entre 4 e 7.

Para representar um *adaptoid*, recorre-se ao predicado toid(+Color,+Pincers,+Legs) que tem como campos, em ordem:

* a sua cor (0 para branco, 1 para preto),
* o número de pinças,
* o número de pernas.

Apresenta-se abaixo a representação em Prolog do estado inicial de jogo (correspondente à figura 1).

game(1, 0, [player(0, 11, 12, 12), player(0, 11, 12, 12)],

[[none, none, none, none],

[none, none, none, none, none],

[none, none, none, none, none, none],

[none, toid(0,0,0), none, none, none, toid(1,0,0), none],

[none, none, none, none, none, none],

[none, none, none, none, none],

[none, none, none, none]],

none-none)

De seguida, apresenta-se um possível estado intermédio (correspondente às figuras 2 e 3).

game(20, 1, [player(2, 7, 8, 7), player(4, 7, 6, 9)],

[[none, none, none, none],

[none, none, none, toid(0,2,2), toid(1,0,0)],

[toid(0,0,0), none, toid(0,0,0), none, toid(0,0,0), none],

[none, none, none, toid(1,3,2), toid(1,0,0), none, none],

[none, toid(0,2,3), none, none, none, toid(1,0,0)],

[none, none, none, none, toid(1,3,1)],

[none, none, none, none]],

moveToid(4-2,5-2)-addLeg(5-2))

Por fim, apresenta-se um possível estado final do jogo (obtido efetuando uma jogada sobre o jogo em estado intermédio apresentado acima).

game(21, 0, [player(3, 8, 10, 10), player(5, 8, 9, 10)],

[[none, none, none, none],

[none, none, none, toid(0,2,2), toid(1,0,0)],

[toid(0,0,0), none, toid(0,0,0), none, toid(0,0,0), none],

[none, none, none, none, toid(1,0,0), none, none],

[none, toid(1,3,2), none, none, none, toid(1,0,0)],

[none, none, none, none, none],

[none, none, none, none]],

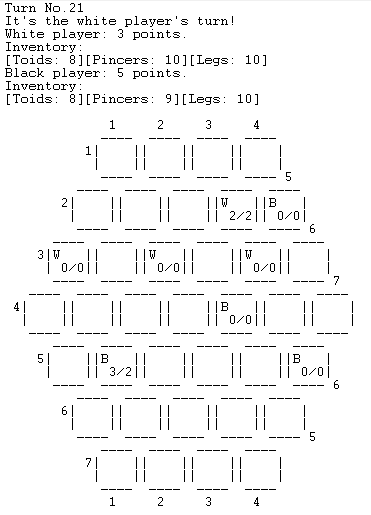
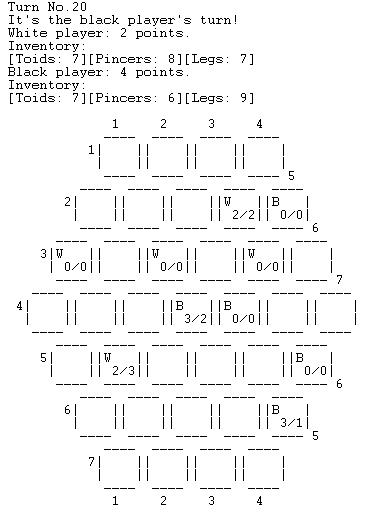
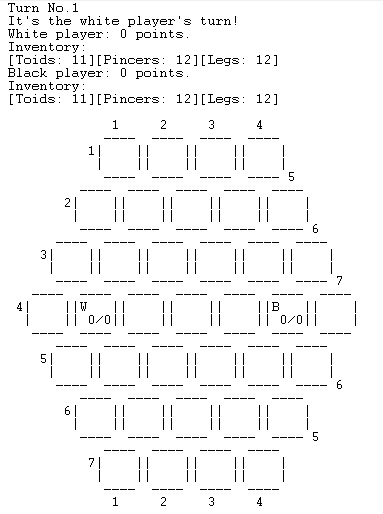
moveToid(4-4,5-2)-none)

## Visualização do Tabuleiro

O predicado display\_game(+Game) em Adaptoid.pl, permite visualizar o turno atual, as estatísticas de ambos os jogadores (pontos e inventário com o número de diferentes peças disponíveis) e o tabuleiro.

No nosso *interface* em modo de texto, um *adaptoid* é representado pelo caracter que representa a cor da peça (W para peças brancas e B para peças pretas) e por dois números que representam respetivamente o número de pinças e o número de pernas que esse *adaptoid* tem no momento.

Apresenta-se de seguida o código das estruturas de dados para visualização de três tabuleiros dados como exemplo na secção anterior (estado inicial do jogo, estado intermédio, e final do jogo).



*Figuras 4, 5 e 6: Visualização dos três tabuleiros de exemplo para o estado inicial, intermédio e final, respetivamente*

## Execução das Jogadas

Cada jogada é composta por duas partes. A primeira parte consiste em mover ou não um *toid.* A segunda parte consiste em colocar um novo *toid* no tabuleiro, adicionar uma pinça ou perna a um *toid* já existente ou não fazer nada.

### **Movimentação de *toids***

Para mover um *toid*, é utilizado o predicado moveToid(+Game, +Line-Col, +DestLine-DestCol, -NewGame). Este predicado é usado para simultaneamente testar se o movimento é legal e, em caso afirmativo, executá-lo, devolvendo em NewGame o novo estado de jogo.

Para testar se o movimento é legal, é testado:

1. se existe um toid nessas mesmas coordenadas, com o predicado isToid(+Elem)
2. se a cor desse *toid* corresponde à cor do jogador ativo, obtendo a cor do jogador ativo e comparando com a cor do toid em questão
3. se o número de pernas desse *toid* permite que ele se mova para as coordenadas de destino, com o predicado canMove(+Game,+Line-Col,:DestLine-DestCol,+Legs)
4. se a casa de destino é capturável (está vazia ou tem peça inimiga), com o predicado toidCapture(+Game,+Elem,+DestElem,-Winner,-NewGame).

Se algum destes predicados intermédios falhar, o movimento não é efetuado. Se nada falhar, o movimento é executado, efetuando as seguintes ações:

1. é capturada a casa de destino usando o predicado toidCapture(+Game,+Elem,+DestElem,-Winner,-NewGame), que recebe o conteúdo da casa de destino, retorna em Winner a peça que venceu a luta (ou none, caso as peças tenham o mesmo número de pinças), e em NewGame o jogo tendo devolvido aos jogadores derrotados as suas peças de *toid*, pinças e pernas.
2. é removida a peça da sua posição antiga usando o predicado setBoardElem(+Game,+Line-Col,+NewElem,-NewGame).
3. é posta na casa de destino a peça “vencedora” da invocação de toidCapture no passo 1.

### **Adição de *toids* ao tabuleiro**

Para adicionar *toids* ao tabuleiro, é utilizado o predicado addToid(+Game, +Line-Col, -NewGame), que testa, pela seguinte ordem:

1. se não existe um toid nessas mesmas coordenadas, com a negação do predicado isToid(+Elem)
2. se o jogador ativo tem peças no seu inventário, com o predicado hasToidPieces(+Player)
3. se existe um toid nas coordenadas adjacentes às coordenadas inseridas, o que permite a sua validação. As coordenadas adjacentes são obtidas através do predicado areAdjacent(+Line-Col,-Line2-Col2)

Em caso de sucesso, são efetuadas as seguintes ações:

1. é posto um *toid* na posição especificada.
2. é decrementado o número de peças de tipo *toid* do jogador ativo, com o predicado decToidPieces(+Player,-NewPlayer).
3. são mortos os *toids* inimigos que não estão alimentados, com o predicado killHungryToids(+Game,-NewGame).

### **Adição de pinças a *toids***

Para adicionar pinças a um *toid*, é usado o predicado addPincer(+Game, +Line-Col, -NewGame), que testa, pela seguinte ordem:

1. se existe um *toid* nessas mesmas coordenadas, com o predicado isToid(+Elem).
2. se esse *toid* têm espaço para adicionar o novo membro, com o predicado hasFreeSlots(+Elem).
3. se esse *toid* é da mesma cor que o jogador ativo.
4. se o jogador ativo têm pinças no seu inventário, com o predicado hasPincerPieces(+Player)

Em caso de sucesso, são efetuadas as seguintes ações:

1. é atualizado o *toid* especificado, com o predicado incPincers(+Elem,-NewElem).
2. é decrementado o número de peças de tipo pinça do jogador ativo, com o predicado decPincerPieces(+Player,-NewPlayer).
3. são mortos os *toids* inimigos que não estão alimentados, com o predicado killHungryToids(+Game,-NewGame).

### **Adição de pernas a *toids***

Para adicionar pernas a um *toid*, é usado o predicado addLeg(+Game, +Line-Col, -NewGame). A sua lógica de implementação é semelhante à do addPincer/3.

### **Atualizações do estado de jogo no final do turno**

No final de cada turno, ou seja, o número de turno é incrementado com o predicado incTurnNo(+Game,-NewGame) e o jogador ativo é alterado com o predicado switchActivePlayer(+Game,-NewGame).

## Final do Jogo

O final do jogo é testado após cada turno pelo predicado testEnd(+Winner) que por sua vez chama o predicado gameOver(+Game, -Winner), que testa todas as situações possíveis de terminação do jogo:

* um dos jogadores tem pelo menos 5 pontos (o número de pontos é obtido com o predicado getPoints(+Player,-Points)) e mais pontos que adversário. Nesse caso, vence o jogador com mais pontos.
* ambos os jogadores têm a mesma pontuação que ultrapassa os 5 pontos. Segundo as regras vence o jogador ativo.
* um dos jogadores não tem *toids* no tabuleiro (o número de peças de *toid* no seu inventário, obtido usando getToidPieces(+Player,-Toids), é igual a 12). Nesse caso, o adversário vence.

## Jogada do Computador

Para a jogada do computador, foram implementados dois tipos de *bots*: *random bot* que executa jogadas aleatórias de entre as jogadas possíveis, e *greedy bot* que executa a jogada possível que mais o favorece naquele momento.

Para o jogador *random*, o predicado que escolhe a nova jogada é o pickRandomNewGame(+Game,-NewGame) e para o jogador *greedy* quem escolhe a nova jogada é o predicado pickGreedyNewGame(+Game,-NewGame).

O jogador *random* começa por obter a lista de todas as jogadas válidas (todos os estados do tabuleiro possíveis) e escolhe uma aleatoriamente (primeiro escolhe aleatoriamente a primeira jogada, e, usando o jogo obtido, escolhe aleatoriamente a segunda jogada).

Para o jogador *greedy*, também se começa por obter a lista de todas as jogadas válidas (todos os estados do tabuleiro possíveis) mas depois é utilizado o predicado getAllBestNewGames(+AllValidNewGames,-AllBestNewGames). Este predicado recebe todos os tabuleiros válidos e cria uma nova lista com todos aqueles que o mais beneficiam (que têm maior valor) recorrendo a um predicado de avaliação do tabuleiro, value(+Game,-Value). No final, o computador escolhe aleatoriamente um dos tabuleiros mais interessantes. Tal como para o jogador *random* é escolhido primeiro o tabuleiro mais interessante tendo apenas eventualmente movido uma peça (primeira jogada). De seguida, é usado o novo jogo para escolher a melhor segunda jogada.

### **Listagem das Jogadas Válidas**

Para ambos estes predicados, como já foi referido, é necessário obter uma lista de todas as jogadas válidas para o jogador ativo.

Isto é feito recorrendo aos predicados getAllValidFirstMoveNewGames(+Game,-AllValidFirstMoveNewGames) e getAllValidSecondMoveNewGames(+Game,-AllValidSecondMoveNewGames) que devolvem listas com todos os estados do tabuleiro possíveis, após a primeira jogada e após a segunda respetivamente.

O modo de funcionamento destes predicados é semelhante: é utilizado o predicado findall(+Template,:Goal,-Bag) para cada tipo de movimento válido na primeira e segunda parte da jogada. Neste caso, os argumentos do findall têm o seguinte significado:

* o Template é um novo estado de jogo,
* o Goal consiste em obter todas as casas válidos do tabuleiro, usando getAllValidCells(-List), seguido da unificação de um par de coordenadas, Line-Col, com um elemento desta lista usando member/2. Por fim, é testado se o movimento, aplicado sobre Line-Col é válido. Para o caso particular do moveToid, não é necessário especificar também a posição de destino pois o predicado canMove(+Game,+Line-Col,:DestLine-DestCol,+NumMoves) unifica com cada uma das posições válidas, visto que o teste de adjacência entre casas envolve gerar todas as posições adjacentes a uma casa, seguida do teste que a outra posição está incluída nessa lista. Isto permite podar de forma significativa a árvore de pesquisa no findall.
* Bag é a lista com todos os estados de jogo válidos.

### **Avaliação do Tabuleiro**

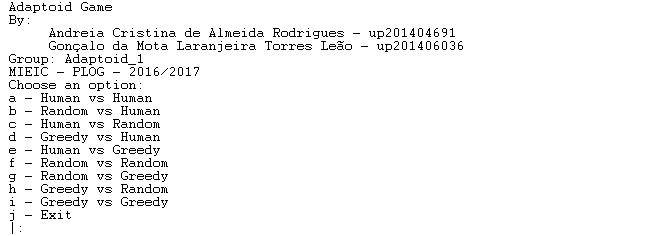
O predicado value(+Game,-Value) tem como objetivo avaliar um jogo, do ponto de vista do jogador ativo. Quanto maior for o valor atribuído ao jogo, mais interessante é de efetuar a jogada que conduz a esse novo estado de jogo. Para auxiliar esta avaliação, definiram-se predicados que incrementam/decrementam o valor do tabuleiro segundo critérios específicos:

* valueGameOver(Game, OldValue, Value) vai testar se a jogada em questão vai permitir acabar o jogo. Se o novo estado corresponder a uma vitória para o jogador ativo, Value será igual a (OldValue + 100). Se corresponder a uma derrota, Value será igual a (OldValue - 100). Se o jogo não acabar com a jogada em questão, o valor não é alterado. Por ser uma condição crítica e de prioridade máxima sobre todos os outros critérios de avaliação do tabuleiro, o valor é modificado usando uma constante (100) que supera qualquer outra combinação de constantes que alteram o valor do tabuleiro, para obrigar o *bot* a tomar a decisão de vencer a partida, ou então de evitar que perca.
* valuePoints(+Game, +OldValue, -Value) vai alterar o valor baseando-se na diferença de pontuação dos dois jogadores. Se a jogada em questão não influenciar em nada as pontuações de ambos os jogadores, o valor não é alterado. Se influenciar, o valor vai ser alterado usando a fórmula Value = OldValue + 20\*(ActivePlayerScore - EnemyPlayerScore).
* valueUnfedActiveToids(+Game, +OldValue, -Value) vai alterar o valor baseando-se na quantidade de *toids* do jogador ativo que não estão alimentados no final da sua jogada. Ter peças por alimentar no final do turno do jogador à qual as peças pertencem não traz nenhuma consequência negativa. Contudo, no final do turno do adversário, qualquer peça não alimentada é eliminada, e o adversário recebe um ponto por cada peça removida do tabuleiro. Assim, é de conveniência para o jogador ativo que não deixe peças suas por alimentar no final do seu turno. O novo valor do tabuleiro, Value, é calculado usando a fórmula Value = OldValue - número de peças do jogador ativo não alimentadas. O número de peças por alimentar é calculado usando o predicado countUnfedActiveToids(+Game,-NumUnfed).

# **Interface com o Utilizador**

O jogo é iniciado com o predicado adaptoid/1, que inicializa o gerador de número aleatórios (para implementar a escolha arbitrária de jogadas para o computador) usando o predicado setRandomSeed e, usando um ciclo repeat, vai exibindo o menu principal e lendo da consola a opção do utilizador.

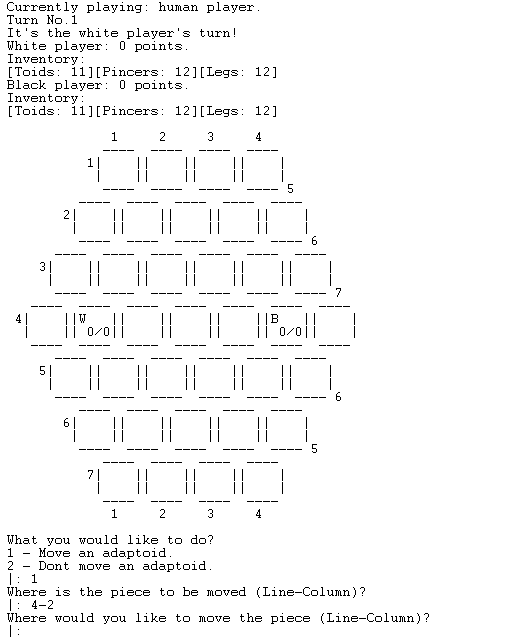
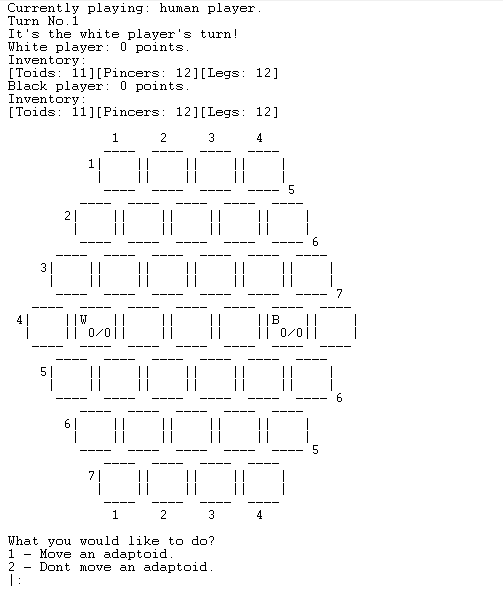
Graças aos predicados getChar/1, getInt/1 e clearBuffer/0 em IOUtil.pl, o *interface* de jogo é mais intuitivo que o *interface* por omissão pois não é necessário escrever um ponto (‘.’) no final de cada *input*.



*Figura 7: Menu principal do jogo*

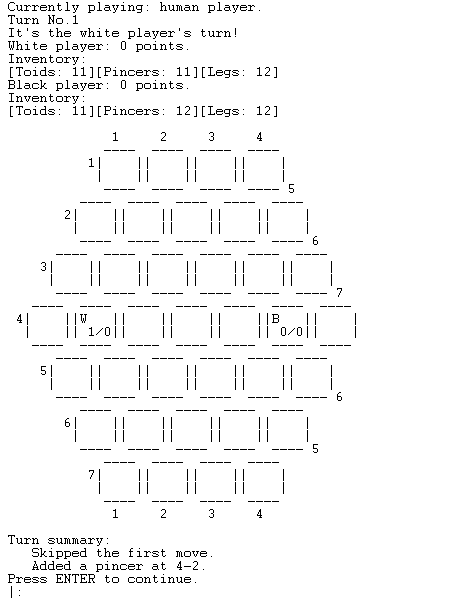
A inicialização de um jogo é feita pelo predicado playGame(+Player1-Player2) que tem como campos os dois jogadores. Tal como apresentado na figura 7, há nove modos de jogo que consistem em todas as combinações de jogadores humanos e jogadores de computador *random* e *greedy* a ocupar os lugares de Player1 e Player2 . A ordem com que os jogadores são introduzidos no predicado playGame/2 determina que tipo de jogador joga o primeiro turno.

Em cada turno, é pedido aos jogadores humanos o que pretendem fazer em cada uma das duas metades do turno (mover ou não uma peça, seguido de adicionar um *toid*, pinça ou perna, ou não fazer nada). Para isso, primeiro, o jogador especifica qual das ações pretende efetuar e depois insere os pares de coordenadas relevantes para a jogada. Cada par de coordenadas deve ser separado por um ‘-’. Caso o *input* tenha um formato inválido ou a jogada especificada seja ilegal, é perguntado novamente ao jogador o que pretende fazer nessa parte da jogada.



*Figuras 8 e 9: Escolha da primeira ação de um turno* e especificação das coordenadas para mover um toid.

No final de cada turno, é mostrado o estado do jogo uma última vez e é esperado que seja premida a tecla “ENTER”. Além disso, é também apresentado um sumário dos movimentos que foram efetuados nesse turno. Isto é essencial para o caso de se jogar com jogadores de computador para se poder acompanhar as suas jogadas.



*Figura 10: Sumário do final de um turno*

Por fim, quando for detectado o final de uma partida, é anunciado o vencedor do jogo e o programa aguarda que se pressione a tecla “ENTER”. De seguida, o programa volta ao menu principal (figura 7).

# **Conclusões**

Este primeiro trabalho prático de Programação em Lógica exigiu bastante esforço, tempo, paciência e trabalho de equipa para cumprir com as suas metas. Houve partes do projeto que se revelaram particularmente desafiantes, nomeadamente, a elaboração do predicado para determinar se duas casas são adjacentes (dado o formato hexagonal do tabuleiro), a implementação das jogadas do computador e o desenvolvimento do interface por linha de comandos (dado o paradigma não-imperativo do Prolog).

Em termos de possíveis melhorias, seria possível implementar algoritmos mais sofisticados para determinar qual a jogada mais interessante de efetuar, dado que uma abordagem *greedy* nem sempre leva à solução ótima, dado que opta sempre pela solução que dá o maior ganho imediato (a curto prazo), sem ter em conta os efeitos de uma escolha a longo prazo.

O grupo considera que foram cumpridos os objetivos deste trabalho, dado que ambos os membros da equipa estão agora mais familiarizados com a linguagem Prolog e o seu paradigma subjacente: o paradigma de programação em lógica.

Como resultado do trabalho destas últimas semanas, foi produzido um programa simples e apelativo que implementa o jogo Adaptoid, um jogo estimulante que promove uma boa prática mental e serve como um excelente passatempo.

# **Bibliografia**

[Andrés, Néstor Romeral. 2008. «Adaptoid». nestorgames.](http://paperpile.com/b/lZgRxP/8bhM) <http://nestorgames.com/rulebooks/ADAPTOID_EN.pdf>[.](http://paperpile.com/b/lZgRxP/8bhM)

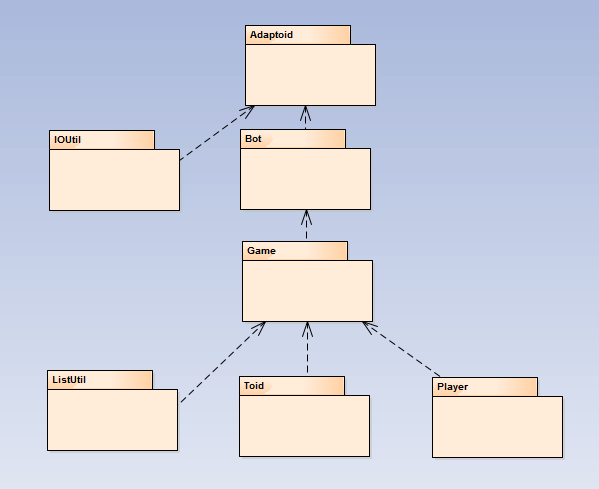
[«Néstor Romeral Andrés». 2016. *boardgamegeek.com*. Acedido Outubro 7.](http://paperpile.com/b/lZgRxP/2tJ5) <https://boardgamegeek.com/boardgamedesigner/9393/nestor-romeral-andres>[.](http://paperpile.com/b/lZgRxP/2tJ5)

[Sandler, Arty. 2016. «Adaptoid». *www.iggamecenter.com*. Acedido Outubro 7.](http://paperpile.com/b/lZgRxP/G8fs) <http://www.iggamecenter.com/info/en/adaptoid.html>[.](http://paperpile.com/b/lZgRxP/G8fs)

[Valentine, Lee. 2016. «Adaptoid». *www.ogrecave.com*. Acedido Outubro 7.](http://paperpile.com/b/lZgRxP/bHVn) <http://www.ogrecave.com/reviews/adaptoid.shtml>[.](http://paperpile.com/b/lZgRxP/bHVn)

# **Anexo 1: Arquitetura da programa**

No diagrama abaixo, pode-se ver o diagrama UML de packages da programa, onde cada package representa um módulo/bloco funcional (associado a um ficheiro .pl). As setas representam as dependências entre módulos.



*Figura 7: Diagrama UML de “packages” do programa*

A tabela seguinte descreve a responsabilidade de cada módulo.

|  |  |
| --- | --- |
| **Módulo** | **Descrição** |
| ListUtil | Módulo com predicados utilitários para aceder e alterar listas uni e bidimensionais. |
| Toid | Módulo com predicados para a manipulação e acesso a objetos da estrutura de dados toid/3. |
| Player | Módulo com predicados para a manipulação e acesso a objetos da estrutura de dados player/4. |
| Game | Módulo com predicados para a manipulação e acesso a objetos da estrutura de dados game/5. |
| Bot | Módulo que implementa as jogadas de computador. |
| IOUtil | Módulo com predicados utilitários para a escrita e leitura da consola. |
| Adaptoid | Módulo de *user interface*, com predicados de alto nível que interagem com o utilizador para jogar o jogo Adaptoid |

*Tabela 1 - Responsabilidade de cada módulo da aplicação*