Asterismo

Relatório Final



Mestrado Integrado em Engenharia Informática e de Computação

Programação em Lógica

Grupo Asterismo 2:

José António Barbosa Fonseca Guerra – up201706421@fe.up.pt

Martim Pinto da Silva - up201705205@fe.up.pt

Faculdade de Engenharia da Universidade do Porto

Rua Roberto Frias, sn, 4200-465 Porto, Portugal

14 De Novembro de 2019

**Resumo**

Este projeto consiste na conceção de um jogo de tabuleiro chamado Asterismo realizado através de um conjunto de regras utilizando a linguagem de programação denominada Prolog.

O jogo escolhido foi um jogo cooperativo de 2 jogadores havendo a possibilidade de se jogar com dois jogadores reais, 1 jogador real e 1 jogador virtual ou 2 jogadores virtuais (2 máquinas). Cada um dos jogadores virtuais tem diferentes níveis de dificuldade (AI) até um nível máximo de 2.

Todas as regras e modos foram implementadas com sucesso para a entrega deste projeto.

Este projeto permitiu uma aprendizagem de forma eficaz para a realização do trabalho, baseando-se muito na pesquisa e na motivação pela parte prática da linguagem de Prolog.

O que se pôde verificar, logo desde o início da utilização desta ferramenta, foi a rapidez e eficácia da realização de qualquer tarefa, nomeadamente problemas de decisão.

Resumindo, foi realizado um jogo complexo numa linguagem nova e completamente diferente do que estávamos habituados mas que há medida do tempo, foi se tornando mais desafiadora e interessante até á final da realização do trabalho. Gostaríamos de ter melhorado ainda mais a parte de interface gráfica, mas sabemos que esse trabalho será desenvolvido mais na unidade curricular de LAIG.

**Índice**

* Resumo….…………………………………………………………………2
* Introdução…………………………………………………………………4
* Jogo Asterismo………………………………………………………....5
* Lógica do Jogo……………………………………………………….....7
  + Representação interna do estado do jogo ……….7
  + Visualização do tabuleiro ……………………………….10
  + Detalhes de implementação ……………………….....11
  + Lista de jogadas Válidas ………………………………….12
  + Execução de jogadas………………………………………..15
  + Final do Jogo……………………………………………………17
  + Avaliação do Tabuleiro…………………………………….20
  + Jogada do Computador…………………………………….21
* Conclusões…………………………………………………………………22

**Introdução**

Este projeto foi desenvolvido no âmbito da unidade curricular de Programação em Logica (PLOG), do 3º ano do curso Mestrado Integrado em Engenharia Informática da FEUP. O objetivo deste projeto foi implementar, através do sistema de desenvolvimento do SICSTUS Prolog, um jogo de tabuleiro com um conjunto de regras pré-definidas pelo criador do jogo.

O relatório segue a seguinte estrutura:

* Jogo Asterismo: Descrição do jogo e principalmente das suas regras.
* Lógica do Jogo: Implementação da lógica do jogo em Prolog, tendo a seguinte estrutura:
  + Representação interna do estado do jogo: Descrição da estrutura utilizada para guardar o estado inicial, intermédio e final do jogo
  + Visualização do tabuleiro: Exposição da interface com o utilizador, descrevendo a visualização do estado do jogo e robustez da interface/validações de entrada.
  + Lista de jogadas válidas: Descrição dos predicados utilizados para a obtenção das jogadas válidas.
  + Execução de jogadas: Exposição dos predicados utilizados para fazer com que o ciclo do jogo funcione.
  + Final do jogo: Descrição dos predicados que fazem a verificação do final do jogo com identificação do vencedor.
  + Avaliação do tabuleiro: Descrição dos predicados que permitem comparar a aplicação de diversas jogadas disponíveis.
  + Jogada do computador: Descrição dos predicados que fazem com que o computador escolha uma jogada válida durante o ciclo de jogo bem como a explicação da implementação das dificuldades 0 e 1 do computador.
* Conclusões: Conclusões do trabalho e opiniões acerca do que poderia ter sido feito para melhorar o trabalho desenvolvido.

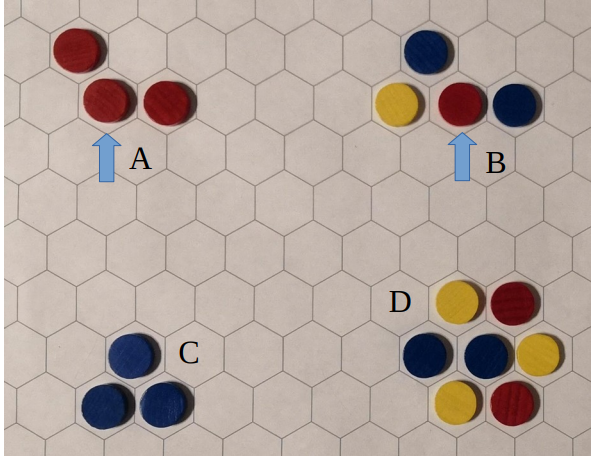
**Jogo Asterismo**

# 

Este jogo de tabuleiro foi criado em 2019 por Giuliano Polverari.

É um **jogo cooperativo** para ser jogado entre 2 pessoas num tabuleiro com peças hexagonais. Há 63 peças de 3 cores diferentes (vermelho, amarelo e azul).

O objetivo principal dos jogadores é obter 5 peças de cada cor cada um, **sem perturbar o equilíbrio da árvore de jogo.**



**Figura 2**: Explicação regras

Regras:

Qualquer peça pode ser retirada, no entanto as peças adjacentes têm de estar **seguras**.

Uma peça está **segura** se estiver ligada a duas peças da mesma cor (situação A) ou ligada a 3 peças de qualquer cor (situação B).

Cada jogador por turno faz apenas uma jogada, ou seja remove 1 peça, se possível.

Cada jogador tem um limite máximo de 5 peças para cada cor.

Ao retirarem peças, os jogadores têm de verificar se **não quebram a árvore**, isto é ao retirar uma peça do tabuleiro não fique com dois ou mais conjuntos de peças separados uns dos outros.

Se os 2 jogadores chegarem as 5 peças de cada cor vencem “o jogo”, senão “o jogo” vence e os jogadores saem derrotados.

**Figura 1**: tabuleiro do jogo

Prémios:

* 2º lugar Cogita 2019 ("melhor jogo abstrato de estratégia”) 5ª edição.



Situação inicial:

As peças são colocadas aleatoriamente pelo tabuleiro.

As peças, que devido a configuração inicial não estejam **seguras**, terão de ser atribuídas a cada jogador da forma aleatória em igual número.

**Figura 3**: Situação inicial

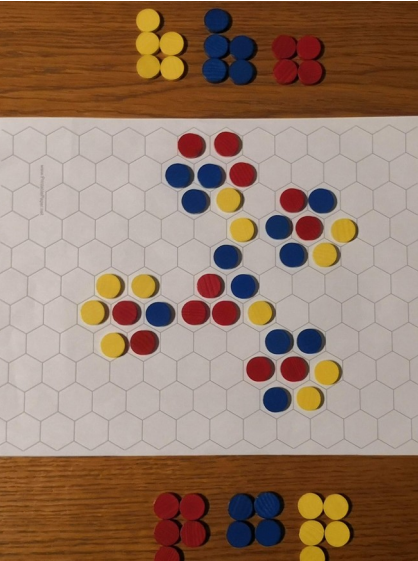


Desenvolvimento:

Vai-se retirando e colocando peças, respeitando as regras de jogo, tendo como objetivo principal acumular 5 peças de cada cor.

Os 2 jogadores jogam cooperativamente, ou seja, têm de acumular os dois 5 peças de cada cor para alcançar a vitória.

O jogador terá de ter especial atenção para não quebrar a árvore do jogo, isto é não poderá haver blocos sem peças adjacentes.



**Figura 5**: Situação final

Situação final:

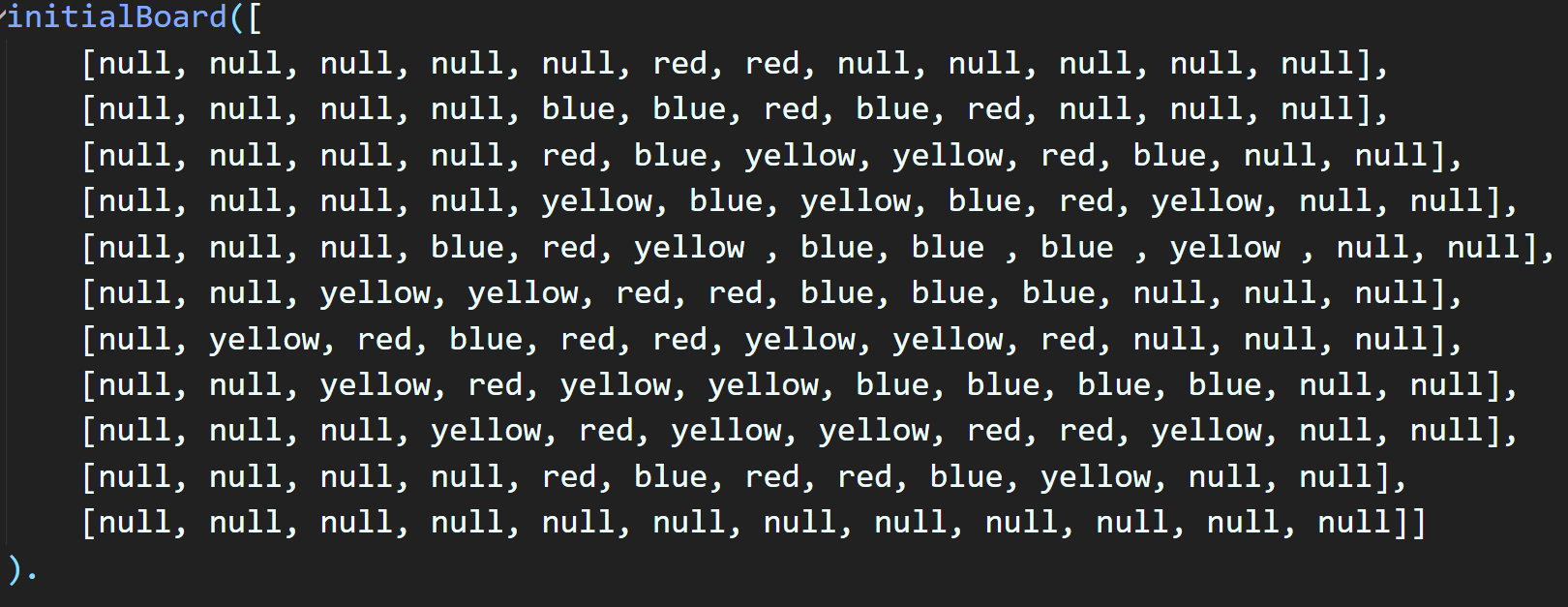
O jogador terá de estar atento a situações em que possui 14 peças, assim como o seu colega de equipa.

Nestas situações, é provável que não seja possível retirar mais peças sem quebrar a árvore do jogo. Nesta condição perdem os jogadores e ganha “o jogo”, tal como neste exemplo. Se por outro lado for possível cada 1 alcançar as 5 peças de cada cor os 2 jogadores vencem “o jogo”.

**Figura 4**: Situação desenvolvimento

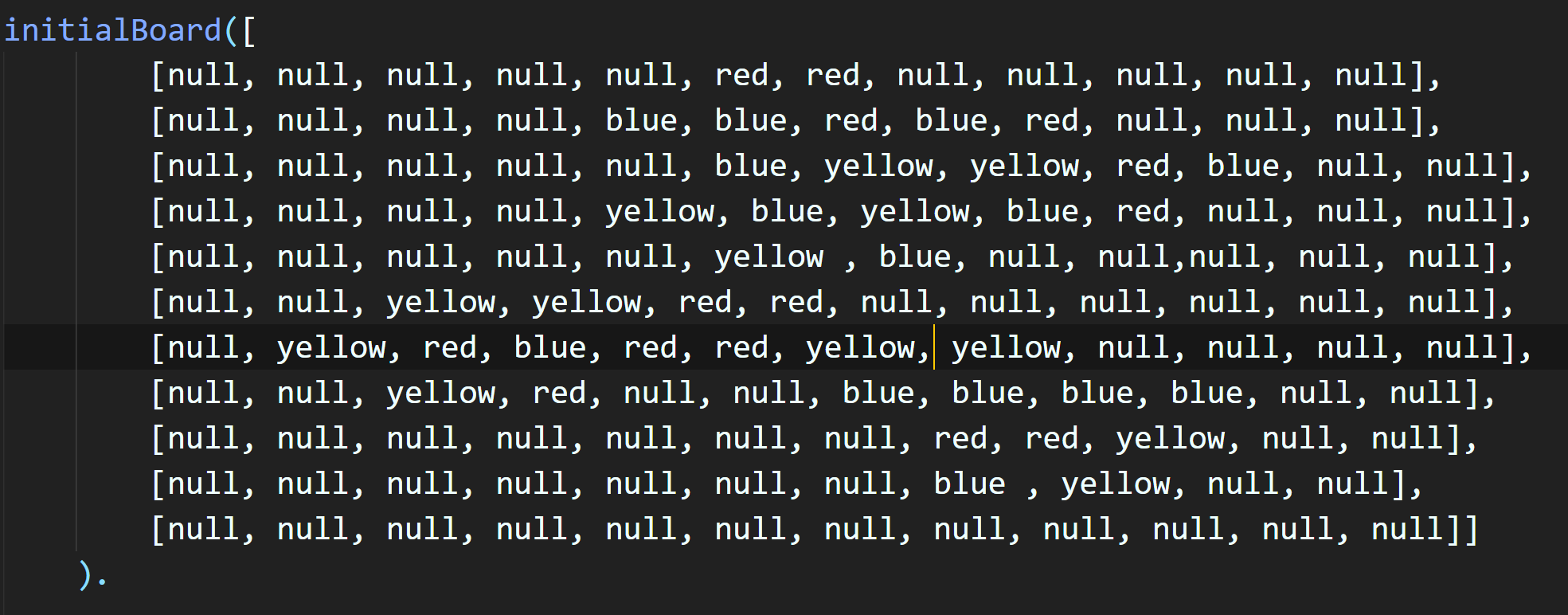
**Lógica do Jogo**

**Representação interna do estado do jogo**

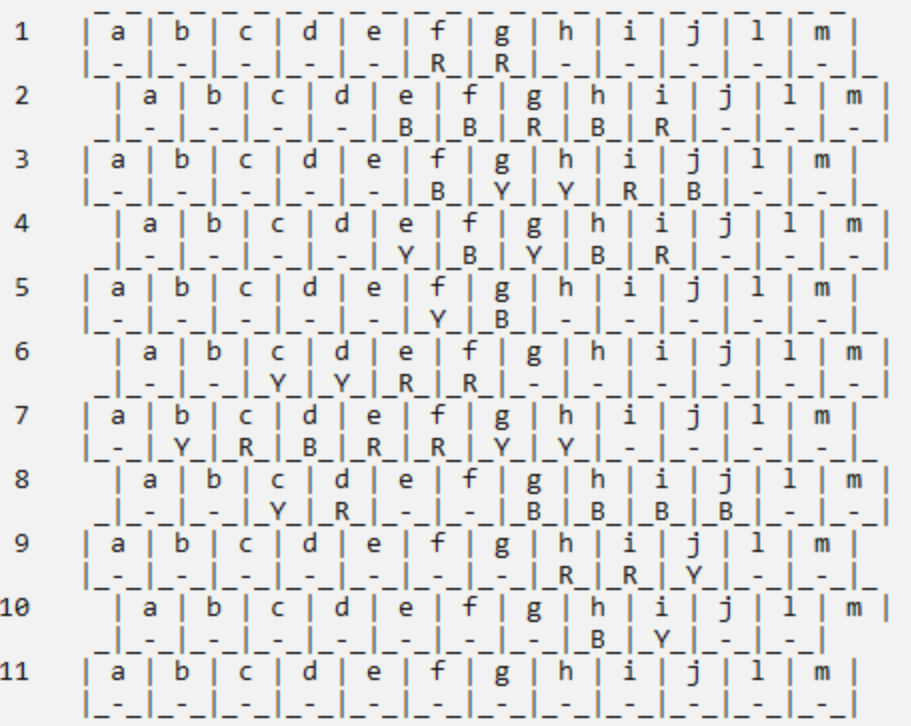
**Situação inicial**

# 

**Figura 6 e 7**: Situação inicial

**Desenvolvimento:**

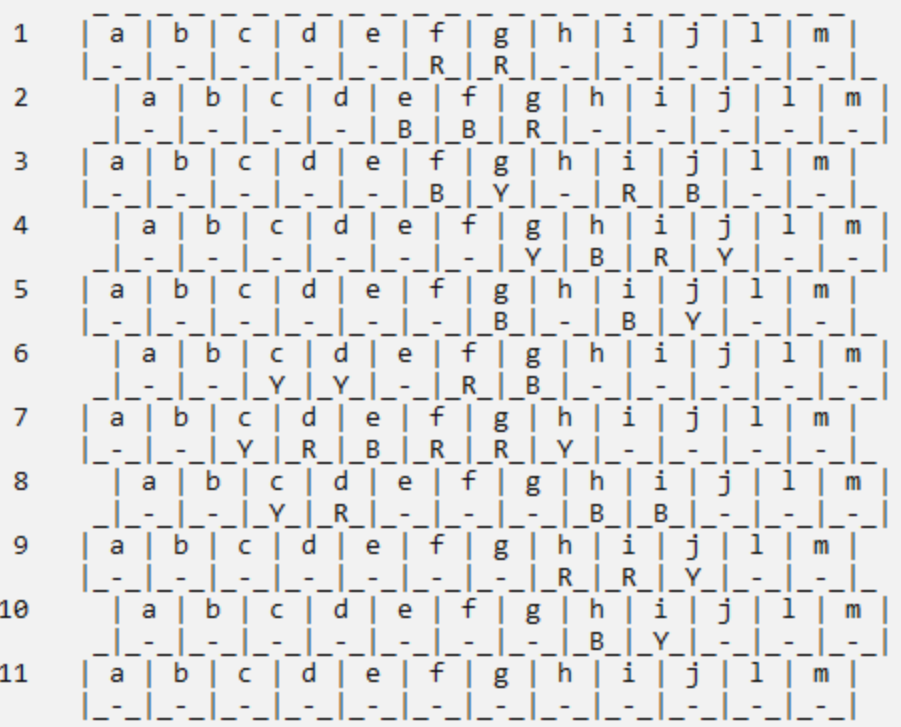
**Figura 8 e 9**: Situação intermédia



**Situação final:**

# 

**Figura 10 e 11**: Situação final



**Visualização do tabuleiro**

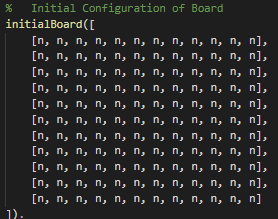
# 

**Figura 12**: predicados responsáveis pelo display do tabuleiro

**Detalhes de Implementação**

É necessário, antes de proceder com o relatório, partilhar alguns detalhes de implementação que são necessários para compreende-lo.

No que toca ao Board (Tabuleiro) uma das razões pela qual não é passado em todos os predicados é porque estamos a trabalhar com um facto chamado **initialBoard** que contém o tabuleiro atual e depois através de **asserts e retracts** muda-se o tabuleiro. Portanto sempre que se necessita do board é necessário meramente chamar o facto initialBoard(Board) com a variável Board onde vai ser recebido o board (tabuleiro) .



**Figura 13**: initialBoard

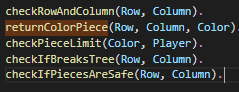
**Lista de jogadas válidas**

Para verificar se uma jogada é válida criou-se um predicado chamado **checkRules (Row, Column,Player, IsMachine),** que se encontra no ficheiro rules.pl, onde se passa a coluna e a linha do tabuleiro, variáveis Row e Column respetivamente, que são valores providenciados pelo jogador para a suposta peça que ele quer retirar. A variável Player serve para identificar o jogador em si (ou máquina) e a variável IsMachine é utilizada sempre ou com o valor 0 ou com valor 1 e serve para indicar se é a Máquina que vai utilizar este predicado (IsMachine é unificado com o valor 1) ou um jogador humano (IsMachine é unificado com 0), isto é feito para que caso seja um jogador humano então se este escolher uma jogada inválida uma mensagem de erro apareça no ecrã, caso seja uma máquina não se quer este comportamento, quer-se que o predicado **checkRules** falhe para que depois ele volte a tentar com um valor novo de Row (linha) e Column (coluna) a seguir. Este predicado **checkRules** verifica se para a posição Row (linha) e Column (coluna) as regras do jogo se verificam, isto é se o jogador efetivamente pode concluir a jogada com a retirada de uma peça.



**Figura 14**:Predicado checkRules()

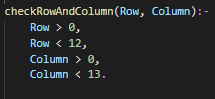
Dentro do checkRules() é necessário verificar diversas condições para que consigamos concluir se com o input do jogador consiguemos retirar uma peça , para tal utilizamos predicados dentro do checkRules() que se encontram no ficheiro game.pl (**returnColorPiece(Row, Column, Color)**), no ficheiro rules.pl ( **checkRowAndColumn(Row, Column)** , **checkIfBreaksTree(Row, Column)** , **checkIfPiecesAreSafe(Row, Column), checkIfBreaksTree(Row, Column**)), e finalmente no ficheiro players.pl ( **checkIfPieceLimit(Color , Player)** ).





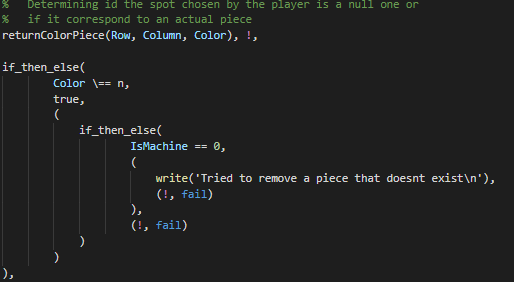
**Figura 15**: predicados utilizados no checkRules

**chekRowAndColumn(Row, Column)** – Verifica se a Row (linha) e a Column (coluna) se encontra dentro de posições possiveis dentro do board.



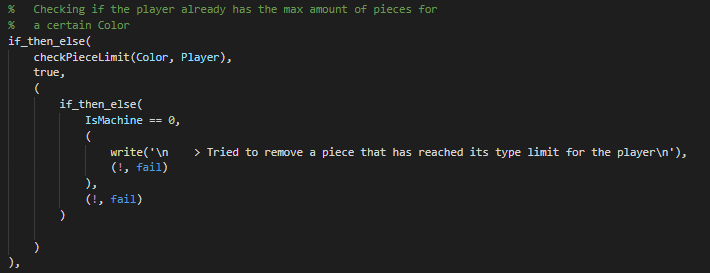
**Figura 16** :Predicado checkRowAndColumn()

**returnColorPiece(Row, Column, Color)** – Retorna a Color (cor) da peça (r, y, b) que se encontra na Row (linha) e Column (coluna) ou retorna ‘n’ (null) caso não haja peça nenhuma na posição, utilizada no checkRules para verificar se uma posição do tabuleiro é nula ou se contém efetivamente lá alguma peça e para saber de que cor a peça é.



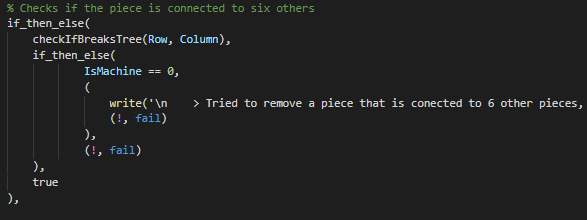
**Figura 17** : Chamada do predicado returnColorPiece() dentro do checkRules()

**checkPieceLimit(Color, Player) –** Verifica se para o jogador com o número Player se para a cor Color o jogador já possui 5 peças dessa cor. Se sim, quer dizer que a peça pode ser retirada e atribuída ao jogador, se não então o predicado falha e o jogador não pode retirar a peça.



**Figura 18**: Chamada do predicado checkPieceLimit () dentro do checkRules()

**checkIfBreaksTree(Row, Column) –** Se uma peça estiver conectada com 6 então a peça não pode ser retirada, esta regra foi definida para que sejam menores as chances de quebrar a árvore do jogo no futuro porém não chega era necessário fazer mais verificações para garantir isso.



**Figura 19**: Chamada do predicado checkIfBreaksTree () dentro do checkRules()

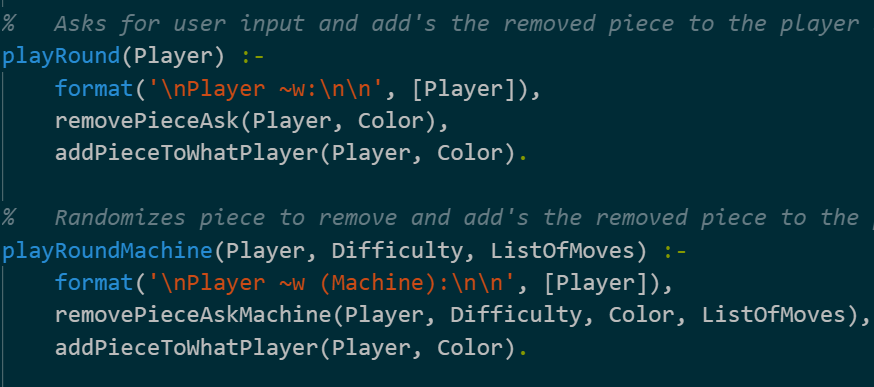
**checkIfPiecesAreSafe(Row, Column)** – Verifica se ao retirar a peça que está na posição Row e Column, se as peças a sua volta ficam ou não desprotegidas, se ficarem desprotegidas então não se pode retirar a peça.

# 

**Figura 20** : Chamada do predicado checkPiecesAreSafe() dentro do checkRules()

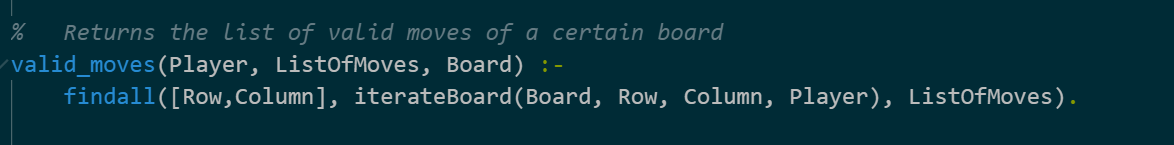
**Execução de jogadas**

O nosso jogo tem um ciclo principal chamado playLoop localizado no ficheiro **game.pl.**

A cada jogada é chamado o predicado playRound ou playRoundMachine dependendo do modo de jogo escolhido. Os modos de jogo variam entre jogador contra jogador, jogador contra máquina e máquina contra máquina.

**Figura 21**: playRound and playRoundMachine

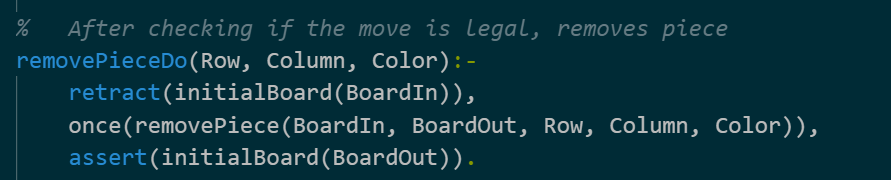
Em ambos os predicados é escolhida a peça a ser retirada (através de input do jogador ou através de geração de uma jogada por parte da máquina).

Para gerar a jogada da máquina são feitos dois níveis de “AI”. No nível 0 é feito uma geração aleatória de uma coluna e linha e se uma peça for possível ser retirada o jogo prossegue. No caso do nível 1 é gerado primeiro uma lista de jogada possíveis e escolhida aleatoriamente uma jogada dessa lista com auxílio ao predicado findall.

**Figura 22**: valid\_moves

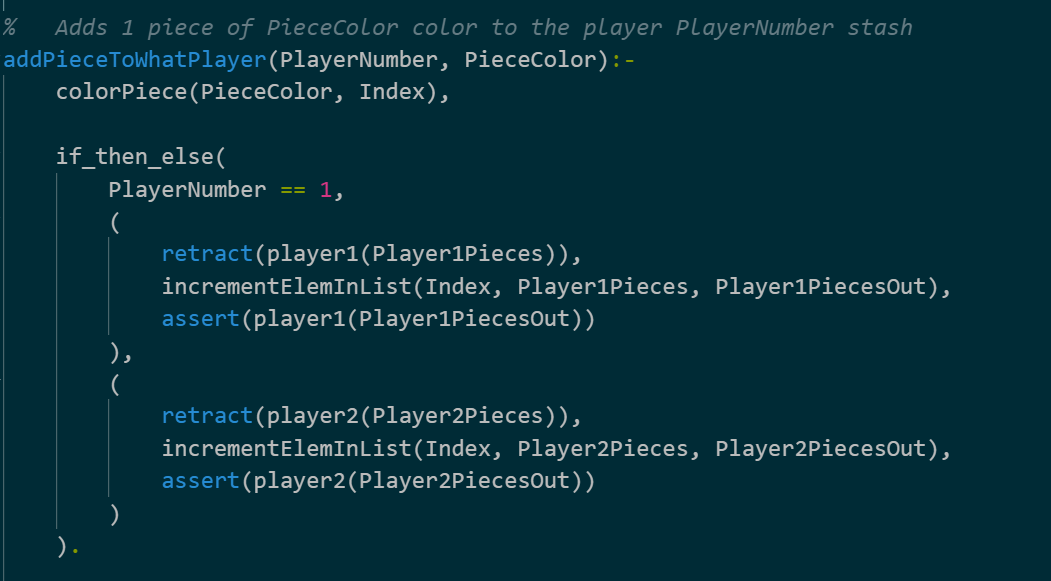
Ambos os predicados removePiecesAsk e removePieceAskMachine, depois de ser escolhida a peça a ser retirada verificam se, primeiro, a jogada está dentro do tabuleiro, e em segundo se todas as regras do jogo são cumpridas. Em ambos os casos, em caso de insucesso é escolhida uma nova jogada e em caso de sucesso a peça retirada é adicionada ao “inventário” de peças do jogador através do predicado removePieceDo.

Para a verificação de jogadas foi criado um ficheiro auxiliar chamado **rules.pl** que faz todo o tipo de verificações. Verifica se está alguma peça na jogada selecionada, se não há nenhuma peça, adjacente à selecionada que fique desprotegida e se a árvore do jogo não é partida (consultar regras no início).

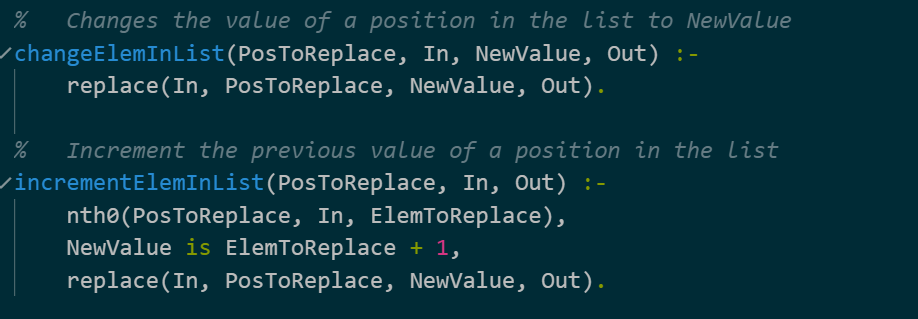


**Figura 23**: removePieceDo

De seguida, é adicionada a peça ao jogador na chamada addPieceToWhatPlayer, na qual é incrementada uma das peças (dependendo da cor da peça selecionada) do inventário do jogador.



**Figura 24**: addPieceToWhatPlayer

Para isso optámos por escolher duas funções de fácil uso nas quais é feito a mudança de um elemento qualquer da lista através de predicados de uso complexo. Este conjunto de predicados, como é partilhado por diversos ficheiros está num documento á parte chamado **shared.pl**

**Figura 25**: playRound and playRoundMachine

Por último, como cada jogador tem obrigatoriamente de retirar uma peça, é verificado, no final do jogo, se os dois jogadores ganharam, ou seja, se no final das duas jogadas têm um total de 10 peças de cada cor, tendo este valor ter de ser distribuído igualmente (5 para cada). É também verificado após cada jogada, se o jogo ganhou, ou seja, se não há mais nenhuma peça possível de ser retirada sem que os jogadores obtenham o total necessário para alcançar a vitória (Não esquecer que este jogo é cooperativo).

**Final do jogo**

Sendo o Asterismo um jogo cooperativo existem duas maneiras distintivas de acabar o jogo, ou os jogadores perdem porque já não há mais jogadas válidas que possam escolher ou os jogadores ganham porque tanto um como o outro foram capazes de retirar 5 peças de cada cor. Para testar esta situação é necessário após cada jogada de um jogador verificar se o jogo já não possui mais jogadas válidas e é necessário ver se depois da jogada do jogador 2 se os jogadores ganharam. Na imagem seguinte encontra-se o ciclo de jogo principal, nas chavetas laranjas encontram-se as verificações se o jogo acabou porque já não existem jogadas válidas e na chaveta vermelha encontra-se a verificação se jogadores ganharam o jogo ou não.

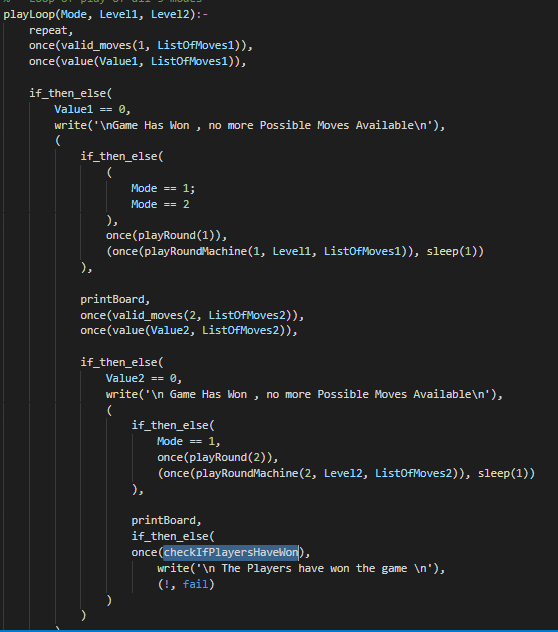


Figura 26– predicado playLoop/ ciclo principal do jogo

Segue-se agora a descrição de como se processa internamente os dois casos de acabar o jogo descritos em cima.

Caso 1: os jogadores ganharem o jogo

Para verificar se os jogadores ganharam, criou-se o predicado **checkIfPlayersHaveWon** que se encontra no ficheiro Players.pl. Este predicado faz uma verificação da quantidade de peças que ambos os jogadores possuem para determinar se ambos têm ou não 5 peças de cada cor, se sim, o predicado retorna verdadeiro e o jogo termina com a vitória dos jogadores, se não o jogo continua.

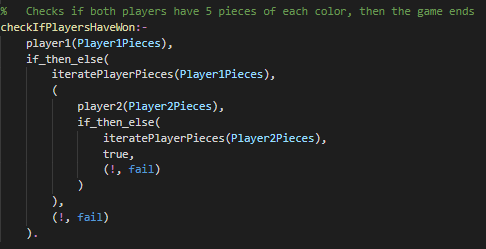


Figura 27– predicado checkIfPlayersHaveWon

Este predicado utiliza os factos player1 e player2 que contêm, tanto um como o outro, para cada um dos jogadores, uma lista do tipo [R, Y, B] (R – nº de peças vermelhas, Y – nº de peças amarelas, B – nº de peças azuis). Cada posição desta lista representa o número de peças de cada cor que o jogador possui/retirou. No início ambas as listas encontram-se na forma [0, 0, 0], a medida que os jogadores vão jogando o jogo estas listas vão sendo atualizadas com **asserts e retracts**.



Figura 28 – factos player1 e player2

O predicado **checkIfPlayersHaveWon** também utiliza o predicado **iteratePlayerPieces(List)** que percorre as listas de peças de cores de cada um dos jogadores e verifica se existe alguma posição destas listas que não tenha o valor 5 , isto é se existe alguma cor que um dos jogadores ainda não tenha colecionado 5 peças. Se isto acontecer, então quer dizer que os jogadores não ganharam o jogo, logo este predicado retorna falso e por sua vez o predicado **checkIfPlayersHaveWon** também retorna falso.

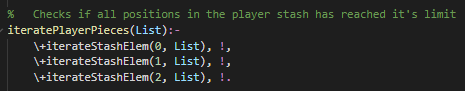


Figura 29– predicado iteratePlayerPieces

Caso 2: os jogadores perderem o jogo

 Para verificar se o jogo acabou com a derrota dos jogadores, no ciclo principal verifica-se o número de jogadas possíveis com ajuda dos predicados **valid\_moves** e **value.** Com o predicado **valid\_moves(Player, ListOfMoves)** descobrem-se as jogadas válidas, que são retornadas na variável ListOfMoves. Com o predicado **value(Value, ListOfMoves)** descobre-se o número de jogadas válidas, que é retornado na variável Value.

Figura 30 – predicado valid\_moves e value

Com a variável Value preenchida a única coisa que falta é verificar se é igual a 0 ou não. Se for igual a 0 o jogo termina com a derrota dos jogadores, se não o jogo continua.

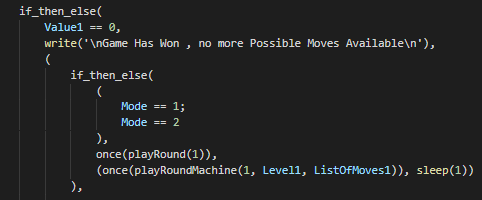
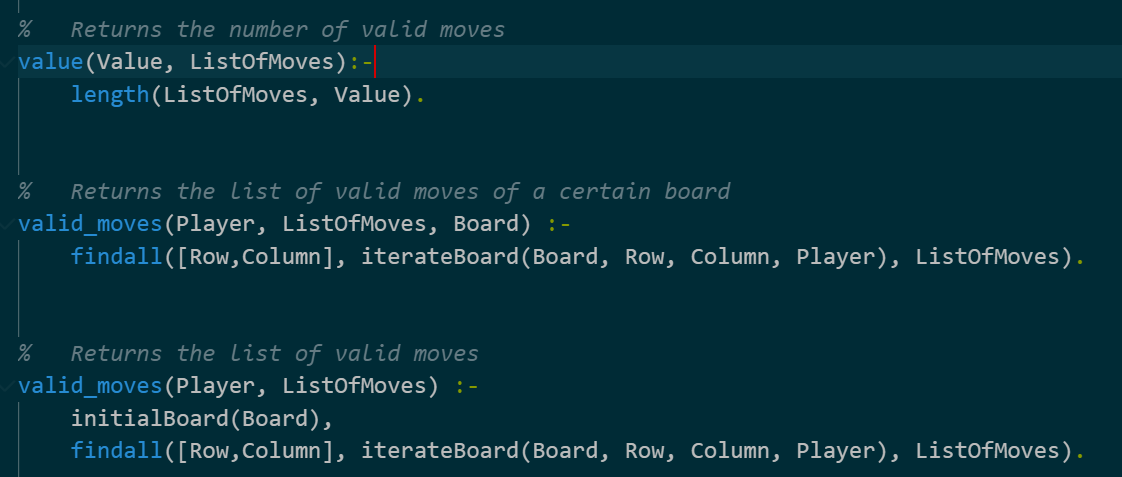


Figura 31 – comparação de Value com 0

**Avaliação do Tabuleiro**

Os 3 predicados mais importantes na avaliação do tabuleiro são os descritos em baixo.

O predicado value avalia um tabuleiro retornando o número possível de jogadas nesse tabuleiro. O predicado valid\_moves cria um tabuleiro com todas as jogadas possíveis através da chamada ao predicado findall.



**Figura 32**: value e valid\_moves

**Jogada do computador**

Ao tratar do nível de dificuldade da máquina 0 ou 1, temos um predicado chamado **removePieceAskMachine(Player, Difficulty, Color, ListMoves)** ficheiro game.plque trata de realizar a jogada da máquina.

O nível 0 da máquina limita-se a dar um número aleatório entre 1 e 11 para a linha e um número aleatório entre 1 e 12 para a coluna (embora no nosso tabuleiro as colunas sejam identificadas com letras, internamente as letras são convertidas para números). De seguida este nível verifica através do predicado **checkRules(Row, Column, Player, IsMachine)** se a peça pode efetivamente ser retirada, isto é, se a retirada da peça respeita ou não as regras do jogo. Se sim, quer dizer que a peça vai ser retirada o jogo com ajuda do predicado **removePieceDo(Row, Column, Color),** se não, o predicado **removePieceAskMachine** é chamado outra vez até que a escolha aleatório de posições encontra uma que seja válida.

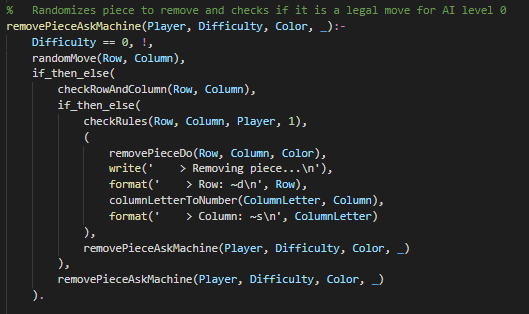


Figura 33 – predicado removePieceAskMachine nível 0

O nível 1 da máquina pega na lista de jogadas válidas produzida pelo predicado **valid\_moves** e meramente escolhe uma jogada aleatório dessa lista. Como todas as jogadas dessa lista são válidas não é necessário fazer o controlo das regras do jogo como se faz no nível de dificuldade 0.

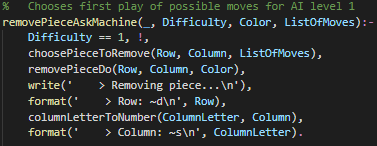


Figura 34 – predicado removePieceAskMachine nível 1

**Conclusões**

O projeto teve como objetivo aplicar o conhecimento adquirido sobre a linguagem Prolog de forma iterativa e desafiadora.

Ao longo do desenvolvimento deste trabalho, encontramos diversos problemas, tais como, usar nos sítios corretos a grande vantagem que é o backtracking e usar efetivamente os cuts. Tentámos também preocuparmo-nos mais com a eficiência (poderíamos tê-lo feito por exemplo usando o predicado inline) mas no final não deu tempo para tudo.

Em suma, foi uma experiência positiva, na qual aprendemos muito mais do que o que foi dado nas aulas teóricas devido ao nosso interesse e dever de pesquisar assuntos não abordados ou não compreendidos totalmente.