Programação 3

PAKTRIS

Henrique Rosa (51923)
Diogo Matos (54466)
17/01/2024

Introdução

O presente relatório visa dar contexto à implantação da nossa solução para o enunciado do trabalho prático da UC Programação 3 do curso de Engenharia Informática. O enunciado em questão tem como problemática a leitura de uma sequência de jogadas e a verificação de que as mesmas cabem num tabuleiro com dimensões exatas. Para tal, foi-nos dada a escolha entre as linguagens de programação Ocaml e Prolog, pelo que escolhemos Ocaml pela sua melhor legibilidade e versatilidade, bem como a proficiência de ambos os integrantes do grupo ser significativamente melhor em Ocaml comparativamente a Prolog.

Jogadas

Como descrito no enunciado, as jogadas são representadas por um tuplo com três elementos sendo a sua syntaxe “ (peça , nrot , ndir) ” , sendo os elementos respectivamente a peça a jogar , o número de vezes que a peça é rodada 90 graus no sentido dos ponteiros do relógio, e quantas vezes a peça é deslocada uma “casa” para a direita. Para implementar a noção de jogadas na nossa solução, foi implementado um *custom data type* (**type jogada = peca * int * int**) (o tipo “peca” será explicado mais adiante).

Leitura da lista de jogadas

A leitura da lista que contém a sequência de jogadas a testar é feita pela função **paktris** (**paktris : (peca * int * int) list -> bool = <fun>**) que aceita uma lista de elementos do tipo jogada e testa cada um deles sequencialmente, chamando-se recursivamente com o novo argumento sendo a cauda da lista da iteração anterior, progredindo pela lista até chegar a uma lista vazia, o que seria o caso de sucesso, retornando então true. Caso alguma das jogadas não caiba (ou seja, caso receba um false das funções que chama, que são respectivas à lógica da queda, posicionamento e verificação “out of bounds” as peças da jogada), retorna false, o que indicaria ao utilizador que a sequência não cabe no tabuleiro, o que coincide com o pedido no enunciado

Tabuleiro

Para a criação do tabuleiro utilizou-se a syntaxe “let board = Array.make_matrix m n 0 ;;” que tira partido do operando “make_matrix” do módulo “Array” para criar uma matriz (array bi-dimensional), com todas as posições iniciadas com o valor zero, e com as dimensões nxm (n=altura , m=largura) , como consta no enunciado. Nesta matriz irão ser representadas e guardadas as jogadas (quando válidas), com valores diferentes de zero.

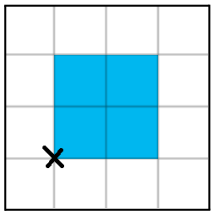
Peças , posicionamento e rotação das mesmas

Para a implementação das peças, implementámos um *custom data type* “pecas” (**type peca = I | O | S | T**) para as diferentes peças poderem ser representadas com as letras designadas no enunciado. Este data type vai ser crucial em quase toda a lógica do código pois permite que a peça de uma jogada seja passada como argumento para ser identificada pelas funções de modo a alterar o seu comportamento para possibilitar que a jogada seja feita.

Já que ndir começa em zero e cada valor a mais indica uma “casa” para a direita no tabuleiro aquando da jogada, então assumimos que o próprio valor de ndir pode ser utilizado como valor para definir a coluna do tabuleiro onde a jogada vai ser feita (ou seja, se ndir for 2, a coluna do tabuleiro onde aquela jogada vai ser feita será a coluna 2 (a começar em zero como é convenção nos arrays)).

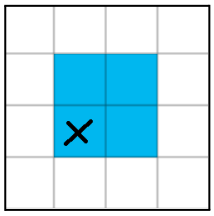
A rotação das peças e a sua implementação foi um tópico de debate pois no enunciado estava designado que o eixo de rotação das peças está no “canto inferior esquerdo”, e haviam duas noções possíveis de canto inferior esquerdo:

Eixo 1



Esta foi a opção considerada inicialmente para o eixo de rotação das peças, mas acabamos por descartar esta opção. Não só percebemos que faz menos sentido do que a segunda opção (eixo 2) como também foi mais difícil de implementar e menos perceptível e didático aquando dos teste manuais (não se percebe bem como é que a peça roda na utilização prática do programa).

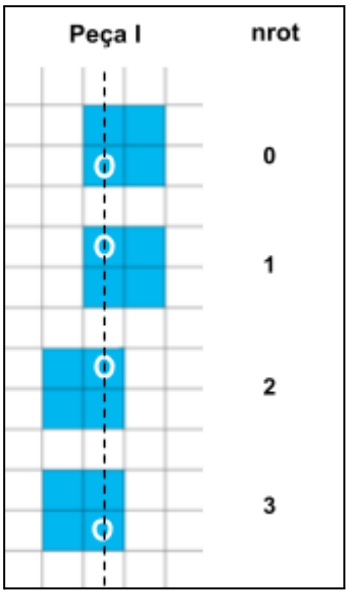
Eixo 2

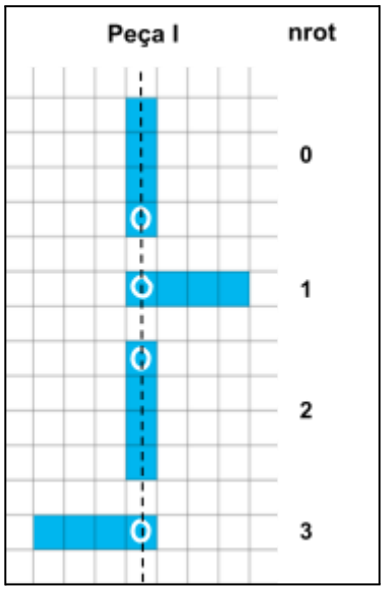


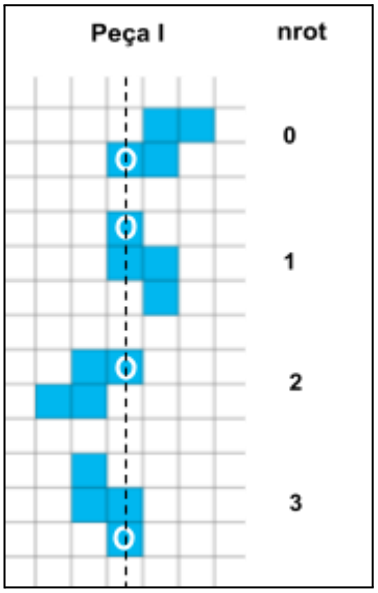
Esta opção faz mais sentido e a rotação que implica é mais perceptível e funcional.

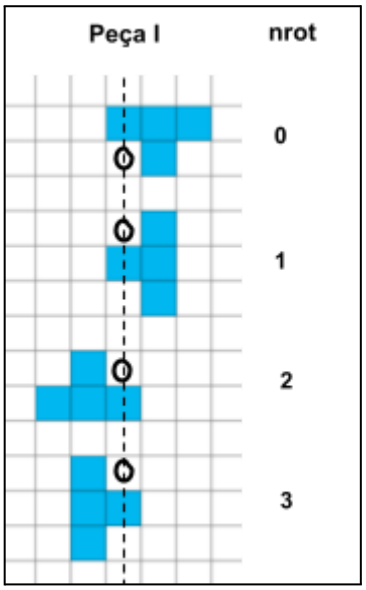
Dentro da código, a lógica referente à rotação das peças nada mais é do que apenas passar o valor de nrot para as funções de afetação do tabuleiro, que consoante essa rotação e um ponto de início, vão modificar os valores da matriz de modo a colocar a peça no tabuleiro e efetuar a jogada,

Como a rotação das peças depende do valor “nrot” de cada jogada, optamos por deixar abaixo uma representação das diferentes peças com os diferentes valores possíveis para nrot. Note-se que apenas incluímos nrot de 0 a 3, pois na nossa solução decidimos passar como argumento para a rotação das peças o módulo de nrot por 4 (nrot % 4) pois se girarmos uma peça 90° quatro vezes, a peça fica igual (4 x 90° = 360° = 0°).
Cada uma das representações abaixo tem também o eixo de rotação (indicado por um círculo) e a posição horizontal do tabuleiro a que consiste a jogada dessa peça (dada por ndir) representada por uma linha tracejada em preto.









Queda das peças e colisões

A lógica da queda das peças aquando de uma jogada é feita pela função **fall** , tendo esta como parâmetros:

- **O tabuleiro (board) :** A matriz que foi falada acima;
- **O "ponto de partida" (x,y) :** nada mais é do que o ponto pelo qual a queda vai começar, sendo este ponto importante pois x representa ndir (a coluna onde a jogada vai ser feita) e y representa o quão acima do tabuleiro a queda tem de começar para que a jogada não comece com a peça dentro do tabuleiro (o que logicamente não faria muito sentido e iria causar problemas). Tiremos como exemplo a jogada (I,0,2), que seria uma jogada com uma peça I vertical - ora, esta peça necessitaria de 4 "casas" acima do tabuleiro para não começar dentro do mesmo, enquanto que a mesma jogada mas com uma peça O apenas necessitaria de dois
- **O array de colisão([|a,b,c,d|]) :** Um array de um, dois, três ou quatro posições, cujos valores vão ser usados para verificar as colisões com as peças e com o fundo do tabuleiro, sendo cada um destes valores a representação unidimensional de quantas "casa" se terá de retirara ao pondo de partida para verificar devidamente as colisões.

A função **fall** vai chamar-se recursivamente , mandando o mesmo tabuleiro e o mesmo array, mas alterando o parâmetro do ponto de partida de (x,y) para (x,y-1), efetivamente "descendo uma casa" no tabuleiro, dai desempenhando a função de queda das peças

A cada iteração, esta função vai verificar por colisões com outras peças ou com o fundo do tabuleiro, chamando a função **check_collision** que tem como parâmetros o ponto de partida e o array de colisão, e vai analisar o tabuleiro em busca de valores que sejam diferentes de zero. Se encontrar algum, o que significa colisão, retorna true, retornando false em caso contrário. Esta função vai utilizar o array como uma representação unidimensional da peça em questão (exemplo, uma peça O seria representada com o array [|2,2|], que seriam as casas que necessitaríamos retirar a "linha" horizontal de y para verificar as colisões dessa peça).

Quando a função **fall** recebe um true da colisão, retorna o par (x,y) do ponto de partida da iteração onde a colisão foi detetada, que vai ser usado para colocar a peça no tabuleiro.

Afetação do tabuleiro

A colocação das peças no tabuleiro vai ser feita pelas funções **set_piece (**uma para cada peça, identificadas pela letra representativa da peça a preceder o nome da função), que, consoante a orientação da peça e o ponto de partida que lhe são fornecidos da função fall, vai modificar os valores no tabuleiro. Os valores colocados no tabuleiro serão diferentes para cada peça : I é representada por 1 , O por 2, S por 3 e S por 4. Isto é feito de modo a possibilitar a diferenciação das peças no tabuleiro aquando da visualização do mesmo.

Visualização e limpeza do tabuleiro

A representação visual do tabuleiro no terminal é feita com a função **print_board** que recebe como parametro o tabuleiro que vai representar no terminal, percorrendo-o posição a posição. Caso a posição esteja vazia (ou seja, caso o seu valor seja 0), a função imprime ".", caso seja uma peça, escreve a letra representante dessa peça com a mesma convenção utilizada inversamente na secção acima.

A limpeza do tabuleiro é feito pela função **clear_board** e consiste apenas em percorrer a matriz do tabuleiro posição a posição, mudando o valor de cada posição para zero, efetivamente limpando o tabuleiro de todas as peças que lá tinham sido colocadas, e possibilitando que seja possível visualizar o tabuleiro quando se introduz uma lista de jogadas e , se for necessário, limpar o tabuleiro para introduzir outra lista, sem ter de se executar o código de novo.

Comentários

Todo o código está devidamente comentado como mandam as boas práticas de programação, tendo estes, aliados ao próprio código, um peso significativo da perceptibilidade do código, pelo que sugerimos que não sejam desprezados aquando da análise do funcionamento do código.

Conclusão e considerações finais

Foi-nos possível Implementar tudo o que o enunciado requer, de forma a concluir de forma satisfatória o desafio proposto, sem deixar nenhuma funcionalidade proposta para trás. Como tal, sentimo-nos orgulhosos e satisfeitos com a nossa solução. -

Decidimos não ir muito além de um código ASCII na representação visual de forma deliberada e consciente, achamos que a aparência da representação no terminal é agradável e completamente funcional.

Consideramos que alguns dos predicados das funções são muito grandes (algumas linhas no código têm mais de 200 caracteres), mas foi também uma escolha intencional, para que a linha seja lida como um seguimento, sem cortes de linha, que podia prejudicar o entendimento do leitor.

Temos também a apontar que tomamos inspiração de excertos de códigos em fóruns sobre programação para o desenho de algumas funções (nomeadamente a função **print_board** e **clear_board**, todas as outras foram completamente originais)

Quanto à limpeza do tabuleiro (função **clear_board**): O enunciado do trabalho não requer esta funcionalidade, mas achámos pertinente incluí-la, pois foi uma ferramenta bastante útil nos testes que realizamos.