Moai

Relatório Intercalar

Universidade do Porto

Faculdade de Engenharia

FEUP

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo 5:
Paulo Jorge de Faria dos Reis - 080509037
Miguel Rossi Seabra - 060509054

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

5 de Novembro de 2012

Resumo

Pretende-se com este relatório apresentar o procedimento seguido pelo grupo de trabalho no desenvolvimento em Prolog do jogo "Moai".
Será disponibilizada informação relativamente ao jogo em termos de regras, descrição do tabuleiro, opções tomadas pelo grupo em pontos onde a informação do criador do daquele não é clara e também a apresentação dos predicados de Prolog que foram utilizados para o desenvolvimento do projeto.
Também é descrito o possível interface para comunicação com um cliente via sockets.

Conteúdo

1 Introdução

A implementação das regras de um jogo de tabuleiro recente, utilizando programação em lógica apresenta um desafio aliciante, ao mesmo tempo que possibilita o contacto e desenvolvimento de capacidades nesta área aos elementos envolvidos no desenvolvimento deste sistema. É também de ressalvar a criação de um programa funcional que permite a realização de partidas de "Moai" onde os intervenientes podem ser humanos ou computadores, e neste caso tendo o sistema capacidades mínimas de realizar jogadas válidas de acordo com as regras. A versão base apenas permitirá a interação em modo de texto através da consolo de Prolog, podendo no entanto ser providenciada a integração com interface gráfico, opção especificada mais adiante.

2 Descrição do Problema

O jogo foi criado por Rey Alicea em 2012 [**?**], e não existe qualquer referência histórica deste. Pelo que nos apercebemos o jogo encontra-se ainda num estádio de revisão do mesmo. A descrição das regras [**?**, **?**] não são claras o suficiente, deixando muito espaço para a interpretação das mesmas, assim a equipa de desenvolvimento das regras em Prolog decidiu utilizar a regras conforme se explica mais adiante.
São necessários dois jogadores para realizar uma partida de "Moai".

3 Arquitetura do Sistema

O projeto foi desenvolvido num único módulo, pois não se considerou necessário a separação dos predicados de acordo com qual tipo de classificação. Existem predicados para cada uma das tarefas necessárias à realização de um jogo de **"Moai"**, os quais são explicados na próxima secção. Bem como a forma como interagem entre si para a obtenção de um resultado útil do programa desenvolvido em Prolog.

Por não ser necessário a qualquer um dos autores não foi desenvolvida a componente de comunicação por sockets, a qual mesmo assim foi acautelada pelo facto de cada um dos principais predicados aceitar como variáveis de entrada, entre outras, o estado do jogo, permitindo desta forma a possibilidade de utilizar o programa de uma forma *stateless*, sendo o estado mantido no cliente. Temos assim um paradigma REST que pode também ser utilizado com a interface adequada para a utilização deste projeto em aplicações Web.

Para efeitos de demonstração das capacidades do programa existe um predicado (*new_game*), que inicia um jogo completo de **"Moai"** utilizando para tal os principais predicados do programa, mas também alguns auxiliares cuja utilidade se fica pela interface com os jogadores e a manutenção de estado do jogo.

3.1 Comunicação por sockets

Numa eventual implementação da comunicação por sockets seriam consideradas as mensagens conforme se descreve de seguida.

Início de comunicação

initialize(X, Y) →

← *ok(Board)*

Pedido de inicialização com especificação das dimensões do tabuleiro pretendido, como resposta é devolvida a representação do tabuleiro (lista de listas), representado por *Board*. Corresponde ao predicado *new_board(X, Y, Board)*, sendo X e Y as dimensões do tabuleiro, respetivamente em colunas e linhas.

Pedido de posicionamento de peão

ini_posicao(IBoard, Piece, X, Y) →

← *ok(OBoard)*

← *invalid*

IBoard é o estado inicial do tabuleiro, *Piece* pode ser "B" ou "P" (corresponde a jogador Branco ou jogador Preto). X e Y são a posição pretendida para colocar o peão. A primeira ação do jogo consiste em cada jogador colocar numa casa livre o seu peão. Utilizando o predicado *set_piece(IBoard, Piece, X, Y, OBoard)* obtemos o tabuleiro (*OBoard*) com a peça do jogador já posicionada se tal for válido, caso contrário a resposta será *invalid*.

Pedido de execução de movimento humano

execute(IBoard, Push, Player, X, Y) →

← *ok(OBoard)*

← *invalid*

IBoard é o estado inicial do tabuleiro, *Push* é a direção de movimento do peão ("s" para aproximar do bloqueador, "n" para afastar), *Player* corresponde ao peão que se pretende mover ("s" para o Preto, "n" para o Branco), X e Y são a posição pretendida para colocar o bloqueador. Trata-se de um pedido de movimento de jogador humano, se o movimento não for possível, responder com *invalid* e não realizar qualquer outra operação, se for um movimento válido devolver novo estado do Tabuleiro (*OBoard*).
Utiliza o predicado *make_a_move(IBoard, Push, Player, X, Y, OBoard)* para dar a resposta.

Pedido de execução de movimento de computador

calculate(IBoard, Level, Player) →

← *ok(Push, Piece, X, Y, OBoard)*

← *invalid*

IBoard é o estado inicial do tabuleiro, *Level* o nível de dificuldade pretendido e *Player* qual o jogador que está a realizar a jogada ("s" para o Preto, "n" para o Branco). Utiliza-se para solicitar um movimento a realizar pela AI a favor de um jogador (Player), em resposta utiliza-se o resultado do predicado *create_a_move(IBoard, Level, Player, Push, Piece, X, Y, OBoard)* onde *Push* é a direção que o peão foi movido ("s" foi aproximado do bloqueador, "n" foi afastado), *Piece* qual o peão que foi movido ("s" para o Preto, "n" para o Branco), X e Y são a posição onde o bloqueador foi colocado e o novo estado do Tabuleiro é (*OBoard*). Se não for possível realizar um movimento devolver *invalid*, nesta situação o cliente deveria fazer uma verificação de final de jogo.

Verificação de final de jogo

← *game_end(Board, Piece)*

← *ok(Player*

Atendendo à forma de definir o vencedor neste jogo, apenas o estado do tabuleiro

(*Board*) não basta para se definir o vencedor. Ambos os peões podem estar a certo momento bloqueados, o jogador que estiver a iniciar a jogada é que será o derrotado porque pelas regras a sua peça tem de ter a possibilidade de se mover no inicio da sua jogada. Por isso tem também de dizer qual o jogador a testar o final de jogo utilizando "P" ou "B" em *Piece*. A resposta obtida do predicado *endgame(Board, Piece)* será **true** se o jogador perdeu ou **false** se ganhou.

Fim da comunicaçao

← *bye*

← *ok*

Informa da vontade de terminar a comunicação, permitindo a libertação de recursos.

4 Módulo de Lógica do Jogo

Para realizar um jogo inicia-se com o predicado *new_game*, o qual apresenta as regras do jogo (*print_help*) e coloca uma série de questões ao utilizador (*setup_game*, as quais permitem inicializar o mesmo. O controlo é então passado para *main_loop*.

O *main_loop* recebe o estado do tabuleiro e o actual jogador. O primeiro jogador é o preto "P" (alternando com o branco "B") e o tabuleiro que foi criado em *setup_game*. A primeira operação é verificar se o jogo terminou com a vitória do jogador anterior, usando *endgame*. De seguida é questionada a posição pretendida para o bloqueador (Coluna e Linha), a ação pretendida (puxar ou empurrar) e o peão sobre o qual se pretende aplicar (preto ou branco). Com esta informação chama-se o predicado *make_a_move* que irá processar as operações pretendidas. É apresentado na consola o novo tabuleiro (*print_board*) e definido o próximo jogador (*next_piece*). Faz-se então uma chamada recursiva ao *main_loop*.

O *make_a_move* vai fazer o parse dos argumentos e consoante o tipo de movimento especificado vai chamar o predicado respetivo para esse movimento. Este último irá utilizar os predicados de pesquisa, clarificados abaixo na "Validação de Jogadas", por forma a tentar encontrar o peão pretendido em linha com o bloqueador colocado pelo jogador nessa mesma jogada. No caso de se encontrar o peão numa das pesquisas o movimento é então realizado com a respetiva atualização do tabuleiro, caso contrário **false** é retornado.

Em caso de jogada válida o tabuleiro é atualizado com a colocação de um bloqueador na posição pedida pelo jogador e com a deslocação do peão solicitado da posição antiga, que fica vazia, para a sua nova posição.

4.1 Visualização do Estado do Jogo

print_board(+Board).

Recebe uma representação do estado do tabuleiro e imprime-o na consola, conforme figura 1.

Board Representação do estado do tabuleiro, na forma de uma lista de listas

(lista de linhas do tabuleiro).

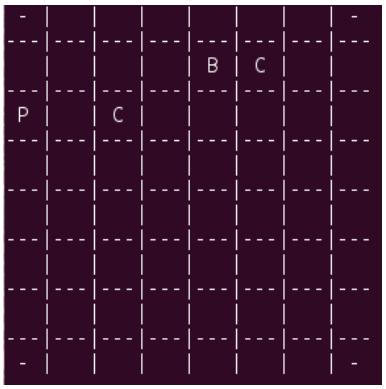

Figura 1: Tabuleiro durante o jogo

4.2 Validação de Jogadas

search_horizontal_forward(+IBoard, +Piece, +XCounter, +YCounter, -PieceX, -PieceY).
search_vertical_forward(+IBoard, +Piece, +XCounter, +YCounter, -PieceX, -PieceY).
search_diagonal_forward(+IBoard, +Piece, +XCounter, +YCounter, -PieceX, -PieceY).
search_reverse_diagonal_forward(+IBoard, +Piece, +XCounter, +YCounter, -PieceX, -PieceY).
search_horizontal_backward(+IBoard, +Piece, +XCounter, +YCounter, -PieceX, -PieceY).
search_vertical_backward(+IBoard, +Piece, +XCounter, +YCounter, -PieceX, -PieceY).
search_diagonal_backward(+IBoard, +Piece, +XCounter, +YCounter, -PieceX, -PieceY).
search_reverse_diagonal_backward(+IBoard, +Piece, +XCounter, +YCounter, -PieceX, -PieceY).
Este conjunto de predicados permite validar verificar se o peão (*Piece*) se encontra na linha que está a ser testada, numa posição que permite o movimento para ou a afastar-se deste.
IBoard é a representação inicial do tabuleiro.
Piece O peão para o qual se pretende fazer a validação ("P" ou "B").
XCounter Coluna onde se pretende colocar o bloqueador (inteiro).
YCounter Linha onde se pretende colocar o bloqueador (inteiro).
PieceX Coluna onde se encontra o peão (caso o movimento possa ser realizado, **false** em caso contrário).
PieceY Linha onde se encontra o peão (caso o movimento possa ser realizado, **false** em caso contrário).

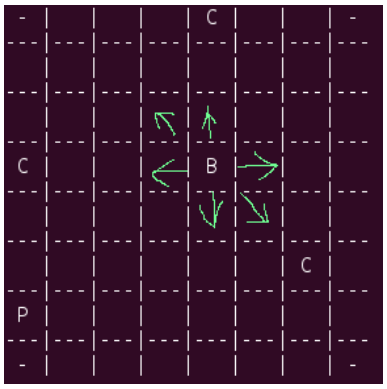

Figura 2: Tabuleiro com Movimentos possíveis

4.3 Execução de Jogadas

move(+IBoard, +XCounter, +YCounter, +Piece, +PushPull, -OBoard).
Cada jogador na sua vez tem de fazer duas ações que por estarem relacionadas são tratadas num único predicado, indicando o estado inicial do tabuleiro e a descrição da jogada a realizar, caso seja válida temos de volta o novo estado do tabuleiro, ou **false** caso contrário.
IBoard é a representação inicial do tabuleiro.
XCounter Coluna onde se pretende colocar o bloqueador (inteiro).
YCounter Linha onde se pretende colocar o bloqueador (inteiro).
Piece O peão para o qual se pretende fazer a validação ("P" ou "B").
PushPull Direção do movimento pretendida ("push" ou "pull").
OBoard é a representação final do tabuleiro.

4.4 Final do Jogo

endgame(+Board, +Piece).
Devido ás regras do jogo, apenas pelo estado do tabuleiro não é possível determinar um vencedor, também temos de saber qual o jogador que vai iniciar a sua ação. Como resposta temos **false** ou **true** conforme o jogador ganhou ou perdeu.
IBoard é a representação inicial do tabuleiro.
Piece O peão para o qual se pretende fazer a validação ("P" ou "B").

4.5 Rotina de jogo

main_loop(+Board, +Piece
Para que se possa jogar "Moai" utilizando apenas o interpretador de Prolog, temos este predicado que controla a interface com o utilizador e mantêm o estado do jogo durante a sua execução.
IBoard é a representação inicial do tabuleiro.
Piece O peão para o qual se pretende fazer a validação ("P" ou "B").

Não foram descritos os predicados auxiliares por não se considerar de utilidade e porque iriam aumentar consideravelmente a dimensão deste relatório sem ganho significativo.

5 Interface com o Utilizador

Não foi implementado o módulo de comunicação com o visualizador. A Interface com o utilizador é feita através de predicados do próprio Prolog.
Para iniciar um novo jogo deve-se utilizador o predicado *new_game*, de seguida

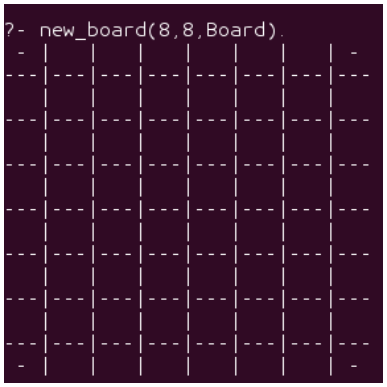

Figura 3: Tabuleiro vazio

devemos informar as dimensões do tabuleiro que pretendemos utilizar, para tal devem ser introduzido um valor inteiro seguido de . (ponto) para cada uma das dimensões. Uma vez que cada jogador pode ser controlado por um humano ou pela AI do computador, temos de responder cada um dos jogadores é humano ou não usando *s* ou *n*.
A primeira ação de cada um dos jogadores, Branco e depois o Preto (representados no tabuleiro por "B" e "P" respetivamente), é a colocação do seu peão no tabuleiro, para tal é perguntado ao utilizador qual a Coluna (X) e a Linha (Y) onde ele o pretende colocar, se for uma posição válida o tabuleiro é atualizado. A partir deste ponto, começando pelo Jogador Branco e alternando com o Jogador Preto, até que se alcance a condição de vitória (derrota) de um deles terminando assim o jogo, realizam-se jogadas de acordo com as regras já enunciadas neste relatório.
Na sua vez, cada jogador informa a Coluna (X) e a Linha (Y) onde pretende colocar o bloqueador (representado por "C" no tabuleiro), se pretende empurrar ("s") ou puxar ("n") o peão em relação ao bloqueador, e também qual o peão sobre o qual pretende atuar, "s" para Preto e "n" para Branco.

Nota: A razão para utilizar "n" e "s" em algumas das respostas quando a resposta adequada deveria ser outra ("P" ou "B" por exemplo) está relacionada com o reaproveitamento em várias situações distintas de código que foi adaptado de predicados obtidos na Internet [**?**].

6 Conclusões e Perspetivas de Desenvolvimento

Utilizar o Prolog para definir as regras de um jogo é algo relativamente simples, sendo apenas necessário a definição de um conjunto de predicados (regras) que descrevam o que é ou não possível fazer no jogo. A comunicação com o utilizador é bem complicada de desenvolver neste contexto, sendo responsável por grande parte do tempo utilizado no desenvolvimento do programa.

A definição de regras passa pela elaboração de factos que definem condições de paragem e clausulas com predicados que permitem o processamento de casos gerais.

Apenas foi desenvolvido o suficiente para possibilitar o jogo entre 2 humanos, a componente de inteligência artificial necessária para que um ou os dois jogadores sejam controlados pelo computador não foi desenvolvida, este é sem dúvida a principal falha e onde reside a possibilidade de melhoria do programa.

A comunicação via sockets passaria também pela criação de um predicado (mais auxiliares) que fizessem a interpretação das mensagens dos clientes e enviassem a resposta a estes, pois a definição da interface está feita, assim como os principais predicados necessários.

Bibliografia

[1] Rey Alicea. Blog do autor. http://reyaliceagames.wordpress.com/, 2012. Online em Outubro 2012.

[2] Rey Alicea. Boardgamegeek. http://boardgamegeek.com/image/1405108/moai, 2012. Online em Outubro 2012.

[3] Eran Kampf. Four in a row game in prolog. http://www.developerzen.com/2004/09/30/four-in-a-row-game-in-prolog/, 2009. Online em Novembro 2012.

Lista de Figuras

A Código

```
%UTILS

%nth(Index to get, Input List, Output)
nth(1,[IH|_],IH):- !.
nth(X,[_|OH],NTH) :-
LX is X-1,
nth(LX,OH,NTH).


%search_piece(IBoard,Piece,X,Y):-
search_piece(IBoard,Piece,X,Y):-
search_piece_y(IBoard,Piece,1,X,Y).

search_piece_y([],_,_,_,_):-
!,false.
search_piece_y([IH|_],Piece,TY,X,TY):-
search_piece_x(IH,Piece,1,X).

search_piece_y([_|IT],Piece,TY,X,Y):-
TTY is TY + 1,
search_piece_y(IT,Piece,TTY,X,Y).


search_piece_x([],_,_,_) :-
!,false.

search_piece_x([Piece|_],Piece,TX,TX):-
!,true.

search_piece_x([_|IT],Piece,TX,X) :-
TTX is TX + 1,
search_piece_x(IT,Piece,TTX,X).


%reverse list( input_list, output_list).
reverse(A,B):- reverse_recursion(A,[],B).
reverse_recursion([A|[]],L,[A|L]) :-
!,true.
reverse_recursion([],L,L) :-
!,true.
reverse_recursion([IH|IT],B,A) :-
reverse_recursion(IT,[IH|B],A).

% Ler inteiros do teclado. (PFR)
read_int(X):-read(X),integer(X),!. % Cut usado para parar ao ler um inteiro.
read_int(X):-writeln('Erro, pretende-se um inteiro!'),read_int(X).

% read a y\n input (PFR)
yesno(s).
yesno(n).
```

```
read_yn(X) :- read(X), yesno(X), !.   % Cut is used to stop when a yesno(X) char is read
read_yn(X) :- writeln('Erro, apenas s ou n!'), read_yn(X).


% Createn a new board
new_board( X, Y, Board ):- create_board(X,Y,IBoard), blank_corners(IBoard,Board),
print_board(Board), !.

create_board( _, 0, [] ):- !.
create_board( X, Y, [Line | Lines] ):- YN is Y-1, create_line(X,0, Line),
create_board(X, YN, Lines).
create_line( X, X, [] ):- !.
create_line( X, LX, [' ' | Ls] ):- LLX is LX+1, create_line( X, LLX, Ls).

blank_corners([Iline|Ilines],Board ):- blank_first_corners(Iline,Oline),
blank_last_corners(Oline, Ilines, Board).

blank_first_corners( [_|IT], ['+'|OT] ):- blank_corner(IT,OT).

blank_corner( [_|[]], ['+'|[]] ):- !.
blank_corner( [IH|IT] ,[IH|OT] ):- blank_corner(IT,OT).

blank_last_corners(Oline, Ilines, [Oline|OT] ):- blank_last(Ilines,OT).

blank_last( [IH|[]], [OH|[]] ):- blank_first_corners(IH,OH), !.
blank_last( [IH|IT], [IH|OT] ):- blank_last(IT,OT).

%-----------------\\----------


%check if it's vacant...
vacant(X,Y,Board) :-
nth(Y,Board,Row),
nth(X,Row,' ').

% set piece on X Y in Board
set_piece( IBoard, Piece, X, Y, OBoard) :-
vacant(X,Y,IBoard),
set_y_piece(IBoard, Piece, X, Y, 1, OBoard),!.

set_y_piece( [IH|IT], P, X, Y,  Y, [OH|IT]):-
set_x_piece(IH,P,X,1,OH),!.
set_y_piece( [IH|IT], P, X, Y,  LY, [IH|OT]) :-
LYY is LY+1,
set_y_piece(IT,P,X,Y,LYY,OT).

set_x_piece( [_|IT], P, X, X, [P|IT] ):- !.
set_x_piece( [IH|IT], P, X, LX, [IH|OT] ):-
XLX is LX+1,
set_x_piece(IT,P,X,XLX,OT).

erase_piece(IBoard, X, Y, OBoard) :-
```

```
set_y_piece(IBoard, ' ', X, Y, 1, OBoard),!.

% START (PFR)
new_game :-
print_help,
setup_game(Board),
start_game(Board).

% HELP (PFR)
print_help :-
writeln('MOAI'),
writeln('====\n'),
writeln('Jogo para 2 jogadores, que vao alternando entre si, ate que um'),
writeln('deixe de conseguir mover a sua peca, perdendo o jogo.'),
writeln('Cada jogador e representado por uma peca no tabuleiro de jogo.'),
writeln('Comum a ambos os jogadores, sao os bloqueadores, pecas colocadas no tabuleiro'),
writeln(', uma em cada jogada. Uma vez colocados ficam fixos ate ao final do jogo.'),
writeln('Joga-se num tabuleiro quadrado, originalmente de 8 por 8 casas.'),
writeln('Esta versao permite qualquer dimensao.'),
writeln('As casas dos cantos sao excluidas do tabuleiro de jogo.'),
writeln('Cada jogada e composta por 2 acoes:'),
writeln('1 - Colocacao de um bloqueador;'),
writeln('2 - Mover 1 peca, afastando ou aproximando-o do bloqueador acabado de colocar.'),
writeln('Apenas se pode mover peca quando esta em linha com o bloqueador, horizontal,'),
writeln('vertical ou diagonal.'),
writeln('O movimento do peca e sempre feito ate encontrar um obstaculo: bloqueador,'),
writeln('peca ou fronteira do tabuleiro.'),
writeln('Um jogador no inicio da sua vez tem de ser capaz de mover a sua peca de acordo'),
writeln('com as regras, mesmo que nao seja essa a jogada que pretende realizar, se tal nao
writeln('for possivel, perde o jogo.'),
writeln('').

% SETUP (PFR)
setup_game(Board) :-
writeln('Quantas linhas pretende no tabuleiro (minimo 5, standard 8): '),
read_int(YSize),
writeln('Quantas colunas pretende no tabuleiro  (minimo 5, standard 8): '),
read_int(XSize),
new_board(XSize, YSize, Board).

init_posicoes(IBoard,OBoard):-
writeln('Jogador Branco indique a sua posição inicial'),
writeln('X: '), read_int(BX),
writeln('Y: '), read_int(BY),
set_piece( IBoard, 'B', BX, BY, OB),
print_board(OB),
writeln('Jogador Preto indique a sua posição inicial'),
writeln('X: '), read_int(PX),
writeln('Y: '), read_int(PY),
set_piece( OB, 'P', PX, PY, OBoard),
print_board(OBoard).
```

```
% START GAME
start_game(Board) :-
init_posicoes(Board,OBoard),
writeln('Comecar a jogar'),
main_loop(OBoard,'P').

endgame(Board,Piece) :-
search_piece(Board,Piece,X,Y),
not(movable(Board,Piece,X,Y)),
!,true.

main_loop(Board,'P'):-
endgame(Board,'P'),
writeln('O Jogador B GANHOU o jogo!').

main_loop(Board,'B'):-
endgame(Board,'B'),
writeln('O Jogador P GANHOU o jogo!').

main_loop(Board,Piece) :-
print_board(Board),
write('Jogador '),
write(Piece),
writeln(' ,onde deseja colocar o Counter?'),
writeln('X: '), read_int(CX),
writeln('Y: '), read_int(CY),
set_piece(Board,'C',CX,CY,OBoard),
print_board(OBoard),
writeln('Deseja fazer Push(s) ou Pull(n)?'),
read_yn(Push),
writeln('Deseja que seja aplicado ao jogador: P(s) ou B(n)?'),
read_yn(Jog),
make_a_move(OBoard,Push,Jog,CX,CY,FBoard),
print_board(FBoard),
next_piece(Piece,OPiece),
main_loop(FBoard,OPiece).


next_piece('P','B').
next_piece('B','P').

%make_a_move(IBoard,Push,Jog,X,Y,OBoard).
make_a_move(IBoard,s,s,X,Y,OBoard):-
move(IBoard,X,Y,'P','push',OBoard).
make_a_move(IBoard,s,n,X,Y,OBoard):-
move(IBoard,X,Y,'P','pull',OBoard).
make_a_move(IBoard,n,s,X,Y,OBoard):-
move(IBoard,X,Y,'B','push',OBoard).
make_a_move(IBoard,n,n,X,Y,OBoard):-
move(IBoard,X,Y,'B','pull',OBoard).
make_a_move(_,P,T,_,_,_):-
```

```
write('wrong move\n'),!,false.


print_board( [LastLine | [] ] ):-
print_line( LastLine),
write('\n\n'), !.
print_board( [Line | Lines ] ):-
print_line( Line),
print_separator(Line),
print_board(Lines).

print_line( [LastPiece | [] ] ):-
write(' '),
write(LastPiece),
write('\n'), !.
print_line( [Piece | Pieces ] ):-
write(' '),
write(Piece),
write(' |'),
print_line( Pieces ).

print_separator( [_|[]] ):-
write('---\n'), !.
print_separator( [_|T] ):-
write('---|'),
print_separator(T).


%search_diagonal(Board,Piece_to_Search, XCounter,YCounter,XPiece,YPiece) =>output:xpiece,y
search_diagonal(Board, P, XC,YC,XP,YP) :-
search_diagonal_forward(Board, P, XC,YC,XP,YP).
search_diagonal(Board, P, XC,YC,XP,YP) :-
search_diagonal_backward(Board, P, XC,YC,XP,YP).
search_diagonal(Board, P, XC,YC,XP,YP) :-
search_reverse_diagonal_forward(Board, P, XC,YC,XP,YP).
search_diagonal(Board, P, XC,YC,XP,YP) :-
search_reverse_diagonal_backward(Board, P, XC,YC,XP,YP).

search_diagonal_forward(Board,_,_,YC,_,_) :-
length(Board,L),
L<YC, !, false.

search_diagonal_forward([H|_],_,XC,_,_,_) :-
length(H,L),
L<XC, !, false.

search_diagonal_forward([H|T], _, XC,YC,XC,YC) :-
length(H,LX),
length(T,LY),
XC==LX,
YC==LY+1,
!, false.
```

```
search_diagonal_forward(Board, P, XC,YC,XC,YC) :-
nth(YC,Board,Row),
nth(XC,Row,P).

search_diagonal_forward(Board, P, XC,YC,XP,YP) :-
TXC is XC + 1,
TYC is YC + 1,
search_diagonal_forward(Board, P, TXC,TYC,XP,YP).



search_diagonal_backward(_,_, XC,_,_,_) :-
XC<1, !, false.

search_diagonal_backward(_,_,_,YC,_,_) :-
YC<1, !, false.

search_diagonal_backward(_,_,XC,YC,_,_) :-
XC==1,YC==1, !, false.

search_diagonal_backward(Board, P, XC,YC,XC,YC) :-
nth(YC,Board,Row),
nth(XC,Row,P).

search_diagonal_backward(Board, P, XC,YC,XP,YP) :-
TXC is XC - 1,
TYC is YC - 1,
search_diagonal_backward(Board, P, TXC,TYC,XP,YP).

%search_reverse_diagonal(Board,Piece_to_Search, XCounter,YCounter,XPiece,YPiece) =>output:
search_reverse_diagonal(Board, P, XC,YC,XP,YP) :-
search_reverse_diagonal_forward(Board, P, XC,YC,XP,YP).
search_reverse_diagonal(Board, P, XC,YC,XP,YP) :-
search_reverse_diagonal_backward(Board, P, XC,YC,XP,YP).

search_reverse_diagonal_forward(Board, _, _,YC,_,_) :-
length(Board,L),
L<YC, !, false.

search_reverse_diagonal_forward(_, _, XC,_,_,_) :-
XC<1, !, false.

search_reverse_diagonal_forward(Board, _, XC,YC,_,_) :-
length(Board,L),
XC==0, YC==L,
!, false.

search_reverse_diagonal_forward(Board, P, XC,YC,XC,YC) :-
nth(YC,Board,Row),
nth(XC,Row,P),
write('found').
```

```
search_reverse_diagonal_forward(Board, P, XC,YC,XP,YP) :-
TXC is XC - 1,
TYC is YC + 1,
search_reverse_diagonal_forward(Board, P, TXC,TYC,XP,YP).



search_reverse_diagonal_backward(_, _, _,YC,_,_) :-
YC<1, !, false.

search_reverse_diagonal_backward([H|_], _, XC,_,_,_) :-
length(H, L),
L<XC, !, false.

search_reverse_diagonal_backward([H|T], _, XC,YC,_,_) :-
length(H, XL),
length(T,YL),
XC==XL, YC==YL+1,
!, false.

search_reverse_diagonal_backward(Board, P, XC,YC,XC,YC) :-
nth(YC,Board,Row),
nth(XC,Row,P).

search_reverse_diagonal_backward(Board, P, XC,YC,XP,YP) :-
TXC is XC + 1,
TYC is YC - 1,
search_reverse_diagonal_backward(Board, P, TXC,TYC,XP,YP).

%search_horizontal(Board,Piece_to_Search, XCounter,YCounter,XPiece,YPiece) =>output:xpiece
search_horizontal(Board, P, XC,YC,XP,YP) :-
search_horizontal_forward(Board, P, XC,YC,XP,YP).
search_horizontal(Board, P, XC,YC,XP,YP) :-
search_horizontal_backward(Board, P, XC,YC,XP,YP).

search_horizontal_forward(Board, _, XC,_,_,_) :-
length(Board, L),
L<XC, !, false.
search_horizontal_forward(Board, P, XC,YC,XC,YC) :-
nth(YC,Board,Row),
nth(XC,Row,P).

search_horizontal_forward(Board, P, XC,YC,XP,YP) :-
TXC is XC + 1,
search_horizontal_forward(Board, P, TXC,YC,XP,YP).

search_horizontal_backward(_, _, XC,_,_,_) :-
XC<1, !, false.
search_horizontal_backward(Board, P, XC,YC,XC,YC) :-
nth(YC,Board,Row),
```

```
nth(XC,Row,P).

search_horizontal_backward(Board, P, XC,YC,XP,YP) :-
TXC is XC - 1,
search_horizontal_backward(Board, P, TXC,YC,XP,YP).


%search_vertical(Board,Piece_to_Search, XCounter,YCounter,XPiece,YPiece) =>output:xpiece,y
%com o OR logico
%search_vertical(Board, P, XC,YC,XP,YP) :- search_vertical_forward(Board, P, XC,YC,XP,YP)
% search_vertical_backward(Board, P, XC,YC,XP,YP).
search_vertical(Board, P, XC,YC,XP,YP) :-
search_vertical_forward(Board, P, XC,YC,XP,YP).

search_vertical(Board, P, XC,YC,XP,YP) :-
search_vertical_backward(Board, P, XC,YC,XP,YP).


search_vertical_forward(Board, _, _,YC,_,_) :-
length(Board,L),
L < YC, !, false.

search_vertical_forward(Board, P, XC,YC,XC,YC) :-
nth(YC,Board,Row),
nth(XC,Row,P).

search_vertical_forward(Board, P, XC,YC,XP,YP) :-
TYC is YC + 1,
search_vertical_forward(Board, P, XC,TYC,XP,YP).

search_vertical_backward(_, _, _,YC,_,_) :-
YC < 1, !, false.
search_vertical_backward(Board, P, XC,YC,XC,YC) :-
nth(YC,Board,Row),
nth(XC,Row,P).

search_vertical_backward(Board, P, XC,YC,XP,YP) :-
TYC is YC - 1,
search_vertical_backward(Board, P, XC,TYC,XP,YP).




%move(Input_Board, XCounter, YCounter, Piece_To_move, PushPull, Output_Board).
move(IBoard, XC, YC, Piece, PushPull, OBoard) :-
move_diagonal(IBoard, XC,YC,Piece,PushPull,OBoard).
move(IBoard, XC, YC, Piece, PushPull, OBoard) :-
move_reverse_diagonal(IBoard, XC,YC,Piece,PushPull,OBoard).
move(IBoard, XC, YC, Piece, PushPull, OBoard) :-
move_horizontal(IBoard, XC,YC,Piece,PushPull,OBoard).
```

```
move(IBoard, XC, YC, Piece, PushPull, OBoard) :-
move_vertical(IBoard, XC,YC,Piece,PushPull,OBoard).

%HORIZONTAL

move_horizontal(IBoard, XC,YC,Piece,'push',OBoard) :-
search_horizontal_forward(IBoard,Piece,XC,YC,XP,YP),
horizontal_forward(IBoard, Piece,YP,OBoard).

move_horizontal(IBoard, XC,YC,Piece,'pull',OBoard) :-
search_horizontal_backward(IBoard,Piece,XC,YC,XP,YP),
horizontal_forward(IBoard, Piece,YP,OBoard).

move_horizontal(IBoard, XC,YC,Piece,'push',OBoard) :-
search_horizontal_backward(IBoard,Piece,XC,YC,XP,YP),
horizontal_backward(IBoard, Piece,YP,OBoard).

move_horizontal(IBoard, XC,YC,Piece,'pull',OBoard) :-
search_horizontal_forward(IBoard,Piece,XC,YC,XP,YP),
horizontal_backward(IBoard, Piece,YP,OBoard).

%VERTICAL

%counter is up fom piece
move_vertical(IBoard, XC,YC,Piece,'push',OBoard) :-
search_vertical_forward(IBoard,Piece,XC,YC,_,_),
vertical_down(IBoard,Piece,XC,OBoard).

move_vertical(IBoard, XC,YC,Piece,'pull',OBoard) :-
search_vertical_forward(IBoard,Piece,XC,YC,_,_),
vertical_up(IBoard,Piece,XC,OBoard).

%counter is down fom piece
move_vertical(IBoard, XC,YC,Piece,'push',OBoard) :-
search_vertical_backward(IBoard,Piece,XC,YC,_,_),
vertical_up(IBoard,Piece,XC,OBoard).

move_vertical(IBoard, XC,YC,Piece,'pull',OBoard) :-
search_vertical_backward(IBoard,Piece,XC,YC,_,_),
vertical_down(IBoard,Piece,XC,OBoard).

%DIAGONAL

move_diagonal(IBoard, XC,YC,Piece,'push',OBoard) :-
search_diagonal_forward(IBoard,Piece,XC,YC,_,_),
top_left(XC,YC,TXC,TYC),
diagonal_down(IBoard,Piece,TXC,TYC,OBoard).

move_diagonal(IBoard, XC,YC,Piece,'pull',OBoard) :-
search_diagonal_forward(IBoard,Piece,XC,YC,_,_),
top_left(XC,YC,TXC,TYC),
diagonal_up(IBoard,Piece,TXC,TYC,OBoard).
```

```
move_diagonal(IBoard, XC,YC,Piece,'push',OBoard) :-
search_diagonal_backward(IBoard,Piece,XC,YC,_,_),
top_left(XC,YC,TXC,TYC),
diagonal_up(IBoard,Piece,TXC,TYC,OBoard).

move_diagonal(IBoard, XC,YC,Piece,'pull',OBoard) :-
search_diagonal_backward(IBoard,Piece,XC,YC,_,_),
top_left(XC,YC,TXC,TYC),
diagonal_down(IBoard,Piece,TXC,TYC,OBoard).


%REVERSE DIAGONAL

move_reverse_diagonal(IBoard, XC,YC,Piece,'push',OBoard) :-
search_reverse_diagonal_forward(IBoard,Piece,XC,YC,_,_),
nth(1,IBoard,Row), length(Row,N),
top_right(N,XC,YC,TXC,TYC),
diagonal_down(IBoard,Piece,TXC,TYC,OBoard).

move_reverse_diagonal(IBoard, XC,YC,Piece,'pull',OBoard) :-
search_reverse_diagonal_forward(IBoard,Piece,XC,YC,_,_),
nth(1,IBoard,Row), length(Row,N),
top_right(N,XC,YC,TXC,TYC),
diagonal_up(IBoard,Piece,TXC,TYC,OBoard).


move_reverse_diagonal(IBoard, XC,YC,Piece,'push',OBoard) :-
search_reverse_diagonal_backward(IBoard,Piece,XC,YC,_,_),
nth(1,IBoard,Row), length(Row,N),
top_right(N,XC,YC,TXC,TYC),
diagonal_up(IBoard,Piece,TXC,TYC,OBoard).

move_reverse_diagonal(IBoard, XC,YC,Piece,'pull',OBoard) :-
search_reverse_diagonal_backward(IBoard,Piece,XC,YC,_,_),
nth(1,IBoard,Row), length(Row,N),
top_right(N,XC,YC,TXC,TYC),
diagonal_down(IBoard,Piece,TXC,TYC,OBoard).




move_piece(B, X, Y, NX, NY, OB):-
  nth(Y,B,ROW),nth(X,ROW,PX),
  set_piece(B,PX,NX,NY,OB1),
  erase_piece(OB1,X,Y,OB),
  true.


/*
```

```
 * horizontal move predicates..
 */

horizontal_forward(IB, Piece,_,OB) :-
search_piece(IB,Piece,X,Y),
horizontal_forward_(IB,X,Y,_,OB).


horizontal_forward_(IB,XP,YP,TB,IB) :-
TX is XP + 1,write('xxxxxxxx1\n'),
not(move_piece(IB,XP,YP,TX,YP,TB)),write('xxxxxxxx2\n').

horizontal_forward_(IB,XP,YP,TB,OB) :-
TX is XP + 1,write('ffffffff1\n'),
nth(YP,IB,Row),nth(TX,Row,' '),write('ffffffff2\n'),
move_piece(IB,XP,YP,TX,YP,TB),write('ffffffff3\n'),
horizontal_forward_(TB,XP,YP,_,OB),write('ffffffff4\n').


horizontal_backward(IB, Piece,_,OB) :-
search_piece(IB,Piece,X,Y),
horizontal_backward_(IB,Piece,X,Y,_,OB).

horizontal_backward_(IB,_,XP,YP,TB,IB) :-
TX is XP - 1,
not(move_piece(IB,XP,YP,TX,YP,TB)).

horizontal_backward_(IB,Piece,XP,YP,TB,OB) :-
TX is XP - 1,
move_piece(IB,XP,YP,TX,YP,TB),
horizontal_backward_(TB,Piece,XP,YP,_,OB).






erase_horizontal_forward( [Piece|IT], Piece, [' '|OT] ) :-
push_h_f(IT,Piece,OT),
!,true.
erase_horizontal_forward( [IH|IT], Piece, [IH|OT] ) :-
erase_horizontal_forward(IT,Piece,OT).


push_h_f( [' '], Piece, Piece):-
!,true.
push_h_f( [' '|[]], Piece, Piece):-
!,true.
push_h_f( [' ',X|IT], Piece, [Piece,X|IT]):-
not(white(X)),!,true.
push_h_f( [' ',' '|IT],Piece,[' '|OT]) :-
```

```
push_h_f( [' '|IT], Piece, OT).


white(' ') :-
true.
white(_) :-
false.


/*
 * Vertical Move predicates
 */



vertical_up(IBoard,Piece,X,OBoard) :-
extract_column(IBoard,X,IL,OL,OBoard),
reverse(IL, REV_IL),
erase_horizontal_forward(REV_IL,Piece,REV_OL),
reverse(REV_OL,OL).

vertical_down(IBoard,Piece,X,OBoard) :-
extract_column(IBoard,X,IL,OL,OBoard),
erase_horizontal_forward(IL,Piece,OL).



/* extract_column(IBoard ,X , ILIST,OLIST, OBOARD).
 *
 * IBoard -> input board
 * X -> index of the column to extract
 * ILIST -> input Column as a List
 * OList -> output Column to manipulate.
 * OBoard -> copy of IBoard but with the column X
 *       with the elements of LIST
 */

extract_column([],_,[],[],[]).
extract_column([IH|IT], X, IL, OL, [IH|OT]) :-
line_extract(IH,X,'+',_,_),
extract_column(IT,X,IL,OL,OT).
extract_column([IH|IT], X, [ILH|ILT], [OLH|OLT], [OH|OT]) :-
line_extract(IH,X,ILH,OLH,OH),
extract_column(IT,X,ILT,OLT,OT).

%line_extract(+ILine,+X,-IX,-OX,-OLine]).
line_extract([IH|IT],1,IH,OH,[OH|IT]) :-
!,true.
line_extract([IH|IT],X,IL,OL,[IH|OT]) :-
TX is X - 1,
line_extract(IT,TX,IL,OL,OT).
```

```
/*
 *   Diagonal moves
 */

diagonal_down(IBoard,Piece,X,Y,OBoard):-
top_left(X,Y,TX,TY),
extract_diagonal(IBoard,TX,TY,IList,OList,OBoard),
erase_horizontal_forward(IList,Piece,OList).


diagonal_up(IBoard,Piece,X,Y,OBoard):-
top_left(X,Y,TX,TY),
extract_diagonal(IBoard,TX,TY,IList,OList,OBoard),
reverse(IList,RIList),
erase_horizontal_forward(RIList,Piece,ROList),
reverse(ROList,OList).


/* extract_diagonal(IBoard ,X,Y, ILIST,OLIST, OBOARD).
 *
 * IBoard -> input board
 * X -> index of the top_left diagonal
 * Y -> index of the top_left diagonal
 * ILIST -> input Column as a List
 * OList -> output Column to manipulate.
 * OBoard -> copy of IBoard but with the column X
 *       with the elements of LIST
 */

extract_diagonal([],_,_,[],[],[]) :-
!, true.
extract_diagonal([IH|IT],X,_,[],[],[IH|IT]):-
LX is X-1,
length(IH,LX),
!, true.

extract_diagonal([IH|IT],X,1,[ILH|ILT],[OLH|OLT],[OH|OT]):-
line_extract(IH,X,ILH,OLH,OH),
TX is X+1,
extract_diagonal(IT,TX,1,ILT,OLT,OT).
extract_diagonal([IH|IT],X,Y,IL,OL,[IH|OT]):-
TY is Y - 1,
extract_diagonal(IT,X,TY,IL,OL,OT).


%top_left(Input_X, Intput_Y, Output_x,Output_Y
%given an XY it gives the top diagonal.
top_left(1,1,2,2).
top_left(1,Y,1,Y).
top_left(X,1,X,1).
```

```
top_left(IX,IY,OX,OY) :-
TX is IX - 1,
TY is IY - 1,
top_left(TX,TY,OX,OY).
```

```
%reverse diagonal


%reverse_diag_down(IBoard, current_X, current_Y,OBoard)
reverse_diagonal_down(IB, X, Y,OB) :-
reverse_diag_down_(IB, X, Y,_,OB).

reverse_diag_down_(IB, X, Y,TB,IB) :-
LX is X - 1,
LY is Y + 1,
not(move_piece(IB,X,Y,LX,LY,TB)).

reverse_diag_down_(IB, X, Y,TB,OB) :-
LX is X - 1,
LY is Y + 1,
move_piece(IB,X,Y,LX,LY,TB),
reverse_diag_down_(TB,LX,LY,_,OB).



%reverse_diag_up(IBoard, current_X, current_Y,OBoard)
reverse_diagonal_up(IB, X, Y,OB) :-
reverse_diag_up_(IB, X, Y,_,OB).

reverse_diag_up_(IB, X, Y,TB,IB) :-
LX is X + 1,
LY is Y - 1,
not(move_piece(IB,X,Y,LX,LY,TB)).

reverse_diag_up_(IB, X, Y,TB,OB) :-
LX is X + 1,
LY is Y - 1,
move_piece(IB,X,Y,LX,LY,TB),
reverse_diag_up_(TB,LX,LY,_,OB).
```

```
/* reverse_extract_diagonal(IBoard ,X,Y, ILIST,OLIST, OBOARD).
 *
 * IBoard -> input board
```

```
 * X -> index of the top_right diagonal
 * Y -> index of the top_right diagonal
 * ILIST -> input Column as a List
 * OList -> output Column to manipulate.
 * OBoard -> copy of IBoard but with the column X
 *       with the elements of LIST
 */
 %y is bottom
extract_reverse_diagonal([],_,_,[],[],[]) :-
!,true.

extract_reverse_diagonal([IBH|_],1,1,[],[], [IBH|[]]) :-
line_extract(IBH,1,'+',_,_),
!,true.

extract_reverse_diagonal([IBH|IBT],1,1,ILH,OLH, [OH|IBT]) :-
line_extract(IBH,1,ILH,OLH,OH),
!,true.

extract_reverse_diagonal( [IBH|IBT] ,X,1, [ILH|ILT],[OLH|OLT], [OH|OT]) :-
line_extract(IBH,X,ILH,OLH,OH),
TX is X - 1,
extract_reverse_diagonal(IBT,TX,1,ILT,OLT,OT).

extract_reverse_diagonal( [IBH|IBT] ,X,Y, IList,OList, [IBH|OT]) :-
TY is Y - 1,
extract_reverse_diagonal(IBT,X,TY,IList,OList,OT).


%top_right(Length(Iboard),IBInput_X, Intput_Y, Output_x,Output_Y
%given an XY it gives the top diagonal.

top_right(N, N, 1, X, 2) :-
X is N-1,
!, true.
top_right(_, X, 1, X, 1) :-
!,true.
top_right(X, X, Y, X, Y) :-
!,true.
top_right(N, X, Y, Ox, Oy) :-
XX is X+1,
YY is Y-1,
top_right(N, XX, YY, Ox, Oy).



%sequential_cell(List, Piece):-
%check if there is 2 valid cells for movement


sequential_cell(List,Piece) :-
sequential_cell_to(List,Piece,0).
```

```
sequential_cell_to([Piece|LT], Piece, N):-
sequential_cell_from(LT,Piece,N).

sequential_cell_to([' '|LT], Piece, N):-
TN is N + 1,
sequential_cell_to(LT,Piece,TN).

sequential_cell_to([_|LT], Piece, _):-
sequential_cell_to(LT,Piece,0).



sequential_cell_from(_, _, 2):-
!,true.

sequential_cell_from([], _, _):-
!,false.

sequential_cell_from([' '|LT], Piece, N):-
TN is N + 1,
sequential_cell_from(LT,Piece,TN).

sequential_cell_from(_, _, _):-
!,false.

%movable(IBoard, Piece, X,Y) :-
%horizontal

movable(IBoard, Piece, _,Y) :-
nth(Y,IBoard,Row),
sequential_cell(Row,Piece).

%vertical
movable(IBoard,Piece,X,_) :-
extract_column(IBoard ,X , IList,_, _),
sequential_cell(IList,Piece).

%diagonal
movable(IBoard,Piece,X,Y) :-
top_left(X,Y,TX,TY),
extract_diagonal(IBoard ,TX,TY, IList,_,_),
sequential_cell(IList,Piece).

%reverse diagonal
movable(IBoard,Piece,X,Y) :-
nth(1,IBoard,Row),
length(Row,Len),
top_right(Len,X,Y,TX,TY),
extract_reverse_diagonal(IBoard ,TX,TY, IList,_,_),
sequential_cell(IList,Piece).
```

```
test_search_piece(B):-
new_board(8,8,B),
set_piece(B,'P',4,5,BO),

print_board(BO),
search_piece(BO,'P',X,Y).



test_movable_reverse_diagonal(B):-

new_board(8,8,B),
set_piece(B,'P',4,4,BO),

print_board(BO),
movable(BO,'P',4,4),
write('correct\n'),


set_piece(BO,'C',3,5,BOB1),
movable(BOB1,'P',4,4),
print_board(BOB1),
write('correct1\n'),


set_piece(BO,'C',6,2,BOBx),
set_piece(BOBx,'C',2,6,BOB2),
movable(BOB2,'P',4,4),
print_board(BOB2),
write('correct2\n'),

set_piece(BO,'C',1,7,BOBs),
set_piece(BOBs,'C',5,3,BOB3),
movable(BOB3,'P',4,4),
print_board(BOB3),
write('correct3\n').


test_movable_diagonal(B):-

new_board(8,8,B),
set_piece(B,'P',4,4,BO),

print_board(BO),
movable(BO,'P',4,4),
write('correct\n'),
```

```
set_piece(BO,'C',5,5,BOB1),
movable(BOB1,'P',4,4),
print_board(BOB1),
write('correct1\n'),


set_piece(BO,'C',2,2,BOBx),
set_piece(BOBx,'C',6,6,BOB2),
movable(BOB2,'P',4,4),
print_board(BOB2),
write('correct2\n'),

set_piece(BO,'C',7,7,BOBs),
set_piece(BOBs,'C',3,3,BOB3),
movable(BOB3,'P',4,4),
print_board(BOB3),
write('correct3\n').



test_movable_vertical(B):-

new_board(8,8,B),
set_piece(B,'P',4,4,BO),

print_board(BO),
movable(BO,'P',4,4),
write('correct\n'),


set_piece(BO,'C',4,5,BOB1),
movable(BOB1,'P',4,4),
print_board(BOB1),
write('correct1\n'),


set_piece(BO,'C',4,2,BOBx),
set_piece(BOBx,'C',4,6,BOB2),
movable(BOB2,'P',4,4),
print_board(BOB2),
write('correct2\n'),

set_piece(BO,'C',4,7,BOBs),
set_piece(BOBs,'C',4,3,BOB3),
movable(BOB3,'P',4,4),
print_board(BOB3),
write('correct3\n').

test_movable_horizontal(B):-

new_board(8,8,B),
```

```
set_piece(B,'P',4,4,BO),

print_board(BO),
movable(BO,'P',4,4),
write('correct\n'),


set_piece(BO,'C',5,4,BOB1),
movable(BOB1,'P',4,4),
print_board(BOB1),
write('correct1\n'),


set_piece(BO,'C',2,4,BOBx),
set_piece(BOBx,'C',6,4,BOB2),
movable(BOB2,'P',4,4),
print_board(BOB2),
write('correct2\n'),

set_piece(BO,'C',7,4,BOBs),
set_piece(BOBs,'C',3,4,BOB3),
movable(BOB3,'P',4,4),
print_board(BOB3),
write('correct3\n').




test_sequencial(_) :-
sequential_cell(['P',' ',' ','B',' ',' ',' ',' ',' '],'P'),write(['P',' ',' ','B',' ',' ',' ',
sequential_cell([' ','P',' ','B',' ',' ',' ',' ',' '],'P'),write([' ','P',' ','B',' ',' ',' ',
sequential_cell([' ',' ','P','B',' ',' ',' ',' ',' '],'P'),write([' ',' ','P','B',' ',' ',' ',
not(sequential_cell([' ',' ','C','P','B',' ',' ',' ',' '],'P')),write([' ',' ','C','P','B',' '
sequential_cell([' ','C','P',' ',' ',' ',' ',' ',' '],'P'),write([' ','C','P',' ',' ',' ',' ',
not(sequential_cell(['P','C',' ',' ',' ',' ',' ',' ',' '],'P')),write(['P','C',' ',' ',' ',' '


test_reverse_diagonal(B) :-
new_board(8,8,B),
test_rev_diag_up(B),
test_rev_diag_down(B).
test_rev_diag_up(B) :-
set_piece(B,'P', 4,6,OB),
print_board(OB),
reverse_diagonal_up(OB,4,6,BOB),
print_board(BOB).


test_rev_diag_down(B) :-
set_piece(B,'P', 2,6,OB),
print_board(OB),
reverse_diagonal_down(OB,2,6,BOB),
```

```
print_board(BOB).



/*
%OLD
test_rev_down(B,IL,OL) :-
new_board(8,8,B),
set_piece(B,'P', 3,6,OB),
print_board(OB),
top_right(8,3,6,X,Y),
extract_reverse_diagonal(OB,X,Y,IL,OL,OBoard),write('IL '),write(IL),write('*\n'),
erase_horizontal_forward(IL,'P',OL),
print_board(OBoard).
%,erase_horizontal_forward([' ',' ',' ',' ','P',' ','+'],'P',OL).
test_rev_up(B,IL,OL) :-
new_board(8,8,B),
set_piece(B,'P', 6,3,OB),
top_right(8,5,4,X,Y),
extract_reverse_diagonal(OB,X,Y,IL,OL,OBoard),
reverse(IL,RIL),
erase_horizontal_forward(RIL,'P',ROL),
reverse(ROL,OL).

test_reverse_diagonal_up(B) :-
new_board(8,8,B),
set_piece(B,'P', 6,3,OB),
print_board(OB),
reverse_diagonal_up(OB,'P',5,4,BOB),
print_board(BOB).

*/




test_diagonal_down(B) :-
new_board(8,8,B),
set_piece(B,'P', 3,3,OB),
print_board(OB),
diagonal_down(OB,'P',2,2,BOB),
print_board(BOB).


test_diagonal_up(B) :-
new_board(8,8,B),
set_piece(B,'P', 6,6,OB),
print_board(OB),
diagonal_up(OB,'P',7,7,BOB),
print_board(BOB).
```

```
test_push(B):-test_push_f(B),test_push_b(B).

test_push_f(B) :-
new_board(8,8,B),
set_piece(B,'P', 5,8,OB),
set_piece(OB,'C', 3,8,BOB),
print_board(BOB),
%push_horizontal_forward(BOB,'P',1,BBOB),
move_horizontal(BOB, 3,8,'P','push',BBOB),
print_board(BBOB).
test_push_b(B) :-
new_board(8,8,B),
set_piece(B,'P', 5,8,OB),
set_piece(OB,'C', 6,8,BOB),
print_board(BOB),
%push_horizontal_backward(BOB,'P',1,BBOB),
move_horizontal(BOB, 6,8,'P','push',BBOB),
print_board(BBOB).


test_vertical(B) :-
test_vertical_down(B),
test_vertical_up(B).


test_vertical_down(B) :-
new_board(8,8,B),
set_piece(B,'P', 8,2,OB),
print_board(OB),
vertical_down(OB,'P',8,BOB),
print_board(BOB).


test_vertical_up(B) :-
new_board(8,8,B),
set_piece(B,'P', 8,6,OB),
print_board(OB),
vertical_up(OB,'P',8,BOB),
print_board(BOB).



test_horizontal_move(B) :-
new_board(8,8,B),
test_pull_f(B),write('pull_f\n\n\n\n\n\n'),
test_push_f(B),write('push_f\n\n\n\n\n\n'),
test_pull_b(B),write('pull_b\n\n\n\n\n\n'),
test_push_b(B),write('push_b\n\n\n\n\n\n').
```

```
test_pull_f(B) :-
set_piece(B,'P', 2,1,OB),
set_piece(OB,'C', 7,1,BOB),
print_board(BOB),
%pull_horizontal_forward(BOB,'P',1,BBOB),
move_horizontal(BOB, 7,1,'P','pull',BBOB),
print_board(BBOB).
/*
test_push_f(B) :-
set_piece(B,'P', 5,1,OB),
set_piece(OB,'C', 3,1,BOB),
print_board(BOB),
%push_horizontal_forward(BOB,'P',1,BBOB),
move_horizontal(BOB, 3,1,'P','push',BBOB),
print_board(BBOB).
*/


test_pull_b(B) :-
set_piece(B,'P', 7,1,OB),
set_piece(OB,'C', 3,1,BOB),
print_board(BOB),
%pull_horizontal_backward(BOB,'P',1,BBOB),
move_horizontal(BOB, 3,1,'P','pull',BBOB),
print_board(BBOB).




test_check(B):- new_board(8,8,B),
set_piece(B,'P', 2,1,OB),
test_diagonal(OB),
test_vertical(OB),
test_horizontal(OB).


test_diagonal(B) :-
search_diagonal(B,'P',3,2,_,_), write('diagonal_backward check\n'),
not(search_diagonal(B,'P',6,6,_,_)), write('diagonal_forward check\n'),
search_diagonal(B,'P',1,2,_,_), write('diagonal_reverse_backward check\n'),
not(search_diagonal(B,'P',2,6,_,_)), write('diagonal_reverse_forward check\n').



test_vertical(B) :- search_vertical(B,'P',2,2,_,_),write('vertival_backward check\n'),
not(search_vertical(B,'P',4,6,_,_)),write('vertival_forward check\n').

test_horizontal(B) :-
search_horizontal(B,'P',4,1,_,_),write('horizontal_backward check\n'),
```

```
not(search_horizontal(B,'P',2,4,_,_)),write('horizontal_forward check\n').
```