```
% TIC TAC TOE
% Game details for use by "play.pl"

:- ensure_loaded('play.pl').

% State representation: [[P1,P2,P3],[P4...],[P7 ... P9]] represents state
%
%         P1 P2 P3
%         P4 P5 P6
%         P7 P8 P9
%
% where each P is one of x, o, or e (empty)

%% Define needed interface functions for mmeval.

%% 1.
%
% * initialize(InitialState,InitialPlyr)
%   - returns an initial game state and Initial player

% Initial state/plyr: no entries on board, player 1 (x) moves first

initialize([[e,e,e],[e,e,e],[e,e,e]],1).

%
% * winner(State,Plyr)
%   - returns winning player if State is a terminal position: If ties
%     allowed, winner could return "Player 0". But see "ties" predicate too.
%
% winning positions (and player who wins)
% simple enumeration OK for such a simple game

winner([[x,x,x],_,_],1).
winner([_,[x,x,x],_],1).
winner([_,_,[x,x,x]],1).
winner([[x|_],[x|_],[x|_]],1).
winner([[_,x,_],[_,x,_],[_,x,_]],1).
winner([[_,_,x],[_,_,x],[_,_,x]],1).
winner([[x,_,_],[_,x,_],[_,_,x]],1).
winner([[_,_,x],[_,x,_],[x,_,_]],1).

winner([[o,o,o],_,_],2).
winner([_,[o,o,o],_],2).
winner([_,_,[o,o,o]],2).
winner([[o|_],[o|_],[o|_]],2).
winner([[_,o,_],[_,o,_],[_,o,_]],2).
winner([[_,_,o],[_,_,o],[_,_,o]],2).
winner([[o,_,_],[_,o,_],[_,_,o]],2).
winner([[_,_,o],[_,o,_],[o,_,_]],2).

% tie: if no empty spaces (terminal) and nobody won, we have a tie
%
% * tie(State)
%   - true if terminal State is a "tie" (no winner)

tie([R1,R2,R3]) :- \+ winner([R1,R2,R3],_),
                   notmember(e,R1), notmember(e,R2), notmember(e,R3).

%
% * terminal(State)
%   - true if State is a terminal
%
% terminal states
terminal(State) :- winner(State,_).
terminal(State) :- tie(State).


%
% * showState(State) prints out the current state of the game
%                     so that the human player can understand where
%                     they are in the game.
%
% how to display state to terminal
showState([R1,R2,R3]) :-
  writeln(R1), writeln(R2), writeln(R3).

% * moves(Plyr,State,MvList)
%   - returns list MvList of all legal moves Plyr can make in State
%
% Generate a list of possible moves
% moves(Plyr,State,MvList)
% - strategy: simply enumerate all nine positions and test if the move
%   is valid; if so, add it to the list

moves(_,State,MvList) :-
  testmoves(9,State,[],MvList).

testmoves(Pos,St,SoFar,MvList) :-
  Pos > 0,
  validmove(_,St,Pos), !,
  NextPos is Pos - 1,
  testmoves(NextPos,St,[Pos|SoFar],MvList).

testmoves(Pos,St,SoFar,MvList) :-
  Pos > 0,
  NextPos is Pos - 1,
  testmoves(NextPos,St,SoFar,MvList).

testmoves(0,_,MvList,MvList).


% * nextState(Plyr,Move,State,NewState,NextPlyr)
%   - given that Plyr makes Move in State, it determines next state
%    (NewState) and next player to move (NextPlayer).
%

nextState(1,Mv,St,NSt,2) :- replPos(Mv,St,x,NSt).
nextState(2,Mv,St,NSt,1) :- replPos(Mv,St,o,NSt).

% replPos(PosNumber,Board,Mark,NewBoard)
% - given a position number on Board, replace whatever mark was on
%   that position with Mark to obtain NewBoard

replPos(1,[[_,P2,P3],R2,R3],Mark,[[Mark,P2,P3],R2,R3]).
replPos(2,[[P1,_,P3],R2,R3],Mark,[[P1,Mark,P3],R2,R3]).
replPos(3,[[P1,P2,_],R2,R3],Mark,[[P1,P2,Mark],R2,R3]).
replPos(4,[R1,[_,P5,P6],R3],Mark,[R1,[Mark,P5,P6],R3]).
replPos(5,[R1,[P4,_,P6],R3],Mark,[R1,[P4,Mark,P6],R3]).
replPos(6,[R1,[P4,P5,_],R3],Mark,[R1,[P4,P5,Mark],R3]).
replPos(7,[R1,R2,[_,P8,P9]],Mark,[R1,R2,[Mark,P8,P9]]).
replPos(8,[R1,R2,[P7,_,P9]],Mark,[R1,R2,[P7,Mark,P9]]).
replPos(9,[R1,R2,[P7,P8,_]],Mark,[R1,R2,[P7,P8,Mark]]).

%
% * validmove(Plyr,State,Proposed)
%   - true if Proposed move by Plyr is valid at State.
```

```
%
% what are the valid moves in every state

validmove(_,[[e,_,_],_,_],1).
validmove(_,[[_,e,_],_,_],2).
validmove(_,[[_,_,e],_,_],3).
validmove(_,[_,[e,_,_],_],4).
validmove(_,[_,[_,e,_],_],5).
validmove(_,[_,[_,_,e],_],6).
validmove(_,[_,_,[e,_,_]],7).
validmove(_,[_,_,[_,e,_]],8).
validmove(_,[_,_,[_,_,e]],9).

%
% * h(State,Val)
%   - given State, returns heuristic Val of that state
%   - larger values are good for Max, smaller values are good for Min
%   NOTE1. that since we doing depth bounded Min-Max search, we will not
%   always reach terminal nodes. Instead we have to terminate with a
%   heuristic evaluation of the depth-bounded non-terminal states.
%   NOTE2. If State is terminal h should return its true value.

h(State,1) :- winner(State,1), !.
h(State,-1) :- winner(State,2), !.
h(State,0) :- tie(State), !.
h(_,0). % otherwise no heuristic guidance used


%
% * lowerBound(B)
%   - returns a value B less than the actual utility or heuristic value
%     of any node (i.e., less than Min’s best possible value)
lowerBound(-2).

%
% * upperBound(B)
%   - returns a value B greater than the actual utility or heuristic value
%     of any node (i.e., greater than Max’s best possible value)

upperBound(2).

% notmember(E,L) is true if E is not a member of list L
notmember(_,[]).
notmember(N,[H|T]) :- N \= H, notmember(N,T).
```