```prolog
% Generic interactive Game shell using Minimax search

% Copyright (c) 2002 Craig Boutilier
% modified for SWI by Fahiem Bacchus


% Human is player 1
% Computer is player 2.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% to utilize the shell, one has to define the rules and states of the
% game. Like the search routines, the shell is designed to take as
% input predicates that tell it, e.g., what are the new states yielded
% by what moves.
%
% In particular the code depends on the following game-specific state
% predicates
%
%
% * initialize(InitialState,InitialPlyr)
%   - returns an initial game state and Initial player
%     (for the initial game state  you can use initBoard(B))
%
% * winner(State,Plyr)
%   - returns winning player if State is a terminal position and
%     Plyr has a higher score than the other player
%
% * tie(State)
%   - true if terminal State is a "tie" (no winner)
%
% * terminal(State)
%   - true if State is a terminal
%
% * showState(State) prints out the current state of the game
%                    so that the human player can understand where
%                    they are in the game.
%                    (You can simply use printGrid(B) here)
%
% * moves(Plyr,State,MvList)
%   - returns list MvList of all legal moves Plyr can make in State
%
% * nextState(Plyr,Move,State,NewState,NextPlyr)
%   - given that Plyr makes Move in State, it determines next state
%    (NewState) and next player to move (NextPlayer). That is, it
%    changes State by playing Move.
%
% * validmove(Plyr,State,Proposed)
%   - true if Proposed move by Plyr is valid at State.
%
% * 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.
%
% * 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)
%
% * 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)
%
% Note that lowerBound and upperBound are static properties of the
% game.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%  MAIN PLAY ROUTINE

play :- initialize(InitState,Plyr), playgame(Plyr,InitState).

% playgame(Plyr,State) - plays the game from State with Plyr moving first
% - tests for a winner; if not, get move from player, determine next State
%   and player, and continue from new state/player

playgame(_,State) :-
  winner(State,Winner), !,
  % winner(State,Winner,Score),
  write('Win by Player number '), writeln(Winner).
  % write('Win by Player number '), write(Winner),
  % write('With Score '), writeln(Score).

playgame(_,State) :-
  tie(State), !,
  writeln('Game ended with no winner!').

playgame(Plyr,State) :-
  getmove(Plyr,State,Move),
  write('The move chosen is : '),
  writeln(Move),
  nextState(Plyr,Move,State,NewState,NextPlyr),
  playgame(NextPlyr,NewState).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% getmove(Player,State,Move)
% If Player = 1, move obtained from stdio
% If Player = 2, move obtained using search

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get move for player 1 (human)
% - show state, ask for move, verify if move is valid
% - if move is invalid, recall getmove until a valid move is input

getmove(1,State,Move) :-
  showState(State),
  write('Please input move followed by a period: '),
  read(Proposed),
  validmove(1,State,Proposed), !,
  Move = Proposed.

getmove(1,State,Move) :-
  writeln('Invalid Move Proposed.'),
  getmove(1,State,Move).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get move for player 2 (computer)
% - do this using minimax evaluation
%
% SET DEPTH BOUND HERE
%  Depth should be set appropriately (last argument of mmeval).

getmove(2,State,Move) :-
```

```
  showState(State),
  writeln('Computer is moving...'),
  mmeval(2,State,_,Move,6,SeF),
  write('Compute Move computed by searching '),
  write(SeF),
  writeln(' states.').

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% mini-max eval
% mmeval(Plyr,State,Value,BestMove,Depth,StatesSearched)
%     - does minimax evaluation
%  of State, assuming move by Plyr (1 = max, 2 = min) to bound Depth.
%  returns Value of the state, as well a BestMove for the player (either
%  the move with max or min value depending on player)
%  Assume evaluation function h


% if State is terminal, use evaluation function
mmeval(_,State,Val,_,_,1) :- terminal(State), !,
  %writeln('Evaluation reached Terminal'),
  h(State,Val).

% if depth bound reached, use evaluation function
mmeval(_,State,Val,_,0,1) :-  !,
  %writeln('Evaluation reached Depth Bnd'),
  h(State,Val).

% FOR MAX PLAYER
% we assume that if player has no moves available, the position is
% terminal and would have been caught above

mmeval(1,St,Val,BestMv,D,SeF) :-
  moves(1,St,MvList), !,
% length(MvList,L),
% write('Evaluating '), write(L), write(' moves at Plyr 1 depth '), writeln(D),
  lowerBound(B), % a value strictly less than worst value max can get
  evalMoves(1,St,MvList,B,null,Val,BestMv,D,0,SeI), % Best so far set to lowerbnd
  SeF is SeI + 1.  %searched the current state as well as

% FOR MIN PLAYER
% we assume that if player has no moves available, the position is
% terminal and would have been caught above

mmeval(2,St,Val,BestMv,D,SeF) :-
  moves(2,St,MvList), !,
% length(MvList,L),
% write('Evaluating '), write(L), write(' moves for Plyr 2 at depth '), writeln(D),
  upperBound(B), % a value strictly less than worst value max can get
  evalMoves(2,St,MvList,B,null,Val,BestMv,D,0,SeI), % Best so far set to upperbnd
  SeF is SeI + 1.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% evalMoves(Plyr,State,MvList,ValSoFar,MvSoFar,Val,BestMv,D,Se,SeF)
%
% - evaluates all moves in MvList for Plyr at State.
% - returns minimax value Val of State by recursively evaluating each
%   successor state, returning BestMv (move that guarantees this value)
% - it has as arguments, the best ValSoFar and best MvSoFar of any other
%   moves that have already been processed (i.e., that have been
%   removed from the current list of moves).
% - a depth bound D is enforced.
%  Se is number of states searched so far.
%  SeF is the total number of states searched to evalute all of these moves.

% if no moves left, return best Val and Mv so far (and number of
% states searched.
evalMoves(_,_,[],Val,BestMv,Val,BestMv,_,Se,Se) :- !.
%       write('No more moves Val = '), write(Val),
%       write(' BestMv = '), write(BestMv), nl.

% otherwise evaluate current move (by calling mmeval on the player/state
% that results from this move), and replace current Best move and value
% by this Mv/Value if value is "better"

evalMoves(1,St,[Mv|Rest],ValSoFar,MvSoFar,Val,BestMv,D,Se,SeF) :-
  nextState(1,Mv,St,NewSt,NextPlyr), !,
%  write('evalMoves 1: '), write(Mv), write(' D='), write(D), write(' S='), write(Se), showS
tate(NewSt),
  Dnew is D - 1,
  mmeval(NextPlyr,NewSt,MvVal,_,Dnew,SeI), !,
  maxMove(ValSoFar,MvSoFar,MvVal,Mv,NewValSoFar,NewMvSoFar),
  SeNew is Se + SeI,
  evalMoves(1,St,Rest,NewValSoFar,NewMvSoFar,Val,BestMv,D,SeNew,SeF).


evalMoves(2,St,[Mv|Rest],ValSoFar,MvSoFar,Val,BestMv,D,Se,SeF) :-
  nextState(2,Mv,St,NewSt,NextPlyr), !,
%  write('evalMoves 2: '), write(Mv), write(' D='), write(D), write(' S='), write(Se), showS
tate(NewSt),
  Dnew is D - 1,
  mmeval(NextPlyr,NewSt,MvVal,_,Dnew,SeI),  !,
  minMove(ValSoFar,MvSoFar,MvVal,Mv,NewValSoFar,NewMvSoFar),
  SeNew is Se + SeI,
  evalMoves(2,St,Rest,NewValSoFar,NewMvSoFar,Val,BestMv,D,SeNew,SeF).

%% Return the max of best so far and the current move.
maxMove(V1,M1,V2,_,V1,M1) :- V1 >= V2.
maxMove(V1,_,V2,M2,V2,M2) :- V1 < V2.
%% Return the min of best so far and the current move.
minMove(V1,M1,V2,_,V1,M1) :- V1 =< V2.
minMove(V1,_,V2,M2,V2,M2) :- V1 > V2.
```