/* This modified version of the Kalah program was developed by Professor Bansal to play with Professor Leon Sterling during Sabbatical to Melbourne University in 1996. The original
version of this program is given in Professor Sterling's book 'The Art of Prolog'.

The program is based upon Alpha beta pruning and a strategy that changes the distribution style
based upon where it is the beginning game, middle game or end game.

It is quite a good program. However, once a while you can beat it.

You are free to copy this program for research and academic purpose including teaching.
If you improve the program to make it more humanistic, please pass on the better copy.
Note that humans and computers have different strategy. Generally human players commit
based upon past experience, and do not explore all the possibilities.

Please do not remove this claimer as it has a historical value in Author's mind */

/* Play framework */

```
     kalah :-
        credits,
        initialize(kalah,Position, Player),
        display_game(Position),
        play(Position,Player, _).

     play(Position, _,Result) :-
        game_over(Position, Result), !,
        announce(Result).
     play(Position,Player,Result) :-
        choose_move(Position,Player,Move),
        move(Move,Position,Position1, Side),
        display_game(Position1),
        next_player(Side, Player, Player1),
        !,
        play(Position1,Player1,Result).

credits :-
        format("This modified version was Developed by Professor Arvind Bansal~n \
  during his sabbatical at Melbourne University in 1996. The original~n \
  program is given in Professor Sterling's book 'The Art of Prolog'~n", []),
```

```
        format("The program varies the strategy based upon beginning, middle, and
end game.~n", []),
        format("Free to copy and distribute for academic and research purpose
only.~n", []),!.

 /* Choosing a move by minimax with alpha-beta cut-off  */

     choose_move(Position,computer,Move) :-
        lookahead(Depth),
        lowest_bound(Alpha),
        highest_bound(Beta),
        alpha_beta(Depth,Position, Alpha, Beta, Move, _),
        nl, writeln(['My move:']), write(Move), nl.
     choose_move(Position,opponent, LegalMove) :-
        nl, writeln(['Your move:']), read(Move),
        (legal(Move, Position) -> LegalMove is Move
        ;otherwise ->
          writeln(['Illegal Move - go again:']),
          choose_move(Position,opponent, LegalMove)
        ).

evaluate_and_choose([Move|Moves], Position, D, Alpha, Beta, Move1, BestMove) :-
     computer_move(Move, Position, Position1),
     alpha_beta(D, Position1, Alpha, Beta, _, Value),
     Value1 is -Value,
     cutoff(Move, Value1, D, Alpha, Beta, Moves,
                            Position, Move1, BestMove).
evaluate_and_choose([], _, _, Alpha, _, Move, (Move, Alpha)).

     alpha_beta(0,Position, _, _, _,Value) :-
        value(Position,Value).
     alpha_beta(D, Position, _, _, _, Value) :-
        D >= 0,
        is_my_side_empty(Position),
        value(Position,Value).
     alpha_beta(D, Position, _, _, _, Value) :-
        D >= 0,
        check_if_finished(Position, FinalBoard),
        value(FinalBoard,Value).
     alpha_beta(D,Position,Alpha,Beta,Move,Value) :-
        D > 0,
        findall(Hole, computer_move(Hole, Position, _), Moves),
        Alpha1 is -Beta,
        Beta1 is -Alpha,
        D1 is D-1,
        Moves = [M|_],
        evaluate_and_choose(Moves,Position, D1,Alpha1,Beta1, M,(Move,Value)).
```

```
     cutoff(Move,Value, _, _, Beta, _, _, _,(Move,Value)) :-
        Value >= Beta.
     cutoff(Move,Value,D,Alpha,Beta,Moves,Position, _,BestMove) :-
        Alpha < Value, Value < Beta,
        evaluate_and_choose(Moves,Position,D,Value, Beta,Move,BestMove).
     cutoff(_,Value,D,Alpha,Beta,Moves,Position,Move1,BestMove) :-
        Value =< Alpha,
        evaluate_and_choose(Moves, Position,D,Alpha,Beta,Move1,BestMove).

/*  Executing a move  */

move(Move, Board, NewBoard, reverse) :-
        is_computer_move(Board), !,
        computer_move(Move, Board, NewBoard).
move(Move, Board, NewBoard, Side) :-
        single_move(Move, Board, NewBoard, Side).

computer_move([Hole|Hs], Board, NewBoard) :-
        board_index(BoardIndex),
        member(Hole, BoardIndex),
        has_stones(Hole, Board),
        single_move(Hole, Board, Board1, Side),
        (Side == same ->
            computer_move(Hs, Board1, NewBoard)
        ;
            NewBoard = Board1, Hs = []
        ).

     single_move(Hole,Board, NewBoard, Side) :-
           has_stones(Hole, Board), !,
           pick_up_stones(Hole, Board, (Stones, PickedUpBoard)),
           board_size(BoardSize),
           distribute_all_stones(Stones, BoardSize, Hole,
                     PickedUpBoard, UpdatedBoard, Side),
           (check_if_finished(UpdatedBoard, FinalBoard) ->
               board_alignment(reverse, FinalBoard, NewBoard)
           ; otherwise ->
                FinalBoard = UpdatedBoard,
                board_alignment(Side, UpdatedBoard, NewBoard)
           ).


stones_in_hole(Hole, board(_, _,  Hs, _, _, _),Stones) :-
        nth_member(Hole, Hs,Stones), Stones > 0, !.
has_stones(Hole, Board) :-
        stones_in_hole(Hole, Board, _).
```

```
pick_up_stones(Hole, Board,  (Stones, NewBoard)) :-
        stones_in_hole(Hole, Board, Stones),
        Board = board(Player, Count,  Hs, K, Ys, L),
        remove_stones(Hole, Hs, Hs1),
        NewBoard = board(Player, Count, Hs1, K, Ys, L), !.

distribute_all_stones(0, _,  _, Board, Board, _) :- !.
distribute_all_stones(Stones, BoardSize, Hole, Board, NewBoard, Side) :-
   Stones > 0, !,
   distribute_once(Stones, BoardSize, Hole, Board, Board1, Stones1, Side),
   distribute_all_stones(Stones1,  BoardSize, 0, Board1, NewBoard, Side).

distribute_once(Stones, BoardSize, Hole, Board, Board2,
                                          RemainingStones, Side) :-
   distribute_player_holes(Stones, BoardSize, Hole,Board,Board1,Stones1, Side),
   distribute_opponent_holes(Stones1, BoardSize, Board1, Board2,
                              RemainingStones, Side).

distribute_player_holes(Stones, BoardSize, Hole,
        board(Player, Count, Hs,K,Ys,L), NewBoard, 0, Side) :-
        Stones =:= BoardSize - Hole + 1, !,
        distribute_from_the_hole(Hole, Stones,Hs,Hs1),
        K1 is K + 1,
        NewBoard = board(Player, Count, Hs1, K1, Ys, L),
        (has_no_stones(Hs1) ->
            Side = reverse
        ;
            Side = same
        ).

distribute_player_holes(Stones, BoardSize, Hole,
                  board(Player,Count, Hs,K,Ys,L), NewBoard, 0, reverse) :-
  Stones < BoardSize - Hole + 1, !,
  distribute_from_the_hole(Hole, Stones,Hs,Hs1),
  FinalHole is Hole + Stones,
  BoardBeforeCapture = board(Player, Count, Hs1, K, Ys, L),
  capture_if_possible(FinalHole, BoardBeforeCapture, NewBoard).
distribute_player_holes(Stones, BoardSize,
    Hole, board(Player, Count, Hs,K,Ys,L), NewBoard, RemainingStones, _) :-
  Stones > BoardSize - Hole + 1, !,
  distribute_from_the_hole(Hole,Stones,Hs,Hs1),
  K1 is K + 1,
  NewBoard = board(Player, Count, Hs1, K1, Ys, L),
  RemainingStones is  Stones + Hole - BoardSize - 1.


distribute_opponent_holes(0, _, Board,Board, 0, _) :- !.
distribute_opponent_holes(Stones, BoardSize, board(Player, Count, Hs,K,Ys,L),
```

```
                    board(Player, Count, Hs,K,Ys1,L), 0, reverse) :-
  Stones =< BoardSize, !,
  distribute(Stones,Ys,Ys1).
distribute_opponent_holes(Stones, BoardSize, board(Player, Count, Hs,K,Ys,L),
               board(Player, Count, Hs,K,Ys1,L), RemainingStones, _) :-
  Stones > BoardSize, !,
  distribute(Stones,Ys,Ys1),
  RemainingStones is Stones - BoardSize.


capture_if_possible(FinalHole, board(Player, Count, Hs, K, Ys, L), NewBoard) :-
   mirror_hole(FinalHole, MirrorHole),
   was_empty_hole(FinalHole, Hs),
   has_some_stones(MirrorHole, Ys), !,
   capture(FinalHole, MirrorHole,
                 board(Player, Count, Hs, K, Ys, L), NewBoard).
capture_if_possible(_, Board, Board).

mirror_hole(MyHole, YourHole) :-
        board_size(BoardSize),
        YourHole is BoardSize - MyHole + 1, !.

was_empty_hole(FinalHole, MySide) :-
          nth_member(FinalHole, MySide, Stones),
          Stones =:= 1, !.

capture(FinalHole, MirrorHole, board(Player, Count, Hs, K, Ys, L),
                                board(Player, Count, Hs1, K1, Ys1, L)) :-
        nth_member(MirrorHole, Ys, Stones),
        K1 is K + 1 + Stones,
        remove_stones(FinalHole, Hs, Hs1),
        remove_stones(MirrorHole, Ys, Ys1).


remove_stones(1, [_|Hs], [0|Hs]) :- !.
remove_stones(N, [H|Hs], [H|Hs1]) :-
        N > 1, !,
        N1 is  N - 1,
        remove_stones(N1, Hs, Hs1).


/*  Lower level stone distribution    */


distribute_from_the_hole(0, Stones, Hs,Hs1) :-
  !, distribute(Stones, Hs,Hs1).
distribute_from_the_hole(Holes, Stones, [H|Hs],[H|Hs1]) :-
  Holes > 0, !, HolesLeft is Holes - 1,
```

```
  distribute_from_the_hole(HolesLeft, Stones, Hs,Hs1).

     distribute(0,Hs,Hs) :- !.
     distribute(N,[H|Hs],[H1|Hs1]) :-
        N > 0, !, N1 is N-1, H1 is H+1, distribute(N1,Hs,Hs1).
     distribute(_,[],[]) :- !.

/*   Evaluation function        */

     value(board(_, Count, Hs, K, Ys,L),Value) :-
        is_early_game(Count), !,
        weight(early, Weight),
        potential_for_recirculation(Hs, Ys, KRec, LRec),
        Value is K - L + Weight * (KRec - LRec).

     value(board(_, Count, Hs, K, Ys,L),Value) :-
        is_middle_game(Count), !,
        weight(middle, Weight),
        recircutable(Hs, Ys, KRec, LRec),
        Value is K - L + Weight * (KRec - LRec).

     value(board(_, Count,  Hs, K, Ys,L),Value) :-
        is_end_game(Count), !,
        weight(end, Weight),
        non_recircutable(Hs, Ys, KNon, LNon),
        Value is K - L + Weight * (KNon - LNon).

is_early_game(Count) :- Count =< 6, !.

is_middle_game(Count) :- Count > 6, Count =< 14.

is_end_game(Count) :- Count > 14.

weight(early, 0.3).
weight(middle, 0.5).
weight(end, 0.8).

potential_for_recirculation(Hs, Ys, KRec, LRec) :-
        potential(Hs, KRec),
        potential(Ys, LRec).


potential(Side, Rec) :-
        board_size(BoardSize),
        MidPoint is (BoardSize + 1) //2,
        MidPoint1 is BoardSize - MidPoint + 1,
        circulate_first_part(MidPoint, Side, LastHalf),
        can_come_back(LastHalf, MidPoint1, BoardSize, Rec).
```

```
circulate_first_part(0, Hs, Hs).
circulate_first_part(N, [Stones|Hs], HsNew) :-
        N > 0,
        distribute(Stones, Hs, Hs1),
        N1 is N - 1,
        circulate_first_part(N1, Hs1, HsNew).

recircutable(Hs, Ys, K, L) :-
        board_size(BoardSize),
        MidPoint is BoardSize //2,
        MidPoint1 is BoardSize - MidPoint,
        recirculate(Hs, MidPoint, MidPoint1, K),
        recirculate(Ys, MidPoint, MidPoint1, L).

recirculate(Hs, 0, MidPoint1, K) :-
        board_size(BoardSize),
        can_come_back(Hs, MidPoint1, BoardSize, K).
recirculate([_|Hs], N, MidPoint1, K) :-
        N > 0, !,
        N1 is N - 1,
        recirculate(Hs, N1, MidPoint1, K).

can_come_back(Hs,  N, BoardSize,  Sum) :-
        can_come_back(Hs,  N, BoardSize, 0, Sum).

can_come_back([], _, _, Sum, Sum).
can_come_back([Stones|Hs], N, BoardSize,  Acc, Sum) :-
         Barrier is 2 * BoardSize - N - 1,
        (Stones > Barrier ->
            Acc1 is Acc + Stones - Barrier
        ; Acc1 is Acc
        ),
        N1 is N - 1,
        can_come_back(Hs, N1, BoardSize,  Acc1, Sum).

non_recircutable(Hs, Ys, KNon, LNon) :-
        board_size(BoardSize),
        nc_count(Hs, 1, BoardSize, 0, KNon),
        nc_count(Ys, 1, BoardSize,  0, LNon).

nc_count([], _, _, Knon, Knon).
nc_count([Stones|Hs], N, BoardSize, Acc,  KNon) :-
         N + Stones =< BoardSize + 1,
        Acc1 is Acc + Stones,
        N1 is N + 1,
        nc_count(Hs, N1, BoardSize, Acc1,  KNon).
nc_count([Stones|Hs], N, BoardSize, Acc, KNon) :-
```

```
        N + Stones > BoardSize + 1,
        Acc1 is Acc - Stones,
        N1 is N + 1,
        nc_count(Hs, N1, BoardSize, Acc1,  KNon).


/*  Testing for the end of the game     */

     game_over(Board, Result) :-
         check_if_finished(Board, FinalBoard), !,
         display_game(FinalBoard),
         conclude_result(FinalBoard, Result).

check_if_finished(board(Player, Count, Hs,K,Ys,L), FinalBoard) :-
  ((has_no_stones(Hs), has_no_stones(Ys)) ->
        FinalBoard = board(Player, Count, [0, 0, 0, 0, 0, 0], K, [0,0,0,0,0,0], L)
  ;has_no_stones(Ys) ->
        sumlist(Hs, LastStones),
        Knew is K + LastStones,
        FinalBoard = board(Player, Count, [0, 0, 0, 0,0, 0], Knew, [0,0,0,0,0,0],
L)
    ;has_no_stones(Hs) ->
        sumlist(Ys, LastStones),
        Lnew is L + LastStones,
        FinalBoard = board(Player, Count, [0, 0, 0, 0, 0, 0], K, [0, 0, 0, 0, 0,
0], Lnew)
    ;otherwise -> fail
    ).


conclude_result(board(_, _, 0,N,0,N), result(draw, 0)) :- !.
conclude_result(board(Player, _, _,K, _, L), result(Opponent, Diff)) :-
          L > K, !,
          Diff is L - K,
          next_player(reverse, Player, Opponent).
conclude_result(board(Player, _, _, K,_,L), result(Player, Diff)) :-
        K > L, !,
        Diff is K - L.


     announce(result(opponent, Diff)) :-
           writeln(['You won by ', Diff, ' stones', '! Congratulations.']).
     announce(result(computer, Diff)) :-
            writeln(['I won by ', Diff, ' stones.']).
     announce(result(draw, _)) :- writeln(['The game is a draw.']).

/*  Miscellaneous game utilities        */
```

```
        nth_member(N,[_|Hs],K) :-
            N > 1, !, N1 is N - 1, nth_member(N1,Hs,K).
        nth_member(1,[H|_], H).

n_substitute(1,[_|Xs],Y,[Y|Xs]) :- !.
n_substitute(N,[X|Xs],Y,[X|Xs1]) :-
  N > 1, !, N1 is N-1, n_substitute(N1,Xs,Y,Xs1).

     next_player(reverse, computer,opponent).
     next_player(reverse, opponent,computer).
     next_player(same, X, X).

     legal(Hole, Board) :-  board_size(Size),
                 0 < Hole, Hole =< Size,
                 has_stones(Hole, Board), !.

member(X, [X|_]).
member(X, [_|Ys]) :- member(X, Ys).

is_computer_move(board(computer, _, _, _, _, _)) :- !.

board_alignment(same, X, X).
board_alignment(reverse, Board, NewBoard) :-
        swap(Board, NewBoard).

     swap(board(computer, Count, Hs,K,Ys,L),
         board(opponent, NewCount, Ys,L,Hs,K)) :-  NewCount is Count + 1.
     swap(board(opponent, Count, Hs,K,Ys,L),
                    board(computer, NewCount, Ys,L,Hs,K)) :-
            NewCount is Count + 1.

     display_game(board(opponent, _, H,K,Y,L)) :-
        reverse(H,HR),  write_stones(HR), write_kalahs(K,L),
        write_stones(Y), nl.
     display_game(board(computer, _, H,K,Y,L)) :-
        reverse(Y,YR),  write_stones(YR), write_kalahs(L, K),
        write_stones(H), nl.

     write_stones(H) :-
        nl, tab(5), display_holes(H).

     display_holes([H|Hs]) :-
        write_pile(H), display_holes(Hs).
     display_holes([]) :- nl.

        write_pile(N) :- N < 10, !, write(N), tab(4).
        write_pile(N) :- N >= 10, !, write(N), tab(3).
```

```
     write_kalahs(K,L) :-
        write(K), tab(34), write(L), nl.


is_my_side_empty(board(_, _, Hs, _, _, _)) :-
        has_no_stones(Hs), !.

     has_no_stones(Side) :- board_size(Size), empty_board(Size, Side).

     has_some_stones(Hole, Side) :-
        nth_member(Hole, Side, Stones),
        Stones > 0, !.

empty_board(0, []) :- !.
empty_board(N, [0|Hs]) :-
        N > 0, !,
        N1 is N - 1,
        empty_board(N1, Hs).

initial_pieces(0, _,  []) :- !.
initial_pieces(N, Stones, [Stones|Hs]) :-
        N > 0, !,
        N1 is N - 1,
        initial_pieces(N1, Stones, Hs).

board_index(BoardIndex) :-
        board_size(BoardSize),
        range(1, BoardSize, BoardIndex).

range(N, M, [N|Bs]) :-
        N =< M, !,
        N1 is N + 1,
        range(N1, M, Bs).
range(N, M, []) :- N > M, !.

reverse(Xs, Ys) :- reverse(Xs, [], Ys).

reverse([X|Xs], Acc, Ys) :- reverse(Xs, [X|Acc], Ys).
reverse([], Ys, Ys).


sumlist(Xs, N) :- sumlist(Xs, 0, N).

sumlist([], N, N).
sumlist([X|Xs], Acc, N) :-
        Acc1 is X + Acc,
        sumlist(Xs, Acc1, N).
```

```
writeln([]) :- nl.
writeln([X|Xs]) :- write(X), writeln(Xs).

lowest_bound(Alpha) :-
        board_size(BoardSize),
        pieces(Pieces),
        Alpha is - (BoardSize * Pieces //2 + 1).
highest_bound(Beta) :-
        board_size(BoardSize),
        pieces(Pieces),
        Beta is  (BoardSize * Pieces //2 + 1).


/*  Initializing        */

     initialize(kalah, board(opponent, 1, MySide, 0, YourSide,0), opponent) :-
        pieces(N),
        board_size(Size),
        initial_pieces(Size, N, MySide),
        initial_pieces(Size, N, YourSide).

     board_size(6).
     pieces(4).
     lookahead(4).


:- kalah.

%  Adapted from Program 21.3  A complete program for playing Kalah in "The Art of
Prolog" by Leon Sterling
```